**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 1](#_Toc71733268)

[2. Bevezetés 3](#_Toc71733269)

[3. Felhasználói dokumentáció 3](#_Toc71733270)

[3.1 Rendszerkövetelmények 3](#_Toc71733271)

[3.2 A program telepítése 4](#_Toc71733272)

[3.3 A program használata 4](#_Toc71733273)

[3.3.1 Belépés az oldalra 4](#_Toc71733274)

[5](#_Toc71733275)

[3.3.2 A felhasználói felület 5](#_Toc71733276)

[3.3.3 *(1)*A File Browser 6](#_Toc71733277)

[3.3.4 *(2)* A megosztás ablak 8](#_Toc71733278)

[3.3.5 A szerkesztő felületről és egy projekt felépítéséről 9](#_Toc71733279)

[3.3.6 Az Editor felépítése 10](#_Toc71733280)

[3.3.7 Editor funkciói részletesen 11](#_Toc71733281)

[4. Fejlesztői dokumentáció 13](#_Toc71733282)

[4.1 Felhasznált technológiák 13](#_Toc71733283)

[4.1.1 Kliens oldal 13](#_Toc71733284)

[4.1.2 Szerver oldal 13](#_Toc71733285)

[4.1.3 Adatbázis 14](#_Toc71733286)

[4.1.4 Fejlesztői környezet 14](#_Toc71733287)

[4.2 Szerkezeti felépítés 15](#_Toc71733288)

[4.2.1 Client-side 16](#_Toc71733289)

[4.2.2 Szerver oldal 32](#_Toc71733290)

[5. Továbbfejlesztési lehetőségek 35](#_Toc71733291)

[5.1 File Manager 35](#_Toc71733292)

[5.2 Editor 36](#_Toc71733293)

[5.3 Felhasználói fiókok 36](#_Toc71733294)

[6. Hivatkozások 37](#_Toc71733295)

# 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 tervezek készíteni, amelyben közösen, akár egyidőben is lehet UML osztálydiagramokat és csomagdiagramokat szerkeszteni.

A program 3 rétegből áll. Egy angular kliensből, Java spring-boot back-end-ből, illetve postgreSQL adatbázisból. Az interaktív szerkesztő felülethez natív, szöveg alapú web socketeket használtam. Az összes többi kommunikáci REST API-n keresztül történik

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

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

A továbbfejlesztéshez ötleteket olvashatunk 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 800-1000mb 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 a chrome-ot, operát vagy firefox-ot ajánlok a használathoz. Az Internet Explorer és Edge böngészők nem támogatottak.

## A program telepítése

* Telepítsük fel A Java 8-hoz tartozó JRE-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. <- ez elv így működik, nem kell semmit mást installálni, mindent betettem a projekt mammába ami kell.

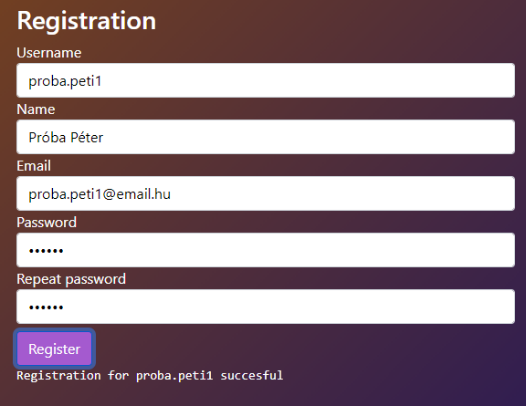
## A program használata

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

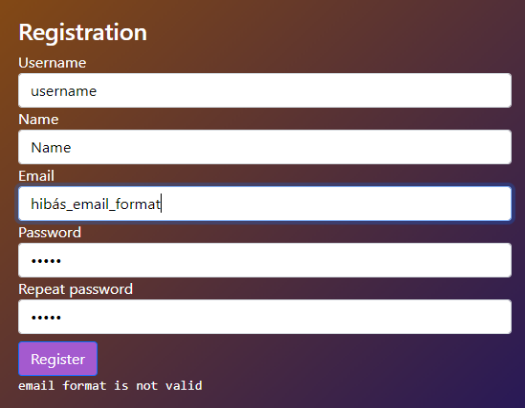
### Belépés az oldalra

|  |  |  |
| --- | --- | --- |
| 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. |

### 



1. ábra: Sikeres regisztáció



2. ábra Hiba a regisztáció során

### A felhasználói felület

Oldalt,illetve felül fordított L alakban a menu sávban a következő opciók vannak: Home, Editor, Files Belépés után a „Home” felületet látjuk, ahol a program köszönt minket.

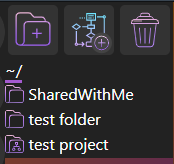
A **“be vagyunk lépve”** feltételt a továbbiakban **“BVL”** jelöléssel jeleztem.

|  |  |  |  |
| --- | --- | --- | --- |
| GIVEN | WHEN | THEN | EXPLANATION |
| BVL | Profile menüpontra kattintunk | **A főképernyőn** megjelennek a Profilunk részletei. | Itt megtekinthetjük, illetve módosíthatjuk az adatainkat; email, jelszó, username, stb… módosítására van lehetőség. |
| BVL | File browser menüpontra kattintunk | **Bal oldali panelen** kinyílik a file browser | link (1) |
| BVL, **nem** az EDITOR felület van megnyitva | Nagy “EDITOR” gombra kattintunk | **A főképernyőn** megjelenik a szerkesztő felület. | Ezen a felületen diagrammjainkat szerkeszthetjük. a fent leírt funkciókat továbbra is használhatjuk a **Bal oldali panelen.** A fájl böngészőben kikereshetjük diagrammjainkat, azokat megnyitva már szerkeszthetjük is. Ez élő socketes kapcsolattal történik, automatikus mentésekkel. Amint valaki más is elkezdi ugyanazt a diagrammot szerkeszteni, auomatikusan bekapcsolódik a session-ba. |

### (1)A File Browser

Itt tudjuk **böngészni** a **projektjeinket**. Vannak velem megosztott, általam megosztott és “privát”(nem megosztott) mappák, projektek.

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 afájlra navigálnak, mint amit a másik felhasználó is lát.

*3. ábra: file browser panel*

A file browser felület 2 részből áll, egy eszköztárból (felül) és nagyobb részt böngészőből (alul). A böngészőben 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.

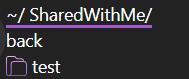
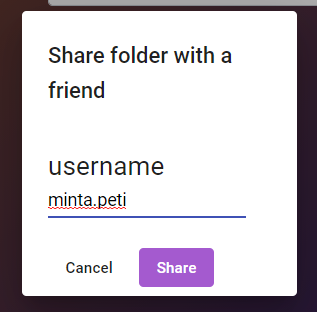
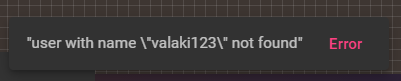
Minden normális fájlkezelőhöz hasonlóan a *File Browser* panel is rendelkezik az alapvető funkcionalitásokkal. Kiválasztás bal kattintással: egy elem, ctrl+ bal klikk-el egyesével lehet jelölgetni illetve jelölést megszűntetni. A kijelölt elemek kék keretet kapnak.

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. | [(2) Megosztás ablak](#_(4)_A_megosztás) |

### 

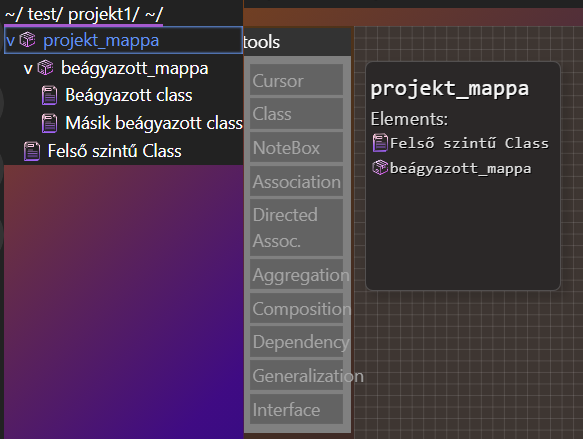
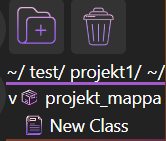
### (2) A megosztás ablak

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.

4. ábra: nem létező felhasználóval akarunk megosztani valamit

5. ábra: megosztás ablak

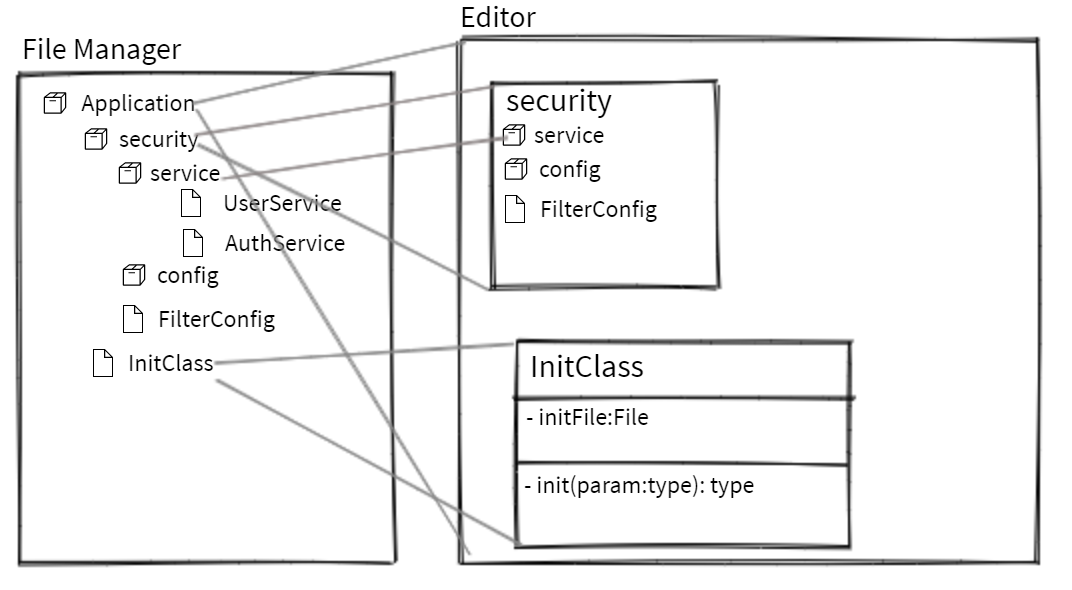
6. ábra: a mappa megosztás sikeres (minta.peti user)

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 session**-ban. Ha egy osztály definíciót létrehozunk, (Toolbox->Class) akkor az a szülő mappa alá virtuálisan bekerül a manager-ben, hogy jobban átlátható legyen, mi hova tartozik. Továbbá a mappa ikonja mellett balra található kis nyilacskával kinyithatók, illetve becsukhatók a mappában található elemek. Osztálydefinítiók szerkesztésekor a nagyszülő diagramban automatikusan frissülnek a csomag definíciók (akár élő kapcsolat esetén is)

7. ábra: Project management

8. ábra: A Manager és az Edtor szoros kapcsolata

### A szerkesztő felületről és egy projekt felépítéséről

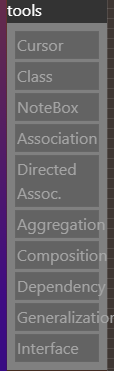
Egy projekt valójában diagrammok struktúrált halmaza. Egy projekt lényegében egy mappa, amelyben almappák(diagrammok) vannak. Egy diagram osztálydefiníciók, megjegyzések, kapcsolatok  **és beágyatott diagrammok** egy csomagja, halmaza. Egy projekt struktúrája pl. a következő képpen nézhet ki: 

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.

#### Toolbox ablak



10. ábra Eszköztár ablak

Itt Osztályleíró dobozok(Class Obect), 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 úgy osztály doboz jön létre. Méretét úgy állíthatjuk be, hogy az egér lenyomása után még lennt 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 kezetet 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, majd 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 szakasztra 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 hibás kéréseket kapunk, akkor a visszaállításokról itt kapunk tájékoztatást. Az egyes sorok fölé húzva az egeret az halvány kiemelést kap, illetve a tárgyban szereplő objektum is az Szerkesztő felületen.



11. ábra: Log ablak, működés közben

### Editor funkciói részletesen

|  |  |  |  |
| --- | --- | --- | --- |
| GIVEN | WHEN | THEN | EXPLANATION |
| Megosztott diagrammot szerkesztünk, egyedül | Valaki más is elkezdi szerkeszteni ugyanazt a diagrammot | Ő is **bekapcsolódik** a szerkesztésbe. | Ilyenkor az elemek **szerkesztését** **zárolásokkal** látja el a program, ezt a szerver a **memóriában** jegyzi. |
| Megosztott diagrammot 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 módisítása), **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, nézetek egymásba ágyazása könnyedén és átláthatóan. A nézetmodel (.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.

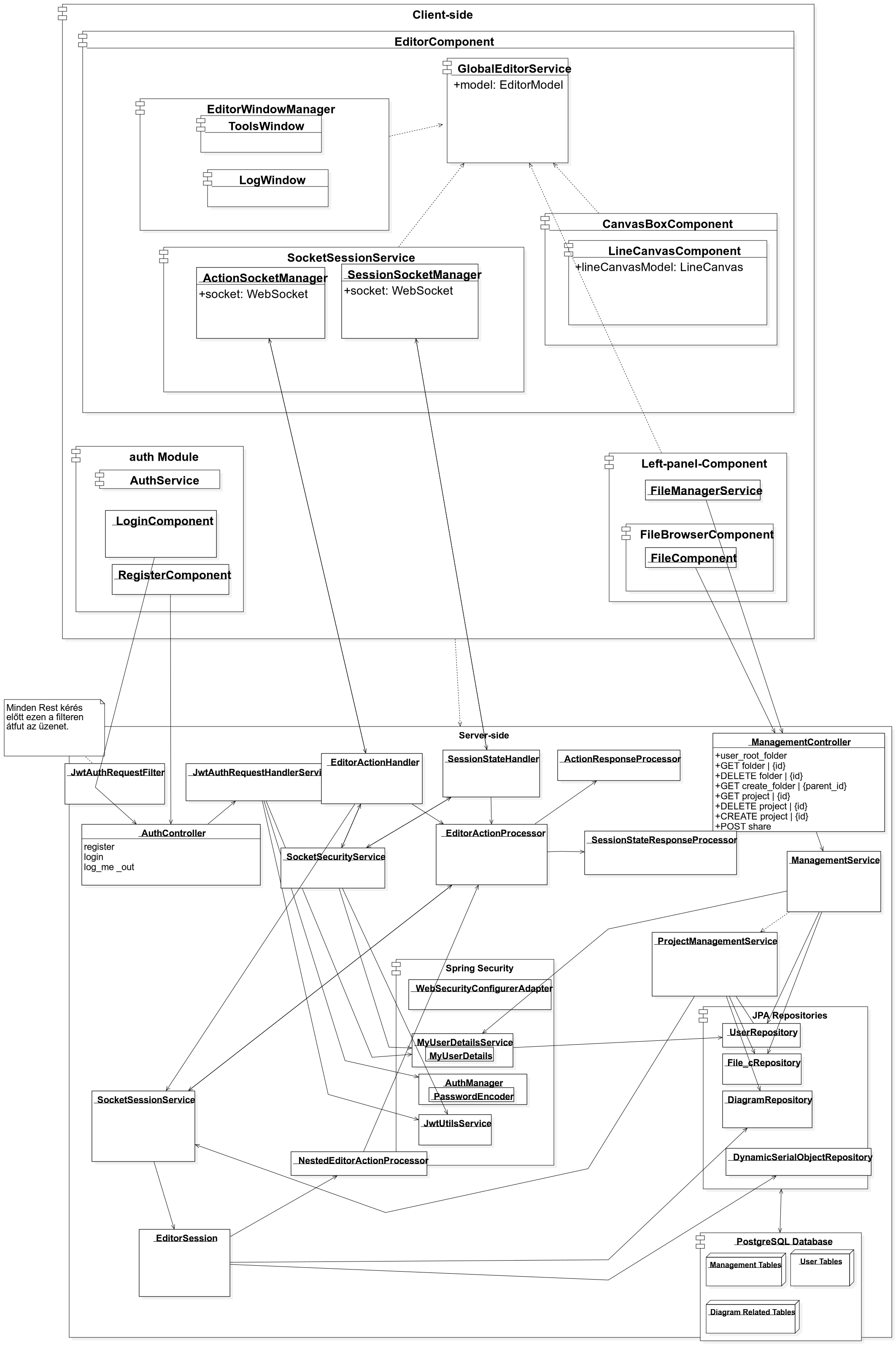
### Adatbázis

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. //dokumentáció hivatkozás

### Fejlesztői környezet

A Kliens oldal fejlesztéséhez Visual studio code-t használtam (<https://code.visualstudio.com>) néhány angular illetve html bővítmény telepítésével. A spring-boot szerver fejlesztéséhez Apache NetBeans 12.0-t(<https://netbeans.apache.org/download/nb120/>), a PostgreSQL adatbázis menedzseléséhez pedig PgAdmin4-et használtam.//irodalomjegyzékbe a linkeket

## Szerkezeti felépítés



NestedActionProcessor hiba van ->SocketSessionService-hez kéne közni, 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 <https://www.flaticon.com> oldalról töltöttem le. Az angular komponenseket úgy terveztem meg, hogy közülük minnél 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 endpoint-ra

##### 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 endpoint-ra

##### AuthService

Ez egy service. A servicek olyan ts osztályok, amelyek más osztályokba injektálhatók, dependency-ként. Ezt a @Injectable kulcsszó teszi lehetővé. Esetünkben az AuthService-nek egy lényeges Függvénye van: logOut(). Értelem szerűen 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. úgyhogy ha valaki időközben valaki megszerzi a tokenünket, 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 tűntesse ki magát.

##### Ősosztályok

* InteractiveItemBase
* InteractiveContainerBase

Az **InteractiveItemBase** egy olyan abstract 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 annak 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 eventet triggereli a service. Erre az eventre két komponens is fel van iratkozva:

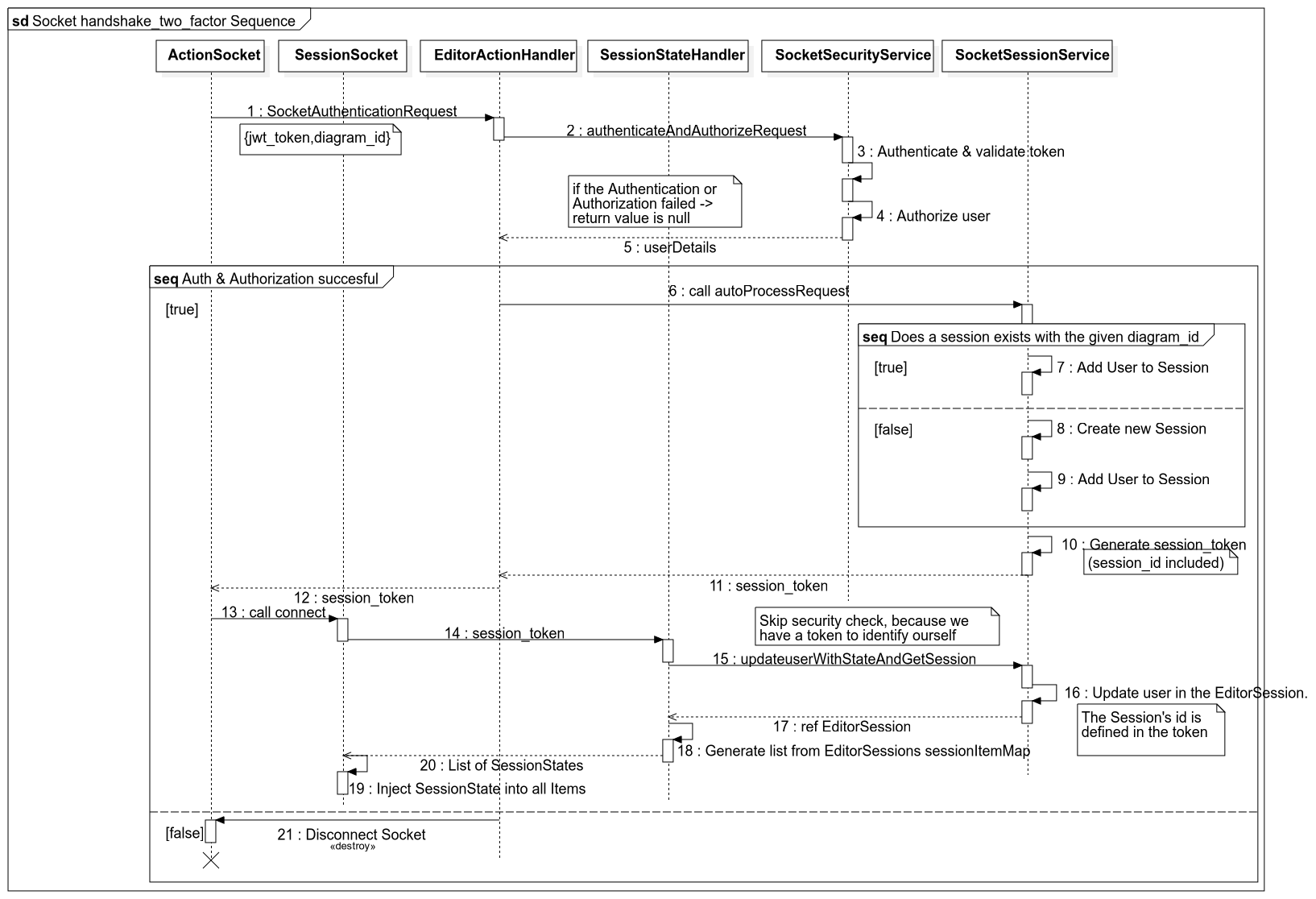
* LineCanvasComponent: az event hatására üríti az előző session vonalait, helyükre a frisseket inicializálja be.
* EditorSocketControllerService: az event hatására Létrehoz 2 socketet és csatlakozik a szerver websocket endpointjaihoz (action és state)

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 a ’canvas\_size\_update’ eseményre, amely tüzeléskor frissíti a canvas és a lineCanvas dimenzióit, hogy az összes objektum megjeleníthető legyen.

##### 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 nyílvántartott elemeket típusuknak megfelelően a következő listákba teszi:

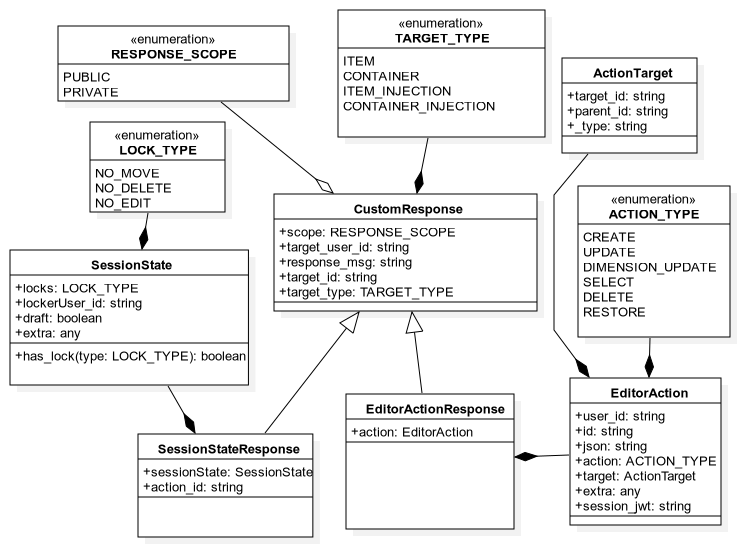
* itemViewModelMap:Pair<String, SessionInteractiveItem>[]
* containerViewModelMap:Pair<String, SessionInteractiveContainer>[]

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 session-hoz a token hashelve tartalmazza a felhasználónevünket illetve a szerver oldali session azonosítóját. Erről mellékelek egy szekvencia diagrammot, alább:

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 áll aktív kapcsolat.

###### ActionSocket

Az interaktív szerkesztőfelület központi hídja ez az osztály. Implementálja a **SocketWrapper\_I interface-t**, amelyben minden fontos függvény deklarálva van, amely egy socketes interakcióhoz kellhet. A connect függvénnyel csatlakoztathatjuk a socketünket a paraméterben kapott címhez. A socket default függvényeit, mint például az onmessage, onopoen vagy onclose, felüldefiniáljuk. Ha a socketes kézfogás, illetve http2.0-s protokollfejlesztés megtörtént, akkor az onopen hatására 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

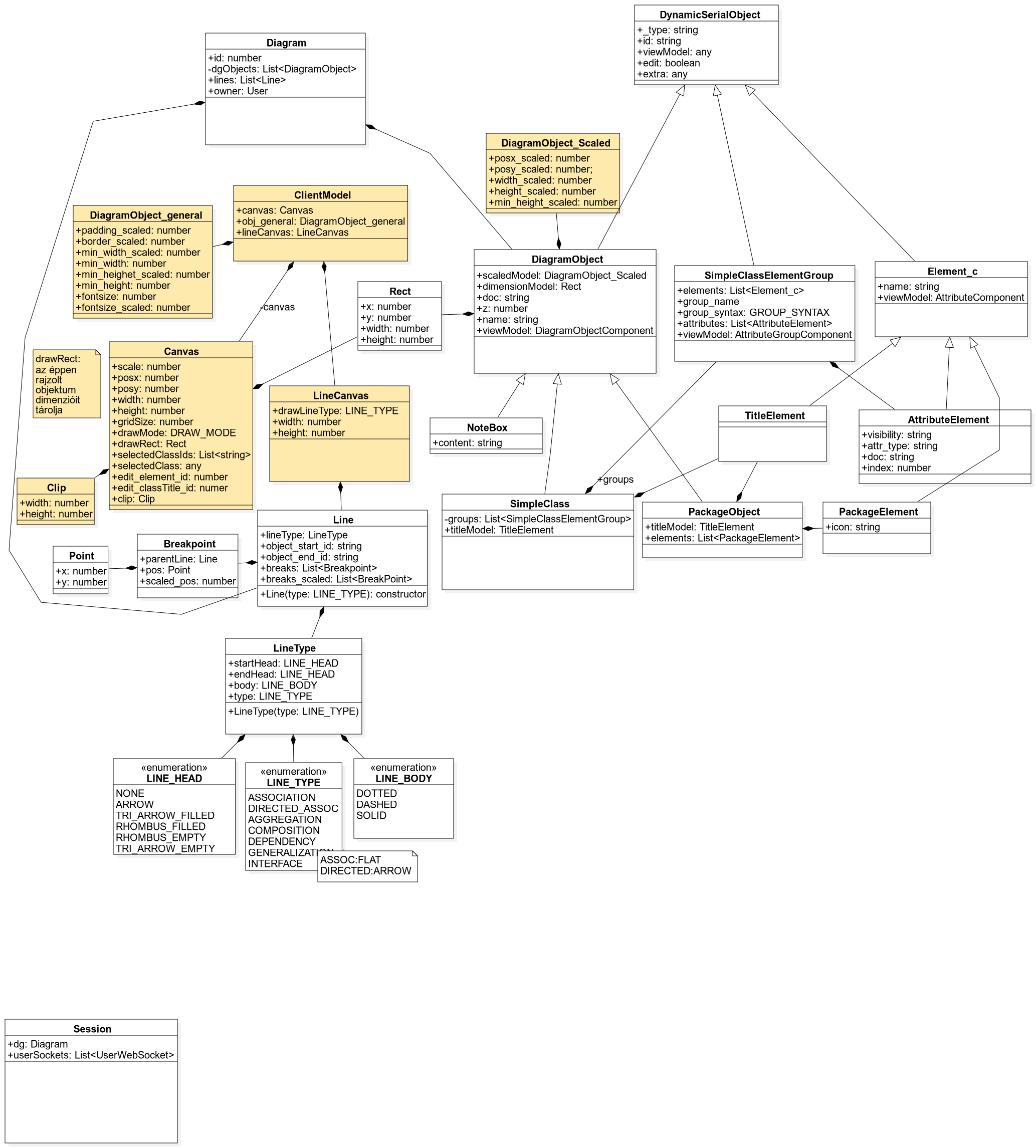
Bármikor, amikor a szerver küld egy akciót, akkor az **onmessage(e:any)** függvény fut le. Az üzenetet a függvény egyből egy EitorActionResponse objektummá alakítja. A **switch**-ben négy féle Akció típust különböztetünk meg:

* ACTIONTYPE.UPDATE
  + Valami módosítás történt. 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 kiparsolt 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 bele tette ’old\_id’ kulccsal a régi azonosítót. Ez alapján kikeresi a nézetmodellt majd frissít rajta. Ha nem mi vagyunk a tulajdonosok, akkor szimplán a szülő 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 handler-t. az oninitmessage 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 hatásáratűnnek el a null kifejezések a szerkesztőből. Az inicializálás után a this.parent.socket.onmessage = this.parent.onmessage; paranccsal átállítja a socket **onmessage** handler-jét az egyedi, általános handler-re. Az **onmessage** 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.



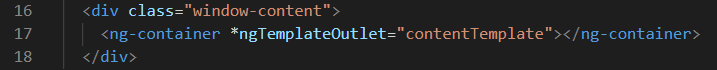
14. ábra: Az alkalmazás adatmodelljei (Objektum orientált megözelítésben)

#### CanvasBoxComponent

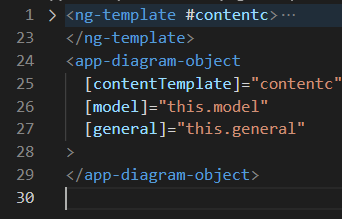
Ez a komponens implementálja a SessionInteractiveContainer interface-t, tehát ő is egy Interaktív konténer, lényegében a Diagram adatmodell nézetmodelljének az egyik fele. A szerkesztői funkcionalitás nagy része ebben a komponensben van leimplementálva. Néhány fonotsabb függvény:

* 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 DiagramObjetumot socketen keresztül, a szervernek.
* corrigateTargetClassPosition(), corrigateTargetClassDimension(): az éppen megfogott doboz pozícióját illetve méretét a tervezőtrácshoz igazítja.
* repositionCanvas(): amikor a vásznat mozgatjuk (egér lenyomása + mozgatás), akkor módosítja a canvas pozícióját.
* corrigateCanvasPosition(): A szerkesztő vásznat vissza igazítja a keretbe (Clip), ha esetleg kihúznánk belőle.
* A CanvasBoxComponent a html template fájljában felsorolja a DiagramObject-ből származtatott dobozokat.

##### DiagramObjectComponent

Minden doboztípusnak külön van komponenstdefiníciója. Az ősosztály megjelenítése a DiagramObjectComponent-ben történik, amely egy 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 kódjába meghívtam az ősosztály selectorját (16. ábra: 24-29-es sor) és Illetve beInjektáltam a gyermek komponens speciális dependenciáit ([model], [general]) , illetve a gyermekosztály belső nézetét, amelyet #contentc azonosítóval láttam el ([contentTemplate]). Ez több szempontból is jó nekünk. Egyrészt az ősoszáj 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. Mindegyiküknek már csak a saját, specifikus fukcionalitását kell megvalósítani a saját fájljaikban.

##### SimpleClassComponent

**SimpleClassComponent**-ben a doboz tartalma a Cím (AttributeComponent), és a csoportok (AttributeGroupComponent). Aztán a csoportokon belül, az attribútumok megjelenítése. Az **AttributeComponent** univerzálisnak számít, ugyanis minden **Element\_c** osztályból származtatott modellt ezzel a komponenssel jelenítek 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 megírta.

##### PackageObjectComponent

Ez a komponens különleges abból a szempontból, hogy ez csupán meglévő információkat összesít más diagrammokból 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 sorolja fel.

##### LineCanvasComponent

A szerkesztői funkciók maradék része ebben a komponensben kaptak helyet. Implementálja a SessionInteractioveContainer interface-t, tehát ez is ez konténer. Miket tárolunk? Vonalakat. Ez a komponens a Vonalak megjelenítéséről gondoskodik. A megjelenítéshez html5 canvas-t használtam 2 dimenziós kontextusban, és a beépített beginPath(), moveTo(x,y), stroke(), stb... funkciókra támaszkodva rajzolom a vonalakat. A komponens az előbb ismertetett CanvasBoxComponent részét képezi, ezért a legtöbb felhasználói eseménykezelést (pl.: kattintás, egérmozgatás, billentyű leütés, stb..) a szülő komponens delegálja és továbbítja. Néhány fontosabb függvény:

###### init()

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, amelytől elkérjük a 2d-s Context-et. Itt különösen vigyáznom kellett arra, hogy a context-et csak akkor próbáljam el 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.

###### drawBegin(e,type), drawMove(e), drawEnd(e):

Ők 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 drawLine() függvény tesz eleget.

drawLine()

Ez a függvény, amennyiben nincsenek töréspontok (BreakPoint) a vonal 2 végpontja közé, amely 2 darab **DiagramObject**, húz a canvas-on egy egyenes vonalat. Ha vannak töréspontok, akkor végig iterál a skálázott töréspontokon, (breaks\_scaled) (mert ugyan is van zoom funkciónk), és azok összekötésével alakul ki a spline.

#### EditorRootComponent

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

#### LeftPanelComponent

Ez a komponens tartalmazza a bal oldalipanelben elhelyezkedő file browsert. a ts fájl tartalmazza a kattintás illetve gomb események kezelését, a szerver rest endpontjait 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 +... {

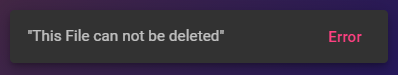
        headers: { 'Authorization': 'Bearer ' + getCookie("jwt\_token") }

      })

      .pipe(catchError(this.handleError<FileResponse>(this, '...', ...)))

      .subscribe((r) => { ... }

A fenti példában látszik, hogy **http GET** kérést indítunk a **default api url**-re, az Authorizációhoz a bejelentkezéskor kapott **token-t mellékelni kell** a kéréshez, különben **403**-as Forbidden http választ kapunk. A további **authorizáció** 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 service néhány http rest kérést intéző függvényt tartalmaz, amelyet közösen használ a FileComponent és a LeftPanelComponent itt nem történik hibakezelés, a nyers választ adják vissza a függvények.

#### FileComponent

Ez a komponens a fájlok megjelenítésére szolgál. A kapott fájl típusa és az ICON enum értéke segítségével szerint dönti el a megjelenítés formáját.

##### A fájlok megjelenítése

A típusok a következők lehetnek:

* FolderDto
* folder
* project
* ProjectFolderDto
* projectFileDto
* ProjectFile
* ProjectFolder

Az osztálytípust a json szövegbe futási időben a \_type mezőbe írja a jackson csomag. Ezt az összes fájltípus ős interfészében definiáltam @JsonTypeInfo és @JsonSubTypes annotációk segítségével. Ha az objektumokat A Jackson saját ObjectMapperjével 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 néhány event 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.

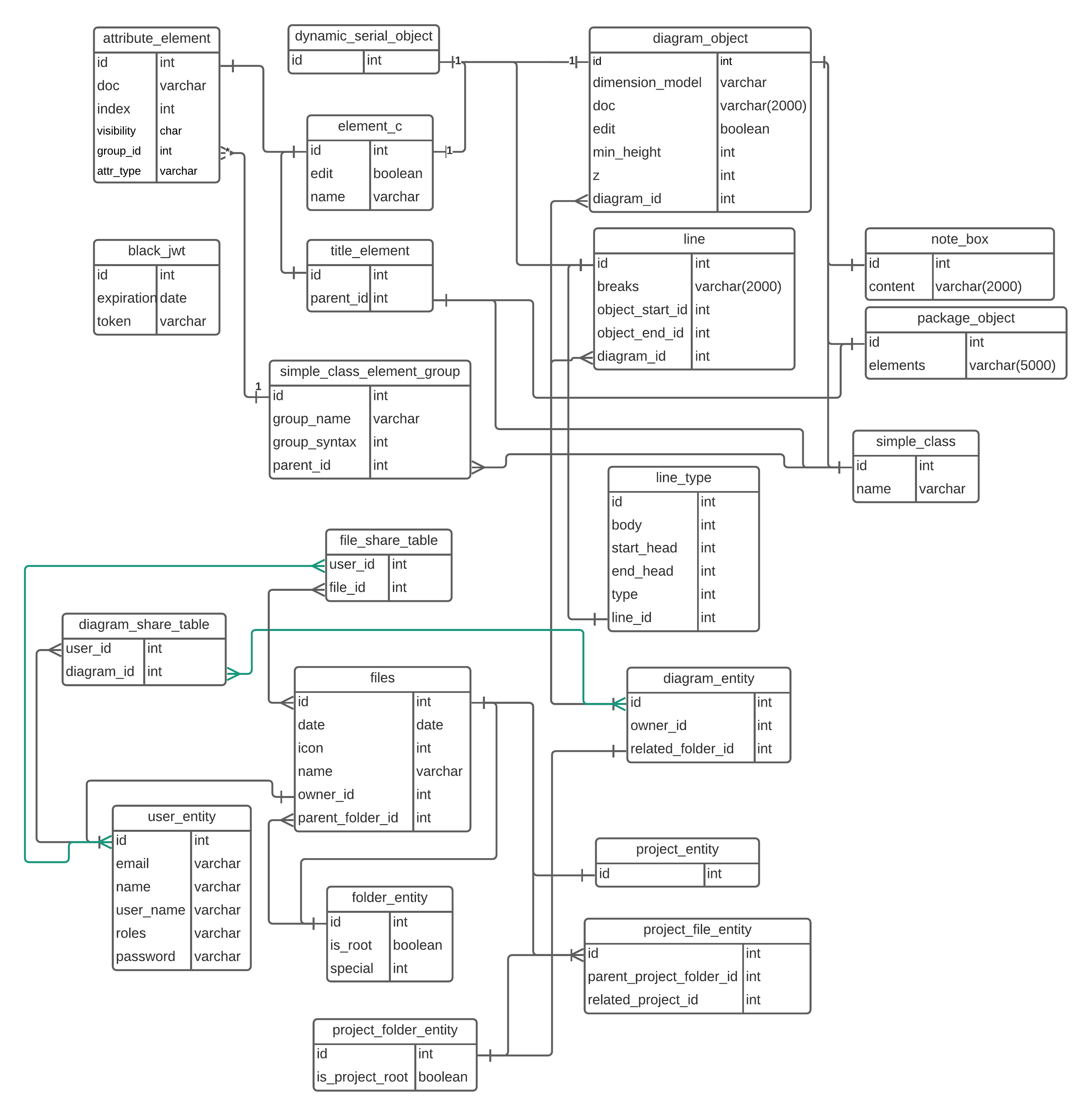
##### ShareDialogComponent

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.



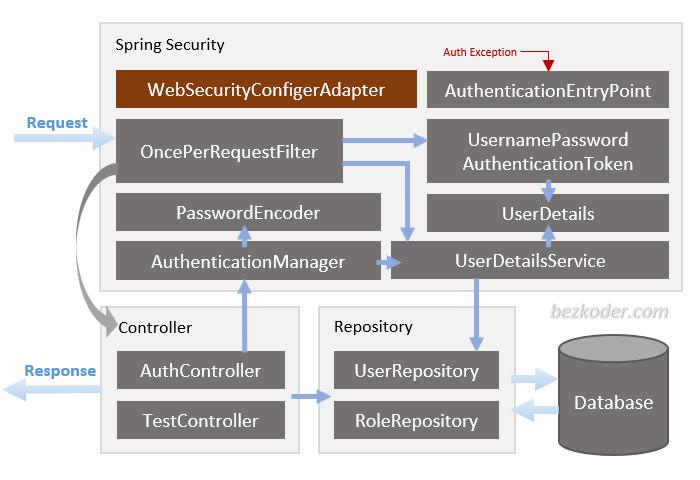
19. ábra: adatbázis táblák diagramja

#### 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 osztállyal kezdeném, mert kiemelten fontosnak tartom a felhasználói adatok és tartalmak védését. a Securty 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:[[1]](#endnote-1)



Itt a Request nyíl jelzi a http kérés beérkezésének helyét. A OncePerRequestFilter esetünkben a JwAuthRequestFilter. ahh erről majd egy sequence diagrammot csinálok.

##### JwtAuthRequestFilter

A filter minden egyes rest hívást megfog, és bele néz 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 service a security rétegben. Feladata, hogy minden tokenekkel kapcsolatos műveletet elvégezzen. itt kiemelnék néhány függvényt:

Az **extractUsername** és **extractExpiration** függvények a paraméterben megadott token-be csomagolt **felhasználónevet,** illetve **lejárati dátumot** csomagolják ki és adják vissza számunkra.

A **createToken** **függvény** a legfontosabb az egész osztályban. Ennek a meghívásával állíthatunk elő hiteles web tokeneket. A claims paraméter egy String kulcsokkal ellátott Object Map. A subject az maga felhasználónév, akire készítjük a token-t. A claims Map-ba bármilyen Serializable Objektumot betehetünk és azt az algoritmus bele írja 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 függvény** a paraméterben megadott token-t fekete listára helyezzük az adatbázisban. Azaz többé már nem használható azonosításra. A feketelistázott tokenek a lejárati dátumot követően egy **trigger** 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 user-t. 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 vissza küldjük 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, kissebb 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 interakciók lekezeléséért felel.

###### ProjectManagementController

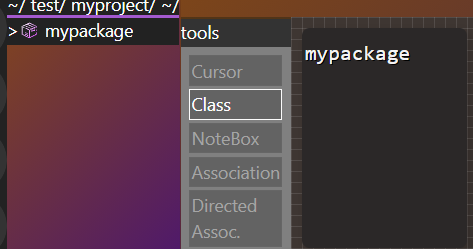
Erről a controllerről nem is mondanék sokat, működése nagyon hasonló a ManagementController-hez, annyi különbséggel, hogy itt a kontroller mögötti logikát a ProjectManagementService adja.

###### ProjectManagementService

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

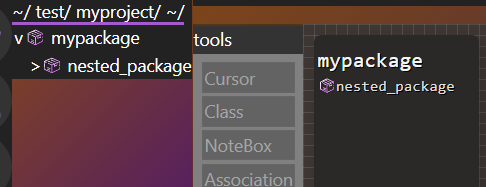
A **createProject függvény** projektek létrehozását végzi. A ProjectEntity konstruktor hívásánál automatikusan létrejön egy gyökér-projektmappa, ehez automatikusan létrejön egy DiagramEntity is.

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.



. 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.



. 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.

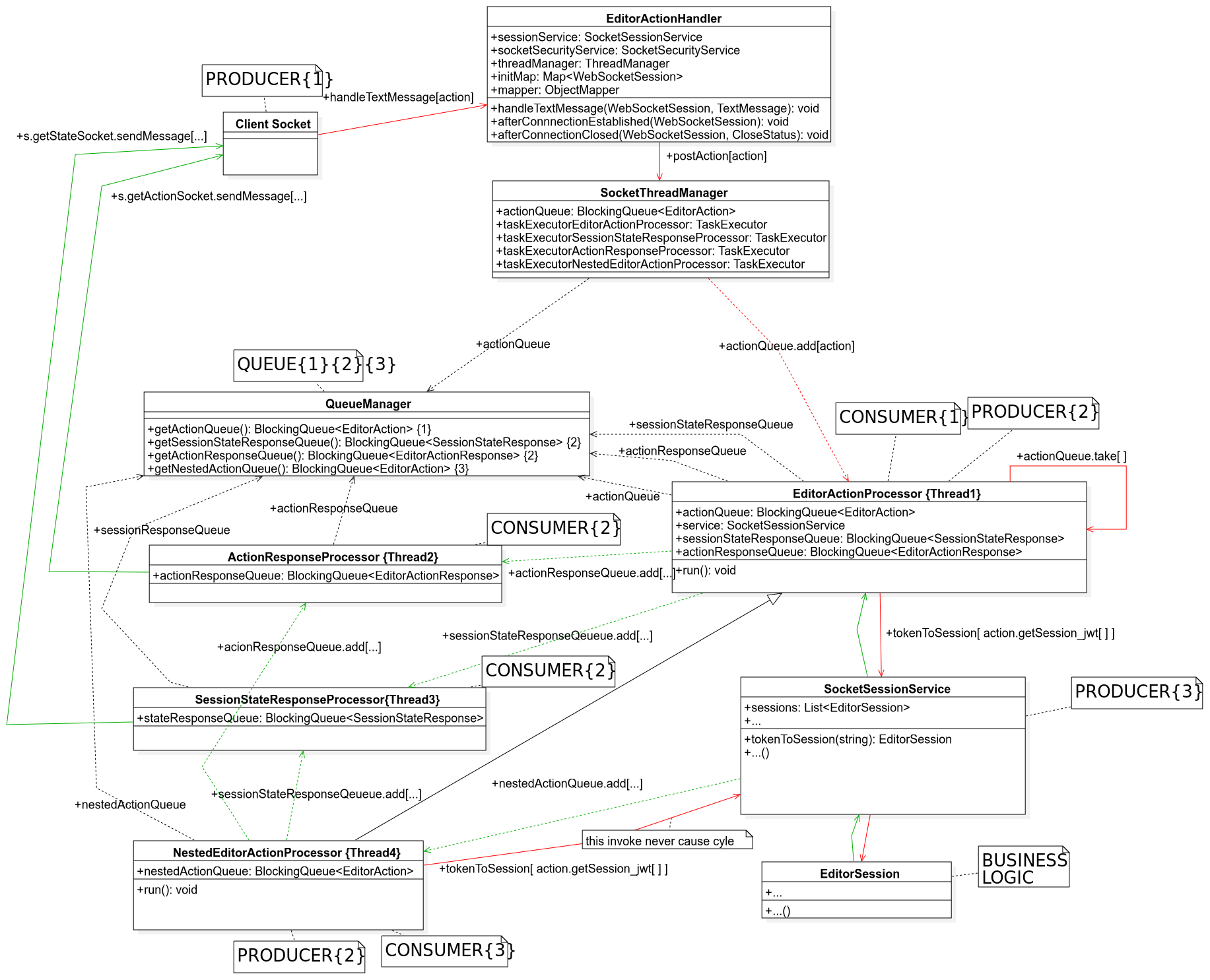
#### Editor

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 képessággel 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[[2]](#endnote-2) 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[[3]](#endnote-3) 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 állapotam 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.

# 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.

/////////

A függvényeket hogyan mutassam be?

* fv1
  + fv1 x-et és y-t tudja

-----------------------  
Az **fv1 függvény** x-et és y-t tudja...

Az **fv2 függvény** x2-t és y2-t tudja.....

---------------------

##### Fv1

Ez a függvény x-et és y-t tudja

//generikus kérdés:

?Szakkifejezések, magyarosítva? váltogatva?

//hibaüzeneteket amiket lehet kapni felhasznlás közben

//táblázatba, hivatkozzunk rá. Hiba elhárítási előírás.  
//a felhasználói, nem csak a program futtatásáról szól.

//help, oldalszámozás

//Mennyire kell részletezni a forrásokat? tutorial videók, stackoverflow posztok, angular példaprojektek, specifikus oldalak, A projektben a free ikonok honnan származnak, Spring security dokumentáció, hogyan kell batch

# Irodalomjegyzék

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

1. <https://www.designmycodes.com/examples/spring-boot-jwt-authentication.html> utolsó elérés dátuma: 2020.05.13 [↑](#endnote-ref-1)
2. <https://docs.spring.io/spring-framework/docs/3.0.0.M3/reference/html/ch04s04.html> [↑](#endnote-ref-2)
3. <https://github.com/FasterXML/jackson-docs> [↑](#endnote-ref-3)