FELADATKIÍRÁS

A feladatkiírást a **tanszék saját előírása szerint** vagy a tanszéki adminisztrációban lehet átvenni, és a tanszéki pecséttel ellátott, a tanszékvezető által aláírt lapot kell belefűzni a leadott munkába, vagy a tanszékvezető által elektronikusan jóváhagyott feladatkiírást kell a Diplomaterv Portálról letölteni és a leadott munkába belefűzni (ezen oldal HELYETT, ez az oldal csak útmutatás). Az elektronikusan feltöltött dolgozatban már nem kell megismételni a feladatkiírást.

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

Varga Ádám Marcell

MULTITENANT ÖTLETKEZELŐ ALKALMAZÁS SPRING ALAPOKON

KONZULENS

Dr. Forstner Bertalan

BUDAPEST, 2024

Tartalomjegyzék

Összefoglaló **7**
Abstract **8**
1 Bevezetés **9**
2 Ötletkezelő alkalmazások bemutatása **11**
3 A feladat részletes specifikálása **15**
3.1 Kezdeti követelmények 15
3.2 Végleges funkcionális követelmények 15
3.2.1 Authentikáció 17
3.2.2 Ötletdobozok 18
3.2.3 Ötletek 20
3.2.4 Kommentek 21
3.2.5 Pontozás 22
3.2.6 Lezárandó és lezárt ötletdobozok 23
3.2.7 Multitenancy 23
3.3 A pontozási rendszer 24
4 Felhasznált technológiák **25**
4.1 Angular 25
4.1.1 Angular CLI 26
4.1.2 Angular Material 27
4.2 ECharts 27
4.3 TypeScript 28
4.4 Spring Boot 28
4.4.1 Gradle 29
4.4.2 Spring Data JPA 29
4.4.3 Spring Security 29
4.4.4 JWT 30
4.4.5 MockK 30
4.4.6 JUnit 31
4.4.7 JaCoCo 31
4.5 MySQL 31
4.6 Figma 32

4.7 Git/GitHub ... 32
5 Tervezés ... 33
5.1 Architektúra tervezés ... 33
5.1.1 Alkalmazás felépítése ... 33
5.1.2 Multitenancy ... 34
5.2 Modell tervezés ... 36
5.2.1 Végleges modell ... 37
5.2.2 User ... 39
5.2.3 IdeaBox ... 39
5.2.4 Idea ... 39
5.2.5 Komment ... 39
5.2.6 ScoreSheet ... 40
5.2.7 ScoreItem ... 40
5.2.8 Egyéb osztályok, Enum-ok ... 40
5.3 UI Tervezés ... 41
Alkalmazás kezdeti drótváza ... 42
6 Implementáció ... 45
6.1 Első iteráció ... 45
6.1.1 Architektúra megvalósítása ... 45
6.1.2 Modell megvalósítás ... 47
6.1.3 Security ... 48
6.1.4 Kommunikáció ... 49
6.1.5 Bejelentkezés/regisztráció ... 51
6.1.6 Navigációs sáv ... 51
6.1.7 Főoldal ... 51
6.1.8 Ötletdoboz ... 52
6.1.9 Ötletek ... 53
6.1.10 Összegzés ... 55
6.2 Második iteráció ... 55
6.2.1 Pontozási rendszer implementálása ... 55
Multitenant architektúra implementálása ... 60
7 Tesztelés ... 64
7.1 Unit tesztelés ... 64
7.2 Integrációs tesztelés ... 65

7.3 End-to-End tesztelés (E2E) 66
8 Értékelés, továbbfejlesztési lehetőségek 67
8.1 Értékelés 67
8.2 Továbbfejlesztési lehetőségek 67
Irodalomjegyzék 69
Függelék 70

HALLGATÓI NYILATKOZAT

Alulírott **Varga Ádám Marcell**, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

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

Kelt: Budapest, 2024. 11. 23.

...…….

Varga Ádám Marcell

Összefoglaló

Az ötletkezelő alkalmazások, más néven Idea Management alkalmazások, olyan digitális eszközök és platformok, melyeket vállalatok és szervezetek használnak az innováció ösztönzésére és az ötleti kreativitás folyamatának támogatására. Ezek az alkalmazások támogatják az ötletek gyűjtését, azok bírálását és üzleti döntések megtételét, segítve ezzel a szervezeteket a rendszeres ötletgenerálásban, a közösségi együttműködésben és az ötletek hatékony menedzselésében.

A dolgozatom célja egy ehhez hasonló ötletkezelő alkalmazás prototípusának megtervezése és lefejlesztése és bemutatása. Az elkészített alkalmazás két részből áll. A felhasználók egy böngészőben elérhető vékonykliensen keresztül kommunikálnak a szerverrel, ami multitenant architektúrát valósít meg, ezzel csökkentve az üzemeltetési költséget és növelve a bérlők közötti adatbiztonságot. A kliens oldali megvalósítás Angular keretrendszerben, a szerver oldali megvalósítás pedig Spring Boot keretrendszert használva, Kotlin nyelven történt meg.

Abstract

Idea management applications, also known as Idea Management applications, are digital tools and platforms used by companies and organisations to stimulate innovation and support the process of idea creativity. These applications support the collection, evaluation and business decision making of ideas, helping organisations to generate ideas on a regular basis, collaborate and manage ideas effectively.

The aim of my thesis is to design and develop a prototype of a similar idea management application and to present it. The developed application consists of two parts. Users communicate with the server through a thin client accessible in a browser, which implements a multitenant architecture, thus reducing the operational cost and increasing the data security between tenants. The client-side implementation is done in Angular framework and the server-side implementation is done using Spring Boot framework in Kotlin language.

1 Bevezetés

A vállalkozások és szervezetek számára egyre fontosabb, hogy az ötleteket, javaslatokat gyorsan és hatékonyan kezeljék, különösen, ha a kreativitás és az innováció a siker kulcsát jelenti számukra. A növekedés során azonban az ötletek áramlása és feldolgozása gyakran megakad; különböző részlegek, csapatok és projektek között nehéz koordinálni a folyamatokat, és fennáll a veszélye annak, hogy értékes kezdeményezések elvesznek. Az elmúlt években egyre több cég választotta az agilis módszertant, amely a szoftverfejlesztés gyors és dinamikus megközelítését jelenti, és arra összpontosít, hogy az iterációk és az újra tervezések nem kudarcok, hanem éppen a hatékony fejlesztést segítik. Ebben a környezetben az ötletkezelő rendszerek szerepe is felértékelődik: a gyors visszajelzés, a folyamatos ötletfejlesztés és a rugalmasság mind az agilis szemlélet szerves részét képezik.

Ez a probléma minden olyan szervezetet érint, amely több csapattal vagy ügyféllel dolgozik együtt, különösen, ha a különböző ügyfelek elkülönítve, de egyazon rendszerben szeretnék kezelni saját ötleteiket. Az igény egy olyan multitenant ötletkezelő alkalmazás iránt tehát valós, amely több bérlő számára biztosít elkülönített adattárolást, de mégis egy közös platformot biztosít, támogatva az agilis fejlesztés követelményeit.

A piacon jelenleg több megoldás is jelen van a témakörben, ám ezeknek mindegyikének vannak előnyei és hátrányai. Ezekről részletesebben az 2. fejezetben írok.

A feladattal kapcsolatban a személyes érdeklődésem egyrészről a felvetett probléma megoldása volt, hiszen az ilyen ötletkezelő alkalmazások fontosak az üzleti életben az innovációs folyamatok kidolgozására és támogatására. Egy ilyen rendszert megvalósítani mind tervezési szempontból, mind technikailag is egy remek lehetőség arra, hogy elmélyítsem a tudásom az adott technológiákban. Különösen érdekes volt számomra a multitenant architektúrával való megismerkedés, ennek implementációs lehetőségei és felhasználási módjai.

A feladatkiírás alapján a dolgozatomban első lépésként a korszerű ötletkezelő alkalmazások áttekintését tartottam szükségesnek azért, hogy megértsem, milyen funkcionalitásokat kínálnak a piacon elérhető megoldások, és ez alapján be tudjam mutatni azokat. Mivel az alkalmazás keretrendszere kötött a feladatkiírásban, így a technológiák, amikkel dolgozni fogok a Spring Boot és az Angular lesz. A dolgozatban

ezt a két keretrendszert fogom alaposabban bemutatni és megmutatni, hogy milyen eszközökkel segítették az alkalmazás tervezését és fejlesztését. A feladatkiírás következő nagyobb egysége ezen ötletkezelő alkalmazás megtervezése. Itt az egyetlen elvárt feltétel az, hogy multitenant, azaz több bérlős architektúrát valósítson meg, így ezzel a résszel külön foglalkozni fogok a dolgozatomban. Ezzel a multitenant architektúrával azonban a biztonság még fontosabb szerepet kap, hiszen az ilyen architektúrákban az adatok nem mindig vannak teljesen szétválasztva a bérlők között. Az utolsó feladatom a tervezés után a megvalósítás volt, ahol egy olyan prototípust kell lefejlesztenem, amiben megtalálhatóak az ötletkezelő alkalmazásokra jellemző funkcionalitások és képességek.

A dolgozatom felépítése ezeket a pontokat sorba véve épül fel. A második fejezetben általánosságban mutatom be az ötletkezelő alkalmazásokat és azok funkcióit, illetve a piacon jelenleg elérhető megoldásokból is hozok példákat. A harmadik fejezetben bemutatom az alkalmazás megvalósulásának részleteit, illetve azt, hogy a funkcionális követelmények hogyan alakultak a 3 tárgy során, ami alatt az alkalmazás elnyerte a jelenlegi alakját. A negyedik fejezetben ezen lefektetett funkcionális követelményekből kiindulva megtervezem az alkalmazás architektúráját és modelljét. Az ötödik fejezetben a tervezett alkalmazás implementációjáról lesz szó, annak technikailag érdekesebb pontjait bemutatva. A hatodik fejezet témája az elkészült alkalmazás tesztelése. A hetedik fejezetben összefoglalom a dolgozat tanulságait, illetve kitérek az alkalmazás továbbfejlesztési lehetőségeire is.

2 Ötletkezelő alkalmazások bemutatása

Az ötletkezelő alkalmazások[1] alapvető célja, hogy strukturált és átlátható módon támogassák az innovációs folyamatokat a szervezeteken belül. Ezek a rendszerek elősegítik, hogy az ötletek és javaslatok ne vesszenek el, hanem szervezett formában kerüljenek rögzítésre, értékelésre és ha jónak bizonyulnak, akkor megvalósításra. Az ötletkezelő rendszerek különösen fontosak az innovációt és kreativitást előtérbe helyező szervezetek számára, legyenek azok akár nagyvállalatok, akár kisvállalkozások vagy nonprofit szervezetek. A rendszer használatával a cégek könnyebben tudnak reagálni a piaci kihívásokra, és egy olyan nyitott platformot biztosítanak a munkavállalók vagy akár az ügyfelek számára, ahol a friss ötletek, újítási javaslatok összegyűjthetők és fejleszthetők. Ezt több kutatás[2] is bizonyítja. Az ötletkezelő alkalmazásoknak vannak tipikus részei, ezek a következők.

Ötletgyűjtés

- Az ötletkezelő rendszerek központi eleme az ötletgyűjtési modul, amely lehetővé teszi, hogy a felhasználók (alkalmazottak, ügyfelek vagy egyéb érintettek) egyszerűen megoszthassák saját ötleteiket egy erre a célra kitalált gyűjtőhelyen.
- Ezen a platformon az ötletek különféle kategóriákba sorolhatók (pl. termékfejlesztés, hatékonyságnövelés, költségcsökkentés), így könnyebben rendszerezhetők és kezelhetők.
- Az ötletgyűjtés lehet nyitott (bárki számára elérhető a szervezeten belül) vagy zárt (csak meghatározott csapatok vagy személyek férhetnek hozzá), attól függően, hogy milyen célra és milyen típusú ötleteket gyűjtenek.

Ötletek értékelése

- Az ötletek értékelése elengedhetetlen ahhoz, hogy a szervezetek a legértékesebb javaslatokat emeljék ki. Az értékelés gyakran pontozási rendszerrel vagy szavazási funkcióval történik, így az ötletek egyszerűen rangsorolhatók.
- A rendszer általában lehetőséget biztosít a felhasználók számára, hogy kifejtsék véleményüket, vagy szavazatukkal támogassanak egy-egy ötletet.

- Az ötletek prioritásának meghatározása egyes rendszerekben akár mesterséges intelligencia alapú ajánlásokkal is történhet, amelyek figyelembe veszik a korábbi sikeres ötleteket és a szervezet aktuális céljait.

Közösségi hozzájárulás

- Miután egy ötlet bekerült a rendszerbe, a többi felhasználó részt vehet annak továbbfejlesztésében. Ezt gyakran közösségi funkciókkal – például kommentelési lehetőséggel, fájlok és linkek megosztásával – támogatják.
- A fejlesztési szakaszban a résztvevők közösen dolgozhatnak az ötlet finomításán, további javaslatokat tehetnek, kiegészíthetik azt, vagy részletesebben kidolgozhatják a megvalósítási tervet.

Elemzési és jelentési modul

- Az ötletkezelő rendszerekben található elemzési modul segítségével a menedzsment nyomon követheti, hogy hány ötlet érkezett be, melyik kategóriában keletkezett a legtöbb ötlet, illetve melyek bizonyultak sikeresnek.
- A statisztikai adatok és riportok alapja lehet a döntéshozatalnak, és lehetővé teszik az ötletkezelési folyamat hatékonyságának mérését, így az esetleges javítási lehetőségek feltárását.

A legtöbb ötletkezelőben a felhasználóknak többféle szerepkörük van. Ezek a szerepkörök befolyásolják, hogy az adott felhasználó mennyi jogosultsággal rendelkezik az alkalmazásban. Általánosságban ezek a szerepkörök fedezhetőek fel az ilyen típusú alkalmazásokban.

- **Alap felhasználók**

  Alapvető hozzáféréssel rendelkeznek, amivel ötleteket tudnak létrehozni, a többi ötletet böngészni, szavazni a nekik tetsző ötletekre és kommenteket fűzni hozzájuk.

- **Moderátorok**

  Ők az ötletfeltöltés és értékelés folyamatának irányítói, akik ellenőrzik az ötletek minőségét, biztosítják, hogy azok megfeleljenek a vállalat irányelveinek, és szükség esetén javítják a tartalmat vagy áthelyezik más kategóriába.

- **Adminisztrátorok**

  A rendszergazdák feladata a felhasználók kezelése, az ötletkezelő rendszer testre szabása, az új funkciók bevezetése, valamint az elemzési modulok kezelése.

- **Döntéshozók**

  A döntéshozók (pl. vezetők vagy projektmenedzserek) azok, akik az ötletek alapján döntéseket hoznak, meghatározzák a megvalósítási prioritásokat, és erőforrásokat rendelnek a különböző projektekhez.

A piacon jelenleg több népszerű ötletkezelő alkalmazás található, ezek közül a legelterjedtebbek az alábbi táblázatban láthatóak.

táblázat 1 - A piacon jelen lévő jelentősebb ötletkezelő alkalmazások

Név	Előnyök	Hátrányok	Link
Accept Mission	• Átfogó platform • Fejlett együttműködés • AI integráció • Gamifikáció	• Költség • Túl sok választási lehetőség • Rosszul kezelt folyamatok kockázata • Új felhasználók számára bonyolult	https://www.acceptmission.com/
BrightIdea	• Komplex Innovációs eszköztár • Csapatmunka támogatása • Felhasználóbarát felület • Átfogó riportolás	• Teljesítményproblémák • Nem intuitív funkciók • Szakértői támogatás szükséges • Nehéz testreszabhatóság	https://www.brightidea.com/

IdeaScale	• Egyszerű, intuitív • Közösségi interakciós funkciók • Skálázhatóság • Folyamatos fejlesztések	• Költséges lehet, mivel az előrehaladott funkciók a drágább csomagokban vannak • Adoptálási nehézségek • Korlátozott integrációs lehetőségek • Testreszabhatósági korlátok	https://ideascale.com/

3 A feladat részletes specifikálása

A feladatkiírás értelmezése és a piackutatás után a saját megoldásom tervezése és megvalósítása volt a következő feladat. Azért, hogy ez a feladat is minél életszerűbb legyen, a konzulensem egy megrendelő szerepkörébe bújt, aki egy ötletkezelő alkalmazást szeretnek készíttetni velem. Nekem, mint kivitelezőnek így az volt a feladatom, hogy először egy tervet és egy prototípust készítsek az alkalmazás alap működéséről, amit majd később agilis módon tovább lehet fejleszteni.

3.1 Kezdeti követelmények

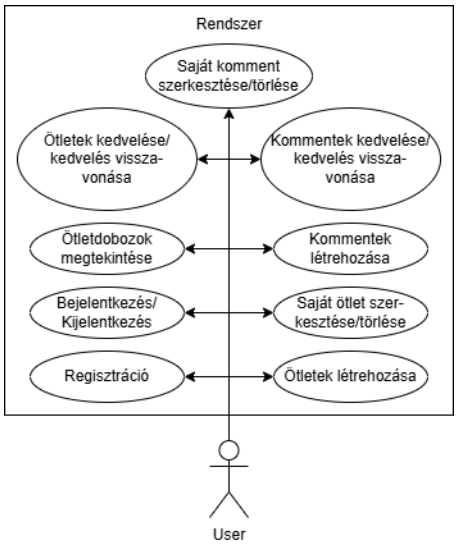
Az alkalmazás első tervezési fázisa az Önálló laboratórium 2 tárgy elején történt meg. Ekkor fektettem le a részben megadott technológiákkal az architektúrámat és írtam le a kezdeti funkcionális követelményeimet. A kezdeti funkcionális követelmények leginkább arra koncentrálódtak, hogy az alkalmazásban legyen user kezelés, a kezdeti modellemhez készüljenek el az alapvető CRUD (Create, Read, Update, Delete) műveletek és a felület legyen felhasználóbarát.

A kezdeti tervezési és implementálási fázis után az alkalmazás legtöbb funkciója elkészült, azonban a megrendelő a pontozási rendszert nem találta megfelelőnek, így egy új iteráció következett, ami a már kész alkalmazás funkcióit bővíti.

3.2 Végleges funkcionális követelmények

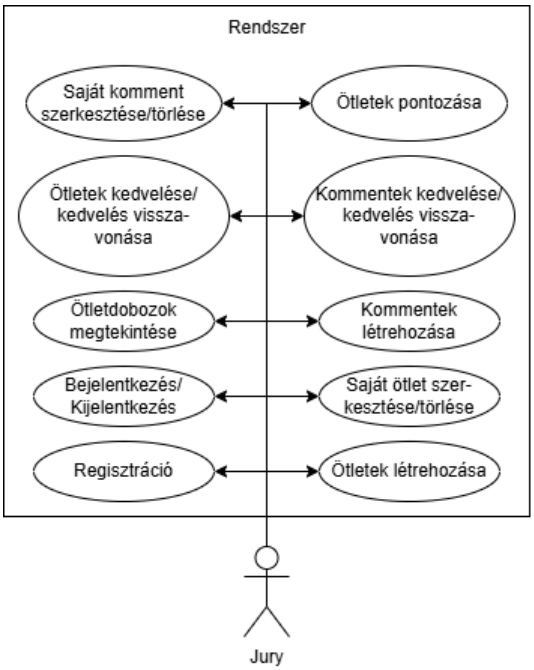
Ebben a második tervezési fázisban történt meg az alkalmazás végleges funkcionalitásának a megtervezése kitérve minden szerepkör jogaira is. Itt történt a jelenlegi bővíthető pontozási rendszer megtervezése, ami már a megrendelő igényeit is kielégítette. A követelményeket nagyobb logikai és funkcionális egységekre bontva mutatom be, ahol szétválasztom a szerepkörökhöz tartozó jogosultságokat is.

A követelmények gyors áttekintése érdekében készítettem use-case diagramokat, amik az egyes szerepkörökre lebontva mutatják be a különböző felhasználók által elérhető akciókat az alkalmazáson belül.



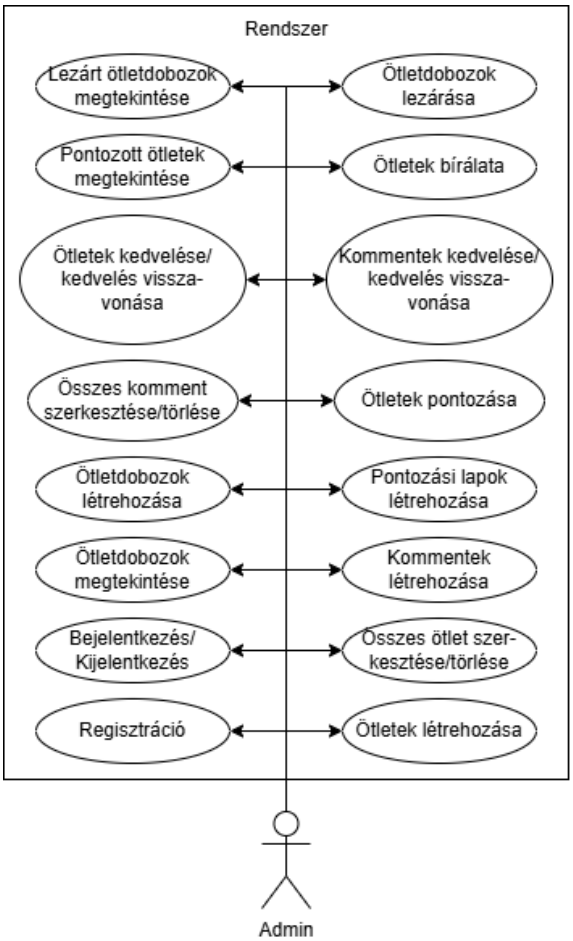
ábra 2 - User felhhasználó által elérhető akciók

A Jury felhasználó minden akciót el tud végezni, amit a User szerepkörrel rendelkező felhasználó tud, ezen felül a pontozási feladatot is megkapja.



ábra 1 - Jury felhhasználó által elérhető akciók

Az admin felhasználó megörökli az összes jogosultságot a User és a Jury felhasználótól, emellett új jogosultságokat is szerez. Szerkesztésnél már nem csak a saját maga által létrehozott objektumokat tudja szerkeszteni, hanem az alkalmazás minden részét.



ábra 3 - Admin felhasználó által elérhető akciók

3.2.1 Authentikáció

Ebben a részben az az alkalmazás authentikációs követelményeit mutatom be. Az olyan követelményekben, ahol egy-egy szerepkör nincs bemutatva, ott úgy kell értelmezni a funkcionalitást, hogy a kihagyott szerepkörnek nincs jogosultsága rá.

3.2.1.1 Regisztráció

User: A User szerepkörrel rendelkező felhasználó legyen képes regisztrálni az alkalmazásba, ha még nincs fiókja. Regisztráció során meg kell adnia a vezeték és keresztnevét, email címét és egy jelszavát.

Ha az adott email címmel az adott bérlőben (tenant-ban) már van regisztrált fiók, akkor ne lehessen új fiókot létrehozni. Ha eddig

nem használt email címmel történik a regisztráció akkor az sikeres és a felhasználó egyből be is jelentkezik az alkalmazásba.

Jury: Ezt a szerepkört nem lehet regisztráció útján betölteni, csak az Admin szerepkörrel rendelkező felhasználók adhatnak Jury szerepkört egy regisztrált User felhasználónak.

Admin: Ezt a szerepkört nem lehet regisztráció útján betölteni, csak az Admin szerepkörrel rendelkező felhasználók adhatnak Admin szerepkört egy regisztrált User felhasználónak.

3.2.1.2 Bejelentkezés

User: A User szerepkörrel rendelkező felhasználó legyen képes bejelentkezni az alkalmazásba abban az esetben, ha már rendelkezik regisztrált felhasználói fiókkal. A bejelentkezéshez meg kell adnia az e-mail címét és a jelszavát. Abban az esetben, ha az adott bérlőben létezik a megadott email és jelszó páros, akkor a bejelentkezés sikeres, ellenkező esetben sikertelen.

Jury: Ezzel a szerepkörrel a bejelentkezési folyamat megegyezik a User szerepkörrel.

Admin: Ezzel a szerepkörrel a bejelentkezési folyamat megegyezik a User szerepkörrel.

3.2.1.3 Kijelentkezés

User/Jury/Admin: A belépett felhasználó legyen képes kijelentkezni az alkalmazásból.

3.2.2 Ötletdobozok

3.2.2.1 Ötletdoboz létrehozása, szerkesztése, törlése

Admin: Az Admin szerepkörrel rendelkező felhasználók legyenek képesek ötletdobozok készítésére az alkalmazáson belül. Ötletdoboz készítésekor meg kell adni az ötletdoboz nevét, leírását kezdő és záró dátumát, illetve, hogy mely Jury szerepkörrel rendelkező felhasználók legyenek alapértelmezett értékelői az ötletdoboznak.

Létrehozás után az Admin szerepkörrel rendelkező felhasználók szerkeszthetik az ötletdoboz adatait, illetve törölhetik is az ötletdobozt. Ezzel törlésre kerül az összes olyan entitás, ami az ötletdobozhoz van kapcsolva (beérkezett ötletek, kommentek a beérkezett ötletekhez, pontozási lapok).

3.2.2.2 Pontozási útmutató az ötletdobozokhoz

Admin: Az Admin szerepkörrel rendelkező felhasználók legyenek képesek pontozási útmutatót készíteni a már létrehozott ötletdobozokhoz.

A pontozási útmutató létrehozásánál tetszőleges számú szempont felvehető, többféle értékelési móddal.

3.2.2.3 Ötletdobozok megtekintése

User/Jury: A User vagy Jury szerepkörrel rendelkező felhasználók legyenek képesek megtekinteni azon ötletdobozokat, amik nyilvánosságra vannak hozva, azaz létre vannak hozva és van hozzájuk rendelt pontozási útmutató. Ezt az alkalmazás nyitó oldalán tehetik meg, ahol keresni is tudnak a listában. Az ötletdobozokra rákattintva megtekinthetik az ötletdoboz alapvető adatait (ki hozta létre az ötletdobozt, mikor lett létrehozva és meddig tart, leírás és az alapértelmezett bírálók), illetve a már beérkezett ötleteket is.

Admin: Az Admin szerepkörrel rendelkező felhasználó is rendelkezik az összes jogosultsággal, amivel a User és a Jury felhasználó. Ezenfelül az Admin látja azon ötletdobozokat is a listában, amikhez még nincs pontozási útmutató rendelve és emiatt nem publikus.

3.2.2.4 Saját ötletdobozok listázása

Admin: Az Admin szerepkörrel rendelkező felhasználók legyenek képesek a saját maguk által készített ötletdobozok listázására. Ezt a profil oldalukon tehessék meg.

3.2.2.5 Ötletdobozok szerkesztése, törlése

Admin: Az Admin jogosultsággal rendelkező felhasználók legyenek képesek szerkeszteni az ötletdobozok adatait, illetve legyenek képesek törölni is az ötletdobozokat.

3.2.3 Ötletek

3.2.3.1 Ötletek létrehozása

User/Jury/Admin: A belépett felhasználó legyen képesek létrehozni ötleteket a publikus ötletdobozokba. Ehhez meg kell adnia az ötlet címét, leírását. Emellett tudjon hozzáadni az ötletekhez plusz bírálókat, ha szükségesnek látja. (Az ötletek automatikusan megöröklik azokat a bírálókat, akik hozzá vannak rendelve az adott ötletdobozhoz, amibe az ötlet létrehozásra kerül).

Ötletet csak akkor lehessen felvenni az ötletdobozba, ha az még nem járt le, azaz nyitott állapotban van.

3.2.3.2 Ötletek szerkesztése

User/Jury: A User vagy Jury szerepkörrel rendelkező felhasználók legyenek képesek szerkeszteni a saját ötleteiket.

Admin: Az Admin szerepkörrel rendelkező felhasználók legyenek képesek szerkeszteni nem csak a saját ötleteiket, hanem minden felhasználó ötletét abban a bérlőben, amibe tartoznak.

3.2.3.3 Ötletek törlése

Admin: Az Admin szerepkörrel rendelkező felhasználók legyenek képesek törölni nem csak a saját ötleteiket, hanem minden felhasználó ötletét abban a bérlőben, amibe tartoznak.

3.2.3.4 Ötletek megtekintése

User/Jury/Admin: A belépett felhasználó legyen képes megtekinteni az egyes ötletdobozokba érkező ötleteket. Az ötletekre rákattintva jelenjen meg az ötlet összes publikus adata (leírás,

pontozás (ha már van), létrehozó felhasználó neve, ötlet státusza, létrehozásának ideje, és a bírálók, akik bíráljak).

3.2.3.5 Ötletek listázása

User/Jury/Admin: A belépett felhasználó legyen képes egy listás nézetben megtekinteni a saját ötleteit. Ezt a saját Profil oldalán tehesse meg.

3.2.3.6 Ötletek kedvelése, kedvelés visszavonása

User/Jury/Admin: A belépett felhasználó legyen képes az ötleteket kedvelni. Ezt a like jelre kattintva teheti meg. Az ötleteken látszódjon, hogy mennyi ember kedveli.

Ha egy felhasználó úgy dönt, hogy mégsem tetszik neki az ötlet, akkor legyen képes a kedvelését visszavonni az ötletről. Ezt a like jel újbóli megnyomásával teheti meg.

3.2.4 Kommentek

3.2.4.1 Kommentek létrehozása

User/Jury/Admin: A belépet felhasználó legyen képes kommenteket létrehozni az egyes ötletekhez.

3.2.4.2 Kommentek megtekintése

User/Jury/Admin: A belépett felhasználó legyen képes megtekinteni az egyes ötletekre érkező kommenteket. Az ötletekre rákattintva a publikus adatok mellett a „Comments” fülön tehesse ezt meg.

3.2.4.3 Kommentek szerkesztése

User/Jury: A User vagy Jury szerepkörrel rendelkező felhasználó legyen képes szerkeszteni a kommentjét. Ha a komment szerkesztett, akkor ez jelenjen meg a felületen.

Admin: Az Admin szerepkörrel rendelkező felhasználók legyenek képesek szerkeszteni nem csak a saját kommentjeiket, hanem minden felhasználó kommentjét abban a bérlőben, amibe tartoznak.

3.2.4.4 Komment kedvelése, kedvelés visszavonása

User/Jury/Admin: A belépett felhasználó legyen képes a kommenteket kedvelni. Ezt a like jelre kattintva teheti meg. Az kommenteken látszódjon, hogy mennyi ember kedveli.

Ha egy felhasználó úgy dönt, hogy mégsem tetszik neki a komment, akkor legyen képes a kedvelését visszavonni a kommentről. Ezt a like jel újbóli megnyomásával teheti meg.

3.2.5 Pontozás

3.2.5.1 Ötletek pontozása

Jury/Admin: A Jury vagy Admin szerepkörrel rendelkező felhasználó legyen képes pontozni a beérkezett ötleteket. Ezt akkor tehesse meg, ha:

- Alapértelmezett bírálója annak az ötletdoboznak, amibe az ötlet érkezett, vagy
- Az ötlethez külön hozzá lett adva, mint bíráló

A pontozást az ötletdobozhoz tartozó pontozási lap kitöltésével teheti meg. A pontozás akkor tekinthető lezártnak, ha minden szempont szerint értékelte az ötletet.

Ha az ötlet pontozva lett (akár 1 bíráló által is) akkor az kerüljön át a SUBMITTED státuszból a REVIEWED státuszba.

3.2.5.2 Pontozott ötletek listázása

Admin: Az Admin szerepkörrel rendelkező felhasználó legyen képes listázni a már pontozott ötleteket.

3.2.5.3 Pontozott ötletek elfogadása, elutasítása

Admin: Az Admin szerepkörrel rendelkező felhasználó legyen képes elfogadni vagy elutasítani a pontozott ötleteket. Ezt csak akkor tehesse meg, ha az adott ötletre minden olyan bíráló adott le pontozást, akinek ez kötelező volt.

Ha egy ötlet elfogadott vagy elutasított státuszba került, akkor ezt a változást kövesse státuszváltozás is (APPROVED, DENIED).

3.2.6 Lezárandó és lezárt ötletdobozok

3.2.6.1 Ötletdoboz lezárása

Admin: Az Admin szerepkörrel rendelkező felhasználó legyen képes lezárni az ötletdobozokat. Ezt csak akkor tehesse meg, ha az ötletdoboz már nem aktív (azaz a lezárási határidő már elmúlt) és minden beérkező ötlet vagy elfogadott, vagy elutasított státuszba került.

3.2.6.2 Lezárt ötletdobozok listázása

Admin: Az Admin szerepkörrel rendelkező felhasználó legyen képes listázni a már lezárt ötletdobozokat és megtekinteni, hogy mely ötletek lettek elfogadva, illetve elutasítva.

3.2.7 Multitenancy

3.2.7.1 Architektúra

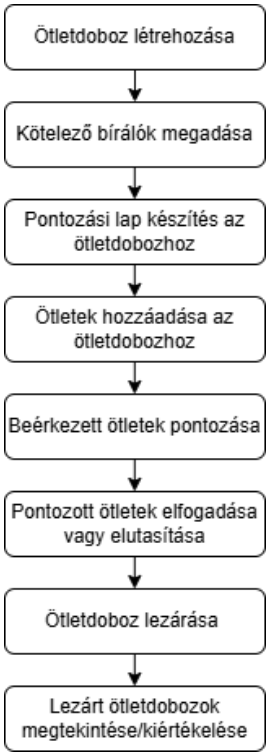
Az alkalmazás valósítson meg valamilyen multitenant architektúrát.

3.2.7.2 Adatizoláció

Az alkalmazást használó bérlők ne érhessék el és ne változtathassák egy másik bérlő által kezelt adatokat.

3.3 A pontozási rendszer

Az alkalmazás pontozási rendszerére érdemes kitérni, hiszen egyrészről ez az alkalmazás egyik, ha nem a leg lényegesebb része, ebből kifolyólag pedig a leg komplexebb folyamat. Ez a rendszer a második tervezési fázisban nyerte el a mostani formáját, ami az ábra 4 - Pontozási folyamat ábrán látható. A folyamat az ötletdoboz elkészítésével kezdődik, ahol meg kell adni az ötletdoboz alapvető adatait, illetve a kötelező bírálókat. Ezután az Admin szerepkörrel rendelkező felhasználó létrehoz hozzá egy pontozási lapot, ami alapján a bírálók majd a bírálatukat készítik a beérkező ötletekre. Ezután az ötletdoboz megnyílik a nyilvánosság elé, bárki szabadon adhat hozzá ötleteket addig, amíg az ötletdoboznak le nem jár az ideje. A bíráló felhasználók már ekkor is értékelhetik az ötleteket. Akkor, amikor egy ötletre minden olyan bíráló leadta a pontozását, akinek ez kötelező volt, akkor az Admin szerepkörrel rendelkező felhasználók elfogadhatják, vagy elutasíthatják az ötleteket. Miután egy adott ötletdobozban minden ötlet el lett bírálva, és az ötletdoboz már nem fogadhat új ötleteket (mert lejárt az ideje), akkor az ötletdoboz lezárásra készen áll, ezt a lépést szintén az admin szerepkörrel rendelkező felhasználók tehetik meg. Lezárás után az admin felhasználók megtekinthetik a lezárt ötletdobozokat.



ábra 4 - Pontozási folyamat

4 Felhasznált technológiák

4.1 Angular

Az Angular[1] egy olyan platform és keretrendszer, amelyet a single-page webalkalmazások (SPA)[3] fejlesztéséhez terveztek. Ez azt jelenti, hogy az alkalmazás egy HTML fájlt tölt be a böngészőben, és futás közben JavaScript segítségével módosítja azt a megjelenítendő nézeteknek megfelelően, nem pedig egy új HTML[2] oldalt tölt be.

Az Angular architektúrája modulokra és komponensekre épül. A komponensekre bontás jelentősége az, hogy egyrészt újra felhasználható részeket lehet írni, másrészt pedig, ha egy funkciót módisítani kell, akkor azt sok esetben elég egy komponensben elvégezni, nem kell egész oldalakat átírni. A modulok logikai részeket csoportosítanak össze. Egy Angular alkalmazás több modulból állhat, de mindenképpen tartalmaz egy gyökérmodult.

Az Angular 2 és az azt követő verziókban a fejlesztés két fő nyelven történik: HTML és TypeScript (TS). Ezenkívül a weboldalunk megjelenését CSS[3] fájlokkal módosíthatjuk. Alapvetően egy komponens egy HTML fájlból, egy TypeScript fájlból (és esetenként egy CSS fájlból) áll. Az HTML fájlt általában sablonfájlnak nevezik, ami meghatározza a komponens szerkezetét, és ezt tölti be a Dokumentum Objektum Modelljébe (DOM)[4], amikor a komponens betöltődik. Ehhez tartozik egy TypeScript fájl, ahol a komponens logikáját és metódusait lehet leírni. A megjelenítendő adatokat itt deklarálják, és itt kezelik a bejövő és kimenő adatokat. Ha a komponenshez tartozik CSS fájl, akkor abban lehet a komponensre szabott stílusokat alkalmazni. Alapvetően az Angular-ban ezek a stílusok csak arra a komponensre vonatkoznak, amelyhez tartoznak, de ezt a funkciót ki lehet kapcsolni.

---

[1] *https://angular.io/guide/what-is-angular*

[2] https://www.w3schools.com/html/html_intro.asp

[3] https://developer.mozilla.org/enUS/docs/Glossary/CSS

4.1.1 Angular CLI

Az Angular CLI (Command Line Interface)[4] egy eszköz, aminek a segítségével Angular alkalmazásokat egyszerűbb létrehozni, fejleszteni és kezelni parancssori utasítások segítségével. Az CLI legfontosabb funkciója az Angular projekt csomagolása. Ezt úgy érjük el, hogy lefordítjuk a TypeScript fájlokat, majd összecsomagoljuk a kapott JavaScript fájlokat, HTML fájlokat, CSS-t (vagy más, esetleg külön fordítást igénylő stíluslapokat, például SCSS-t). Az összecsomagolás előnye, hogy az egész projektet csak néhány fájl importálásával lehet használni, nem szükséges mindegyiket külön importálni.

A fejlesztés támogatásához az CLI számos parancssori utasítást nyújt a fejlesztő számára, közülük a legfontosabbak:

- **ng new**

  Új Angular projektet hoz létre. Utána paraméterként írható név, ami a projekt megnevezése lesz. A generálás során több opciót is meg lehet adni parancssori kérdés-válasz formában

- **ng generate**

  A meglévő projektbe generál új komponenseket, szolgáltatásokat, direktívákat, moduloka és még sok mást.

- **ng serve**

  Az Angular alkalmazás futtatása, alapértelmezetten a localhost:4200-on. Ha fut az Angular projekt, akkor az automatikus változásdetektálás miatt, ha a fájlok változnak, az oldal frissül, ami megkönnyíti a dinamikus fejlesztést.

- **ng build**

  Az alkalmazás lefordítását és összecsomagolását végzi el. A parancs végrehajtása után az alkalmazás készen áll, hogy egy szerveren kiszolgálja a felhasználókat.

---

[4] https://angular.io/cli

4.1.2 Angular Material

Az Angular Material[5] egy UI függvénykönyvtár, amit a Google fejleszt. Előre definiált komponenseket és felületi elemeket tartalmaz, amelyek könnyen és egyszerűen alkalmazhatóak az alkalmazásokban. A komponensek a Material Design irányelveit követik, elérhetőek több változatban, de személyre szabásra is nyújt lehetőséget a könyvtár. Használatuk egy csomagkezelővel (például npm) történő telepítés után nagyon egyszerű és kényelmes. A komponensek modulokra vannak osztva, melyeket külön-külön lehet importálni a projektbe, vagy egy adott modulba a projekten belül, így az alkalmazásba csak az kerül bele, amire valóban szükség van.

4.2 ECharts

Az ECharts[6] egy rugalmas és teljesítményorientált grafikon- és diagramkészítő könyvtár, amely széleskörű lehetőségeket kínál az adatok vizualizálására. Az ECharts használatával különféle diagramtípusokat, például vonaldiagramokat, oszlopdiagramokat és kördiagramokat lehet létrehozni és prezentálni komplex adathalmazokat. Az Angular keretrendszerrel való integrációja zökkenőmentes, így garantálva a fejlesztők számára, hogy könnyedén beépítsék a grafikonokat az alkalmazásaikba, kihasználva az Angular dinamikus adatkezelési képességeit.

Az ECharts használata Angular-ban lehetőséget biztosít a vizuális elemek testre szabására, így a felhasználói élmény fokozása érdekében a diagramok dinamikusan reagálhatnak az adatok frissítésére. A könyvtár támogatja az interaktív elemeket, mint például a tooltipok és a zoom funkciók, amiknek a segítségével a felhasználók részletesebb információkat érhetnek el a diagramok vizsgálata során. Az ECharts és Angular kombinációja jó megoldást a modern webalkalmazások számára, amelyek igényes és interaktív adatmegjelenítést akarnak használni.

---

[5] https://material.angular.io/

[6] https://echarts.apache.org/en/index.html

4.3 TypeScript

A TypeScript[7] egy nyílt forráskódú, JavaScript alapú programozási nyelv, amit a Microsoft fejlesztett ki. A nyelv kiemelkedő tulajdonsága, hogy támogatja mind az explicit, mind az implicit típusokat, ami miatt objektumorientált. A TypeScript kódja JavaScriptre fordul le, így a meglévő JavaScript kód is működik TypeScript fájlokban. Ez különösen előnyös a webfejlesztés területén, mivel a legtöbb weboldal kliensoldali kódja JavaScript. Ráadásul a fejlesztőknek nem szükséges más nyelvet tanulniuk a szerveroldali fejlesztéshez sem, mivel a Node.js JavaScriptet futtat. Ennek köszönhetően egyetlen nyelv használatával, kizárólag TypeScript segítségével lehet teljes (full stack) alkalmazást írni.

4.4 Spring Boot

A Spring Boot[8] egy rendkívül népszerű keretrendszer, amely a Java platformra épül, és célja a webalkalmazások gyors fejlesztésének megkönnyítése. A Spring Boot a Spring keretrendszerre épít, és automatikus konfigurációt, beépített szervereket, valamint előre definiált sablonokat kínál, amelyek lehetővé teszik a fejlesztők számára, hogy minimális beállítással és konfigurálással hozzanak létre robusztus alkalmazásokat. A Spring Boot nagy előnye, hogy támogatja a microservice architektúrát, amely lehetővé teszi a könnyű skálázhatóságot és karbantartást.

A Spring Boot tökéletesen integrálható a Kotlin[9] programozási nyelvvel, amely modern szintaxist és nullabilitáskezelést biztosít, ezzel segítve a fejlesztést. A Kotlin használata a Spring Boot alkalmazásokban segít abban, hogy tisztább, kifejezőbb és biztonságosabb kódot írjon a fejlesztő, miközben megőrzi a Java ökoszisztéma előnyeit, mint például a gazdag könyvtárak és eszközök elérhetőségét. A Spring Boot és a Kotlin kombinációja ideális megoldást a fejlesztők számára, lehetővé téve a gyors, könnyen karbantartható és hatékony alkalmazások létrehozását, amelyeket az iparági standardoknak megfelelően terveztek.

---

[7] https://en.wikipedia.org/wiki/TypeScript

[8] https://spring.io/projects/spring-boot

[9] https://kotlinlang.org/

4.4.1 Gradle

A Gradle[10] egy fejlett eszköz, amit arra találtak ki, hogy automatizálja a szoftver projektek build folyamatát. A Gradle kiválóan támogatja a Spring Boot projektek kezelését, különösen a komplex függőségek és konfigurációk esetén. A Gradle rugalmassága és gyorsasága teszi a legjobb jelöltté a Spring Boot alkalmazások fejlesztésére és karbantartására, mivel könnyedén kezelhetők a build-szkriptek és a függőségek. A Gradle támogatja a deklaratív függőségkezelést a build.gradle fájlban, amely leegyszerűsíti a szükséges Spring Boot és egyéb könyvtárak integrálását.

4.4.2 Spring Data JPA

A Spring Data JPA[11] egy hatékony és kényelmes eszköz, amely a Spring keretrendszer részeként a Java Persistence API (JPA) fölé építkezik, ezzel biztosítja a relációs adatbázisokkal való egyszerű és intuitív interakciót. A Spring Data JPA célja, hogy minimalizálja a boilerplate kód mennyiségét az adatkezelés során, így a fejlesztők hatékonyabban dolgozhatnak. A keretrendszer támogatja a lekérdezések automatikus generálását a metódusok elnevezésén alapulva, emellett rugalmasan integrálható különböző adatbázis-kezelő rendszerekkel. A Spring Data JPA segítségével a komplex lekérdezések végrehajtása, a tranzakciók kezelése és a paginálás (lapozható adatelérés) leegyszerűsödik. Emiatt a fejlesztőknek csak speciális esetekben kell közvetlenül SQL lekérdezésekkel foglalkozniuk, az alap eseteket meg lehet oldani kód nélkül. A Spring Data JPA egy remek megoldás a modern Java vagy Kotlin alapú alkalmazások adatkezelési igényeinek kielégítésére, hiszen gyorsítja a fejlesztési folyamatot és növeli a produktivitást.

4.4.3 Spring Security

A Spring Security[12] a Spring ökoszisztéma egyik legelterjedtebb biztonsági keretrendszere, amely robusztus és rugalmas megoldást kínál a hitelesítés és jogosultságkezelés megvalósítására. A Spring Security integrálása a Spring Boot projektekbe egyszerűen történik, például a spring-boot-starter-security függőség

---

[10] https://gradle.org/

[11] https://spring.io/projects/spring-data-jpa

[12] https://spring.io/projects/spring-security

hozzáadásával, amely automatikus konfigurációt nyújt az alapvető biztonsági funkciókhoz. A keretrendszer támogatja az autentikáció különböző módjait, például az OAuth2 és JWT-alapú hitelesítést, amelyek különösen hasznosak modern alkalmazásokban.

A Spring Security segítségével egyedi szabályokat és hozzáférési szinteket definiálhatunk, amelyek segítségével a felhasználói szerepkörök kezelése és az érzékeny adatok védelmét leegyszerűsödik. Ennek a technológiának az integrálásával a Spring Boot alkalmazások biztonságosabbá tehetők olyan eszközökkel, mint például a jelszavak titkosítása, az automatikus session-kezelés és az URL-szintű hozzáférés-ellenőrzés.

4.4.4 JWT

A JSON Web Token (JWT)[13] egy kompakt és elterjedt, URL-barát adatstruktúra, amelyet gyakran használnak azonosítás és információbiztonság céljából webalkalmazásokban. A JWT három részből áll: a fejlécből, a törzsből (vagy payload), és a titkosított aláírásból. A fejléc a token típusát és az aláírási algoritmust tartalmazza, a törzs pedig a felhasználói adatokat vagy más kiegészítő információkat (mint például felhasználói jogosultságokat). A titkosított aláírás biztosítja az adat integritását és hitelességét. JWT-t leggyakrabban arra használják, hogy a szerver hitelesített módon felismerje a felhasználót további bejelentkezés nélkül, így támogatva a stateless, azaz állapotmentes kommunikációt a kliens és szerver között.

4.4.5 MockK

A MockK[14] egy modern és elterjedt mocking könyvtár, amelyet kifejezetten a Kotlin nyelvhez terveztek. A könyvtár lehetővé teszi a tesztelés során használt objektumok és függőségek egyszerű és rugalmas utánzását. A MockK lehetőséget biztosít a fejlesztőknek arra, hogy könnyen és gyorsan készítsenek mock objektumokat, stubbokat és spy-ket, támogatva a különböző tesztelési mintákat és technikákat. A könyvtár előnye, hogy kihasználja a Kotlin nyelv funkcionális tulajdonságait, mint például a lambda kifejezéseket és a korosztályos programozást, így intuitív és kifejező szintaxist kínál a mock-objektumok definiálásához. Emellett a MockK jól integrálható más tesztelő

---

[13] https://jwt.io/

[14] https://mockk.io/

keretrendszerekkel, például a JUnit-tel és a Kotest-tel, így a Kotlin fejlesztők számára ideális eszközzé válik a tesztelés során felmerülő bonyodalmak kezelésére, javítva ezzel a kód megbízhatóságát és karbantarthatóságát.

4.4.6 JUnit

A JUnit[15] egy népszerű, nyílt forráskódú keretrendszer a Java alkalmazások egységtesztelésére (Unit tesztelés). A keretrendszer feladata, hogy az egyes kódrészek (például metódusok) funkcionálisan tesztelve legyenek ezzel segítve a hibák gyors azonosítását és javítását. A JUnit egyszerű annotációkkal, mint például @Test és @BeforeEach, könnyíti meg a tesztek írását és a tesztelési folyamat szervezését. A JUnit támogatja az automatizált tesztfuttatást, aminek köszönhetően könnyen használható a folyamatos integrációs (CI) rendszerekben is. Egyszerűsége és széles körű eszköztámogatottsága miatt az egyik legelterjedtebb tesztelési eszköz a Java fejlesztők körében.

4.4.7 JaCoCo

A JaCoCo (Java Code Coverage)[16] egy nyílt forráskódú eszköz, amely a Java alkalmazások tesztlefedettségét méri. Segítségével meghatározható, hogy a tesztek a kód hány százalékát fedik le, beleértve a sorokat, ágakat, metódusokat és osztályokat. Könnyen integrálható olyan build eszközökbe, mint például a Maven vagy a Gradle, és támogatja a CI/CD folyamatokat is. A JaCoCo HTML, XML és CSV formátumú vizuális jelentéseket generál, amelyek kiemelik a nem tesztelt részeket, ezzel támogatva a fejlesztőket a tesztelési lefedettség növelésében és a nem használt kódrészletek azonosításában.

4.5 MySQL

A MySQL[17] egy relációs adatbáziskezelő rendszer, amit először a MySQLAB cég fejlesztett, de ma már az Oracle tulajdonában áll. Még ma is az egyik legelterjedtebb és

---

[15] https://junit.org/junit5/

[16] https://www.eclemma.org/jacoco/

[17] https://hu.wikipedia.org/wiki/MySQL

legtöbbet használt adatbázis-kezelő, ami elérhető az interneten. Mivel relációs adatbázis, az adatokat táblákban tárolja, a táblákon belül sorokba rendezve.

4.6 Figma

A Figma[18] egy modern, webes felhasználói felülettervező alkalmazás akár drótvázak (wireframe) akár teljes UI design-ok tervezésére. A vektorgrafikus tervezőfelületen több forma, alakzat elkérhető, emellett minden alakzatnak beállíthatunk különböző tulajdonságait. Ezek a tulajdonságok az egész alap funkcióktól, mint például a szín, vonalvastagság, egészen az igen specifikus részletekig terjednek, mint például a különböző effektek (árnyékok, homályosítás és még sok más). A felület lehetőséget biztosít rétegek, rétegcsoportok kialakítására, amitől sokkal átláthatóbb lesz a készülő felület. Egy adott alakzatcsoportból komponensek készíthetőek, amik utána újra felhasználhatóak, illetve változtatásukra minden 11 előfordulásnál megváltoznak. Ezen komponenseknek lehetnek variánsai is, amik színesítik a látványtervet. A felületen lehetőség van prototípusokat létrehozni a design-hoz, amitől az interaktív lesz. Beállítható minden alakzatra, hogy például kattintásra melyik nézet jelenjen meg, milyen adatok jelenjenek meg. Ez a tervezési folyamat során, illetve például egy megrendelői bemutatásnál sok félreértést el tud oszlatni ez a funkció.

4.7 Git/GitHub

A Git[19] napjaink legelterjedtebb verziókezelő rendszere, amely minden fejlesztő számára lokális adattárat hoz létre, ahol a kódbázist tárolják. A felhasználó által végzett változtatásokat helyileg menti, amit később majd hozzá lehet adni a közös kódbázishoz (merge). Az elavult kód elkerülése érdekében a fejlesztők frissíthetik a saját adattárukat, mielőtt új módosításokat végeznének el. A kódütközések csökkentése érdekében a Git ágakon (branch) alapul: egy-egy ágat például funkciók vagy modulok fejlesztésére használnak, és a hibátlan ágakat a fő ágba (master) olvasztják be. A projektben a GitHubot[20] használtam adattárolásra, amely a Git egyik népszerű, ingyenes szolgáltatója.

---

[18] https://www.figma.com/

[19] https://git-scm.com/

[20] https://github.com/

5 Tervezés

5.1 Architektúra tervezés

5.1.1 Alkalmazás felépítése

Az alkalmazás tervezése során három rétegű architektúra[5] mellett döntöttem, mivel ebben a megoldásban jól el tudom szeparálni a fő komponenseket (kliens oldal, szerver oldal, adatréteg)

A megjelenítési (kliens oldal) réteghez Angular 14-et használtam, mivel ebben a keretrendszerben már volt előzőleg tapasztalatom. Az Angular keretrendszer egy jó választás közepes és nagy méretű projektek kliens oldali megvalósításához, mivel rengeteg könyvtár létezik hozzá a különböző komponensek gyors megvalósítására, amik nem mellesleg követik a Material design alapelveit. Ezáltal a fejlesztés gyors ütemben tud haladni.

Az üzleti logikai réteg megvalósításához a Spring Boot keretrendszer mellett döntöttem, mivel ez a keretrendszer már régóta az egyik legjobban elterjedt szerver oldali keretrendszer tele olyan beépített megoldásokkal, amik segíthetnek az alkalmazás megvalósításában. Emellett a Spring támogatja a Kotlin nyelvet, amely egy sokkal modernebb, olvashatóbb nyelv a Java-hoz képest, így a fejlesztés benne sokkal egyszerűbb és gyorsabb.

Az kliens oldali réteget és az üzleti réteget REST[6] alapú kommunikációval terveztem megoldani, ami jelenleg a legjobban elterjedt megoldás web alkalmazások kliens és szerver oldali kommunikációjának megvalósítására.

Az adatrétegben már a tervezés elején eldöntöttem, hogy adatbázisként MySQL-t szeretnék használni, mivel ezzel is volt már tapasztalatom, emellett ez is egy piacon már bevált, jól támogatott relációs adatbázis megoldás. A MySQL remek Spring Boot támogatással rendelkezik, így az architektúrám rétegeinek integrációja sem okoz kihívást.

A kezdeti fázisban azt a döntést hoztam, hogy amíg magát az ötletkezelő alkalmazást fejlesztem, addig elég egy kapcsolatos, egy sémás adatréteg megoldás. Ez az architektúra az ábra 5 - Kezdeti architektúra ábrán látható.

ábra 5 - Kezdeti architektúra

5.1.2 Multitenancy

A multitenant adatréteg megvalósítását az alkalmazás elkészülte után terveztem. Ezt az új adatréteg implementálását egy kutatás előzte meg, ahol megismerkedtem a multitenant architektúrák 3 alfajával, azok működésével, előnyeikkel és hátrányaikkal.

Bérlőnkénti Adatbázis

Ebben az implementálási modellben minden bérlőnek saját adatbázisa van, amihez csak ő férhet hozzá. Ez a legmagasabb szintű elkülönítés, amit multitenant architektúrában el lehet érni. Ez a módszer akkor ajánlott, ha az alkalmazás szigorú adatbiztonságot követel meg.

Előnyei:

- Legmagasabb szintű elszigeteltség; ha egy adatbázis leáll, a többi bérlőre ez nincs hatással.
- Könnyen skálázható horizontálisan további adatbázisok hozzáadásával.

Hátrányai:

- Magasabb költségek, mivel minden adatbázis külön erőforrásokat igényel.
- Nehezebben kezelhető nagy skálán, mivel több adatbázist kell felügyelni.

Bérlőnkénti séma

Ebben az implementálási modellben minden bérlő ugyan azt az adatbázist használja, de minden bérlőnek külön adatbázis séma van fenntartva. Ez egy közepes szintű elkülönítést eredményez, ami jó egyensúlyt nyújt az adatbiztonság és a költségek között olyan alkalmazások számára, amelyek nem igényelnek teljes adatbázis-szintű elkülönítést.

Előnyei:

- Kezelhetőbb, mint a "Bérlőnkénti adatbázis" modell, mivel csak egy adatbázist kell felügyelni. Könnyen skálázható horizontálisan további adatbázisok hozzáadásával.
- Költséghatékonyabb, mivel a bérlők ugyanazt az adatbázist osztják meg.

Hátrányai:

- Az adatbázis erőforrásai szétoszlanak a bérlők között, így növekedés esetén fokozódhat a verseny az erőforrásokért.
- Bonyolultabb lehet a mentés és visszaállítás, mivel a sémák egy adatbázison belül függenek egymástól.

Megosztott adatbázis, megosztott séma

Ebben az implementációban minden bérlő ugyan azt az adatbázist és sémát használja. A bérlőket egy discriminator oszlop választja el (pl tenant_id). Ez egy alacsony szintű elkülönítést eredményez, mivel az adatokat csak szűrőkkel védik. Ez a megoldás ideális SaaS alkalmazásokhoz, ahol nagyszámú, kisebb bérlő van, és kevés az egyedi igény.

Előnyei:

- A leghatékonyabb erőforráshasználat; alacsonyabb költségek és könnyebb skálázhatóság.
- Egyszerű adatbázis-kezelés, mivel csak egy sémát kell karbantartani.

Hátrányai:

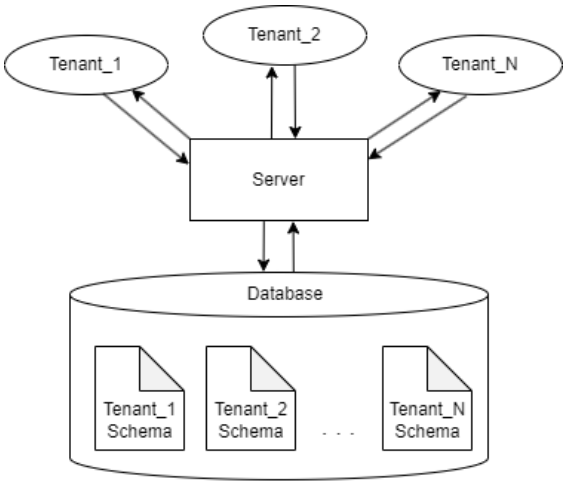
- Nehezebb biztosítani az adatok biztonságát, mivel minden bérlő ugyanazokat a táblákat használja.
- Teljesítményproblémák jelentkezhetnek az adatbázis növekedésével, különösen nagy bérlőszám és komplex lekérdezések esetén.

A tervezés során a **Bérlőnkénti séma** és a **Megosztott adatbázis, megosztott séma** implementációkban mélyültem el, ezeket teszteltem egy kis projekten, hogy át tudjam majd ültetni a leendő kész alkalmazásba.

Azért esett erre a két módra a választás, mivel az alkalmazás alapvetően nem üzleti titkok és értékes üzleti döntések megvitatására lesz készítve, így nem szükséges a legnagyobb biztonsággal kezelni az adatokat. Másrészről viszont, mivel ez egy off-the-shelf szoftver megoldás, így fontos, hogy sok bérlő esetén is megfelelően működjön és költséghatékony legyen.

A **megosztott adatbázis, megosztott séma** implementáció elsőre egy könnyebb megoldásnak tűnt, így ezt próbáltam ki először. A Spring keretrendszer támogatja ezt a multitenancy implementációt, vannak beépített @Filter dekorátorok, amelyekkel meg lehet jelölni Entity szinten a discriminator oszlopot a táblákban. Ennek a megoldásnak az implementációja során azonban abba a problémába ütköztem, hogy felül kell írni az alap Spring által biztosított EntityManagert, ami szimpla lekérdezéseknél nem okoz problémát, de bonyolultabb lekérdezéseknél előfordulhatnak problémák a tranzakciók ütemezésével. Ezekre biztosít a Spring megoldást, de amíg teszteltem nem sikerült hiba mentesre felüldefiniálni az EntityManagert.

Így jutottam a második és a jó megoldásnak bizonyuló **Bérlőnkénti séma implementálásához.** Ezt a megoldási módszert is támogatja a Spring Boot. Ennek részletesebb implementálását az 0 fejezetben fejtem ki. Az alábbi, ábra 6 - Végleges architektúra ábrán látható a végleges architektúra a kicserélt adatréteggel.



ábra 6 - Végleges architektúra

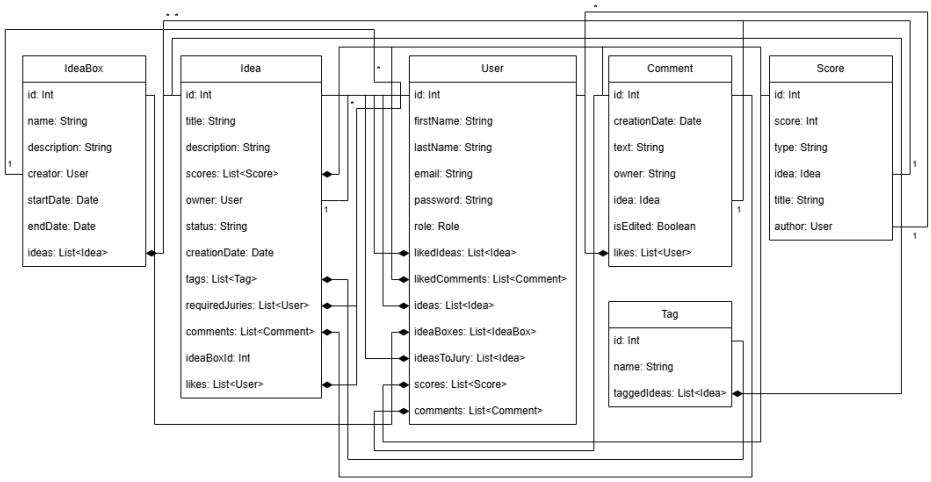
5.2 Modell tervezés

A kezdeti tervezési fázisban az alkalmazás összes nagyobb osztálya modellezésre került, kivéve természetesen a pontozást. Már ekkor is figyelembe vettem a Spring Boot

által kínált lehetőségeket és a modellt a Spring Data JPA segítségével terveztem meg. A Spring Data JPA beépített ORM-je segít abban, hogy a modellemet modern, dokumentum alapú struktúrában tudjam definiálni, de a motorház alatt ez a modell relációs adatbázisként jelenjen meg és így kerüljön mentésre a választott adatbázisban.

A kezdeti modellem a következőképpen állt elő. Ez a modell (ábra 7 - Az alkalmazás modellje az első tervezési fázis végén) elég jól lefedte kezdetben az alkalmazás funkcionális követelményeit, azonban a pontozási rendszere korántsem volt olyan robosztus, mint amilyet a megrendelő elvárt volna. Emiatt ez a rész nem is lett implementálva az első fázisban.

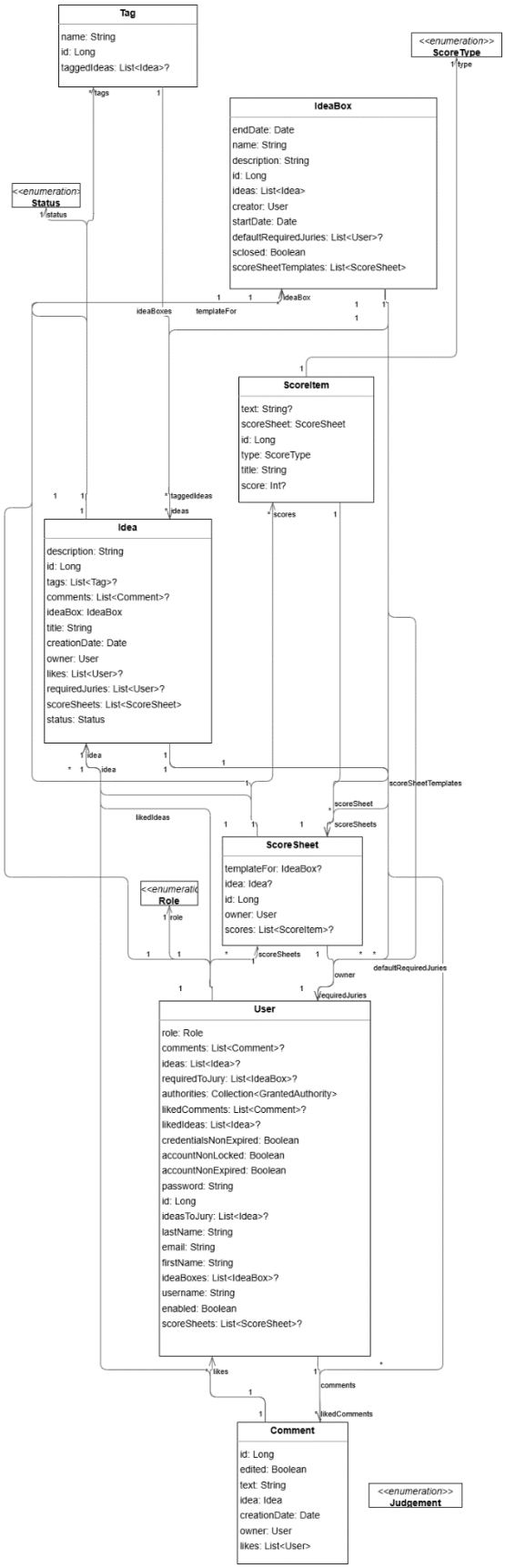
A modell revíziójára a Diplomatervezés 1 tárgy elején került sor, ahol a pontozási rendszer nagy változáson ment át. Az alapvető modell többi részén csak apróbb változásokat kellett eszközölni.



ábra 7 - Az alkalmazás modellje az első tervezési fázis végén

5.2.1 Végleges modell

A második tervezési fázis végén kialakult a végleges adatmodell, amit az alkalmazás használ. Ezt a következőkben mutatom be alfejezetenként. A teljes modell és a közöttük lévő kapcsolatokat az ábra 8 - Az alkalmazás UML diagramja, IntelliJ-ből exportálva szemlélteti.



ábra 8 - Az alkalmazás UML diagramja, IntelliJ-ből exportálva

5.2.2 User

A User osztály felel azért, hogy a felhasználók adatait kezelje. Az osztály megvalósítja a userDetails osztályt, ami a Spring részét képezi és segít a biztonság és fiókkezelés megvalósításában.

Ezt a User Entitást használja az összes felhasználó akármilyen szerepkörrel rendelkezik. Így előfordulhat, hogy egyes listák üresek, például a User szerepkörrel rendelkező felhasználónak nem lesznek ötletdobozai, pontozási lapjai és üres lesz a requiredToJury és ideasToJury listái is.

5.2.3 IdeaBox

A következő elengedhetetlen osztály az ötletdobozok osztálya. Az ötletdoboz tárolja a rá beérkező ötleteket, és egységes pontozási keretet biztosít azok értékelésére. Tárolja továbbá azokat a bíráló felhasználókat is, akiknek kötelező minden ötletet pontozni, ami az adott ötletdobozba érkezik.

Az ötletdoboz modell szinten lehetőséget biztosít rá, hogy akár több pontozási lap szerint is lehessen értlkelni az ötleteket, ezt azonban az alkalmazás jelenlegi állapota nem támogatja.

5.2.4 Idea

Az Idea osztály modellezi az ötleteket az alkalmazásban. Ez is egy elengedhetetlen része az alkalmazásnak, ha nem a legfontosabb. Ez tárolja a felhasználók által írt ötleteket, illetve a leendő értékeléseket is, amit az ötlet majd kapni fog. Ezen felül Tageket is tartalmaz, ami az ötletek csoportosítására szolgál. Ez a funkcionalitás inkább csak egy előkészítés az alkalmazás ilyen irányú továbbfejlesztésére. Az ötlet objektum tárolja még a rá érkező kommenteket is, illetve azt, hogy kiknek tetszik az ötlet.

Az ötleteknek lehet külön-külön is megadni bírálót, ebben az esetben nem csak azok a bírálók fogják értékelni az ötletet, ami az ötletdobozból eredendően elvárt, hanem az itt definiált plusz bírálók is.

5.2.5 Komment

A kommentezési rendszer nem befolyásolja közvetlenül az alkalmazás lényegi működését, azonban jelentősen hozzájárul a felhasználói élmény és a felhasználói kapcsolódáshoz. Ennek a funkciónak az lenne a lényege, hogy a felhasználók

megoszthassák a véleményüket és visszajelzést adjanak egymásnak ezzel téve interaktívabbá az alkalmazást és a szervezeten belüli közösséget. Ezt a funkcionalitást implementálja a komment modell, ami tárolja a komment üzenetét, a felhasználót, aki írta, a komment készülésének dátumát, illetve azt is, hogy kiknek tetszik ez a komment.

5.2.6 ScoreSheet

A ScoreSheet osztály szerepe a pontozási lapok megvalósítása az alkalmazáson belül. Ebben az osztályban vannak tárolva tetszőleges számban a pontozási szempontok, amikkel az ötletek értékelése egységes lesz.

A pontozási lapnak 2 felhasználási lehetősége van az alkalmazáson belül. Lehet pontozási útmutatóként használni, ekkor az ötletdobozhoz lesz rendelve. Ez a template alapján fogják tudni a pontozást elvégezni a bírálók. A második, és egyben többet használt felhasználás maga a pontozási lap. Mivel az osztály scoreItem objektumokat tárol, így ezeknek már eleve lehet pontot adni.

5.2.7 ScoreItem

Ez az osztály felelős azért, hogy mérni és bírálni lehessen az ötleteket egyes szempontok szerint. A scoreItem felépítéséből adódóan egy egységes, többször felhasználható objektum, amit akár bővíteni is lehet.

5.2.8 Egyéb osztályok, Enum-ok

Ezek az osztályok és Enum-ok a fő osztályok működését segítik, nagyobb funkcionalitással nem rendelkeznek.

5.2.8.1 Tag

Ez az osztály felel azért, hogy az ötleteket kategóriákba lehessen sorolni.

5.2.8.2 Role

Ez egy Enum osztály, amivel az egyes felhasználók szerepköreit lehet jelölni az alkalmazásban. Három értéket vehet fel, ezek a **USER**, **JURY** és **ADMIN**.

5.2.8.3 ScoreType

A ScoreType is egy Enum osztály, amit a pontozási szempontok típusát határozzák meg. Az alkalmazás jelenleg 2 típust használ, ezek a **STAR** és **SLIDER**.

- A **STAR** típusú pontozási szempontban egy csillagtól öt csillagig lehet értékelni az ötletet a szempont szerint. A pontozás végleges értéke a score változóban van tárolva 1-5ig egy integer változóban.
- A **SLIDER** típusú pontozási szempontnál egy csúszkán lehet értékelni az ötletet az adott szempont szerint egytől tízig. Ennek az értéke is a score változóban tárolódik integerként.

A ScoreItem és ScoreType felépítésének köszönhetően az alkalmazás támogatja többféle pontozási metódus létrehozását.

5.2.8.4 Status

A Status is egy Enum osztály az alkalmazásban, ez felel azért, hogy az ötletek státuszát nyomon lehessen követni. Négy értéket vehet fel, ezek a következők:

- **SUBMITTED**:

  Az ötlet létre lett hozva az ötletdobozban, de még nem kapott egy értékelést sem. Ez minden ötlet kezdőértéke.

- **REVIEWED**

  Az ötletet már legalább egy bíráló pontozta. Ezzel a művelettel az ötlet automatikusan a Reviewed státuszba kerül.

- **APPROVED**

  Amennyiben az összes értékelés után az Admin úgy dönt, hogy elfogadja az ötletet, akkor annak státusza APPROVED lesz.

- **DENIED**

  Amennyiben az összes értékelés után az Admin úgy dönt, hogy elutasítja az ötletet, akkor annak státusza DENIED lesz.

5.3 UI Tervezés

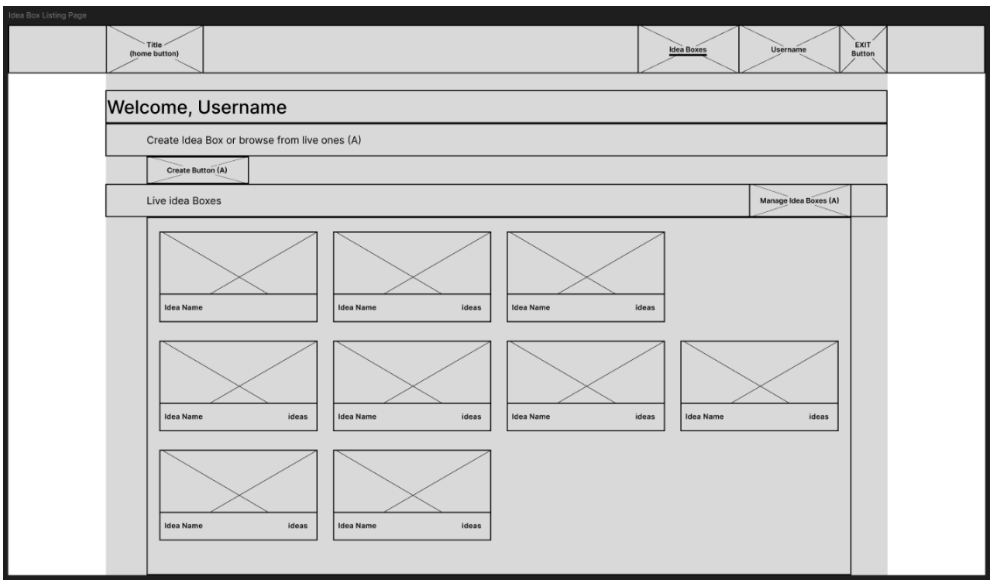
Az alkalmazás funkcionális követelményeinek definiálása és a modell megalkotása után a felhasználói felület vázolásával folytattam a tervezést. Felhasználói felületek tervezésénél elterjedt és bevált módszer a drótvázak készítése, így én is ezzel kezdtem a munkát.

Alkalmazás kezdeti drótváza

Az alkalmazás első tervezési fázisában, a modell vázolása után a UI tervek elkészítése volt a következő és egyben utolsó tervezési lépés az implementáció elött.

A tervezés során a Figma nevű tervező szoftvert használtam. A Figma egy széles körben elterjed UI tervező szoftver, amivel könnyen és gyorsan lehet akár drótvázakat, akár már kész terveket is készíteni az alkalmazásunkhoz.

Az ötletkezelő alkalmazás felületét úgy próbáltam megtervezni, hogy azt könnyen és intuitívan lehessen használni. Az első felület, amit megterveztem az alkalmazás főoldala, ami az ábra 9 - Idea Box Listing Page ábrán látható.

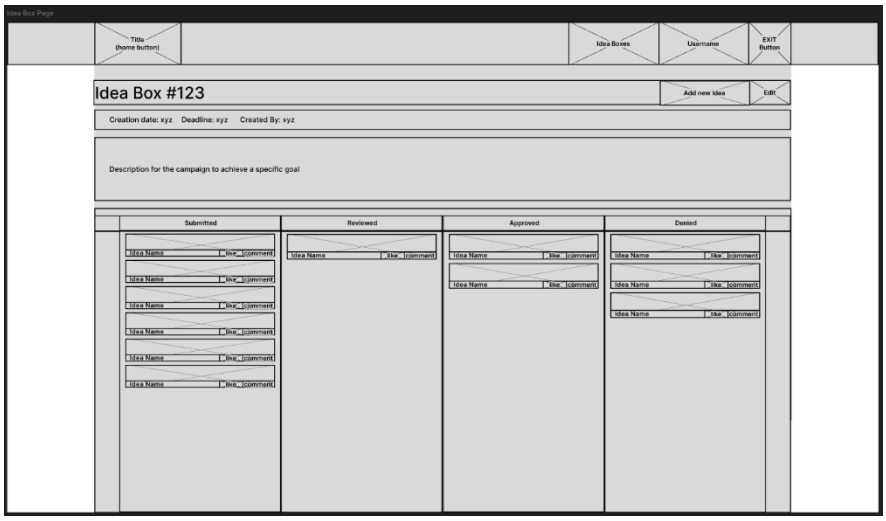


ábra 9 - Idea Box Listing Page

Ez a felület felel a nyitott ötletdobozok listázásáért. Az ötletdobozok a tervek szerint kártyák lennének, amik egy középen elhelyezkedő grid-ben vannak listázva. Ezek a kártyák kattinthatóak, kattintás után az ötletdoboz teljes oldala jelenik meg. Az oldalon található még a belépett felhasználó neve, egy ötletdoboz létrehozásáért felelős gomb, illetve egy gomb, amivel a rendszerben szereplő összes ötletdoboz szerkeszthető.

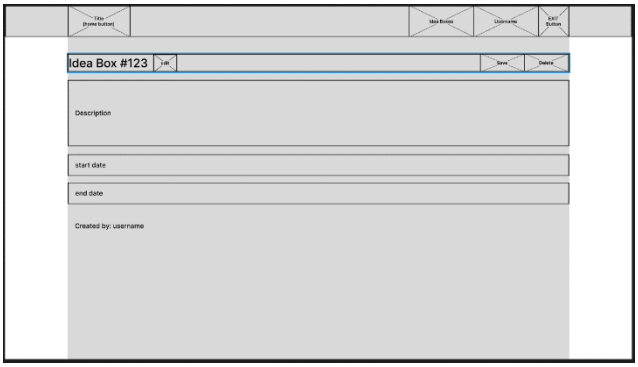
A következő felület az Ötletdoboz oldala, amely a kártyákra kattintva jelenik meg. Itt látható az ötletdoboz neve, hasznos információi és adatai. Gombok, amikkel új ötletet lehet felvenni a dobozba vagy szerkeszteni a doboz adatait. Ezek alatt láthatóak az ötletek, amik 4 oszlopba vannak szervezve aszerint, hogy éppen hol tartanak az elbírálási folyamatban. Az ötletek kártyákkal vannak felsorolva, ezek a kártyák – hasonlóan, mint

az ötletdoboz kártyákhoz – kattinthatóak, kattintás után az ötlet teljes oldalára navigálják a felhasználót. A felület terve az ábra 10 - Idea Box Page ábrán látható.



ábra 10 - Idea Box Page

Az ötletdobozhoz tartozik egy create/edit oldal is. Ez egy egyszerűbbfelület, ahol az ötletdoboz adatait lehet megadni a létrehozáshoz, vagy ezen adatokat módosítani szerkesztéskor. Ez a felület a ábra 11 - IdeaBox create/edit ábrán látható.

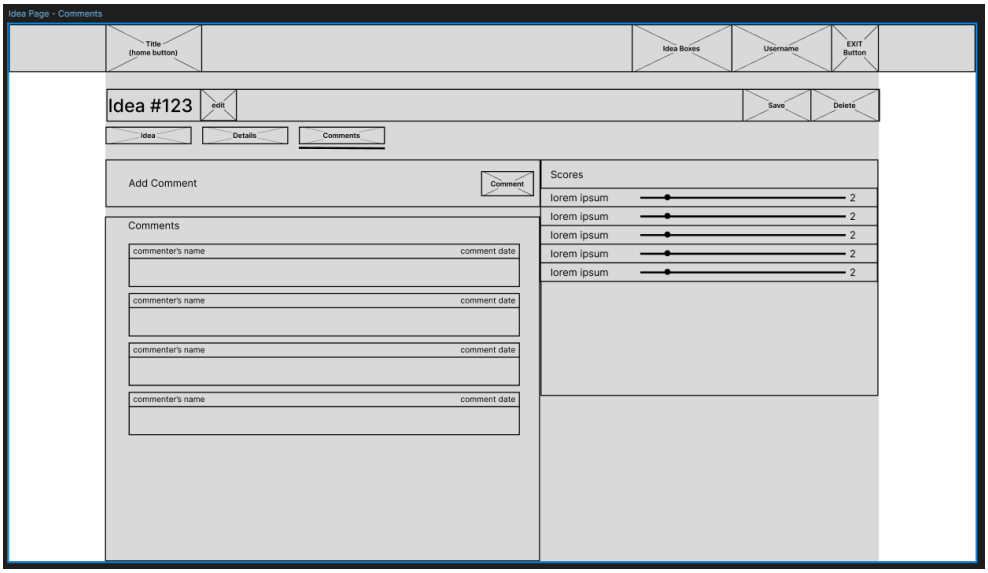


ábra 11 - IdeaBox create/edit

A következő lényeges oldal az ötletek részletes oldala. Ez az oldal az ábra 12 - Idea Page kommentek aloldallal képen látható. Ezen az oldalon jelennek meg a felhasználók által készített ötletek neve, leírása és egyéb információi. Az oldalt 2 részre osztottam. A bal oldalon egy 3 aloldalból álló lapozható komponens van. Itt az első oldalon az ötlet leírása található az Idea fülön. A Details fülön az ötlet alap adatait olvashatjuk. A harmadik, Comments fülön böngészhetjük az ötletre érkezett kommenteket. A jobb oldalon láthatjuk az ütletre érkezett pontozásokat, ha vannak.

Az ötletre érkező kommenteket listázó oldalon láthatjuk a kommentezési rendszer drótvázát is. A felületen lesz egy beviteli mező, ahova a saját kommentjét írja a felhasználó, a Comment gomb megnyomásával pedig publikálja azt. Ha sikeres a publikáció, akkor az megjelenik az beviteli mező alatti listában. A listában szintén

kártyákkal vannak reprezentálva a kommentek. Az egyes kártyákon a kommentelő neve és a kommentjének a szövege szerepel.



ábra 12 - Idea Page kommentek aloldallal

A tervezés során bizonyos oldalakra nem tértem ki, mint például a user Profile oldal vagy a pontozás, mivel ezeknél még nem voltam biztos benne, hogy pontosan milyen funkcionalitásokat és követelményeket fog hozni a második tervezési fázis. Egyes oldalak, mint például a login vagy regisztrációs felület viszont nem hordozott olyan lényeges funkciót, amit egy előre megfontolt dizájn során ki kellene találni, így ezekről az oldalakról azért nem készült terv.

Ezek a UI tervek az alkalmazás második tervezési fázisában néhány esetben elavulttá váltak, de nagy részben a jelenlegi alkalmazás vázát képezik. Az implementáció során ezeket a terveket vettem alapul. Az új funkcionalitások implementálásánál is arra törekedtem, hogy az itt lefektetett design-tól ne üssenek el az új komponensek.

A következő, Implementációs fejezetben látható, hogy hogyan alakult az alkalmazás kinézete a kezdeti tervekhez képest.

6 Implementáció

Ahogy azt már írtam, az alkalmazás két tervezési és implementációs iteráció után nyerte el a mostani alakját. Erről a két iterációról fogok írni ebben a fejezetben, kiemelve a lényegesebb és érdekesebb részeket a kliens és szerver oldalon.

6.1 Első iteráció

Az első iterációban a fejlesztés fő fókusza egy működő prototípus elkészítése volt, ami biztonságos módon kezeli a felhasználókat és megvalósítja az alapvető funkciókat, amiket egy ötletkezelő alkalmazásban el lehet érni. Ehhez a következő feladatokat tűztem ki magamnak:

- A tervezett három rétegű architektúra alapjainak lefektetése
- A modell leképezése kódra
- Az alapvető CRUD funkcionalitások megvalósítása mind kliens oldalon, mind szerver oldalon
- Felhasználókezelés és alapvető biztonsági funkciók implementálása

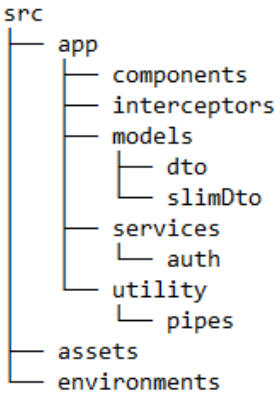
A következő fejezetekben ezen pontok implementálásának lépéseit mutatom be.

6.1.1 Architektúra megvalósítása

6.1.1.1 Kliens oldal

Az alkalmazás kliens oldali megvalósításához Angular CLI segítségével létrehoztam egy új projektet, amibe importáltam az Angular Material könyvtárat. Emellett CSS fájlok helyett SCSS fájlok alkalmazása mellett döntöttem arra az esetre, ha szükség lenne a bővebb funkcionalitásra, amit az SCSS kínál.

Az alkalmazás kliens oldali implementációjának a könyvtárszerkezetét az ábra 13 - Kliens oldali könyvtárszerkezet szemlélteti.



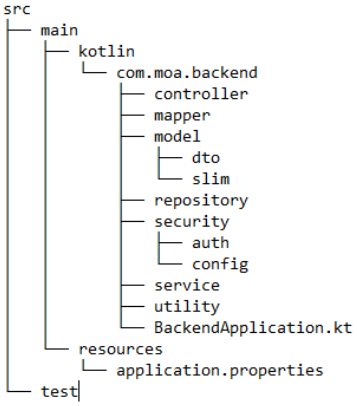
ábra 13 - Kliens oldali könyvtárszerkezet

A komponens mappában találhatóak az alkalmazásban használt komponensek, oldalakra lebontva, azokon belül pedig funkcionalitásra vagy aloldalakra lebontva. Az interceptors mappában vannak tárolva a kommunikáció során használt interceptorok, amelyek segítségével a http header kezelés egyszerűsödik. A modell mappán belül tárolok egy dto mappán és egy slimDto mappát is. Ezekről bővebben a modell bemutatásánál fogok írni. A services mappa tárolja a kommunikációt lebonyolító service osztályaimat, amikkel a szerver oldallal folytatott kommunikáció van megoldva. Az Utility mappában hasznos, többször használt függvények, pipe-ok vannak definiálva. Ezen felül az alkalmazás az alap Angular fájlstruktúrát követve rendelkezik egy assets és egy environments mappával is.

6.1.1.2 Szerver oldal

Az alkalmazás szerver oldalának kiinduló projektjét a Spring Initializr segítségével készítettem el. Ebben a kiinduló projektben már eleve bele volt generálva a Spring JPA, MySQL dependency, illetve több más hasznos könyvtár.

Az alkalmazás szerver oldali implementációjának mappaszerkezetét az ábra 14 - Szerver oldali könyvtárszerkezet szemlélteti.



ábra 14 - Szerver oldali könyvtárszerkezet

Az application.properties fájlban definiáltam az első adatbázis kapcsolatomat, mivel ebben a stádiumban csak egy MySQL adatbázishoz és egy sémához kapcsolódott az alkalmazás. Ezután kialakítottam itt is a mappaszerkezetet, ami keretet ad az implementációmnak.

Szerver oldalon a megvalósítás során a Controller-Service-Repository (CSR)[7] mintát követtem, ezáltal az alkalmazás átlátható és könnyen bővíthető. A kérések a különböző filterek után először a kontrollerekbe érkeznek be, ahol a végpontjaim vannak definiálva. Ezekből a végpontokból vannak meghívva a service-ben implementált funkcionalitások. A service mappában tárolt függvények az esetek legnagyobb részében valamilyen adatot akarnak elérni az adatbázisból vagy éppen oda akarnak új adatot menteni, módosítani. Ezekhez a repository mappában tárolt osztályokat használom, amiket a spring felismer és leképez SQL parancsokra, ezáltal könnyítve az adatbázissal való kommunikációt. A modellem kóddá való leképezése a model mappában található. A

mapper mappában találhatóak olyan segédosztályok, amikkel a modellem különböző elérési rétegeit tudom konvertálni. Emellett egy security mappában találhatóak meg a Spring Security által használt osztályok, amelyekkel a biztonságos felhasználókezelést oldottam meg. Az utility mappában egyéb hasznos függvények vannak definiálva.

6.1.2 Modell megvalósítás

A modell megvalósításánál törekedtem arra, hogy a rétegek közötti kommunikáció során a lehető legkevesebb adat utazzon, ezzel nem csak az adatátviteli költséget redukálom, de az adatok rejtésével nagyobb adatbiztonságot is teremtek az alkalmazásban. Emiatt az adatrétegek közötti kommunikációnál a DTO (Data Transfer Object) mintát használtam. A megvalósításomban ez 3 féle modellt jelent minden entitásomhoz. A három típusom a teljes osztály, az osztályDto és az osztálySlimDto. Az implementálás során az volt az irányelvem, hogy a teljes osztályokat csak akkor használom, amikor adatbázisból olvasok, illetve, ha oda mentek. A Dto modellek a fő kommunikációs osztályaim, ezek utaznak a szerver és kliens oldal között. Ezek az osztályok általában nem rendelkeznek az összes változóval. Mivel sok esetben a küldeni kívánt Dto osztályokban listák is vannak, ezért ezekben a listákban bevezettem a SlimDto osztályokat. Ezek az osztályok a tartalmazzák a legkevesebb adatot, leggyakrabban csak egy id-t, illetve egy nevet a megjelenítéshez, listákat egyáltalán nem tartalmaznak. Ezzel a megoldással az adatok nem tudnak végtelen ciklusú listákba kerülni.

6.1.2.1 Kliens oldal

A kliens oldali modell implementálásnál csak a Dto és a slimDto modelleket készítettem el, hiszen ennek a rétegnek csak ezek az adatok kellenek. Kódban ezek a modellek interfészekkel az ábra 15-ön látható módon vannak implementálva.

```
export interface User {
  id?: number;
  firstName?: string;
  lastName?: string;
  email?: string;
  role?: string;
  likedIdeas?: IdeaSlimDto[];
  likedComments?: CommentSlimDto[];
  ideas?: IdeaSlimDto[];
  ideaBoxes?: IdeaBoxSlimDto[];
  comments?: CommentSlimDto[];
}
```

```
export interface UserSlimDto {
  id?: number;
  firstName?: string;
  lastName?: string;
  email?: string;
  role?: string;
}
```

ábra 15 - User osztály interfészei (Dto, slimDto)

6.1.2.2 Szerver oldal

Szerver oldalon is hasonlóképpen vannak implementálva a modellek. Itt a Dto és a SlimDto mellett megjelenik a teljes osztály is. Ezek a teljes osztályok @Entity

annotációval vannak ellátva annak érdekében, hogy a Spring Entity Manager regisztrálja őket és ezzel megkönnyítse az ORM munkáját.

6.1.3 Security

Az alkalmazás biztonságát ebben a stádiumban a Spring Security és a JWT (Json Web Token), illetve a kliens oldali korlátozások biztosították.

6.1.3.1 Kliens oldal

Kliens oldalon a biztonság inkább megjelenítésben – vagy annak hiányában – valósul meg. Olyan oldalakat, amiket a felhasználó nem érhet el az alkalmazás webes felülete nem jelenít meg. Ezt úgy oldottam meg, hogy bejelentkezés után a JWT-ben tárolt role alapján rögzítettem a felhasználónak szerepkörét és ez alapján egyszerű *ngIf direktívákkal van blokkolva egyes komponensek megjelenítése.

6.1.3.2 Szerver oldal

A spring Security és JWT mechanizmust a szerver oldalon implementáltam a security mappában található fájlokban. A rendszer a már ipari szinten jól bevált mintát követi, miszerint az új felhasználóknál regisztrációkor a jelszava nem egyszerű string-ként van elmentve az adatbázisban, hanem egy kódolt stringként, ezzel biztosítva a jelszavak biztonságát egy esetleges adatszivárgásnál.

Vannak olyan végpontok az alkalmazásban, amiket csak bizonyos szerepkörrel lehet elérni (például a pontozás – Jury+Admin, vagy az ötletdobozok készítése – csak Admin). Ezeknek a végpontoknak a védelmét a SecurityFilterChain-ben implementáltam. Itt szabályozni lehet antMatcher függvényekkel azt, hogy milyen végpontot milyen jogosultsággal lehet elérni.

A regisztrált felhasználók bejelentkezésekor JWT generálódik, amiben tárolom a felhasználó nevét, e-mailjét és szerepkörét. Ez a JWT fogja autentikálni a felhasználó további kéréseit mindaddig, amíg a JWT érvényes. Ezt az autentikációt egy JwtAuthenticationFilter osztály bonyolítja le. Ez minden kérés beérkezésekor lefut még azelőtt, hogy a kérés elérne a Controllerbe.

6.1.4 Kommunikáció

A kliens oldal és a szerver oldal közötti kommunikáció REST-en keresztül történik. Annak érdekében, hogy egységes üzeneteket tudjak küldeni szerver oldalról, létrehoztam egy WebResponse objektumot, amivel a kliens oldalon az adatok fogadása és a felugró Toast üzenetek kezelése egyszerűsödik. A WebResponse-nak, ahogy az az ábra 16-on is látható 3 változója van. A code változóban tárolom a HttpStatus-t, a message-ben üzenetet lehet definiálni, ami megjelenik a kliens oldalon Toast üzenetként, a data változóban pedig az éppen kért adatot lehet elküldeni.

```
data class WebResponse<T> (   Ádám Varga
    val code: Int = HttpStatus.OK.value(),
    val message: String = HttpStatus.OK.name,
    val data: T? = null
)
```

ábra 16 - WebResponse osztály

Az alkalmazás funkciójinak megvalósításánál mindenhol törekedtem arra, hogy egységesen legyenek a kérések és válaszok kezelve. Egy ötletdoboz létrehozásán bemutatva az alkalmazás kliens és szerver oldali kommunikációja a következőképpen néz ki.

Kliens oldalon a Form megfelelő kitöltésével engedélyezetté válik a Create gomb, ennek megnyomásakor az a Typescript fájlban elindul a kérés a kliens oldali service felé. A service-ben definiált hívások $ jellel vannak zárva, ezzel indikálva azt, hogy ezek a függvények nem szinkron hívások, így feliratkozást igényelnek. A service-ben a hívás egy http POST kérésként elküldésre kerül a backend felé, válaszként pedig egy olyan WebResponse objektumot vár, amiben a data változóban egy IdeaBoxDto van. A kérés kiküldése elött azonban a headerHandler interceptor még beteszi a kérés header-jébe a felhasználó saját JWT-jét és beállítja a típusát Bearer tokenre.

```
//idea-box-create-edit.component.ts
create() {
  let ideaBox = this.IdeaBoxForm.value;
  ideaBox.defaultRequiredJuries = this.selectedJuries;
  ideaBox.scoreSheetTemplate = null;
  this.ideaBoxService
    .createIdeaBox$(ideaBox)
    .pipe(untilDestroyed(this))
    .subscribe((res: WebResponse<IdeaBoxDto>) => {
      console.log(res);
      if (res.code == 200) {
        this.snackBar.ok(res.message);
        this.router.navigateByUrl('/idea-boxes');
      } else {
        this.snackBar.error(res.message);
      }
    });
}

//ideaBox.service.ts
createIdeaBox$(ideaBox: IdeaBoxDto): Observable<WebResponse<IdeaBoxDto>> {
  return this.http.post<WebResponse<IdeaBoxDto>>(
    `${this.apiUrl}/idea-box`,
    ideaBox
  );
}
```

ábra 17 - Front End: Create IdeaBox

A kérés ezután elérkezik a szerver oldalra, ahol a már tárgyalt biztonsági filterek lefutnak. Abban az esetben, ha a felhasználó megfelelő JWT-vel küldte el a kérést, és szerepköréből adódóan van jogosultsága elérni az adott végpontot, akkor a kérés eljut a

kontrollerbe, jelen esetben az IdeaBoxControllerbe. Szerver odlalon a kontrolelreknek alapvetően egyszerű a működésük, csak meghívják az adott funkcionalitáshoz tartozó függvényt a megfelelő service-ben.

A service - mint ahogy az ábra 18 is mutatja– ellenőrizheti az adott felhasználó jogosultságait, ha kritikus funkcionalitásról van szó. Amennyiben a felhasználó olyan kérést próbál elérni, amihez nincs jogosultsága a WebResponse-ban az üzenet mezőben kap értesítést arról, hogy miért nem sikerült a kérést végrehajtani. Amennyiben azonban a felhasználó rendelkezik a megfelelő jogosultságokkal, akkor a függvény tovább halad, a legtöbb esetben kommunikál az adatbázissal a modellekhez tartozó Repositorykon keresztül, majd visszatér egy WebResponse-al, amibe becsomagolja a kért adatot. Jelen példában az ötletdoboz a beküldött adatokkal mentésre kerül, az elmentett objektum pedig visszatér a klienshez. Ebben a példában látszódik a mapperek használata is, ahogy átalakítják az utazó adatot Objektummá, illetve vissza is alakítják a visszaküldésnél.

```
//IdeaBoxController
@PostMapping("/idea-box")  Ádám Varga
fun createIdeaBox(@RequestBody box: IdeaBoxDto): ResponseEntity<*> {
    return ideaBoxService.createIdeaBox(box)
}

//IdeaBoxService
fun createIdeaBox(box: IdeaBoxDto): ResponseEntity<*> {  Ádám Varga +1
    val authentication = SecurityContextHolder.getContext().authentication
    if (authentication.authorities.find{ auth -> auth.authority.toString() == "ADMIN"} == null) {
        logger.info { "Unauthorized user ${authentication.name} tried to create IdeaBox" }
        return ResponseEntity.ok(
            WebResponse<IdeaBoxDto>(
                code = HttpStatus.UNAUTHORIZED.value(),
                message = "User is unauthorized to do this action!",
                data = null
            )
        )
    }

    val data = ideaBoxMapper.modelToDto(ideaBoxRepository.save(ideaBoxMapper.dtoToModel(box)))

    logger.info { "MOA-INFO: IdeaBox created with id: ${data.id}. IdeaBox: $data" }

    return ResponseEntity.ok(
        WebResponse<IdeaBoxDto>(
            code = HttpStatus.OK.value(),
            message = "Idea Box successfully created!",
            data = data
        )
    )
}
```

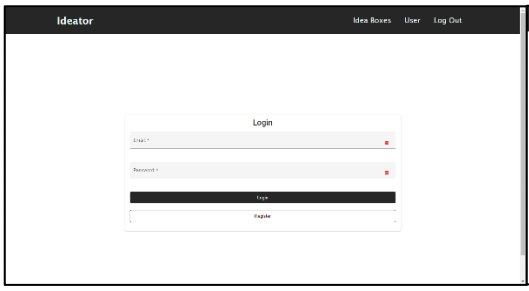
ábra 18 - Back End: Create IdeaBox

Amikor az adat visszatér a kliens oldalra, akkor a service hívásra felíratkozott függvény értesülést kap, és „kicsomagolja” az adatot. Amennyiben a WebResponse státuszkódja 200, akkor a felhasználó sikeres zöld SnackBar-t lát. Ha bármilyen hibával tér vissza a kérés akkor hibás piros SnackBar-al tudatja ezt a felhasználóval. Ezt láthatjuk az ábra 17-en is.

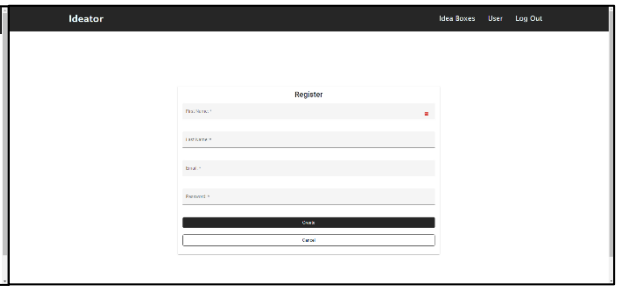
Az alkalmazás szinte összes kommunikációja ezen az elven működik, mint a most bemutatott kérési folyamat.

6.1.5 Bejelentkezés/regisztráció

Az első oldal, amivel a felhasználó találkozik az alkalmazás használatakor a bejelentkezési felület, illetve, ha még nincs fiókja, a regisztrációs felület. Ezek a kezdeti specifikációk alapján lettek implementálva. Az ábra 20 és ábra 19 ezeket mutatja.



ábra 20 - Bejelentkezési felület



ábra 19 - Regisztrációs felület

A felület működése magától értetődő, amennyiben a felhasználónak van fiókja, úgy az email címével és jelszavával belép, amennyiben nincs fiókja, úgy a regisztráció gombra kattintva átkerül a regisztrációs oldalra, ahol az adatai megadása után a regisztrációra kattintva elkészül a fiókja. Mind a két akció után a felhasználó belépve kerül az alkalmazás főoldalára.

6.1.6 Navigációs sáv

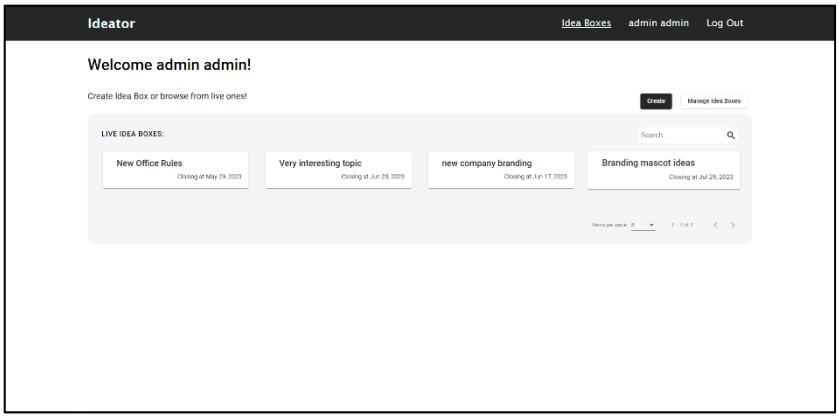
Az oldalon a felső sorban található a navigációs sáv. Ezen bal oldalon megtalálható az ötletkezelő alkalmazás neve, jobb oldalon pedig navigációs gombok. Az Idea Boxes gombra kattintva a főoldalra jut a felhasználó. A nevére kattintva a Profil oldalára kerül. A Log Out gombbal pedig kilép az alkalmazásból és visszajut a belépési képernyőre.

6.1.7 Főoldal

Az alkalmazás főoldala az a hely, ahol a belépett felhasználók böngészhetik a jelenleg nyitott ötletdobozokat. Ez a felület az ábra 21-en látható. A nyitott ötletdobozok a középső grid-es listázó dobozban jelennek meg. Ez a grid az Angular Material Grid komponens segítségével lett létrehozva. Az ötletdobozok között lehet keresni is a keresőmezővel, ekkor szerver felé olyan kérés indul, amiben az adott karaktersorra szűr az adatbázis. A visszakapott ötletdobozok lapozható formában érkeznek a kliens oldalra, ami lapozható és az egyes lapok mérete is szabályozható. A lapozható formátumhoz

szerver oldalon a Pageable beépített osztályt használtam, ami a Spring Framework beépített megoldása az ilyen jellegű funkciók megvalósítására. A főoldalon a kártyákra húzva az egeret azok animálva kicsit megnőnek, illetve, ha rájuk kattint a felhasználó, akkor megnyitják az adott ötletdoboz teljes oldalát.

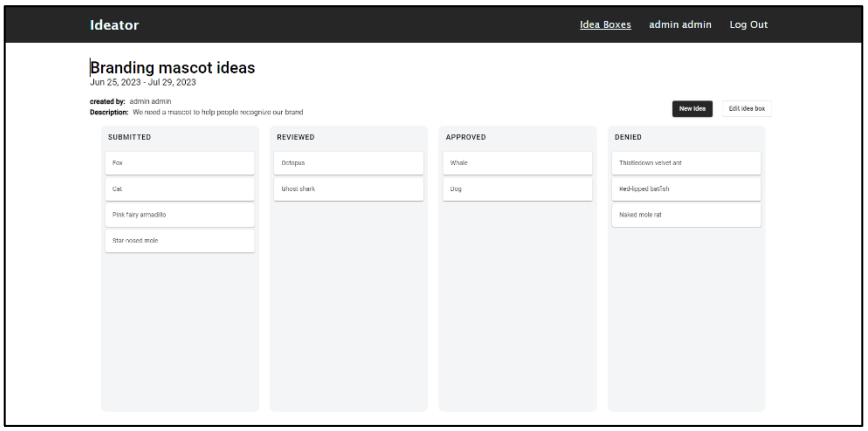
Amennyiben a bejelentkezett felhasználó Admin szerepkörrel rendelkezik, akkor egy Create és egy Manage Idea Boxes gomb is megjelenik, amikkel új ötletdobozt lehet létrehozni, illetve a meglévőketszerkeszteni.



ábra 21 - Az alkalmazás főoldala

6.1.8 Ötletdoboz

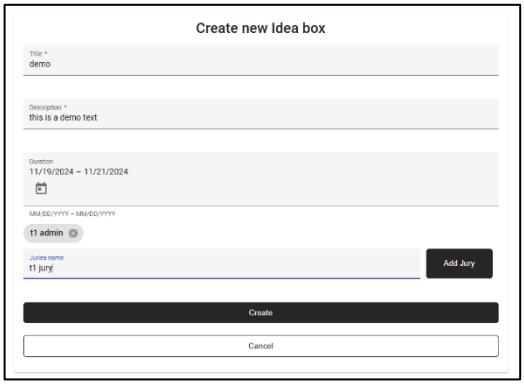
A kezdeti specifikáció és a UI design alapján az ötletdoboz tartalmát részletező oldalon elkészítettem egy Kanban táblához hasonló, 4 oszlopból álló komponenst. A táblában az ötletek kártyák formájában a státuszuk alapján kerülnek a 4 oszlop valamelyikébe. A kártyákon az ötlet címe olvasható és hasonlóan, mint az ötletdoboznál, itt is megnagyobbodik a kártya akkor, ha a kurzort ráhúzzuk. Kattintásra itt is megnyílik az ötletet részletező felület. Emellett az oldalon látható az ötletdoboz kezdési és zárási dátuma, létrehozója és az ötletdoboz leírása. A felületet az ábra 22 mutatja be.



ábra 22 - Ötletdoboz részletes oldala

Az oldalon megtalálható ezen felül egy „New Idea” gomb, ami minden felhasználó számára elérhető, és egy „Edit Idea Box” gomb, amivel az Admin szerepkörrel rendelkező felhasználó módosíthatja az ötletdobozt.

Az ötletdobozok létrehozása és szerkesztésére létrehoztam egy komponenst (ábra 23), ami mind a két funkcionalitást el tudja látni. A komponens attól függően, hogy melyik módban üzemel vagy a createIdeaBox$() vagy az editIdeaBox$() service függvényt hívja meg.

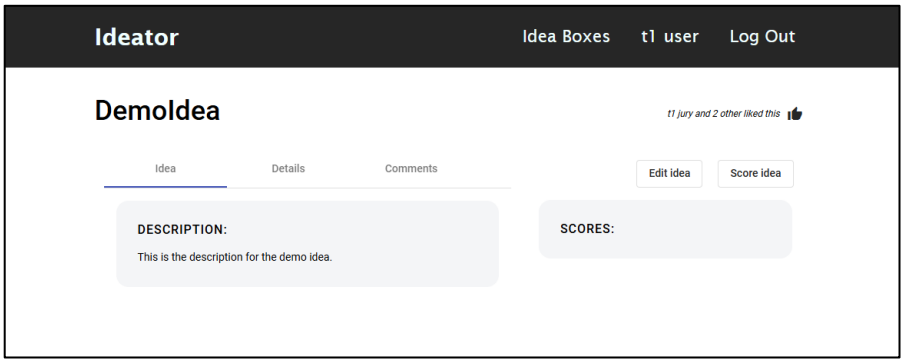


ábra 23 – Ötletdoboz Create/Edit komponens

Az alkalmazás összes olyan részénél, ahol egy objektumot kell létrehozni ugyan ezt a create/edit egyesített komponens megoldási módszert alkalmaztam.

6.1.9 Ötletek

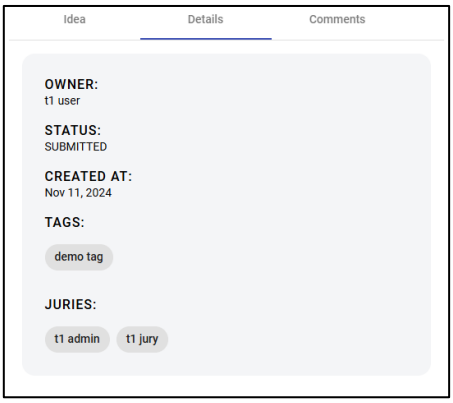
Az oldal implementálásánál a megtervezett UI volt a kiinduló alap, eszerint bontottam két nagyobb komponensre és 3 kisebbre a felületet, ahogy az az ábra 24-en is látható. A két nagyobb komponens a jobb és bal oldalon helyezkedik el a felületen, a bal oldalsó komponens pedig 3 kisebb komponensből áll elő, amik egy Angular Material Tabs komponensel vannak elválasztva. Ez a három tab az *Idea*, *Details* és *Comments*.



ábra 24 - Ötletdoboz részletező oldala

Az *Idea* oldalon ebben a fázisban az ötlet leírása van megjelenítve, de ez a felület kiváló helyet biztosít a jövőben több adat megjelenítésére is.

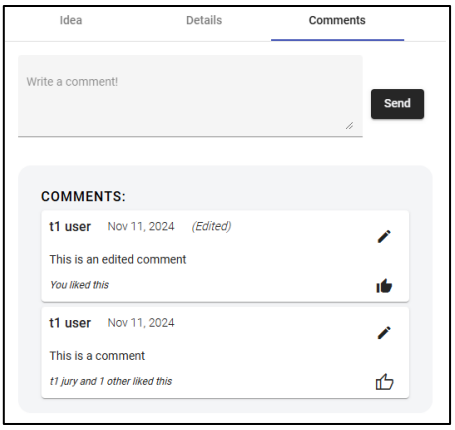
A *Details* oldalon (ábra 25) megtalálhatóak az ötlet alap adatai, ahogy az a képen is látszik. A Tag-ek és a Jury-k felsorolása az Angular Material Chip komponensével lett létrehozva.



ábra 25 - Ötlet details tab

A *Comments* oldal (ábra 26) egy kicsit bonyolultabb, mint az előző két oldal, hiszen itt több komponens is jelen van az implementációban. A felhasználó egy Angular Material Textarea komponensbe beleírhatja a kommentjét, amit a „Send” gombbal közzé tehet. Ekkor a komment megjelenik az alatta található Comments listában. Ebben a felsorolásban a kommentek a képen látható kártyákként jelennek meg, feltüntetve a kommentelő nevét, a komment készítésének időpontját, a komment szövegét és azt is, hogy kik kedvelték a kommentet. Ha egy adott kommentnek a felhasználó a tulajdonosa, akkor megjelenik a kis ceruza ikon a jobb felső sarokban ezzel indikálva azt, hogy a komment szerkeszthető. Ha egy komment szerkesztve lett, akkor a kártyán megjelenik az (Edited) szöveg, így a többi felhasználó is látja azt, hogy ez nem az eredeti komment.

A kommenteket minden felhasználó kedvelheti. Ezt a like ikonnal tudja megtenni. Ha egy felhasználó kedvel egy kommentet, akkor megjelenik a neve a like jel mellett, illetve a like jel teli lesz. Amennyiben egy kommenten a belépett felhasználó az egyetlen kedvelő, akkor nem a neve van kiírva, hanem az, hogy „You” (Te). Amennyiben egynél többember is kedvel egy kommentet, akkor mindig az utolsó kedvelő ember neve van kiírva, a többi ember csak számszerűsítve jelenik meg, ahogy az az ábra 26 képen is látható. Amennyiben egy felhasználónak mégsem tetszik a komment, akkor a like ikon újbóli megnyomásával eltávolíthatja a kedvelését a kommentről. Ekkor a like jel ismét csak körvonal lesz, illetve a neve is lekerül a listából, és a szám csökken eggyel. Ezzel a mechanizmussal működik az ötletek kedvelése/kedvelés visszavonása is, ennek az indikátora az ötlet jobb felső sarkában található.



ábra 26 - Ötlet comments tab

A jobb oldalon lévő scores komponensben jelennek meg az értékelések akkor, amikor az ötlet már értékelve lett legalább egy bíráló által, de ez a funkció még inkább

csak helyfoglalónak van itt, a tényleges funkció csak a második iterációban került implementálásra.

6.1.10 Összegzés

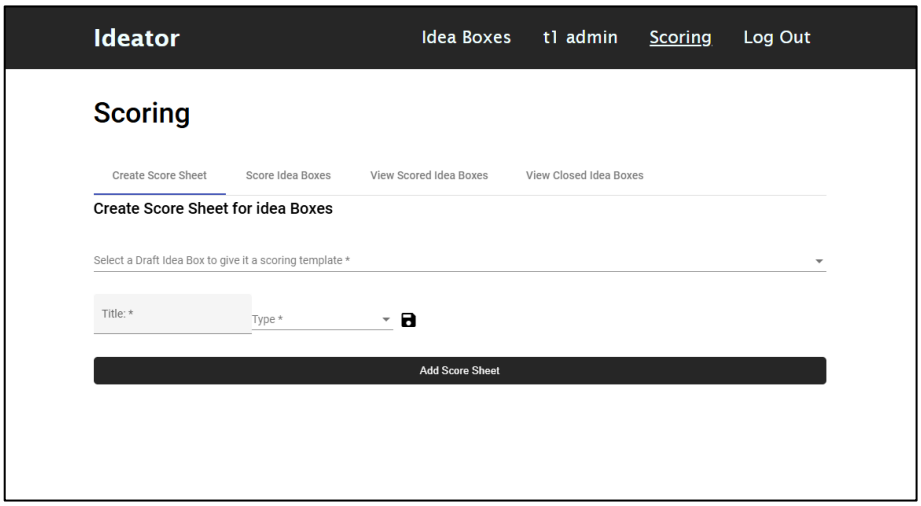
Ennek az iterációnak a végére az alkalmazás váza elkészült, több funkcionalitás teljesnek mondható, azonban sok dolog hiányzott még. Az alkalmazás pontozási rendszerére készült egy terv, azonban ezt a megrendelő bővíteni akarta, így ez a rész újra tervezésre szorult. Emellett az alkalmazás még nem valósítja meg a multitenant architektúrát sem, így ez is a következő iterációra maradt.

6.2 Második iteráció

A második iterációban két fontos funkció került implementálásra és több kisebb hibajavításra is sor került. Először a hiányzó pontozási folyamat készült el mind szerver oldalon, mind kliens oldalon. Ezután jött a multitenant architektúra megvalósítása, ami leginkább szerver oldalon hozott változásokat az alkalmazásba, kliens oldalon csak kisebb módosításokat kellett alkalmazni, hogy kihasználja az új lehetőségeket. Ebben a fejezetben ezeknek a funkcionalitásoknak az implementálásáról fogok írni.

6.2.1 Pontozási rendszer implementálása

Miután elkészült a pontozási rendszer folyamatának megtervezése nekiláttam az implementálásnak. Először kliens oldalon kezdtem el az új komponensek készítését, keretet adva a pontozási rendszernek. Ehhez az első lépés a Scoring aloldal elkészítése volt (ábra 27). Ezen az oldalon és ennek aloldalain történik minden pontozással kapcsolatos folyamat.



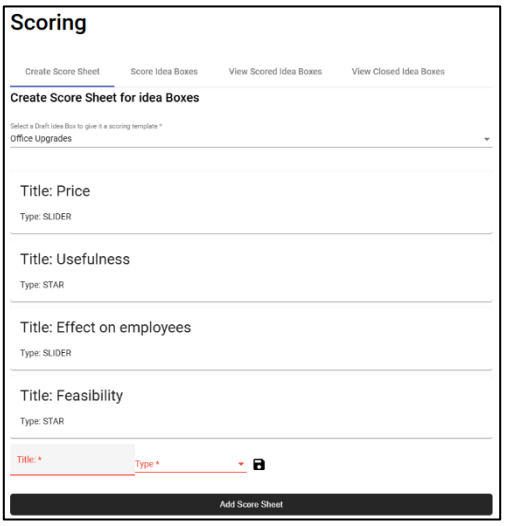
ábra 27 - Scoring oldal

Ezt az oldalt a User felhasználói körrel rendelkező felhasználók nem látogathatják, nem is jelenik meg nekik a navigációs sávban. Az oldal 4 részre oszlik, ám ezek az aloldalakon is vannak korlátozások.

Jury felhasználói körrel csak a „Score Idea Boxes” aloldal látogatható, admin szerepkörrel mind a négy.

6.2.1.1 Create Score Sheet aloldal

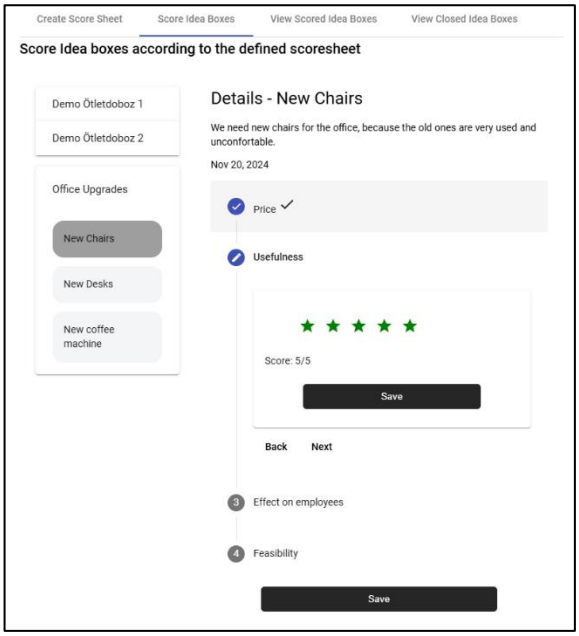
Ezen az oldalon, ahogy az ábra 28-on is látható, az ötletdobozokhoz lehet pontozási lapot adni. Ezt csak akkor lehet megtenni, ha az ötletdoboz még nem publikus, azaz még csak létre van hozva. Ezek az ötletdobozok közöl a „Select a Draft Idea Box…” lenyíló ablakban tudunk böngészni. Amikor kiválasztottuk a kívánt ötletdobozt, utána kezdhetjük szerkeszteni a pontozási szempontokat. A Title mezőbe kell beírni a szempont címét, a Type mezőbe pedig a szempont pontozási módszerét lehet kiválasztani. Az alkalmazás jelenleg két pontozási módszert támogat, a STAR és SLIDER. A mentés ikonra kattintva ez a szempont hozzáadódik a pontozási laphoz, alatta ismét megjelenik üresen a form komponens. Ezzel a dinamikus módszerrel nincs megszabva az, hogy mennyi szempontot adunk a pontozási laphoz. Amikor elkészült a felhasználó a pontozási lappal, akkor azt elmentheti az „Add Score Sheet” gomb megnyomásával. Ezzel az ötletdoboz megnyílik a nyilvánosság elé.



ábra 28 - Pontozási lap szerkesztése

6.2.1.2 Score Idea Boxes

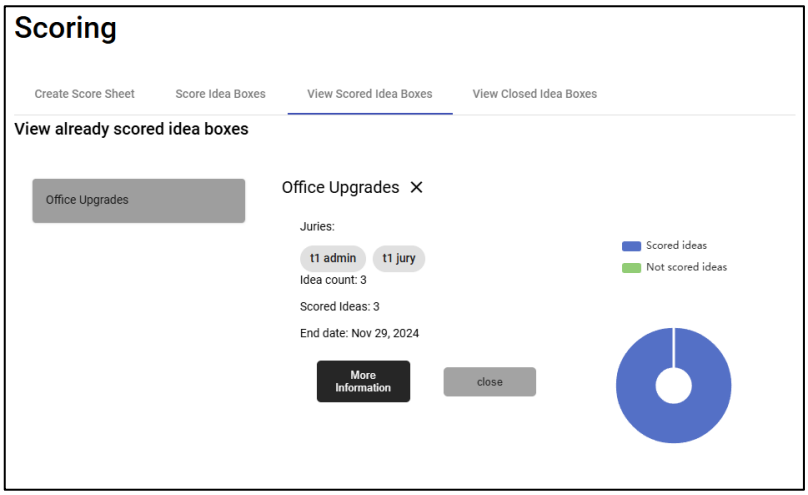
Ez az aloldal (ábra 29) felel a beérkező ötletek pontozásáért. Az oldal két részletre van osztva, a bal oldalon egy listát láthatunk azokról az ötletdobozokról, amikben már vannak ötletek és a belépett felhasználó (Jury vagy Admin) van jogosultsága az ötletet és ötletdobozt értékelni. Az ötletdobozok a listában kattintásra lenyilnak és láthatóvá válnak a beadott ötletek. Ha ezek valamelyikére kattint a felhasználó, akkor a jobb oldalon megjelenik a pontozási felület. Itt az ábra 29-en felül látható az ötlet címe, leírása és a létrehozás dátuma. Alatta egy vertikális Angular Material Stepper komponenst használva vannak felsorolva a pontozási lapban meghatározott szempontok és az azokhoz tartozó pontozási stratégia, amit a szempont pontozási módszere határoz meg. A képen látható példán egy csillagos pontozási módszer látható. A pontozó felhasználó az adott módszert használva (vagy a csillagokra kattintva, vagy a csúszkát húzva) beállíthatja a pontot amit az ötletre szeretne adni az adott szempont alapján. Ezután a Save gomb megnyomásával ezt a pontot elmentheti, és átléphet a következő szempontra a Next gomb megnyomásával, vagy ha szeretne vissza is léphet a Back gomb megnyomásával. A már értékelt elemek mellett ekkor megjelenik egy pipa is. Amikor az ötlet minden szempont szerint pontozva lett, akkor a bíráló leadhatja a pontozási lapját a Save gomb megnyomásával. Ezt csak egyszer teheti meg, amennyiben már értékelte az adott ötletet, akkor a felület hibát ad vissza egy SnackBar-ral.



ábra 29 - Ötlet pontozó felület

6.2.1.3 View Scored Idea Boxes

Ezen a felületen, amit az ábra 30 mutat be, tudják az admin felhasználók elbírálni a már pontozott ötleteket. Ez egy komplexebb felület, sok új funkcionalitással, amik a döntéshozást is segítik. Ez a felület is két részre oszlik, hasonlóan, mint az előző. A bal oldalon ismét egy lista található azokról az ötletdobozokról, amikben már vannak pontozott ötletek. A jobb oldalon pedig egy részletező felület található az egyes ötletdobozokról.



ábra 30 - Ötletdoboz abban az esetben, amikor minden ötlet pontozva lett minden bíráló által

Ezen a jobb oldalon látható az ötletdoboz neve, a bírálók felsorolása ismét Angular Material Chip-ekkel, az ötletek száma és az, hogy ezek közül az ötletek közül mennyi van már minden bíráló által pontozva. Ezen felül az ötletdoboz lejáratának dátuma is kiírásra került. Ezek alatt található két gomb. A „More Information” gomb az ötletdoboz egy részletesebb statisztikai felületét hozza be, a „Close gomb” pedig az ötlet lezárására szolgál. Az ötletet csak abban az esetben lehet lezárni, ha minden ötlet egyénileg elfogadásra vagy elutasításra került, és az ötletdoboz már nem fogad új ötleteket, azaz elérte a lejárati dátumát. Ezen az oldalon ezen felül található még egy kördiagram, ami a számszerűen kiirt, ötletek pontozására vonatkozó adatokat vizualizálja. Ehhez a diagramhoz (és a részletező oldalon lévőkhöz is) az ECharts nevű könyvtár Angular implementációját használtam. Ezzel a könyvtárral könnyen és egyszerűen lehet látványos diagramokkal színesíteni a kliens oldalunkat. HTML szinten egy egyszerű sor írásával lehet megjeleníteni a diagramokat. Ennek a diagramnak a megjelenítése:

```
<div echarts [options]="option" ></div>
```

Az option property a TypeScript fájlban van definiálva az ECharts

dokumentációja[21] alapján. A „More Information” gomb kattintása után az ábra 31-en látható felület fogadja a felhasználót.

ábra 31 - Ötletdoboz információs és elbíráló felülete

Ezen a felületen (ábra 31) több funkció is elérhető. Az ötletdoboz címe mellett látható egy visszafelé mutató nyíl, erre kattintva visszajutunk az előző oldalra. Ez alatt három oszlopot készítettem, amik számszerű adatokkal segítik a döntéshozást. Az első oszlopban a megadott szempontok alapján lehet vizsgálni az ötleteket. Itt egy oszlopdiagramon vannak vizualizálva az ötletekre kapott pontok. Az oszlopdiagram teljesen automatikusan működik, ha csillagos értékelésű az adott szempont akkor egytől ötig tart a diagram, ha csúszka alapú, akkor egytől tízig. Ez mellett található egy radar diagram, ami az ötletek szerint mutatja azt, hogy az egyes bírálók a megadott szempontokra mennyi pontot adtak. A mellékelt ábra 31-en látható, hogy 2 bíráló van az ötletdobozon, így kétféle színű vonal van, három értékelési szempont van, ezért a radar három tengelyen mozog. Az oldalon található még egy kapcsoló, amivel ki és be lehet kapcsolni a diagramokon megjelenített számokat.

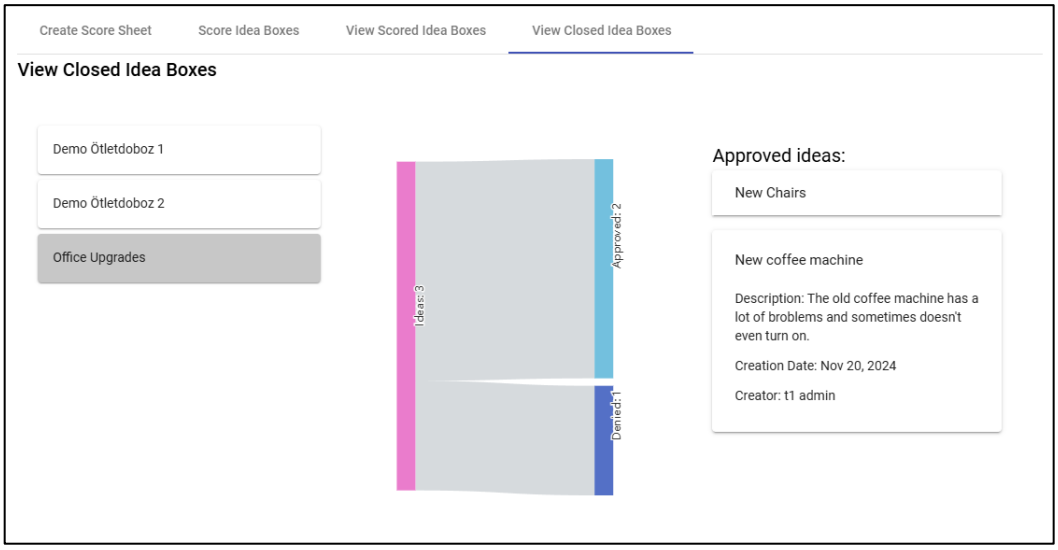
Az oldal jobb oldalán helyeztem el azt a panelt, amin az ötletet el lehet fogadni, vagy elutasítani. Ez fölött az ötlet leírása és egyéb információi is megtalálhatóak. Miután egy ötlet elfogadott vagy elutasított állapotba kerül, azon már nem lehet változtatni. Ekkor a két gomb elszürkül és nem is lehet rányomni. Ezen a panelen mindig az az ötlet van

[21] https://echarts.apache.org/examples/en/index.html

megjelenítve, ami a középső panelen ki van választva a legördülő menöből. Ezeket a diagramokat is az ECharts könyvtár segítségével készítettem.

6.2.1.4 View Closed Idea Boxes

Ez az aloldal az utolsó felület a Scoring oldalon, ezt az ábra 32 mutatja be. Itt a lezárt ötletdobozokat lehet vizsgálni és statisztikát készíteni arról, hogy a kiválasztott ötletdobozban milyen arányban voltak elfogadva az ötletek. Ez az oldal is két részből épül fel, a bal oldalon a lezárt ötletdobozok listáját jelenítettem meg, a jobb oldalon pedig egy ismét ECharts segítségével készített Sankey diagramot. A diagram bal oldalán láthatjuk az összes beérkezett és elbírált ötletek számát, ami kettéválik aszerint, hogy az mennyi ötlet lett elfogadva, illetve elutasítva. A diagram mellett az elfogadott ötletek listája látható. Ezek a kártyák kattinthatóak, kattintásra kinyílnak és az ötlet alap adatait mutatják.



ábra 32 - Lezárt ötletdoboz

Multitenant architektúra implementálása

Ebben a fejezetben az alkalmazás multitenant architektúrájának kialakítását fogom bemutatni. Ahogy azt a tervezésnél lefektettem, egy séma alapú szétválasztást implementáltam, ami azt jelenti, hogy az alkalmazás egyadatbázissal áll kapcsolatban, és azon az adatbázison belül minden bérlőnek külön adatbázis séma van biztosítva. Ez a módszer egy tökéletes középút az adatszeparáció és a költségek között.

Az implementáció legnagyobb részben a szerver oldalon történt, a kliens oldalon csak kisebb változásokat kellett eszközölnöm, így a bemutatást először a szerver oldalon kezdem.

6.2.1.5 Szerver oldali implementáció

A multitenant architektúra kialakításához a fileokat a com.moa.backend.multitenancy csomagba szerveztem. Az első lépés az volt, hogy a szerver tudomást szerezzen arról, hogy a beérkező kérések melyik bérlőtől származnak. Ezt jelenleg egyszerűen az X-TenantId nevű header-ben adom át kliens oldalon. Szerver oldalon létrehoztam egy filtert (TenantFilter.kt), ami ezt a header-t olvassa. Annak érdekében, hogy a kérés során az alkalmazás tudja, hogy milyen bérlővel kell dolgozni, ezt a beérkező tenantId-t eltárolom egy TenantContext objektumban, ami ThreadLocal típusú currentTenant propertyvel rendelkezik. Azért esett a választásom a ThreadLocal-ra, mivel ez egy beépített és jól működő és biztonságos Java alapú mechanizmus arra, hogy adatokat tároljunk rövid ideig (a kérés idejéig) egy Thread-en. A TenantContext objektumnak létrehoztam még egy getter és setter függvényt is, hogy ezt a privát currentTenant-ot biztonságosan tudjam kezelni, illetve egy clear() függvényt is definiáltam, hogy a kérések végén tudjam törölni a tartalmát.

Azután, hogy az alkalmazás el tudja dönteni, hogy melyik bérlő adatait kell majd elérnie, az adatelérés implementálása volt a következő feladat. Ezt három új fájl létrehozásával oldottam meg, ezek a fájlok a JpaConfig.kt, CurrentTenantIdentifierResolver.kt és TenantConnectionProvider.kt.

CurrentTenantIdentifierResolver

Ennek a komponensnek a feladata az aktuális bérlő azonosító meghatározása. Ehhez a már bemutatott TenantContext objektumot használom, ebből olvasom ki az éppen aktuális bérlő id-ját.

TenantConnectionProvider

Ez a komponens az adatbázis kapcsolat kezeléséért és a megfelelő adatbázis séma kiválasztásáért felel. Ehhez a MultitenantConnectionProvider interfészt valósítom meg. A getConnection függvény az aktuális tenantId alapján hoz létre adatbáziskapcsolatot. Első lépésben a megadott sémát ellenőrzi és ha szükséges, akkor létrehozza azt, utána a USE SQL utasítással a kiválasztott sémára vált. A relaseConnection függvény lezárja az aktuális adatbázis kapcsolatot a beérkező kérés végén, ezzel szabadítva fel erőforrást. Abban az esetben amikor nem áll rendelkezésre bérlő azonosító, akkor a getAnyConnection és releaseAnyConnection hívódik meg, amik a hatására egy általános adatbáziskapcsolatot kapok és kezelek.

JpaConfig

A jpaConfig feladata a Spring JPA konfigurálása és a tranzakciókezelés. Itt két fontos függvény található, az entityManagerFactory és a transactionManager.

Az entityManagerFactory hozza létre az entitáskezelőt, amihez az általam definiált jpaPropertiesMap-et használja. A jpaPropertiesMap-ben van beállítva az általam definiált tenantConnectionProvider és identifierResolver, emellett itt van beállítva a séma alapú multitenancy működés is. A factory-ban emellett az is be van állítva, hogy az entityManager hol keresse az alkalmazásban azokat az entitásokat, amiket tárolni szeretnék az adatbázisban.

A transactionManager beállít egy JpaTransactionManager tranzakciókezelőt, amely biztosítja, hogy a tranzakciók megfelelően működjenek és a JPA összhangban legyen a multitenancy konfigurációval.

6.2.1.6 Kliens oldali implementáció

Kliens oldalon az alkalmazást headerHandler fájlját kellett átírnom olyan módon, hogy most már ne csak a JWT-t helyezze el a headerben, hanem a kliens által tárolt TenantId-t is. Annak érdekében, hogy a TenantId-t tárolni tudjam, kliens oldalon is készítettem egy TenantContexthez hasonló tároló objektumot, itt egy service keretein belül. A TenantService tenantId proprety-je tárolja kliens oldalon az előre beállított tenantId-t. Ennek a servicenek is készítettem getter és setter függvényeket, illetve clear függvényt is. A getter függvény a tenantId header-be helyezésénél hívódik meg, a clear pedig amikor a felhasználó kilép az alkalmazásból. A setter jelenleg akkor hívódik meg,

amikor a felhasználó belép az alkalmazásba. Mivel a jelenlegi alkalmazás egy prototípus, ezért a login felületre vettem fel egy tenant választó dropdown beviteli mezőt. Ezzel lehet kiválasztani a kliens oldalon, hogy a felhasználó melyik tenant-ba szeretne belépni.

7 Tesztelés

Ebben a fejezetben az alkalmazás tesztelésének menetét mutatom be. Az alkalmazás tesztelését három féle módszerrel folytattam, ezek az Unit tesztelés, integrációs tesztelés és End-to-End tesztelés volt. Ezeket a következő alfejezetekben fejtem ki.

7.1 Unit tesztelés

Az alkalmazás Unit tesztelését szerver oldalon hajtottam végre. A tesztelés során JUnit 5-öt és MockK-ot használtam. A tesztek írása során a komponensek izolációjához a @MockkBean annotációt használtam, ezen felül nagy hangsúlyt fektettem itt a szerepkör alapú tesztelésre is. Azoknál a service függvényeknél, ahol volt értelme, ott készítettem több tesztet is egy adott funkcionalitáshoz annak érdekében, hogy több szerepkörrel rendelkező felhasználó lekérdezését tudjam tesztelni és ellenőrizni.

A tesztek írásánál törekedtem arra, hogy a teszt neve mindig jól leírja a tesztelni kívánt funkcionalitást, ezért az elnevezésnél az „It Should” mintát követtem. Ez azt jelenti, hogy a tesztek leírják a tesztelendő funkcionalitás elvárt viselkedése egy specifikus esetben. Erre egy példa az ábra 33-on látható látható teszt címe.

```
@Test   Ádám Varga
fun `deleteIdeaBox() should return 200 when IdeaBox is successfully deleted`() {
    val ideaBoxId = 1L
    val authentication = mockk<Authentication>(relaxed = true)

    val securityContext = mockk<SecurityContext>(relaxed = true)
    SecurityContextHolder.setContext(securityContext)

    every { securityContext.authentication } returns authentication
    every { authentication.authorities } returns listOf(SimpleGrantedAuthority( role: "ADMIN"))

    every { ideaBoxRepository.deleteById(ideaBoxId) } returns Unit

    val response: ResponseEntity<*> = ideaBoxService.deleteIdeaBox(ideaBoxId)

    val responseBody = response.body as WebResponse<String>
    assertEquals(HttpStatus.OK.value(), responseBody.code)
    assertEquals( expected: "Idea Box successfully deleted!", responseBody.message)
    assertEquals( expected: "Idea Box successfully deleted!", responseBody.data)

    verify { securityContext.authentication }
    verify { ideaBoxRepository.deleteById(ideaBoxId) }

    SecurityContextHolder.clearContext()
}
```

ábra 33 - Példa teszt - Ötletdoboz törlése

A képen (ábra 33) látható teszt jól szemlélteti a tesztjeim felépítését. A szerepkörök ellenőrzését a SecurityContext mockolásával oldom meg. Az every {}

kódblokkokban vannak definiálva az esetlegesen meghívott egyéb komponensek visszatérési értékei. Ezután megtörténik a tényleges tesztelni való függvény hívása, amit az elemzés követ. Az elemzésben összehasonlítom az elvárt és visszakapott eredményeket, illetve vizsgálom azt is, hogy a függvény meghívott-e olyan függvényeket a törzsben, amiket meg kellett, hogy hívjon a helyes működésnél. A teszt végén törlöm a SecurityContextHodlert.

Összesen az alkalmazás szerver oldalához 214 Unit tesztet írtam, amik sikeresen lefutnak. A kódom lefedettségének vizsgálásához a JaCoCo nyílt forráskodó könyvtárat használtam. Ez egy webes felületen jeleníti meg grafikusan a kódlefedettséget (ábra 34).

backend Sessions

backend

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.moa.backend.service		82%		69%	188	463	114	1,089	58	217	16	92
com.moa.backend.model		53%		55%	107	192	58	244	70	147	1	11
com.moa.backend.mapper		68%		41%	90	172	104	426	28	104	0	7
com.moa.backend.security.config		45%		5%	41	52	67	110	24	35	0	4
com.moa.backend.model.dto		47%		n/a	71	121	10	67	71	121	1	7
com.moa.backend.model.slim		36%		n/a	57	72	7	43	57	72	0	7
com.moa.backend.multitenancy		46%		16%	17	31	37	76	14	28	1	8
com.moa.backend.controller		89%		54%	26	122	2	121	6	98	3	14
com.moa.backend.security.auth		78%		n/a	5	21	0	17	5	21	0	3
com.moa.backend.utility		78%		n/a	3	12	0	12	3	12	0	4
com.moa.backend.model.dto.utility		42%		n/a	4	5	0	3	4	5	0	1
com.moa.backend		20%		n/a	1	2	2	4	1	2	1	2
Total	4,092 of 13,673	70%	328 of 804	59%	610	1,265	401	2,212	341	862	23	160

Created with JaCoCo 0.8.10.202304240956

ábra 34 - JaCoCo felülete és az alkalmazás lefedettsége

7.2 Integrációs tesztelés

Az alkalmazás integrációs tesztelését bizonyos esetekben Postman-nel végeztem el. Ez az API tesztelés manuálisan lettek elvégezve és inkább csak ellenőrzés szempontjából volt fontos. Ezt a technikát az olyan funkcionalitások tesztelésére használtam, ahol vagy szerepkörök, vagy bérlők közötti adatbiztonságot teszteltem. Olyan esetekben is használtam, amikor meg akartam bizonyosodni, hogy egy adott funkció nem csak kliens oldalon van-e védve, hanem szerver oldalon is. Emellett az alkalmazás fejlesztése során itt többször használtam egy-egy funkcionalitás gyors tesztelésére.

7.3 End-to-End tesztelés (E2E)

Az alkalmazás End-to-Enmd tesztelését manuálisan végeztem el. Az alkalmazás kliens oldalát használva az összes lényeges funkciót leellenőriztem a lehető legtöbb szerepköri kombinációban annak érdekében, hogy megbizonyosodjam arról, hogy az alkalmazás valóban úgy működik-e, ahogy az a funkcionális követelményekben le volt fektetve. Ennél a tesztelésnél figyeltem arra is, hogy ne csak a funkcionalitásokat ellenőrizzem le, hanem azt egyes felhasználók valóban csak azokat a gombokat, oldalakat látják, amiket a szerepkörük megenged

8 Értékelés, továbbfejlesztési lehetőségek

Ebben a fejezetben összefoglalom és értékelem a munkámat, majd ajánlok továbbfeljesztési lehetőségeket.

8.1 Értékelés

A diplomatervem írása során egy olyan ötletkezelő alkalmazás prototípusát terveztem meg és fejlesztettem le, amely rendelkezik az ötletkezelő alkalmazások alapvető funkcionalitásaival. Az alkalmazás képes a felhasználó és szerepkörök kezelésére, ötletek létrehozására, ezek pontozására és elbírálására. Emellett kezdetleges üzleti intelligenciai igényeket is kielégít az ötletek összehasonlítására és a legjobb döntés meghozására. Az alkalmazás architektúrája megvalósítja a több bérlős modellt a séma alapú adatszeparációval, ezzel fokozott biztonságot nyújtva a bérlők számára.

Az alkalmazás fejlesztése közben sok új technológiával ismerkedtem meg mélyebben, főleg a biztonságos felhasználókezelés és a több bérlős architektúra implementálása során. Emellett lehetőségem volt mind kliens oldalon, mind szerver oldalon egy nagyobb méretű projekten dolgozni, ahol a komponensek újra felhasználása és az implementáció struktúrája már nagyobb hangsúlyt kapott, mint egy kisebb demó alkalmazásnál. Az fejlesztés utáni tesztelés is tanulságos volt, hiszen egy ekkora alkalmazás tesztelését már nem lehet csak manuális, E2E teszteléssel lebonyolítani, így az Unit tesztek világába is betekintést nyertem.

A fejlesztés során a konzulensemmel imitált agilis fejlesztésbe is belekóstolhattam azzal, hogy visszacsatolásokkal több iterációban terveztem és fejlesztettem az alkalmazást.

8.2 Továbbfejlesztési lehetőségek

Az elkészült alkalmazás a második fejlesztési fázis után is még egy prototípusnak tekinthető. Az alap funkcionalitások megvannak, ám sok olyan funkcionalitás jutott az eszembe a fejlesztés során, amik a folyamatokat gördülékenyebbé és használhatóbbá tehetnék. Ilyenek például:

- User szerepkörök bővítése divíziókra, amikkel az egyes munkacsoportok saját ötletdobozokat is létrehozhatnak azért, hogy a szervezeten belül a felhasználók csak egy kisebb csoportja férhessen hozzá.
- Bírálatot segítő megoldások továbbfejlesztése: Itt a határ a csillagos ég, a mostani diagramokon felül be lehetne építeni egy sokkal összetettebb értékelést segítő rendszert, ami akár AI segítségével tudja a döntéshozót segíteni historikus adatok és egyéb, a szervezeten belül elérhető adatok és BI rendszerek vizsgálatával.
- Az egyes bérlők felületei között jelenleg nincs különbség. Ezt lehetne továbbfejleszteni azzal, hogy a bérlők testre szabhassák a felületüket a saját színeikkel, képeikkel és beállításaikkal. Akár a szerepkörökhöz tartozó jogosultságaikat is beállíthatnák a saját felületükön.

Irodalomjegyzék

[1] "Ötletkezelő alkalmazások" *https://www.worldscientific.com/doi/epdf/10.1142/S1363919603000878*

[2] Mikelsone, E., Spilbergs, A., & Segers, J.-P. (2021). Benefits of Web-based Idea Management System Application. Idea Management Journal, 10(2), 45-60. *https://bibliotekanauki.pl/articles/1879852.pdf*

[3] „Single-Page application (SPA)" *https://link.springer.com/chapter/10.1007/978-1-4302-6674-7_1*

[4] „DOM (Document Object Model)" *https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction*

[5] „Három rétegű architektúra" *https://www.oreilly.com/library/view/software-architecture-patterns/9781098134280/*

[6] „REST" *https://www.oreilly.com/library/view/restful-web-apis/9781449359713/*

[7] „Controller-Service-Repository" *https://www.amazon.com/Professional-Java-Applications-Nicholas-Williams/dp/1118656466*

Függelék