Győrffy András

Témalaboratórium beszámoló:

Alkalmazásfejlesztés React és ASP.NET Core alapokon

Konzulens: Benedek Zoltán

Budapest, 2020

1. Bevezetés

Az alkalmazásfejlesztés React és ASP.NET Core alapokon témalabor keretében egy full stack, feladatkezelésre alkalmas webalkalmazást fejlesztettem. Bár maga a C# nyelv, illetve a React, mint Javascript könyvtár nem voltak ismeretlenek számomra, a félév során mégis sok újdonságot tanultam, mint például a REST API fejlesztés ASP környezetben, illetve a Typescript nyelv és Redux alapú állapotkezelés a Reactben.

A webalkalmazásom középpontjában a különböző állapotú teendők állnak, ezeket lehet létrehozni, módosítani és törölni, illetve különböző nézeteken megtekinteni. Az időtartammal, vagy határidővel rendelkező elemek egy naptáron is láthatók, illetve swimlane-eken is, ahol függőben – folyamatban – kész állapotok követhetőek a feladatok oszlopok közötti mozgatásával. A feladatokhoz lehet órákat / kurzusokat rendelni, és ezeket színkódokkal ellátani. A teendők között lehetőség van keresni is, illetve címkéket létrehozni és hozzájuk rendelni. Több munkaterület is létrehozható különböző projektekhez, amelyekhez meg lehet hívni más felhasználókat.

A forráskód elérhető a GitHubon.

Felhasznált technológiák

1.1. NET Core

A .NET Core a Microsoft egyre nagyobb népszerűségnek örvendő, nyíltforráskódú, több platformot is támogató (a .NET Frameworkkel ellentétben Linux, illetve MacOS-n is futó) futtatható alkalmazásfejlesztési környezete. Bár támogatottsága jelenleg még nem teljesen éri el a .NET Frameworkben megszokottat, modern .NET alapú alkalmazásfejlesztésnél teljesen egyértelmű a használata, hiszen a Microsoft által közzétett ütemterv alapján a (közel)jövőben ez teljesen lecseréli a Frameworkot.

1.2. ASP .NET Core Web API

Az ASP .NET a Microsoft webalkalmazásfejlesztési keretrendszere, melynek segítségével nemcsak REST API, hanem kliensoldalon megjeleníthető webtartalmakat lehet előállítani, akár szerveroldali rendereléssel (Razor Pages), vagy újabban SPA (Single Page Application) formában is, a Blazor segítségével. Bár ezek szintén érdekes technológiák, a kliens oldalon a később taglalt React könyvtárat használtam, az ASP .NET-nek csak a REST API képességeit használtam ki.

1.3.MongoDB és MongoDB .NET Driver

Az alkalmazásom adatbázisául a MongoDB-t választottam, pusztán amiatt, mert NodeJS környezetben már voltak vele korábbi tapasztalataim. A MongoDB egy NoSQL, azaz nem relációs adatbázis, amelyben táblák helyett kollekciók, sorok helyett pedig JSON-szerű dokumentumok vannak, amelyeknek sémája nem szabott. Egy ilyen adatbáziskezelőrendszer számos előnnyel rendelkezik többek között a skálázhatóság terén, ám ezeket egy ilyen méretű projektnél nyilvánvalóan nem használtam ki.

A kódbázishoz a MongoDB által kiadott, hivatalos .NET Driver Nuget csomagot használtam, amely az objektumok dokumentumokra (és vissza) való leképezést segítette. A csomag támogatta az általam az alkalmazásban sok helyen használt aszinkron programozási paradigmát, és az Entity Frameworkhöz hasonló adatelérést engedett meg. Véleményem szerint azonban funkcionalitásában elmarad például a mongoose nevű Javascript könyvtártól, illetve a sémamentesség is sokkal kisebb előny ahhoz képest, mint amikor Javascriptet használunk szerveroldalon, hiszen az erősen típusos C#-ban úgyis definiálnunk kell biztos sémákat típusokkal, és emiatt nem lehet használni a nagyon kényelmes „populate” funkciót,

amely lekérdezéskor egy dokumentumban tárolt ObjectID által azonosított másik dokumentumot keres ki a megadott kollekcióból, és lecseréli vele az ObjectID-t.

1.4. JSON Web Token

A felhasználók autentikációjához / autorizációjához JSON Web Tokeneket használtam, ami egy digitálisan aláírt tokenben tárol JSON formában információt. Előnye, hogy nem kell a szerveroldalon sessionökkel foglalkozni, hanem csak a token dekódolásával meg lehet kapni a felhasználó információit (mi az azonosítója, neve, email címe stb.), és a digitális aláírás biztosítja azt, hogy más ne tudhassa ezeket az adatokat módosítani.

1.5. React

A Javascript a webfejlesztés során megkerülhetetlen, minden weboldal működéséhez szükséges, ám a kívánt kinézet és viselkedés eléréséhez számos keretrendszert és könyvtárat találunk. Manapság a legnépszerűbb opciók a komponens alapú megoldások, mint például a React, a Vue és az Angular. Ezek közül igazából csak a Google által fejlesztett Angular, illetve a korábban ott dolgozó, és kiábrándult Evan You által fejlesztett Vue keretrendszerek ténylegesen, a Facebook Reactje pedig valójában csak egy Javascript könyvtár, amely alapból nem tartalmaz állapotkezelést, útvonalkezelést, vagy http kéréseket, csak és kizárólag a DOM-ba való renderelésért felelős. Valódi keretrendszerré ezt például a Create React App teszi, amely átvállalja a React kód sima Javscriptté alakításának oroszlánrészét, feladatunk ezután „egyszerű”, mindössze meg kell írnunk a komponenseket, és a fejlesztés alatt álló oldalunk minden fájl mentésével lefordul és frissül a böngészőnkben. Én is ezt a megoldást használtam a projekt készítése közben, kiegészítve a Create React App sablont a lejjebb taglalt Typescripttel.

A React könyvtár nemrégiben hatalmas koncepcionális változáson ment keresztül, ami során a komponensek típusa változott meg osztályokról függvényekre. A régi, osztálykomponenses rendszer mögött álló gondolatot nagyon egyszerű megérteni objektumorientált gondolkodással: minden komponens egy osztály, amely a React.Component osztályból származik le, és rendelkezik különböző, felülírható életciklus függvényekkel, amelyek például felcsatoláskor / lecsatoláskor hívódnak meg, illetve egy kötelező render fügvénnyel, aminek visszatérése értéke kb. HTML tartalom, azaz az ismerős HTML tagek és egyéb komponensek.

Ezzel szemben a funkcionális komponensek olyasmik, mint az osztálykomponensek render függvényei, azaz HTML tartalmat adnak vissza, de ezelőtt még ún. Hookokkal

(horgok) lehetőség van állapotot kezelni (useState), különböző változásokra feliratkozni (useEffect), és még rengeteg másra. Bár ez az új módszer ijesztőnek tűnhet, főleg objektumorientált környezetből érkezőknek, ez lesz a React jövője és más előnyök mellett jelentős hatékonyságot is hoz az osztályokhoz képest.

Nekem igazából csak az osztálykomponensekkel volt tapasztalatom, így a félév során célom volt minél mélyebben megismerkedni a funkcionális komponensekkel és horgokkal. Bár a projekt végére úgy érzem, megfelelően elsajátítottam az új paradigma alapjait, mégis kicsit kevésbé találom kifejezőnek azt. Például, ha a célunk egy olyan logika írása, ami a komponens első megjelenítésekor egyszer lefut, akkor a korábbi componentDidMount() függvény felüldefiniálása helyett egy useEffect horgot kell alkalmaznunk, amelynek függőségtömbje üres – mivel lefutása nem függ semmitől, ezért csak egyszer fog lefutni a felcsatoláskor, és utána többet nem.

1.6. Typescript

A Typescript a Microsoft által fejlesztett kiegészítés a Javascripthez, ami statikusan típusossá teszi azt. A nyelv maga egy bővebb halmaza a Javascriptnek, és arra is fordul át, tehát könnyedén integrálható Javascript alapú projektekbe Természetesen egyesek szerint ez a Javascript rugalmas koncepcióval gyökeresen szembemegy, mindenesetre számomra hatalmas segítség volt, hiszen a fejlesztőkörnyezet és a fordító rögtön felhívják a figyelmet arra, hogy valamilyen probléma van (inkompatibilis típusok, egy változó lehet, hogy nem definiált, pedig annak kéne lennie), és ez sok-sok időt spórol meg, amit a lefordított kód tesztelésével töltött volna az ember.

1.7. Redux és Redux Toolkit

A Redux egy manapság igen elterjedt Javascript könyvtár az állapotkezelésre. Fő előnye, hogy az állapotot tároló „raktár” (store) bárhonnan elérhető az alkalmazáson belül, legyen szó React komponensekről, vagy egyszerű Javascript (Typescript) modulokról. Használata nélkül könnyen belefuthatunk olyan megoldásokba, hogy a komponensfánkon valamilyen adatra több helyen is van szükség (például a belépett felhasználót megjeleníti a navigációs sáv is, de a profil oldalon is látszik stb.), így azt kénytelenek vagyunk a legfelső közös ősbe elhelyezni, és propként átadni. Ez akkor lesz igazán átláthatatlan és kezelhetetlen, ha az ősnek több komponensen keresztül kell átadnia azt annak a gyermeknek, akinek szüksége van rá – nem is beszélve arról, hogy a köztes komponenseknek semmi közük nincsen ahhoz az adathoz, amit tovább kell adniuk.

Többek között ezt a problémát oldja meg a mindenhonnan elérhető Redux store, amely tehát az alkalmazás állapotát tárolja, legyen szó http kérések eredményéről, vagy valamilyen belső információról (kattintott-e a felhasználó már egy bizonyos helyre stb.). A Redux Toolkit, a Create React Apphez hasonlóan, nagyban leegyszerűsíti a Redux konfigurálását, és a React Redux csomaggal együtt a funkcionális komponensek horgain keresztül teszi lehetővé a store elérését és különböző akciók kiadását.

1.8. Axios

Az Axios egy, a Promise API-n alapuló http kéréseket kezelő könyvtár. Segítségével ezek jóval könnyebben használhatóak, mint a Javascript beépített fetch megoldásával, mert például automatikusan konvertálja a kéréseket, válaszokat Javascript objektumokká, és a http státusz kódokat kivételekké tudja alakítani, ami a hatékony hibakezeléshez elengedhetetlen.

1.9.MomentJS

Az először 2011-ben MomentJS hosszú ideje az első számú választása az idő és dátumkezelő könyvtárat kereső fejlesztőknek. A közelmúltban azonban számos kritika érte egyrészt a mutabilitása miatt, másrészt az internacionalizáció és időzónák által okozott nagy mérete miatt, így a fejlesztőcsapat befejezte a további fejlesztést, a továbbiakban csak karbantartani fogják a kódbázist.

Emiatt természetesen nem ajánlott új projektekben ezt a könyvtárat használni, a MomentJS weboldalán a fejlesztői is legalább négy különböző alternatívát ajánlanak fel. Normális esetben én is ezek egyike közül választottam volna, de az alkalmazásom egy fontos részét képezte a naptár nézetet megvalósító könyvtár, ami viszont MomentJS-t használt, így egyszerűbbnek találtam a programom egészében egységesen ezt alkalmazni.

1.10. Elastic UI

Bár a lehetőség meg van rá, a legtöbb weboldal fejlesztésekor annak kinézetét gyakran nem a nulláról készítik el a fejlesztők, hanem valamilyen UI könyvtárat használnak, amelyek különböző CSS osztályokat és kinézeti elemeket definiálnak. React környezetben további elvárás, hogy egy ilyen könyvtár komponenseket definiáljon, amiket megfelelő propokkal lehet kontrollálni és testre szabni. Ilyen könyvtárból számos található, mint például a React Bootstrap könyvtárral kiegészített Bootstrap, vagy a Material UI, én azonban a kevésbé ismert Elastic UI-t választottam, mert ezzel korábbi tapasztalatom még nem volt. A könyvtár maga rengeteg komponenst tartalmaz, a megszokott gombok, formok, flex elrendezések és

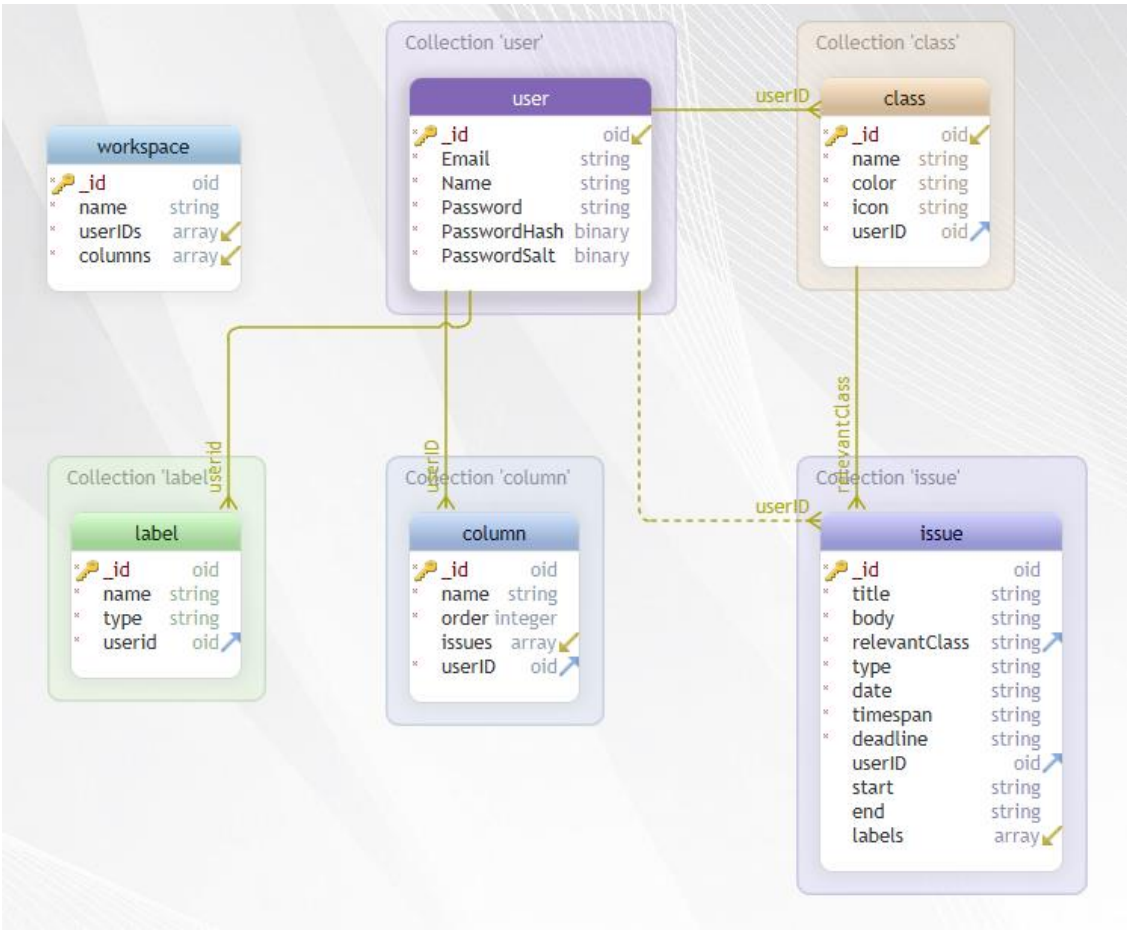
felugró ablakok mellett saját ikonok, markdown szerkesztő, grafikonok és túrák (az oldal ismertetésére szolgáló több lépésből álló felugró ablakok) is megtalálhatóak benne, és kezelésük is könnyű volt. Az ilyen könyvtárak nagy ereje igazából abban rejlik, hogy hogyan szabja őket testre az ember, alakítja egyedivé a formákat és színeket, azonban a mostani projekt keretében ezt a lehetőséget nem használtam ki, hanem az (egyébként így is kifinomult) alap témával dolgoztam.

2. Felépítés

Az elkészített webalkalmazás frontend és backend részre válik, ezeket egymástól teljesen függetlenül, más technológiákkal készítettem el. A backend oldalon a fent említett ASP .NET Core Web API minta projektjéből indultam ki.

A backend oldal architektúráját leginkább az adatbázis sémái határozzák meg. Hat darab MongoDB kollekció mindegyikéhez különböző sémák tartoznak, mégpedig: Class (a feladatokhoz hozzárendelhető osztályok), Column (a swim lane nézeten megjelenő oszlopok), Issue (maguk a feladatok), Label (feladatokhoz hozzárendelhető címkék), User (az alkalmazás felhasználói), Workspace (munkaterület, amely ismeri a hozzárendelt felhasználókat és oszlopokat).

Az adatbázis sémája a következő ábrán látható:



Az adatbáziskapcsolatért a Database nevű osztály felelős, illetve a rajta található property-k segítségével érhetőek el az egyes kollekciók. A különböző dokumentumokat a Repository modell szerint minden sémához külön repository-ból érhetjük el, amelyek

függőséginjektálás segítségével érik el az adatbázist. A repository-k nyilvános függvényei a legtöbb esetben valamilyen CRUD operációért felelősek, azaz létrehoznak, visszaadnak egy keresett, vagy az összes dokumentumot, frissítik, vagy törlik azokat. Egy gyakori privát metódus azonban a mongoose könyvtárból irigyelt Populate függvény megvalósítása.

Nyilvánvaló elvárás egy komplex alkalmazás esetében, hogy az adatbázis sémája ne határozza meg teljesen a külvilággal való kommunikációt, hanem kifelé logikus, könnyen átlátható és kezelhető formátumú adatokat kínáljunk, viszont az adatbázisunkban ezeket hatékonyan tároljuk. Az én esetemben az egyes dokumentumok gyakran tárolnak egy, vagy több referenciát (MongoDB ObjectID) formájában más dokumentumokra – például egy munkaterület az összes hozzá rendelt felhasználót. Ám amikor le akarunk kérdezni egy munkaterületet a kliensoldalról, nem lenne szerencsés, ha csak egy hosszú betű- és számsort kapnánk, ami alapján további lekérdezéssel kellene kiderítenünk, hogy pontosan ki a felhasználó, azaz például mi a neve – hiszen ezt fogjuk felhasználni a felhasználói felületen.

Ezért tehát a sémák nagy részének lesz egy kliensoldali párja, amely ugyanazzal a névvel rendelkezik, csak előtte a Client szócska áll (pl. ClientWorkspace), amelyeknek property-jei között nem ObjectID-ket tároló listákat, hanem a teljes dokumentumot leképező osztályokból (pl. ClientUser) álló listát találunk. Ezek előállításáért felelősek az egyes repository-k Populate metódusai, amelyek kikeresik a listában talált azonosítók alapján a megfelelő dokumentumokat, és felcsatolják azokat a kliensnek szánt objektumra.

A felhasználókat kezelő repository és séma némileg egyedi, ugyanis az új felhasználó létrehozásakor (regisztrációkor) annak jelszavát természetesen nem mentjük el egyszerű szövegként, hanem byte tömb formájában egy hash-t és sót (salt) tárol, amelyek alapján egy beérkező bejelentkezési kérés alapján el lehet dönteni, hogy helyes jelszót adott-e meg a felhasználó. Ezeket az adatokat a regisztrációkor (Register függvény) generáljuk, és a belépésnél ellenőrizzük (Authenticate), a kérések és válaszok sémáit a könnyebb kezelhetőség érdekében egy-egy közös osztályba vettem (AuthenticateRequest, AuthenticateResponse, RegisterRequest).

Miután egyszer belépett a felhasználó, nem lenne szerencsés, ha minden egyes alkalommal, amikor új kérést küld a szerver felé, újra be kellene írnia a felhasználónevét és jelszavát, ezért bejelentkezés után a technológiáknál taglalt JSON Web Tokenek segítségével fogjuk azonosítani a bejövő kéréseket. Ezeket szintén a UserRepository állítja elő, és elhelyezi benne a felhasználó azonosítóján kívül a nevét és email címét is, így a tokennek a

dekódolása is elegendő ahhoz, hogy minden fontos információt megkapjunk a felhasználóról, és ne kelljen az adatbázishoz nyúlni. A digitális aláírás pedig biztosítja, hogy a tokenbe ágyazott adatok tartalmát biztosan mi írtuk és más nem változtatta meg.

A REST API-n elérhető egyes végpontokat a szintén minden sémához külön kontrollerek szolgáljak ki. Ezek is függőséginjektálással kapják meg a szükséges repository-t, és a különböző attribútumokkal ellátott függvényeik szolgálják ki az adott végponthoz beérkező kéréseket. Nagy meglepetés itt sincs, a végpontra érkező Get igével lekérdezhetjük az összes dokumentumot, ha az URL-ben mögé odatesszük az azonosítót is, akkor csak egy bizonyosat (ha az létezik) stb. Egyes kontrollerek rendelkeznek bonyolultabb logikát megvalósító végpontokkal, ilyen például a regisztráló és bejelentkező függvények a felhasználók esetében, vagy a munkaterületekhez hozzárendelt felhasználók változtatása. Ezeket azonban szinte minden esetben továbbküldi a kontroller a megfelelő repository-nak, az üzleti logika már ott van megvalósítva, viszont, ha ezen a szinten hiba történik, azt a kontroller kezeli le és értesíti a felhasználót, hogy valami baj történt (404 státuszkóddal, ha nem létezik a lekért elem vagy 400 Bad Request, ha olyan feladatot akarnánk elvinni egy oszlopból, ami igazából nincs is rajta stb.).

A kontrollerek és repository-k implementálása során mindig a C# által támogatott aszinkron paradigmát követtem, azaz minden függvény szignatúrájában szerepel az async kulcsszó és egy Task<T> megfelelően parametrizált sablonnal tér vissza, amelyet meghíváskor az await kulcsszóval kell bevárni. Szerencsére a MongoDB Driver is támogatja ezt a paradigmát, úgyhogy az egész alkalmazásban egységesen tudtam használni.

A frontend oldal felépítése ennél egyszerűbb, a konfigurációs és fordítás részt a Create React App megoldja, így lényegében csak a weboldal felépítésével kell foglalkozni. Ezt a részt négy különböző részre szedtem.

1. **API**: ebben a mappában a különböző végpontokhoz csatoló modulok találhatóak. Ezek az előző fejezetben bemutatott axios könyvtárt használják a http kérések lebonyolításához, és az egyes függvényei az adott kontrollerek különböző végpontjaihoz intéznek lekéréseket és adják vissza a kapott adatokat, illetve, minden fájl elején Typescript interfész formájában található az adott végpont sémája (Issue, User stb.).

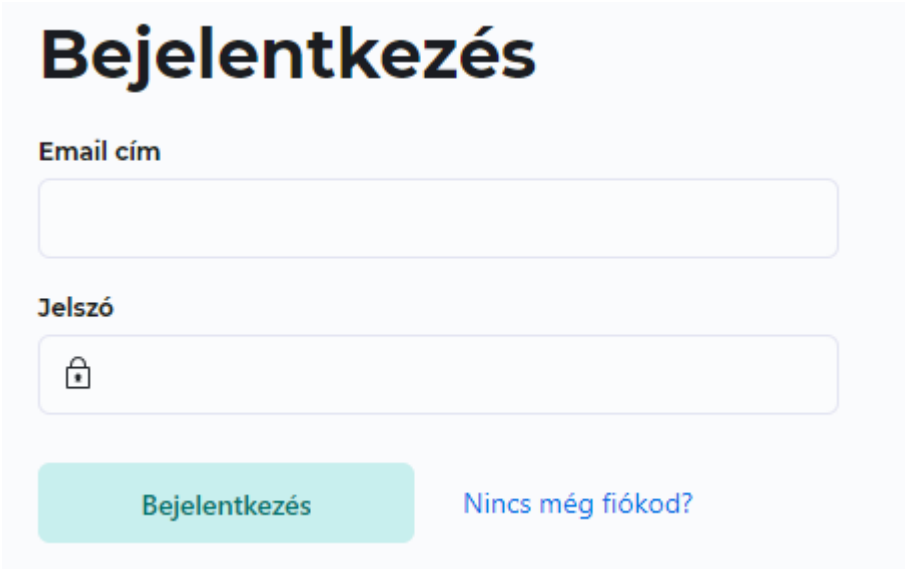
2. **Components**: itt található a kliensalkalmazás érdemi része, a kezelőfelületet leíró komponensek. A fő nézetekért felelősek a gyökérkönyvtárban találhatók, míg a felugró ablakokért, a naptár részeiért, illetve a swim lane nézetet felépítő egyes részkomponensek. Ezeknek működését a következő fejezetben részletezem.

3. **Store**: itt a Redux raktár konfigurációja és implementációja található a Redux Toolkit könyvtár segítségével. Minden típushoz külön fájlban található egy-egy slice, azaz objektum redukáló (reducer) függvényekkel – ezek tárolják az egyes típussal kapcsolatos állapotot (feladatok listája, címkék listája stb.) és tesz elérhetővé reducereket, amelyekkel különböző akciókat lehet küldeni az alkalmazás bármely részéről. Esetemben ezeknek túlnyomó többsége API hívást jelent, amelyet asyncThunk-ok segítségével oldottam meg – ezek az aszinkron függvények a kapott paramétereket átadják az API könyvtárból beimportált megfelelő modul megfelelő függvényének és visszaadják az eredményt, amelyet egy ún. extraReducer kezel le az adott slice-on belül, és a megfelelő módon változtat a tárolt állapoton. Egyetlen kivétel a felhasználókat kezelő reducer, amely csak az adott, bejelentkezett felhasználó adatait tárolja, illetve felelős a bejelentkezés és kijelentkezés során a kapott token eltárolásáért (törléséért), és ezt beállítani az axioson.

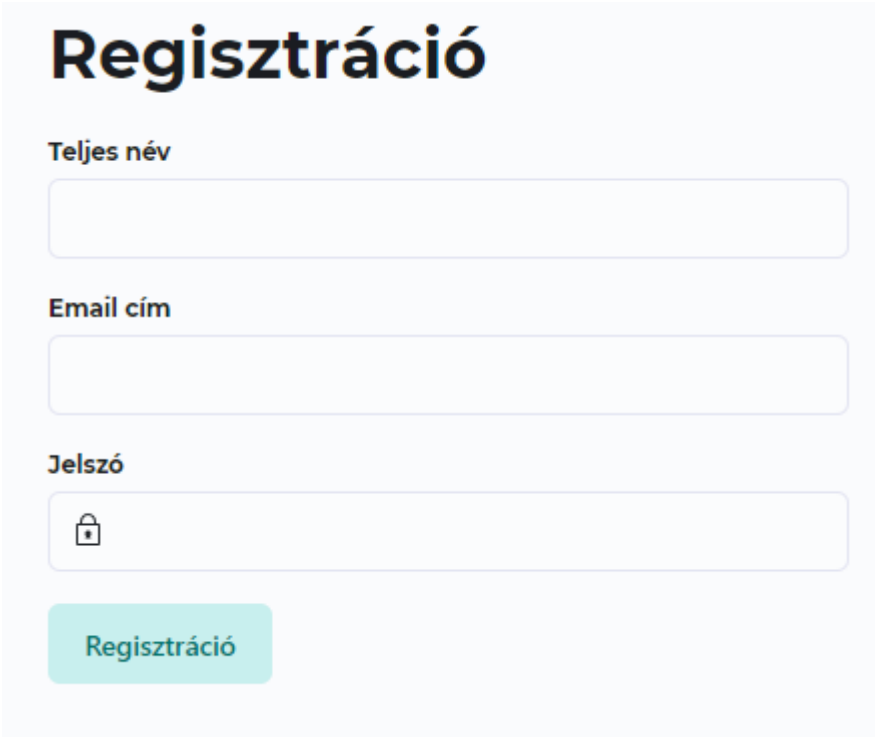
4. **Styles**: ide a CSS modulok fájljai kerültek, amelyben az extra stílusokat írtam le. Bár az alkalmazás nagy részében az Elastic UI beépített stílusait használtam, egyes komponensek megvalósításához szükségem volt extra stílusok leírására, és ezeket modulok formájában szerveztem, amelyeket, mint bármilyen Javascript modult lehet importálni, és a rajta levő tulajdonságokkal lehet az egyes osztályok nevét elérni.

3. Funkciók

Az alkalmazás elindításakor a bejelentkezési oldalra érkezünk.

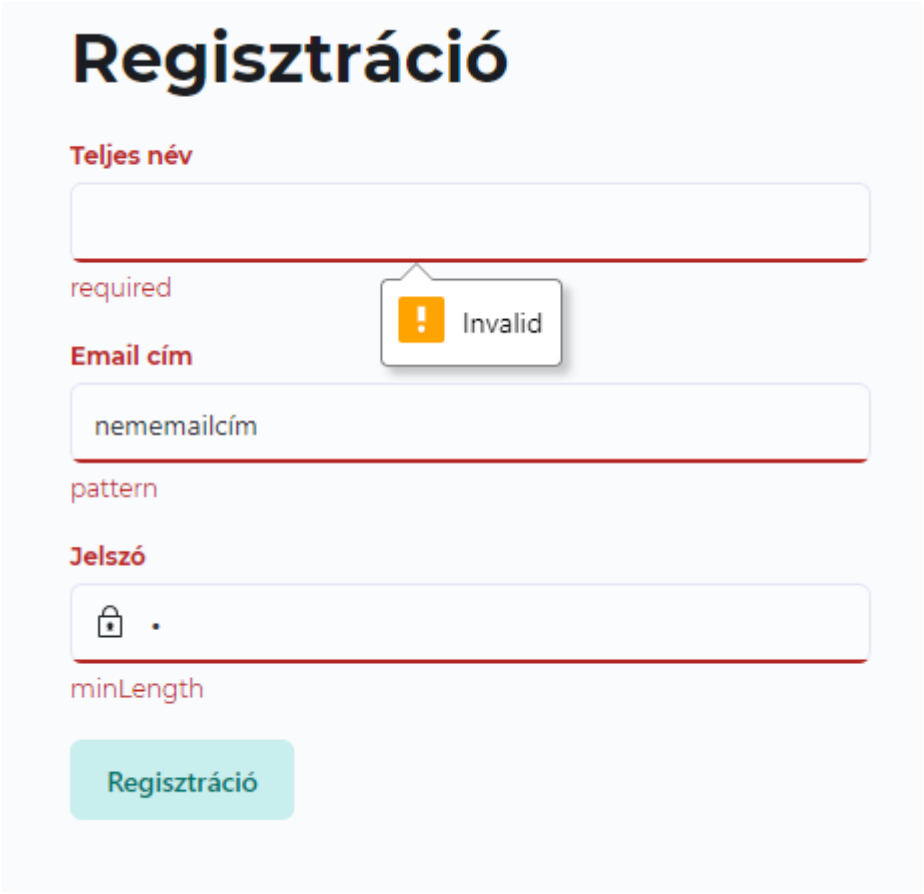


Itt lehetőség van megadni a felhasználónevet és jelszót, illetve, ha valaki még nem rendelkezik fiókkal, át ugorhat a regisztrációs oldalra is.

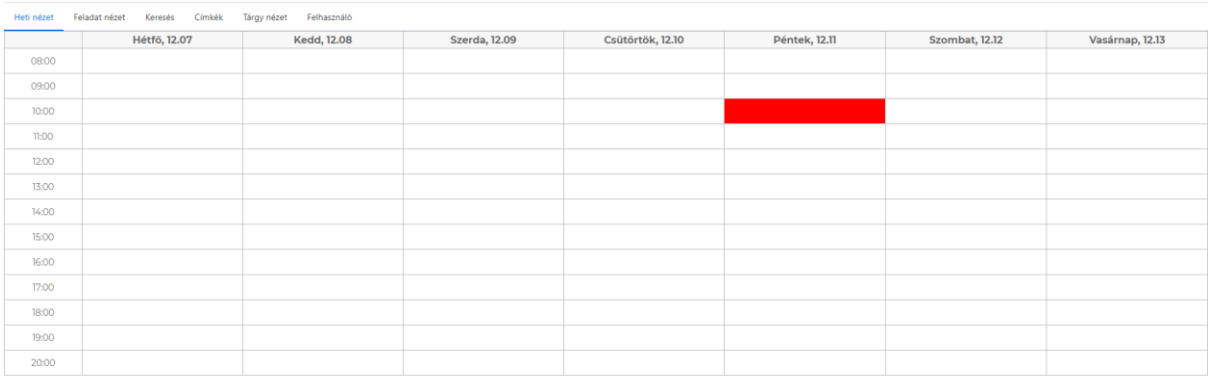


Itt bizonyos validációs követelményeknek meg kell felelni, azaz a név nem lehet üres, az email címnek megfelelő formájúnak kell lennie, és a jelszónak legalább öt karakterből kell

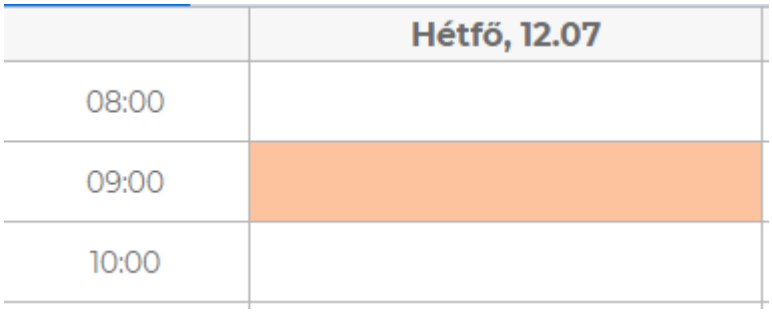
állnia. Ha az űrlap kitöltése helytelen, az egyes bemenetek pirossal kerülnek aláhúzásra, illetve alattuk megjelenik a hiba oka.



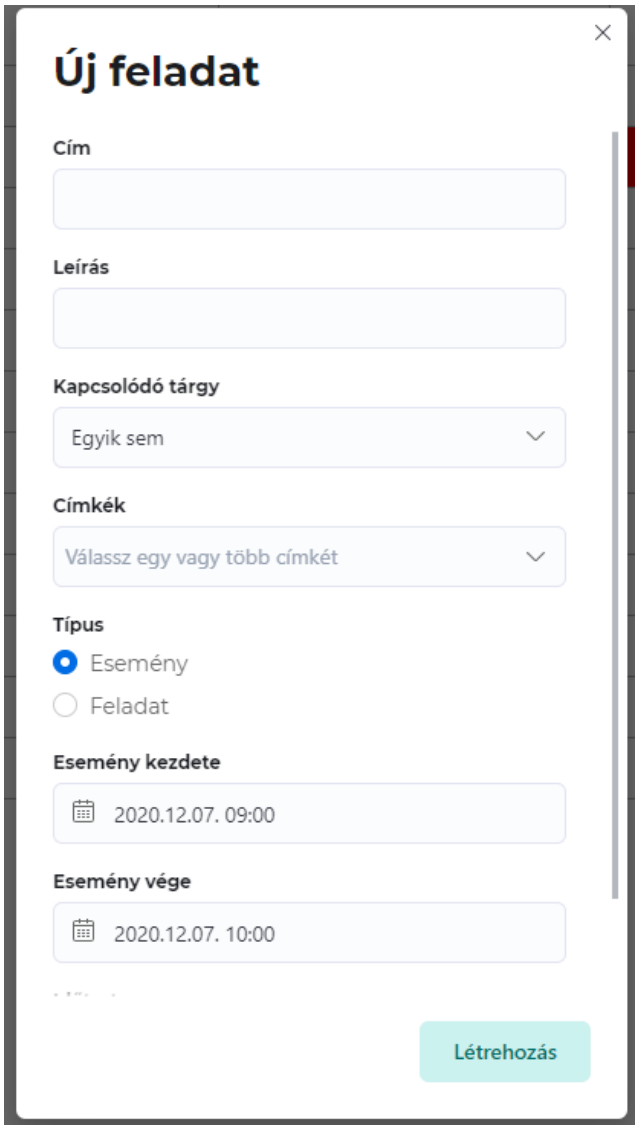
Bejelentkezés után a főoldalra kerülünk, ahol a fenti sorban található opciókkal tudunk navigálni a különböző nézetek közül. Az első ezek közül a naptár nézet.



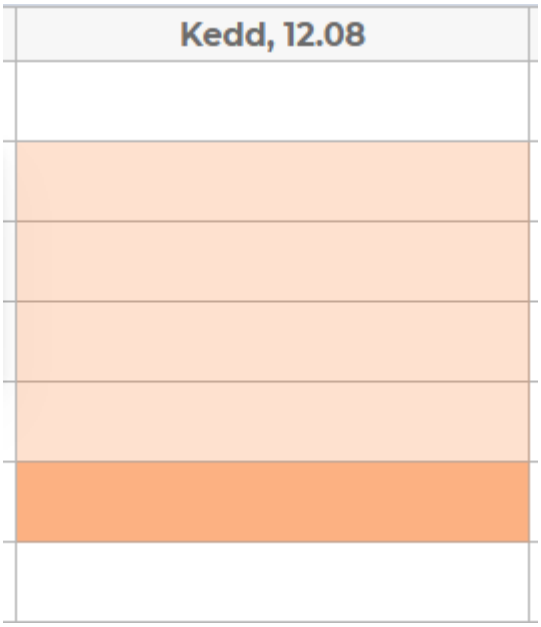
A naptárban az adott hét napjai találhatóak, és reggel nyolctól este nyolcig minden órához egy-egy cella. Az aktuális óra piros színnel van kiemelve. Ha rávisszük az egeret az egyik cellára, hívogató narancssárga kiemelést látunk, ami invitál új esemény felvételére.



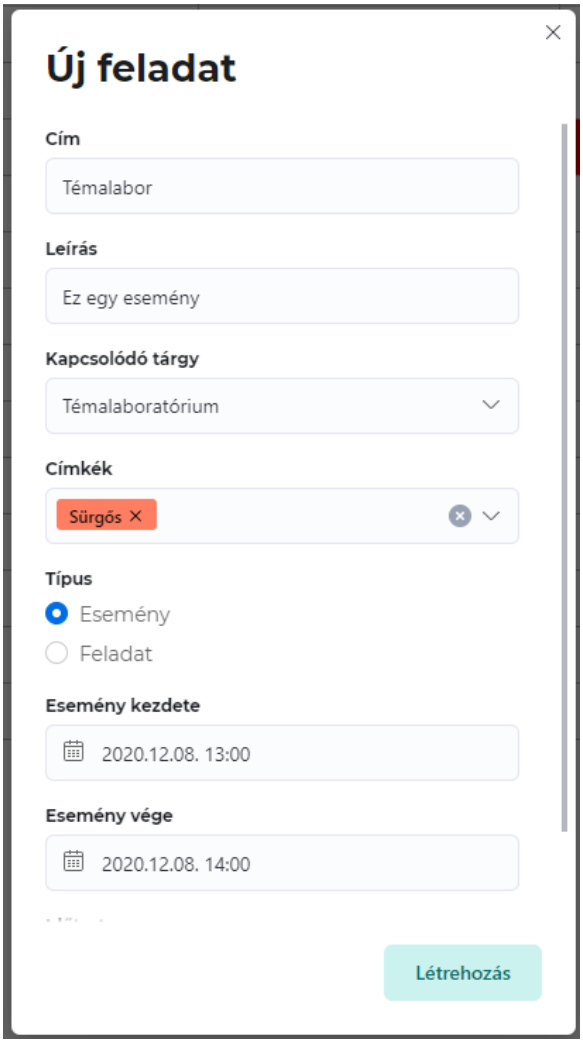
Klikkelés után egy felugró űrlapon adhatjuk meg az adott esemény adatait, természetesen a kezdetének és végének időpontja már ki van töltve.



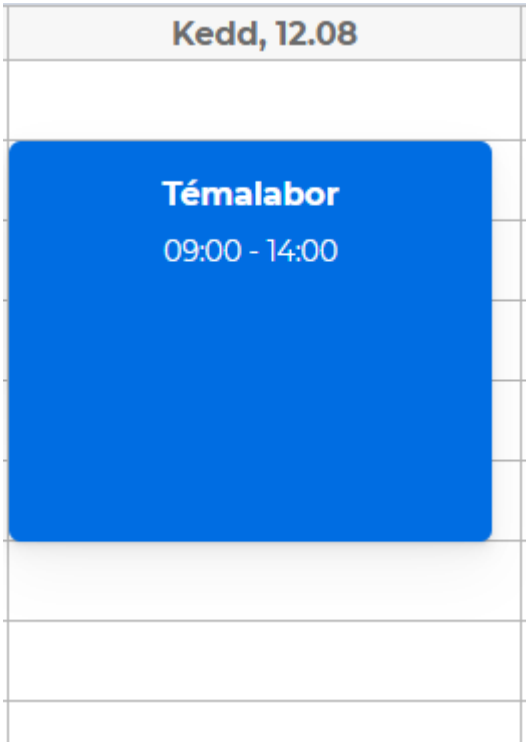
Az egeret lenyomva tartva nemcsak egy órát, hanem egy hosszabb időtartamot és kijelölhetünk.



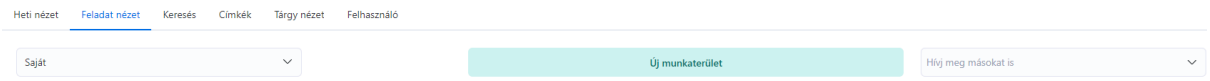
Az űrlapot kitölthetjük tetszőleges adatokkal, ezek közül csak a cím kötelező, a leírás, kapcsolódó tárgy és címke opcionálisak.



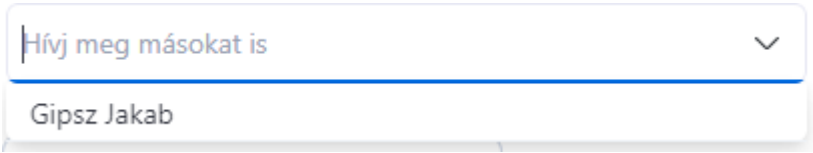
Ezután már az esemény meg is jelent a naptárban.

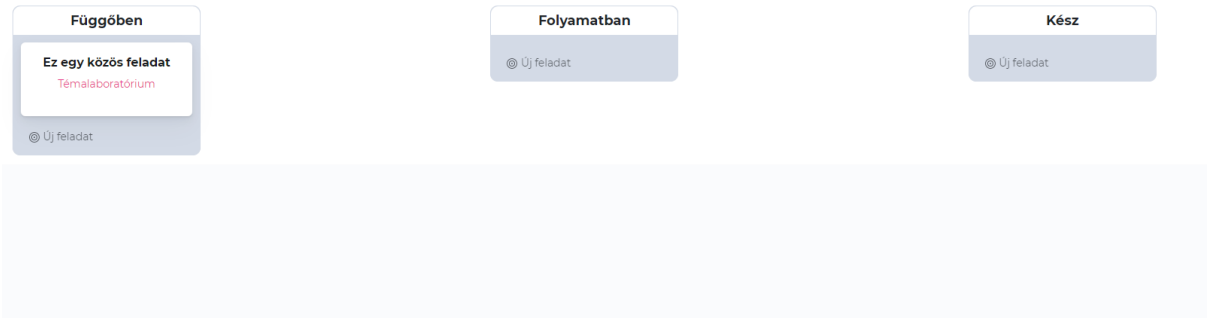


A következő oldal a feladat nézet.

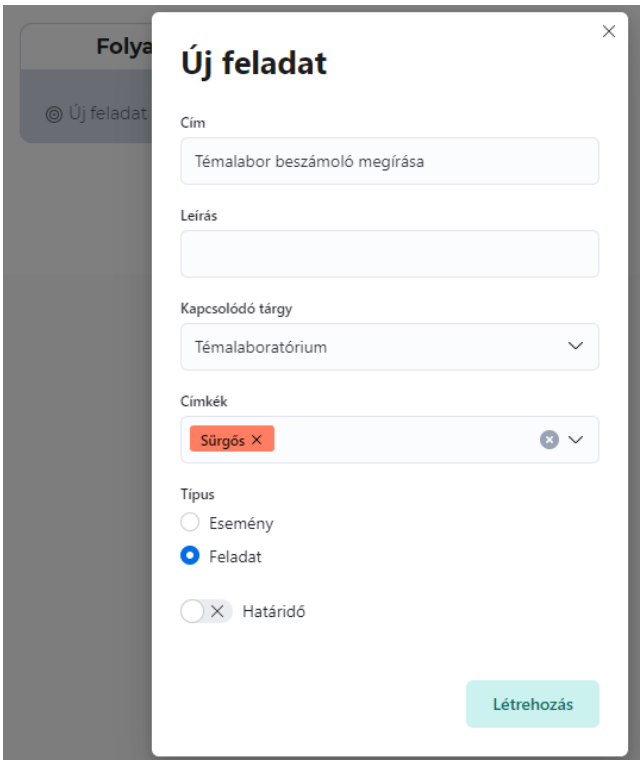


Az oldal tetején ki lehet választani a munkaterületet. A munkaterület egy, csak a feladat nézetre érvényes gyűjtemény, amelyben például különböző projektekhez lehet vezetni a feladatokat. Egy felhasználó bármennyi munkaterületet létrehozhat, és meghívhat más felhasználókat kollaborálni. A jobb oldalon található dobozba lehet felvenni más felhasználókat, akik ezután automatikusan hozzáférnek az adott munkaterülethez.





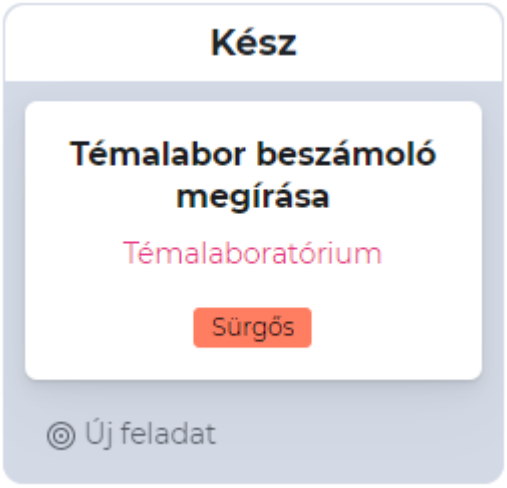
A munkaterületen swim lane-ek találhatók, amiken a feladatok állapotát lehet követni. Bármelyiken az Új feladat gombra kattintva a már ismerős felugró űrlap jön elő. Ha kapcsolódó tárgyat is megadunk, ennek neve a hozzá tartozó színnel kiemelve fog megjelenni a feladat neve alatt. A címkék hasonló módon választhatóak egy listából, egyszerre több is megadható, és szintén megjelennek a feladaton, a tárgy alatt (ha van).



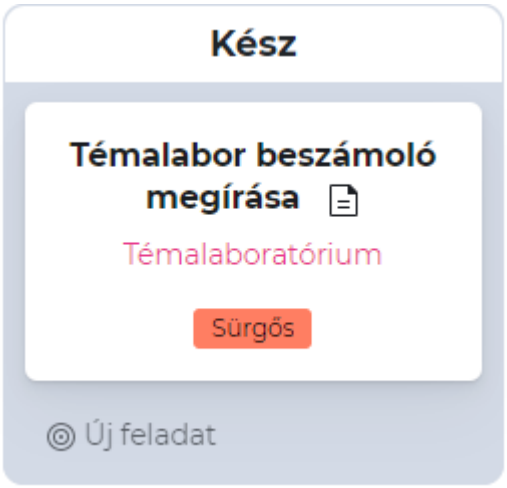
Létrehozás után az adott oszlopon jelenik meg a feladat.

A feladatok a bal egérgomb lenyomva tartásával húzhatóak át az egyik oszlopból a másikba.

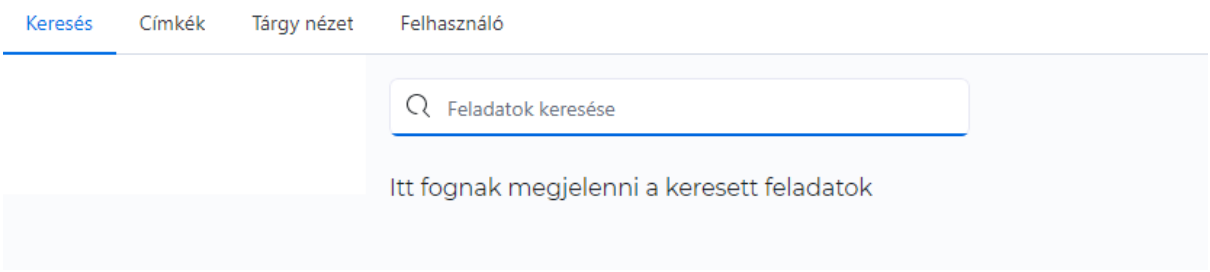
A gomb elengedése után a feladat átkerül a másik oszlopra.



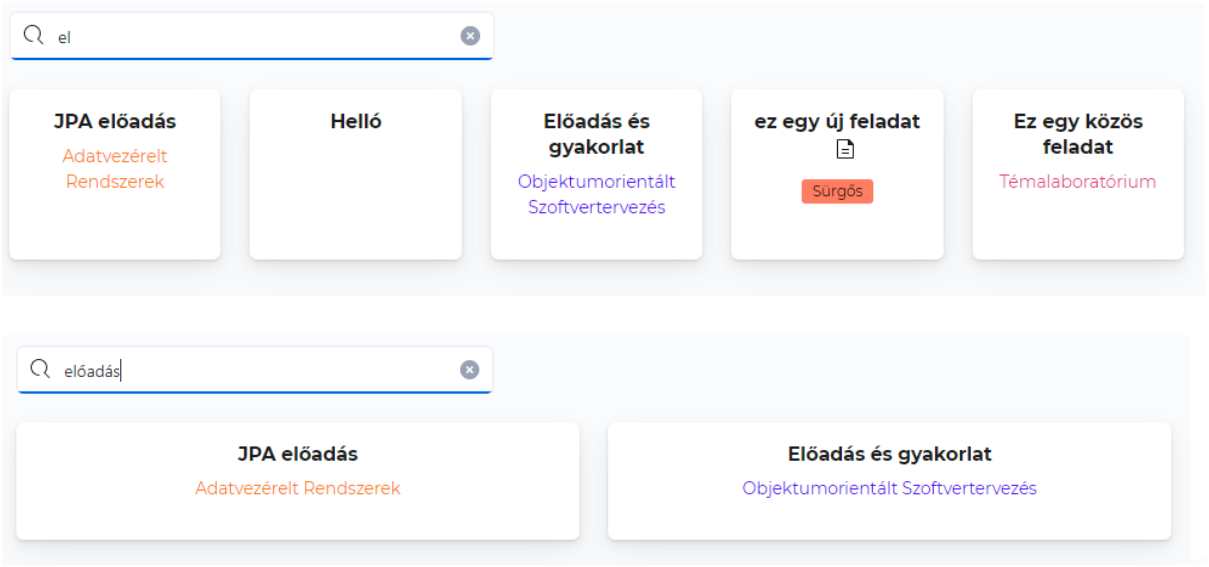
Ha megadtunk egy leírást is a feladatnak, megjelenik egy kis ikon ezt jelezve.



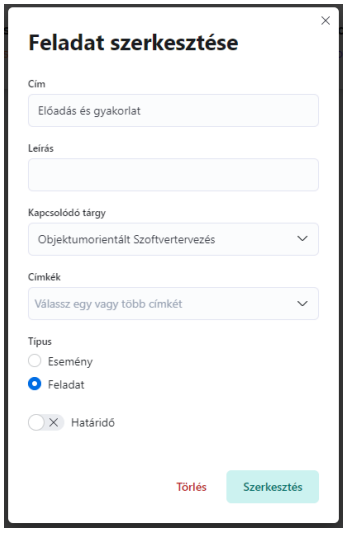
A következő oldalon az összes feladat között tudunk keresni.



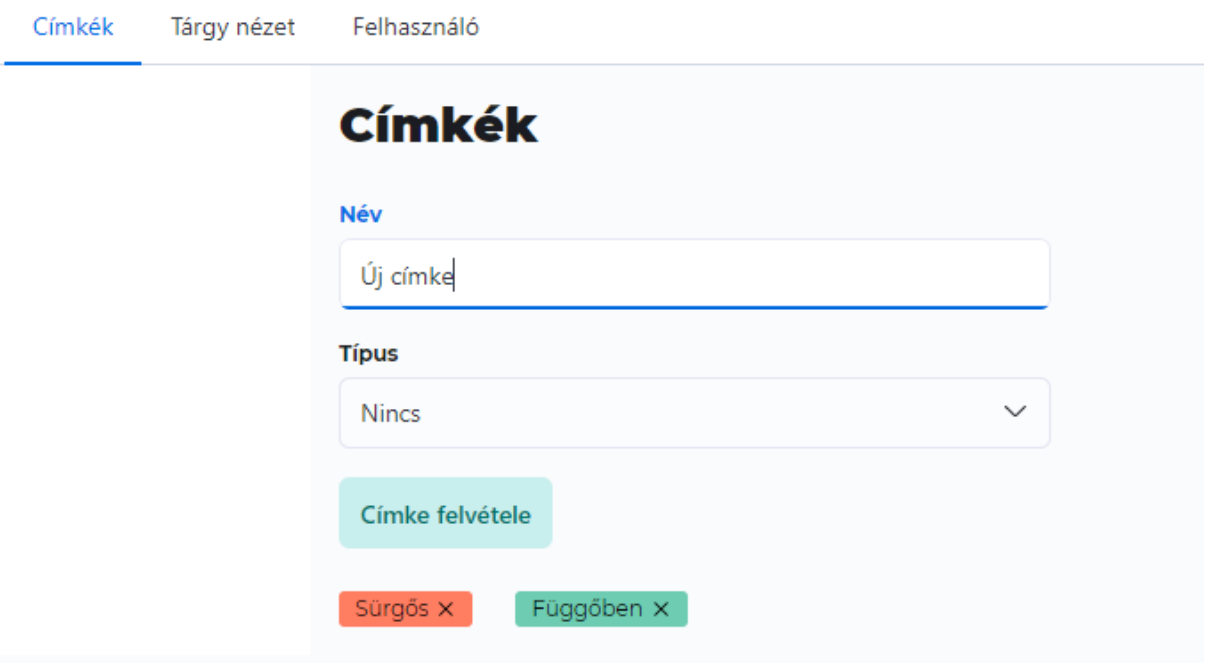
A frissítés folyamatos, azaz minden beírt karakterrel automatikusan indul a keresés és láthatóak a találatok.



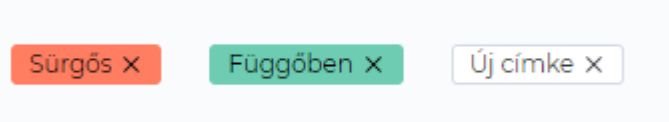
A feladatok természetesen innen is szerkeszthetőek.

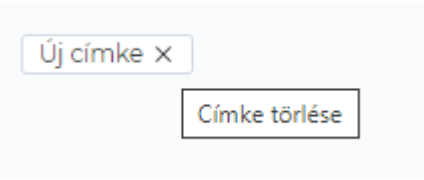


A következő nézet a címkéké. Itt láthatjuk az összes korábban felvett címkét, illetve vehetünk fel újabbakat a nevükkel és opcionálisan típusukkal (állapot, prioritás, stb.).

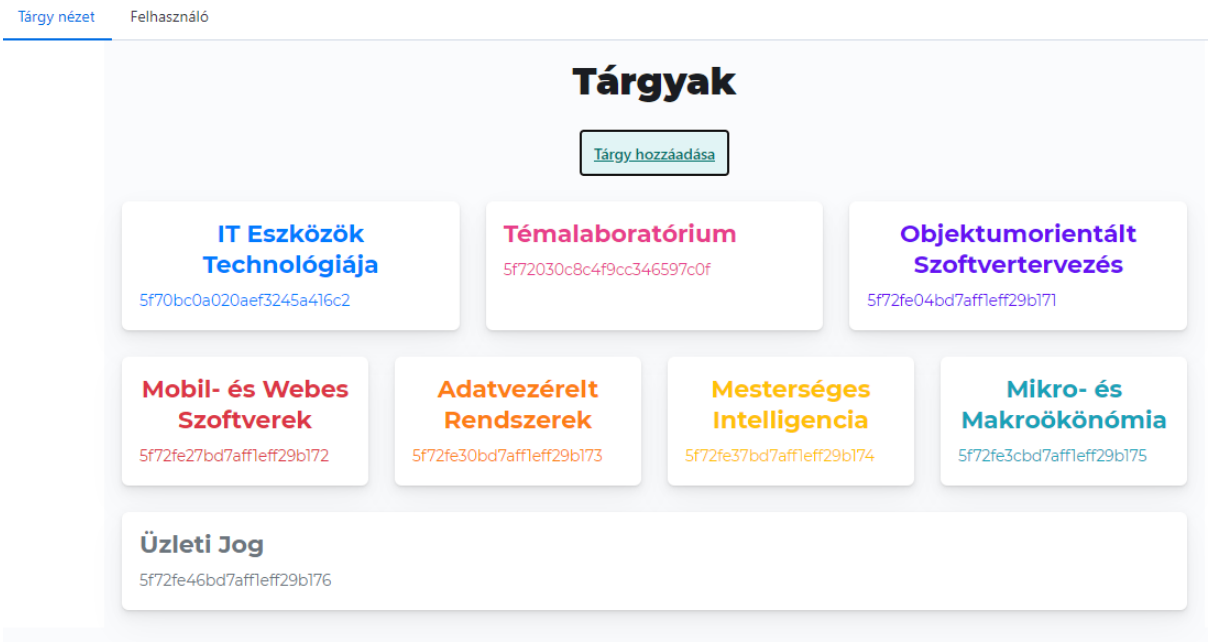


A címkéket pedig az X-re kattintva törölhetjük.

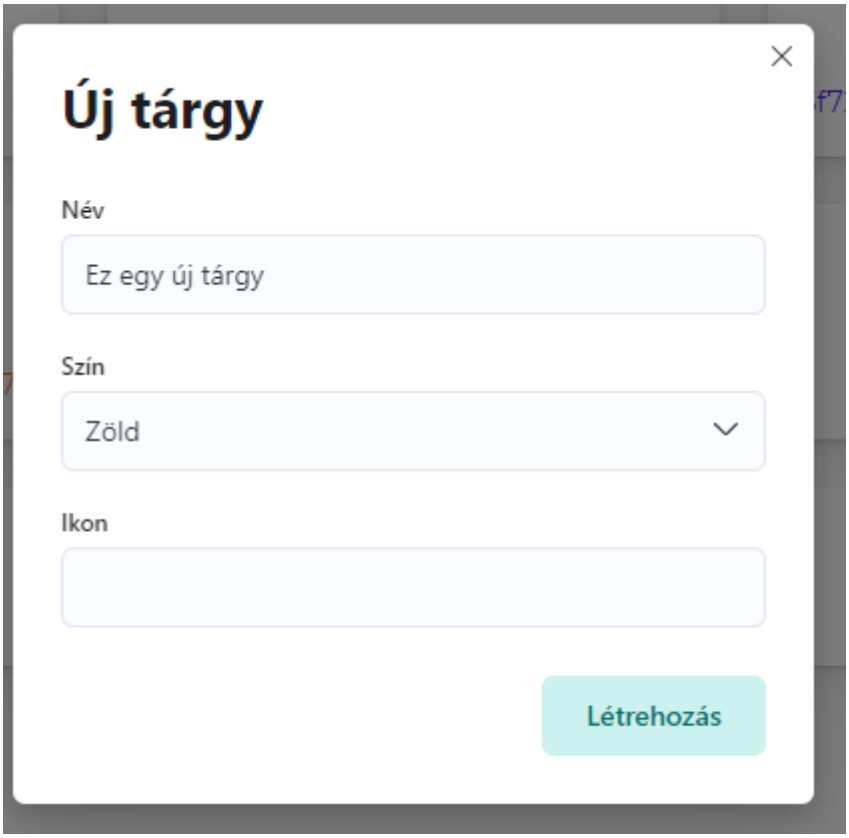




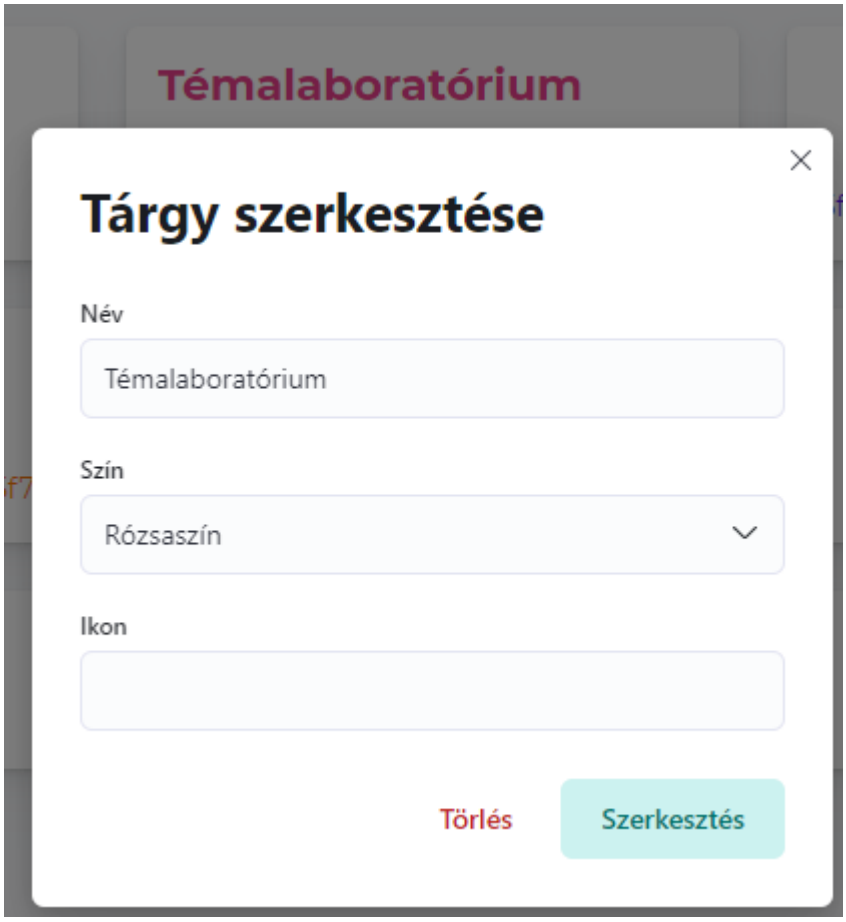
A tárgy nézeten az egyes feladatokhoz hozzárendelhető tárgyak találhatóak.



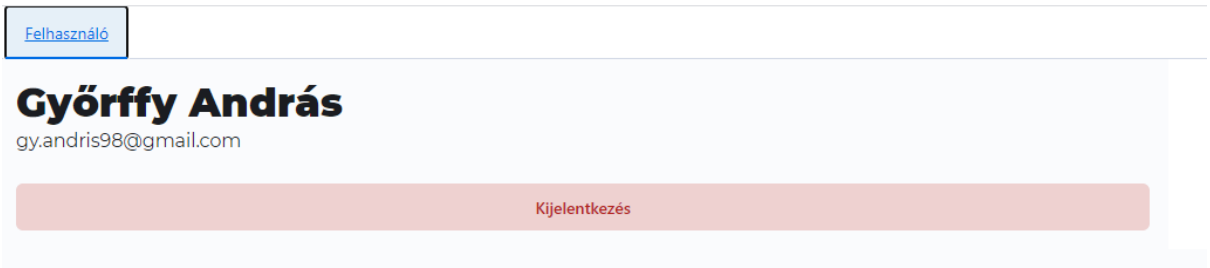
Az tárgy hozzáadására kattintva pedig egy felugró űrlap segítségével hozhatunk létre új tárgyat, szín hozzáadásával pedig könnyen megkülönböztethetővé tehetjük őket.



Hasonló módon van lehetőség a tárgyak szerkesztésére is.



Végül a felhasználó fülön a belépett felhasználóval kapcsolatos információkat láthatjuk, azaz nevet, email címet, illetve egy kijelentkezés gombot, ami a felhasználót kilépteti és átirányítja a bejelentkezés oldalra.



4. Limitációk

Az alkalmazás készítése során leginkább új technikák, technológiák megismerésére koncentráltam, nem feltétlenül a tökéletes, vagy mindenre kitérő működésre. A következőkben felsorolok néhány olyan hiányosságot, amelyeket egy nagyobb lélegzetű projektben biztosan kijavítottam volna.

A naptár nézeten nem lehet navigálni az egyes hetek között, mindig csak az adott hét látható. Az itt látható eseményeket nem lehet szerkeszteni.

A feladat nézeten a munkaterületek behozása miatt architekturális probléma alakult ki a Reduxban, ami abból fakad, hogy az egyes feladatok, oszlopok és munkaterületek bár külön reducerhez tartoznak, redundáns módon egymást tárolják, ezért a keletkezett változások nem feltétlenül futnak át megfelelően, például a szerkesztések eredménye nem látható azonnal, csak az oldal újratöltésével. A munkaterülethez rendelt más felhasználók nem tudják az egymás által készített feladatokat kezelni, csak azt, amik ők hoztak létre (jogosultság probléma), illetve a változásokat sem látják, csak újratöltés után.

A címke nézeten található űrlap nem állítja vissza magát új címke felvétele után.

A tárgy nézeten, bár az űrlapon és az adatbázisban szerepel az ikon lehetőség, ezek végül sehol nem jelennek meg.