



Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Hálózati Rendszerek és Szolgáltatások Tanszék

Leltározási feladatokat segítő, grafikus felülettel rendelkező adatbázis tervezése és fejlesztése

SZAKDOLGOZAT

Készítette
Király Bálint Martin

Konzulens
Schulcz Róbert

2020. december 9.

Tartalomjegyzék

Kivonat	i
Abstract	ii
1. Bevezetés	1
2. Feladat specifikáció	2
2.1. Funkcionális követelmények	2
2.1.1. Regisztráció	2
2.1.2. Bejelentkezés	2
2.1.3. Kijelentkezés	2
2.1.4. Raktár épületek kezelése	2
2.1.5. Tárolók kezelése	2
2.1.6. Eszközök kezelése	3
2.1.7. Raktár térképes nézettel	3
2.1.8. Tároló térképes nézettel	3
2.1.9. Kereshetőség	3
2.1.10. Naplózás	3
2.2. Nem funkcionális követelmények	3
2.2.1. Webes párhuzamos működés	3
2.2.2. Üzemeltetéssel szemben támasztott követelmények	4
3. Wireframe-ek	5
3.1. Bejelentkezés és regisztráció	5
3.2. Keresés	5
3.3. Raktár nézet	5
3.4. Tároló nézet	6
4. Választott technológiák	8
4.1. Frontend	8
4.1.1. Angular	8
4.1.2. React	8
4.1.3. Vue	8
4.1.4. Trendek	9
4.1.5. Konklúzió	9
4.2. Backend	10
4.2.1. NodeJS	10
4.3. Kommunikációs megoldások	10
4.3.1. REST API	11
4.3.2. GraphQL	11
4.3.2.1. Query	11

4.3.2.2.	Mutation	11
4.3.2.3.	Subscription	11
4.3.3.	Konkluzió	11
4.4.	Adatbázis	12
4.4.1.	Sequelize	12
4.4.2.	TypeORM	12
4.4.3.	Mongoose	12
4.4.4.	Prisma	12
4.4.5.	Konkluzió	12
4.5.	Közös technológiák	13
4.5.1.	TypeScript	13
5.	Architektúra	14
5.1.	Adatbázis séma	14
5.2.	Backend felépítése	14
5.3.	Frontend felépítése	16
5.4.	Architektúra összefoglalása	17
6.	Fejlesztést segítő eszközök	18
6.1.	Continuous Integration	18
6.1.1.	Statikus kódellenőrzés	18
6.1.1.1.	Lintér	18
6.1.1.2.	Sercurity check	19
6.2.	Git hook	19
6.3.	Continuous Delivery	20
6.3.1.	Heroku	20
6.3.2.	Vercel	20
7.	Alkalmazás fejlesztése és működése	21
7.1.	Backend	21
7.1.1.	GraphQL Playground	21
7.1.2.	Modellek és kapcsolatok	21
7.1.3.	Resolver felépítése	22
7.1.4.	Autentikáció és autorizáció	23
7.1.5.	Képek kezelése	24
7.2.	Frontend	24
7.2.1.	Útvonalválasztás	24
7.2.2.	Kódgenerálás	26
7.2.3.	Validáció	26
7.2.4.	Felhasználói felület	27
8.	Tesztelés	30
8.1.	Frontend tesztelés	30
8.2.	Backend tesztelés	31
9.	Üzemeltetés	32
9.1.	Docker	32
10.	Összefoglalás	34
10.1.	Tovább fejlesztési lehetőségek	34
10.1.1.	Keresés	34
10.1.2.	Egyedi tulajdonságok kezelése	34

10.1.3. 3D-s térkép nézet	35
Ábrák és táblázatok jegyzéke	36
Irodalomjegyzék	36

HALLGATÓI NYILATKOZAT

Alulírott *Király Bálint Martin*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot 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 autentikált felhasználók számára) közzétegye. 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.

Budapest, 2020. december 9.

Király Bálint Martin
hallgató

Kivonat

Rengeteg raktárkezelő alkalmazás érhető el a piacon, azonban a térképes nézet egy ritka funkciónak számít ezekben a rendszerekben. Az erre alkalmas szoftverek rendszerint előredefiniált térképpel dolgoznak. A szakdolgozatom célja egy olyan raktár kezelő rendszer tervezése és fejlesztése, amely lehetővé teszi az eszközök pozíciójának pontos meghatározását a raktáron belül, mindezt dinamikusan. A rendszer segítségével a felhasználók képesek a raktár méretét és elrendezését is meghatározni, így a pontos igényeknek megfelelő alkalmazást kaphatnak.

Abstract

There is a great deal of warehouse management applications available in the market, however, the map view is a rare feature in these systems. The warehouse management software usually wordks with a predefined maps. The aim of my dissertation is to design and develop a warehouse management system that allows to determine the exact position of items within the warehouse. With the help of the system, users can also set the size and layout of the warehouse, so they can get the application that suits their exact needs.

1. fejezet

Bevezetés

A szakdolgozat során a feladatom egy olyan grafikus felülettel ellátott leltár rendszer tervezése és fejlesztése volt, melynek segítségével az alapvető leltári funkciókon felül könnyedén behatárolhatjuk a leltárba vett eszközök pontos helyzetét a raktárunkon belül egy grafikus “térkép” segítségével.

A dolgozatom során első körben egy részletes specifikációt készítettem, melyben taglaltam az alkalmazással szemben támasztott funkcionális és nem-funkcionális követelményeket. Ezután megterveztem a webalkalmazás felhasználói felületét, egyszerű wireframe-ek segítségével.

A specifikáció és a wireframe-ek elkészítése után kiválasztottam a használni kívánt technológiákat és megterveztem az alkalmazás architekturális felépítését.

Az alkalmazás két fő részből áll, frontend és backend. Utóbbi tartalmazza az üzleti logikát és az adatbázis kommunikációt, míg a frontend az adatok lekéréséért és megjelenítéséért felel, valamint a felhasználói interakciók által eljuttatja a módosításokat a backend részére. Ezek fejlesztését párhuzamosan végeztem. Minden egyes funkciónak először elkészítettem a backend oldali implementációját, majd hozzáálltam a frontend oldali kód fejlesztésének. Sok esetben szükséges volt a backend módosításra a frontend fejlesztése közben is.

Az alkalmazás tervezésén és fejlesztésén kívül az üzemeltetés előkészítését is elvégeztem, valamint bevezettem olyan megoldásokat, melyek a fejlesztés minőségét segítették elő.

Végül a tesztelésre fektettem a hangsúlyt, amely közben a felmerülő hibák javítását eszközöltem az alkalmazásban.

A dolgozat felépítése is ennek megfelelően készült el, hogy az olvasó számára is könnyedén követhető legyen a tervezés és fejlesztés folyamata valamint a köztük lévő összefüggés.

2. fejezet

Feladat specifikáció

A fejezet kitér az alkalmazással szemben támasztott funkcionális és nem funkcionális követelményekre.

2.1. Funkcionális követelmények

2.1.1. Regisztráció

Az elkészítendő alkalmazásban legyen lehetőség felhasználói fiókot létrehozni egy regisztrációs oldal segítségével. A regisztráció során a felhasználó nevét, email címét, jelszavát valamint a jelszavának megerősítését kérjük. További fontos követelmény, hogy egy email címhez csak egy felhasználó tartozhat, így az alkalmazásnak szükséges elvégezni ezt az ellenőrzést is.

2.1.2. Bejelentkezés

A rendszer csak autentikált felhasználók számára legyen elérhető. A regisztráció során megadott email cím és jelszó segítségével a látogatónak képesnek kell lennie autentikálnia magát, ezáltal hozzáférni az alkalmazásban tárolt eszközökhöz. Nem autentikált felhasználóknak csak a bejelentkezés és a regisztráció opciókat kínáljuk fel.

2.1.3. Kijelentkezés

A bejelentkezett felhasználónak biztosítsunk lehetőséget a munkamenet megszüntetésére, annak folytatásához csak újbóli bejelentkezéssel legyen lehetősége.

2.1.4. Raktár épületek kezelése

Az alkalmazással szemben követelmény, hogy képesnek kell lennie több raktár (raktár épület) kezelésére. Ez alatt értjük a raktár létrehozását, szerkesztését, valamint az ezekhez tartozó jogosultságok menedzselését. A raktárról tároljuk a méreteit, a nevét és természetesen a szerkesztésre jogosult felhasználók listáját.

2.1.5. Tárolók kezelése

Minden raktárba tárolók helyezhetők. A tárolókat a nevükkel, méretükkel és a raktáron belüli pozíciójukkal együtt rögzíthetjük. Amennyiben a felhasználó rendelkezik a megfelelő jogosultsággal az adott raktáron belül, lehetősége van a tárolók szerkesztésre, létrehozására és törlésére.

2.1.6. Eszközök kezelése

A hierarchia harmadik szintjén helyezkednek el az eszközök. Minden eszköz rendelkezzen az alábbi tulajdonságokkal. Név, amely az egyszerű azonosítást és a kereshetőséget biztosítja. Érték, ami az eszköz raktárba vétele kori értékét tartalmazza. Mérete és a tárolón belüli pozíciója. Minden egyes leltárba vett eszközhöz legyen lehetőségünk a kiadások rögzítésére. Minden kiadáshoz egy összeg és egy leírás tartozik, amely a kiadás okának és mértékének magyarázatára szolgál.

2.1.7. Raktár térképes nézettel

A raktárakban a tárolók elhelyezkedését jelenítsük meg egy felülnézeti, térképes nézet formájában is. A tárolók mozgathatóságát nem szükséges megvalósítani ezen a térképen. Ennek oka, hogy míg az eszközök pozíciója gyakran változik a tárolók fixen telepítve vannak. Ennek a funkciónak a nem implementálása felesleges félreértések elkerülését is szolgálja.

2.1.8. Tároló térképes nézettel

Minden tároló oldalán jelenítsünk meg egy képet a tároló tartalmával. A tárolt eszközöket egyszerű téglalappal reprezentáljuk. Fontos követelmény, hogy a ezen a nézeten legyen lehetőség az eszközök mozgatására is. A mozgatást a felhasználó a mozgatni kívánt elemre kattintva, majd az egér bal billentyűjét lenyomva tartva mozgathatja a tárolón belül. A fent leírt módszer segítségével legyen lehetőség egy ideiglenes tárolóba rakni. Ezt az ideiglenes tárolót jelenítsük meg minden (az adott raktárban lévő) tároló oldalán, az eszközök tárolók közötti mozgatását megvalósítva.

2.1.9. Kereshetőség

Az alkalmazásban legyen lehetőség a leltárba vett eszközök közötti keresésre. Fontos, hogy a keresés segítségével ne csak az adott elemet kapjuk meg hanem annak az elhelyezkedését is könnyedén le tudja kérdezni a felhasználó.

2.1.10. Naplózás

Egy alkalmazásnál általában fontos követelmény, hogy képesek legyünk nyomon követni az adatbázisba történt változások okait, különösen igaz ez egy raktár rendszer esetén. A rendszer naplózzon minden olyan eszközökkel kapcsolatos felhasználói interakciót, amely adatbázis művelethez vezet. Tehát az eszköz létrehozását, szerkesztését és törlését, a megleltetések naplózása nem szükséges.

2.2. Nem funkcionális követelmények

Az alkalmazással szemben természetesen nem csak funkcionális követelményeket támasztunk. A nem funkcionális követelmények figyelembevétele is legalább annyira fontosak egy alkalmazás tervezésénél és fejlesztésénél, mint a funkcionális követelmények.

2.2.1. Webes párhuzamos működés

Az elkészülő programmal szemben támasztott követelmények közé tartozik az egyidejűleg több felhasználó kiszolgálása webböngésző segítségével. Erre legyen lehetőség az összes modern böngészőből, azonban elegendő a böngészők legfrissebb változatának a támogatása.

2.2.2. Üzemeltetéssel szemben támasztott követelmények

Továbbá fontos követelmény, hogy az alkalmazás üzemeltetéséhez ne legyen szükség speciális hardware-re vagy speciális operációs rendszerre. Elindításához elegendőnek kell lennie egy átlagos személyi számítógép hardware készlete. A fejlesztést és üzemeltetést tegyük lehetővé Linux, MacOS és Windows operációs rendszereken is.

3. fejezet

Wireframe-ek

A wireframe-ek egy alkalmazás tervezésénél nagyon gyakran elmaradnak, pedig igenis fontos szerepük van. Rengeteg olyan dologra világíthatnak rá, amire egyébként nem gondolnánk és segíti a kommunikációt a fejlesztő(k) és a megrendelő között.

A dolgozatba csak a főbb képernyők wireframe-ét helyeztem el. Ezeknek az elkészítéséhez a Figma¹. névre keresztelt webes alkalmazást használtam. Ennek segítségével az egyszerű wireframe-ektől kezdve komplex prototipizált design terveket is készíthetünk.

3.1. Bejelentkezés és regisztráció

A bejelentkezés és regisztráció oldalak (3.1. ábra) felépítése azonos, egyedül a beviteli mezőkben térnek el. A navigációs sávban csak a bejelentkezés és a regisztráció opciók közül választhatunk. A képernyő közepén mind a két esetben egy formot jelenítünk meg a szükséges adatokat bekérő beviteli mezőkkel.

3.2. Keresés

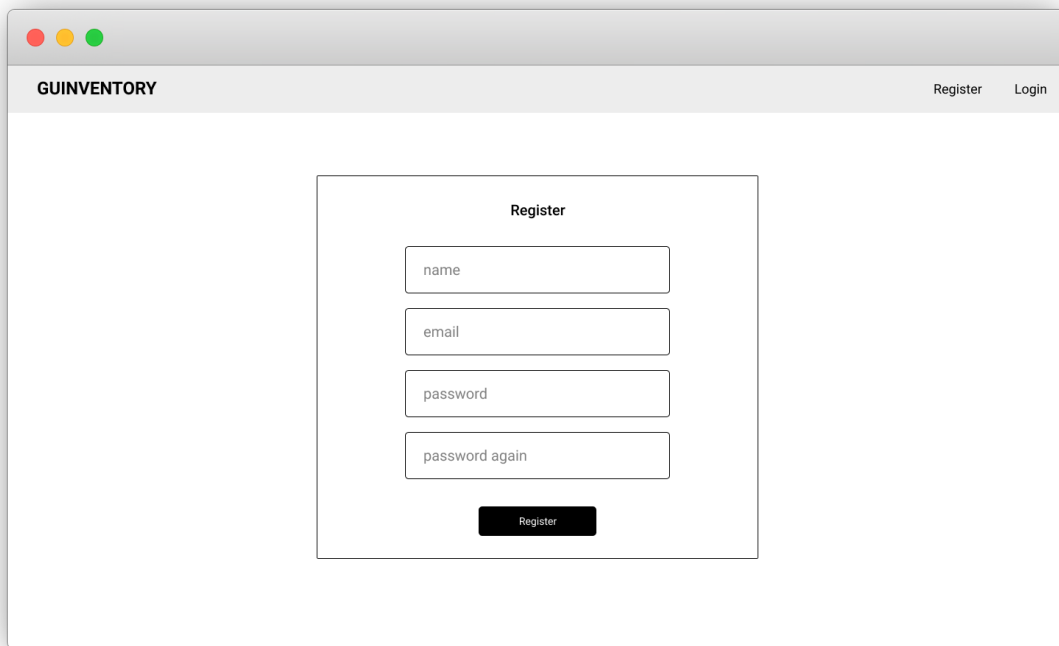
A keresést, annak érdekében, hogy az alkalmazás bármely részéről könnyedén elérhető legyen a felső navigációs sávba helyeztem el. A wireframe (3.2. ábra) alapján látszik, hogy kereséskor egy legördülő listában jelennek meg az eredmények, így az aktuális oldal elhagyása nélkül láthatjuk a keresett eszköz helyét.

3.3. Raktár nézet

A raktár oldalán (3.3. ábra) láthatunk egy térképes nézetet és egy listát is a tárolókról. A térképes nézeten a kurzort a tároló fülé mozgatva megjelenítjük annak nevét a könnyebb azonosítás érdekében.

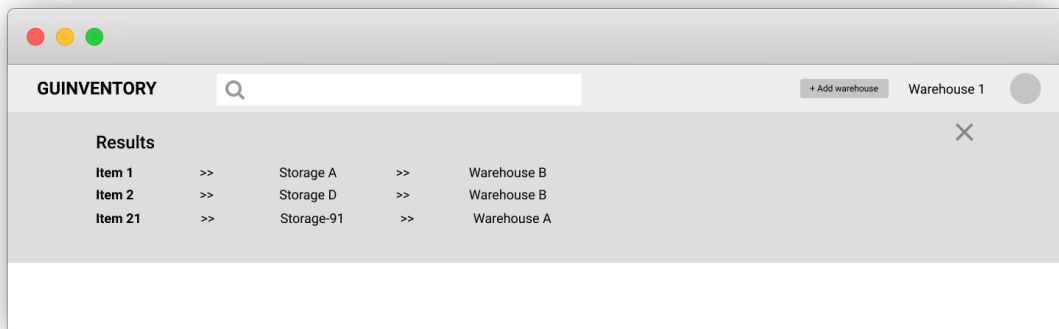
Ezen felül a navigációs sávban láthatjuk az éppen kiválasztott raktárat, ahol egy legördülő menü segítségével azonnal választhatunk másik raktárat is, amennyiben több raktárhoz is van hozzáférésünk. A raktár választó mellett megjelenítünk egy gombot, amellyel új tárolót hozhatunk létre.

¹A Figma hivatalos weboldala: <https://figma.com/>



The wireframe shows a web browser window titled 'GUINVENTORY'. In the top right corner, there are links for 'Register' and 'Login'. The main content area features a centered 'Register' form. This form contains four input fields stacked vertically, labeled 'name', 'email', 'password', and 'password again'. Below these fields is a black 'Register' button.

3.1. ábra. Regisztráció wireframe



The wireframe shows a web browser window titled 'GUINVENTORY'. It includes a search bar with a magnifying glass icon. To the right of the search bar are buttons for '+ Add warehouse' and 'Warehouse 1', followed by a close button (X). Below the search bar, a 'Results' section displays a table of search results. The table has four columns: item name, a separator ' >> ', storage location, another separator ' >> ', and warehouse name. The results are as follows:

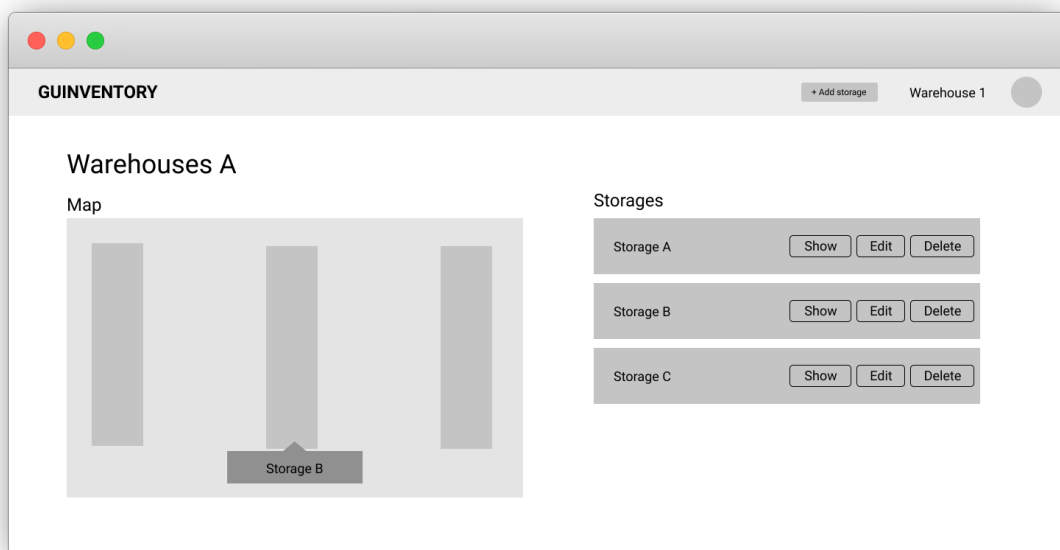
Results				
Item 1	>>	Storage A	>>	Warehouse B
Item 2	>>	Storage D	>>	Warehouse B
Item 21	>>	Storage-91	>>	Warehouse A

3.2. ábra. Keresés wireframe

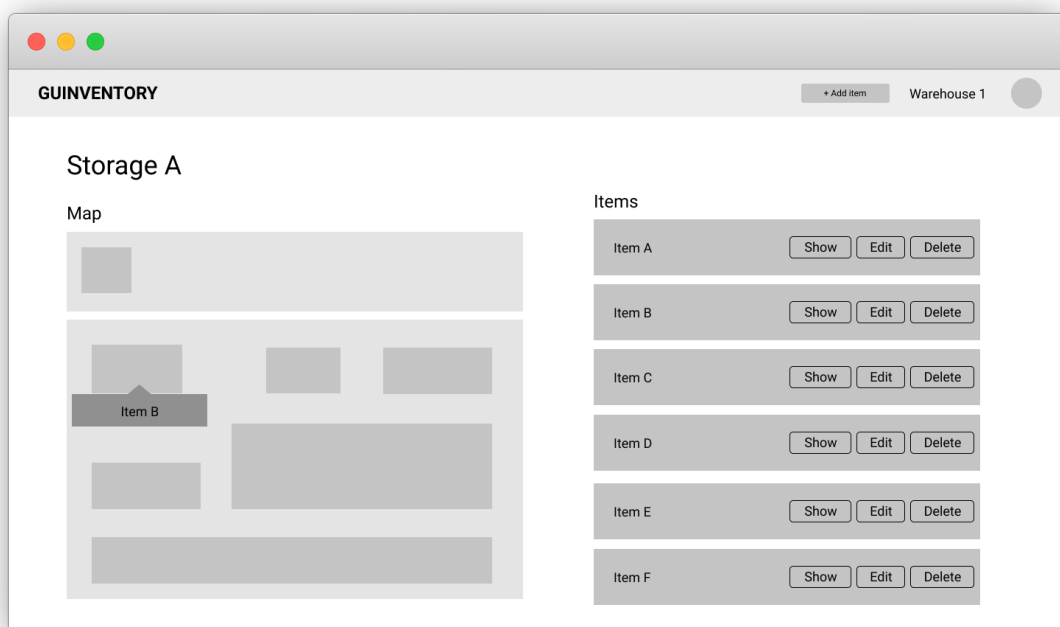
3.4. Tároló nézet

A tároló nézete (3.4. ábra) nagyon hasonlít a raktáréhoz, azonban itt kiegészítésként a térkép felett megjelenítünk egy másik tárolót is. Ez a feladatspecifikációban megkövetelt tárolók közötti eszköz mozgatását teszi lehetővé.

A navigációs sávban itt is megjelenítjük a raktár választó gombot, azonban a tároló létrehozása helyett, itt az eszköz felvétele gombot találhatjuk.



3.3. ábra. Raktár wireframe



3.4. ábra. Tároló wireframe

4. fejezet

Választott technológiák

Ennek a fejezetnek a keretein belül a felhasznált technológiák kiválasztásának szempontjait és folyamatát mutatom be. Először a frontend technológiát választottam ki ugyanis az alkalmazásnak ez a része a bonyolultabb a raktárak térképes nézetének kezelése miatt.

4.1. Frontend

A piacon jelenleg 3 meghatározó keretrendszer/könyvtár érhető el, melyek segítségével webes alkalmazások felhasználói felületét készíthetjük el.

Ezek az Angular, a React és a Vue. Bár mind a három keretrendszer célja ugyanaz, mégis nagy eltéréseket tapasztalhatunk a kódbázisban, felépítésben és a fejlesztők filozófiájában.

4.1.1. Angular

Az Angular a Google által 2010-ben elindított keretrendszer, fejlesztésbe és karbantartásába azóta rengeteg a másik nagy cég is csatlakozott[1]. Filozófiája, hogy egy általános felhasználásra felkészített alapot ad a fejlesztők kezébe. Tartalmazza a formok validációját, az állapot kezelést, a routing-ot, a felhasználói input-ok kezelését és ezeken felül még rengeteg más olyan dolgot is, ami hasznos lehet egy webalkalmazás fejlesztéséhez.

4.1.2. React

Az Angular-hoz hasonlóan a React mögött is egy nagy cég áll. A Facebook 2013 óta fejleszti és tartja karban a keretrendszert[6]. Az Angular-ral szemben a React, filozófiája szerint csak egy könnyű súlyú keretet ad. Emiatt talán nem is nevezhetjük keretrendszernek, sokkal inkább csak egy könyvtár. Azonban ennek és a fejlesztők által implementált virtuális DOM-nak köszönhetően sokkal jobb sebesség érhető el vele. Természetesen az, hogy csak egy keretet ad nem okoz semmilyen hátrányt. Ugyanis rengeteg hivatalos csomag érhető el hozzá, melyek megvalósítják az Angular által is nyújtott megoldásokat.

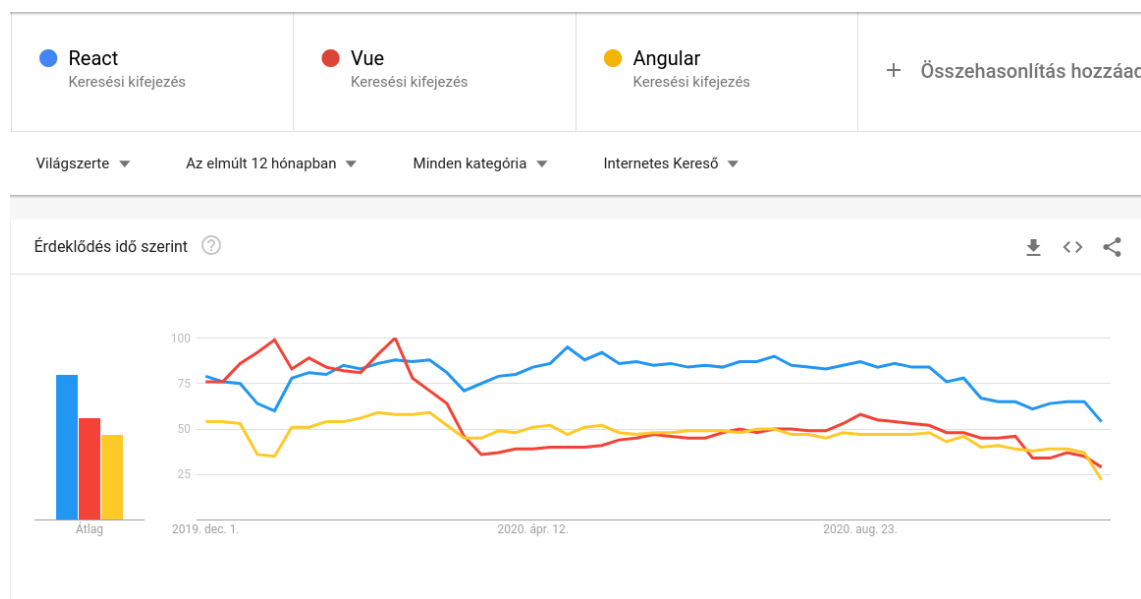
4.1.3. Vue

A három keretrendszer közül a legújabb és legkevesebb fejlesztői erőforrással rendelkező opció. A fejlesztését 2014-ben kezdte a Google egyik korábbi mérnöke, aki a mai napig meghatározó szerepet tölt be a projektben[8]. Filozófiáját tekintve a korábban tárgyalt két rendszer között helyezhető el. Többet tartalmaz, mint a React, de korán sem annyit, mint az Angular. Itt is találkozhatunk a virtuális DOM-mal, melynek köszönhetően nagyon

gyors a működése. Az összehasonlított keretrendszerek közül ezzel a legkönnyebb elkezdni a fejlesztést, egy egyszerű alkalmazás elkészítése nem igényel sok ismeretet a HTML, CSS és JavaScripten kívül. Természetesen komplex alkalmazásokat is készíthetünk vele és ipari környezetben is remekül megállja a helyét, azonban ilyen esetben komolyabb tervezésre van szükség, ugyanis teljesen szabad-kezet kapunk az alkalmazás felépítését illetően.

4.1.4. Trendek

A döntés meghozása előtt végeztem egy kisebb kutatást a napjainkban tapasztalható trendekről. Ehhez a Google Trends¹ és az NPM Trends² szolgáltatásait vettem igénybe. Előbbi segítségével a Google keresések számát tudjuk összehasonlítani, míg utóbbival az NPM³ csomagkezelő oldalról történő letöltések számát.



4.1. ábra. Google Trends – keresések összehasonlítása.

A Google keresések alapján a korábbi években az Angular egyértelműen uralta a piacot, azonban a vezető szerepet mára már átvette tőle a React, ahogy az a grafikonon (4.1. ábra) is látszik.

A Google keresések nem mutattak szignifikáns különbséget, azonban az NPM letöltések számában már jelentős eltéréseket tapasztalhatunk. A grafikonról (4.2. ábra) könnyedén leolvasható, hogy több, mint 4-szer annyi a letöltések száma a React esetén, mint a másik 2 keretrendszerénél.

4.1.5. Konklúzió

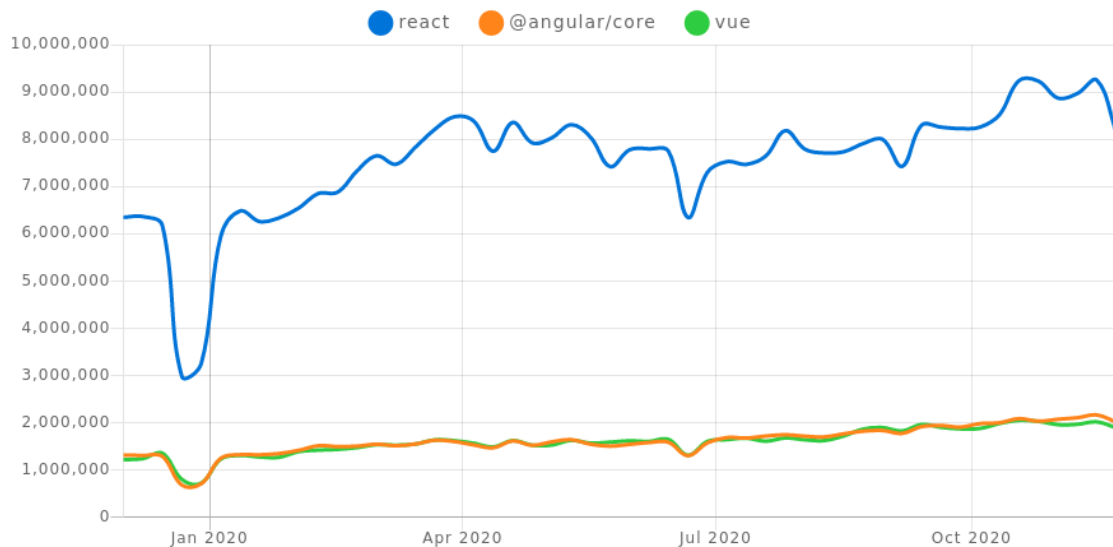
A fent felsoroltakat alapul véve végül a React-re esett a választásom. A virtuális DOM és az aktív fejlesztői közösség hatalmas előnyt jelent a fejlesztés és az üzemeltetése során is. A döntést továbbá segítette, hogy ezzel a keretrendszerrel rendelkezem tapasztalattal, korábbi fejlesztéseim során minden problémát remekül meg tudtam oldani a segítségével.

¹A Google Trends hivatalos weboldala <https://trends.google.com/>

²Az NPM Trends hivatalos weboldala <https://www.npmtrends.com/>

³Node Package Manager

Downloads in past 1 Year ▾



4.2. ábra. NPM Trends – letöltések összehasonlítása.

4.2. Backend

A kliens oldali technológia kiválasztása után a szerveroldali keretrendszer kiválasztására került sor. Itt sokkal több népszerű és az iparban is használt rendszer érhető el. Ilyen például a DotNet, a Spring, a Laravel, a Ruby on Rails, a Django és még sok más. A mennyiségük miatt ezeknek összehasonlítása nagyon nehéz, így megpróbáltam egyszerűsíteni a folyamatot. Az elsődleges szempont a kiválasztás során az, hogy a választott frontend technológiával a legegyszerűbben és a legjobban tudjon együtt működni. Természetesen a felsorolt és a nem felsorolt keretrendszerek is működnek React-tel probléma nélkül. Azonban egy kiemelkedik közülük azáltal, hogy a programozási nyelve azonos a React-ével. A NodeJS segítségével frontend és backend oldalon is használhatunk JavaScriptet. Az azonos programozási nyelv felveti a kódmegosztás lehetőségét is a két komponens között.

4.2.1. NodeJS

A NodeJS története több, mint 11 éve indult Ryan Dahl keze által. A projekt a Google által fejlesztett V8 JavaScript motor segítségével teszi lehetővé JavaScript futtatását web böngészőn kívül, így alkalmassá tették a nyelvet backend oldali felhasználásra is. A NodeJS gyors ütemben fejlődött, 2011-ben már a Microsoft is kivette a részét a fejlesztésből, napjainkra pedig az egyik legnépszerűbb technológia webes környezetben.[2]

4.3. Kommunikációs megoldások

A frontend és a backend technológiák kiválasztása után a következő lépés a köztük történő kommunikáció mikéntjének eldöntése volt. Itt szerencsémre sokkal kevesebb opció közül kellett választanom. Napjainkban két fő irányvonal figyelhető meg. Ezek a REST API és a GraphQL.

4.3.1. REST API

A REST feloldása REpresentational State Transfer, ami magyarra fordítva Reprezentatív Állapot Átvitelt jelent. Ez – ahogy a nevéből is következtetni lehet – próbálja kifejező módon átvinni az adatot a kliens és a szerver alkalmazások között. Ezt úgy valósítja meg, hogy ajánlást tesz a végpontok nevére és típusára rendeltetésük szerint.

Művelet angolul	Művelet magyarul	HTTP üzenet típusa	Végpont
Create	Létrehozás	POST	/users
Read	Megtekintés	GET	/users/:id
Update	Módosítás	POST	/users/:id
Delete	Törlés	DELETE	/users/:id
List	Listázás	GET	/users

4.3. táblázat. Példa egy entitáson végezhető műveletekre a REST API elvei szerint

4.3.2. GraphQL

A GraphQL egy lekérdező nyelv, amely a jelenleg elterjedt REST API-s megoldásokat próbálja leváltani/kiegészíteni. A megszokott REST API-val ellentétben GraphQL-nél csak egyetlen egy végpont létezik, valamint csak POST típusú HTTP kéréseket használunk.

Az összes kérést erre a végpontra küldjük a megfelelő tartalommal, melyet a POST kérés törzsében (body) helyezünk el.

A bevett REST API-s megoldással szembeni hatalmas előnye, hogy mindig azt kapjuk amit kérünk. A POST kérés törzsében elhelyezett GraphQL operation pontosan meghatározza, hogy milyen entitások milyen tulajdonságait szeretnénk visszakapni. Ez a GraphQL operation nagyon hasonlít a JSON formátumra, azonban egy-két dologban eltér attól. Lehetőségünk van több entitásból is adatot lekérni egyetlen kéréssel, így csökkentve a HTTP üzenetek számát.

A kéréseket minden esetben egy (vagy több) úgynevezett resolver szolgálja ki nekünk. A resolver-ekből 3 fő típust különböztetünk meg Query, Mutation és Subscription. A REST API elveihez hasonlóan a GraphQL esetén is kapunk ajánlásokat arra, hogy milyen adatbázis művelethez milyen típusu resolvert használjunk.

4.3.2.1. Query

Adatok lekérésére szolgál.

4.3.2.2. Mutation

Ahogy a nevéből is következtethetünk rá, főként adatok módosítására és létrehozására szolgál.

4.3.2.3. Subscription

A standard GraphQL implementáció tartalmazza a websocket kommunikációt is. A subscription-ök segítségével lehetősége van a kliensnek feliratkozni bizonyos eseményekre, melyek bekövetkeztéről azonnal értesül socket kapcsolaton keresztül.

4.3.3. Konkluzió

A fent leírt szempontokat figyelembe véve a végső választásomat a GraphQL mellett tettem le. Tanulmányaim során rengetegszer találkoztam REST API-t használó vagy

annak megvalósítását követelő feladattal, így ezen opció választása esetén nem mélyítettem volna el a tudásomat egy kevésbé ismert, azonban mégis remek technológiában.

4.4. Adatbázis

Az alkalmazás nem rendelkezik olyan követelménnyel, amely komoly adatbázis műveletet igényel. Ezért a választási szempont elsődlegesen az volt, hogy a már kiválasztott technológiákkal együtt a lehető legkényelmesebb és legjobb fejlesztési élményt nyújtsa. Ennek következménye az volt, hogy adatbázisok helyett ORM⁴ rendszerek összehasonlítását végeztem el. Az ORM egy absztrakciós réteget helyez az adatbázis és az alkalmazás közé, így elfedve a lekérdező nyelvet, ez nagyobb biztonságot és gyorsabb fejlesztést eredményez.

Az ORM-ek többsége rengeteg dologban hasonlít, így az összehasonlítás során leginkább a különbségekre fókuszáltam.

4.4.1. Sequelize

Több, mint 7 éve elérhető ORM rendszer, az ötös verziótól beépített TypeScript támogatással érkezik. A sequelize-cli segítségével generálhatunk modelleket és migrációkat is.[4]

4.4.2. TypeORM

Fejlesztése 5 éve kezdődött, azonban a mai napig sem érte el az 1.0-ás verziót. Beépített TypeScript támogatással rendelkezik és a Sequelize-hoz hasonlóan generálhatunk modelleket és migrációkat.[5]

4.4.3. Mongoose

A Mongoose ORM a mongoDB kezeléséhez több, mint 10 évvel ezelőtt létrehozott csomag. Mivel a mongoDB egy document alapú NoSQL adatbázis, így még fontosabb az adatok validációja.[3] A mongoose séma természetesen kezeli ezt, azonban a migrációk meglévő adatok esetén problémás és sok hibalehetőséget rejt. A TypeScript támogatás megoldható de szintén körülményes.

4.4.4. Prisma

2020 júniusában jelentették be a 2.0-ás verziót, amely az 1.0 alapoktól újraírt változata. Teljes körű TypeScript támogatás érkezik és rengeteg extra funkcióval. A migrációs generálása automatikus, így a séma módosítása után automatikusan generálható bármilyen fejlesztői beavatkozás nélkül. Természetesen, ha adatmigráció is szükséges, azt nekünk kell kezelniük. A Prisma egy beépített Studio nevű grafikus interface-szel érkezik, így teljes grafikus hozzáférést kapunk az adatbázishoz bármilyen külső megoldás nélkül.

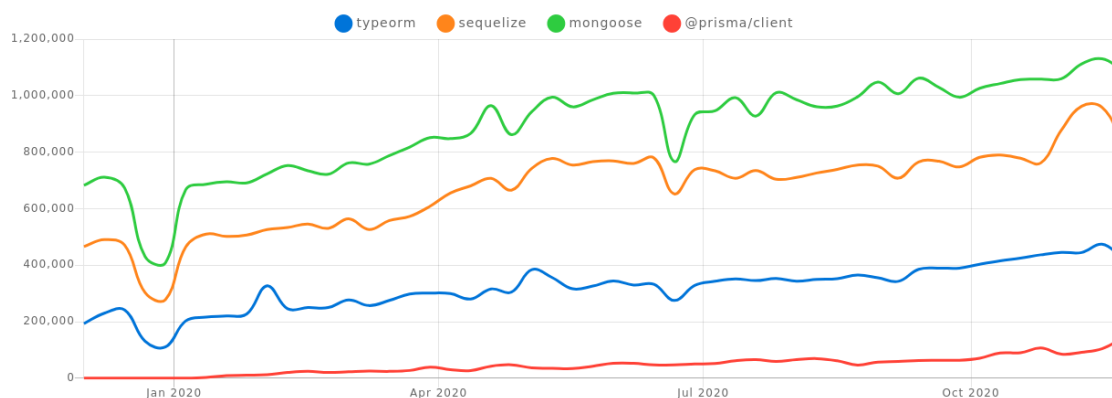
4.4.5. Konklúzió

Habár a trendek (4.4. ábra) alapján a Prisma jócskán el van maradva népszerűségben az összehasonlításban részt vett többi ORM-től, mégis erre esett a választásom. A korábban felsorolt előnyök és a növekvő népszerűség miatt úgy érzem, hogy a lemaradása egyedül az újdonságának köszönhető.

⁴Object-Relational mapping.

Tekintve, hogy fejlesztése még mindig egy korai stádiumban van az adatbázis kezelők listája szűkös. A fejlesztés kezdetekor csak PostgreSQL, MySQL és SQLite volt támogatott, utóbbi kisebb hiányosságokkal. A dolgozat írása közben bejelentették a MSSQL támogatást is. A választáskor a MySQL és PostgreSQL között kellett döntenem. Az adatbázis felépítése nem bonyolult, így személyes preferencia alapján a PostgreSQL mellett döntöttem.

Downloads in past 1 Year ▾



4.4. ábra. NPM Trends – ORM-ek letöltésének összehasonlítása.

4.5. Közös technológiák

Ahogy azt a korábbi fejezetben is említettem a NodeJS-nek köszönhetően a backend és a frontend oldali alkalmazás azonos nyelvet használ, a JavaScriptet. A JavaScript egyik nagy hátránya az, hogy gyengén típusos nyelv (természetesen ezt bizonyos esetekben tekinthetjük előnynek is). Ennek a megoldására JavaScript helyett TypeScriptet használtam az alkalmazás megvalósításához.

4.5.1. TypeScript

A TypeScript egy – a Microsoft által fejlesztett – nyílt forráskódú nyelv, amely JavaScriptet egészíti ki statikus típus definíciókkal. Mondhatjuk, hogy a JavaScript egy superset-je.

A típusok segítségével hamarabb észrevehetjük a hibákat az alkalmazásunkban. Azonban fontos megjegyezni, hogy a típusok definiálása opcionális, ezért TypeScript mellett érdemes valamilyen linter-t használni, amely figyelmezteti a programozót ha elmulasztja a típusdefiníciók használatát. Minden érvényes JavaScript kód egyben egy érvényes TypeScript kód is, ez részben az elhagyható típusdefiníciók miatt igaz.

Annak érdekében, hogy probléma nélkül futtathassuk a TypeScript kódunkat a böngészőkben minden kódot JavaScript-re transzformálunk. Erre több megoldás is létezik, ilyen például a Babel vagy a TypeScript compiler.[7]

A NodeJS-nek köszönhetően használhatjuk backend oldali nyelvként is, így a frontend és a backend közös nyelvet használhat, amely akár a kódmegosztás lehetőségét is felveti.

5. fejezet

Architektúra

Az alkalmazás 3 fő részre bontható frontend, backend valamint adatbázis. E három réteg együttesen felel azért, hogy a felhasználótól a böngészőn keresztül érkező interakciókat kezelje és az állapotot tárolja.

5.1. Adatbázis séma

Az adatbázis migrációját nem kellett manuálisan végrehajtanom a Prisma-nak köszönhetően. A Prisma – amellett, hogy kezeli a migrációkat – egy absztrakciós réteget ad az adatbázisunk és az alkalmazásunk közé olyan szinten, hogy teljesen elfedi az adatbázist a fejlesztő elől.

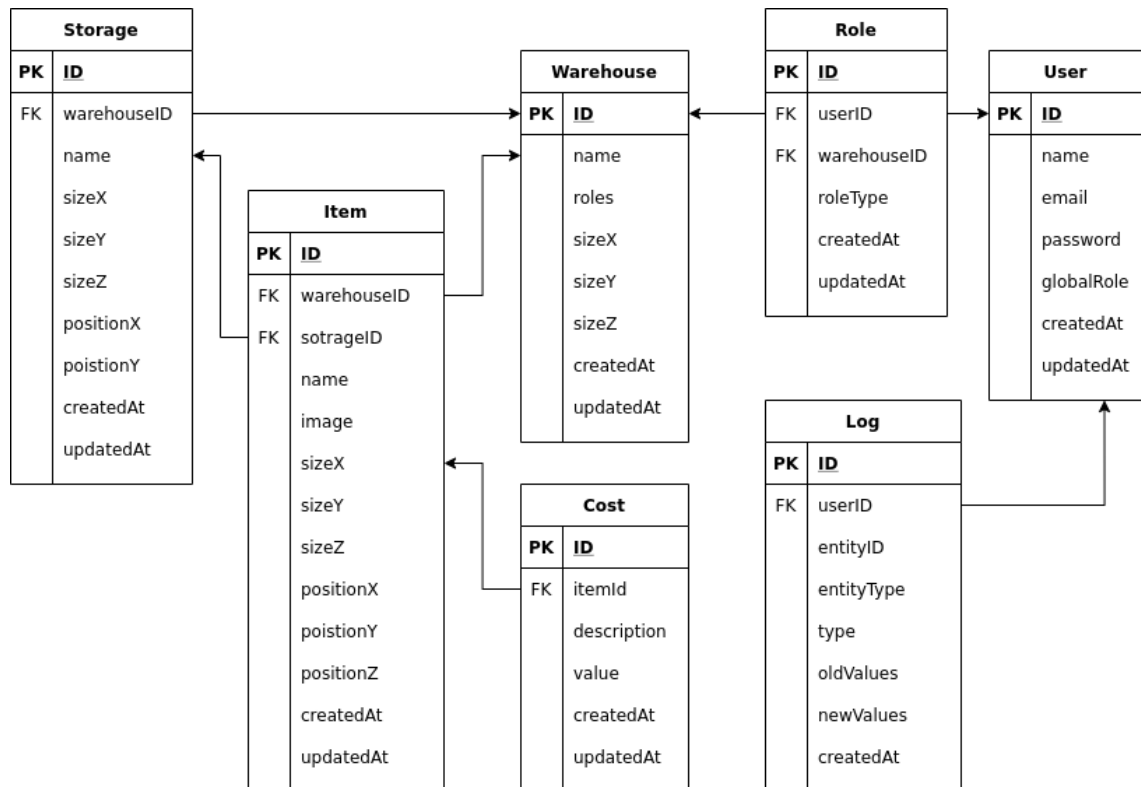
A tervezés során ennek ellenére mégis relációs adatbázist terveztem, majd ezt ültettem át a Prisma által megkövetelt sémába. Tanulmányaim során ezzel a tervezési metodikával találkoztam és alkalmazása rögzült, rutin szerűvé vált, így először nehéz volt más szemszögből megközelíteni a problémát.

Az adatbázis sémát a követelményeket figyelembe véve hoztam létre. Az ábrán (5.1. ábra) látható, hogy egy eszköz (Item) tartozhat tárolóhoz (Storage) és raktárhoz (Warehosue) is. Erre azért volt szükség, mert a követelmények között szerepelt az ideiglenes tároló fogalma, ami a tárolók közötti eszköz mozgatás megoldását szolgálja. Ennek megvalósítására új entitás bevezetése felesleges volt. Az opciók között szerepelt még, hogy a tárolók entításban egy kitüntetett tárolót használunk erre a célra, azonban ez rengeteg elágazást eredményezett volna a kódban. Így egyszerű tervezési döntés értelmében, ezek a mozgatás során nem tartoznak tárolóhoz csak a raktárhoz, amely ezáltal betölti az ideiglenes tároló szerepét.

A naplózás megvalósításánál törekedtem a generikus megoldás létrehozására. A napló (Log) tábla csak egy tényleges kapcsolatot tartalmaz a naplóbejegyzés létrejöttét kiváltó felhasználóra mutatva. A későbbi fejlesztések megkönnyítése érdekében az eszközök és a naplóbejegyzések között nincs tényleges adatbázis kapcsolat. A kapcsolatot az entitás azonosítója (entityID) és entitás típusa (entityType) határozza meg, így biztosítva, hogy bármilyen entitáshoz készülhessen naplóbejegyzés egységes módon.

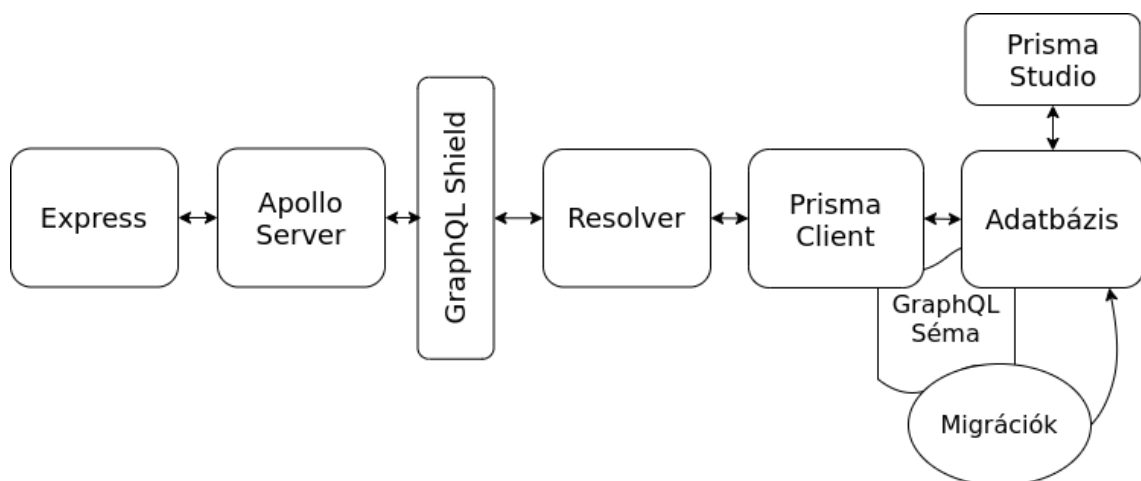
5.2. Backend felépítése

Az alkalmazás üzleti logikáját megvalósító rész az architekturális tervezés értelmében egy NodeJS-re épülő rendszer. Az alkalmazás egyetlen egy végpontot ajánl a kliensek számára. A kéréseket egy express server fogadja, a feldolgozásának mikéntjéről pedig egy Apollo server gondoskodik, itt történik meg a GraphQL elemzése és ez alapján a megfelelő kódrészlet futtatása. Az Apollo server lehetőséget nyújt middleware-ek



5.1. ábra. Adatbázis séma

definiálásra, melyek minden kérés kiszolgálása előtt végrehajtódnak. Erre a lehetőségre épít a GraphQL Shield nevű könyvtár, aminek segítségével minden egyes GraphQL műveletre megadhatunk ahhoz szükséges előfeltételeket egyszerű szabályok segítségével. Ilyen szabályokkal valósítottam meg a teljes autorizációt és az autentikáció ellenőrzését is.



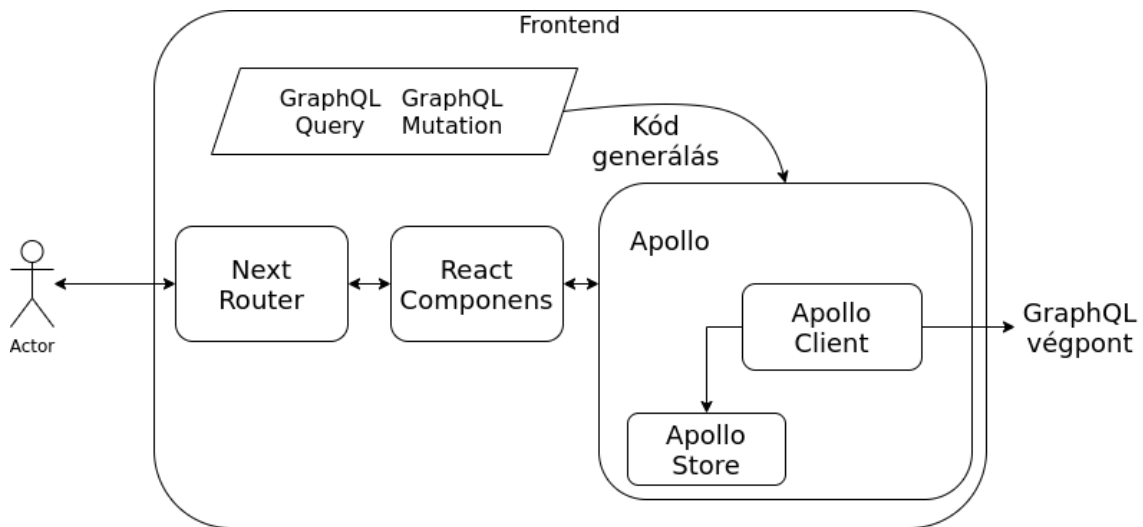
5.2. ábra. Backend felépítése

A megfelelő kódrészlet és a middleware-ek futtatása után – a Prisma-án keresztül – az adatbázishoz fordulunk adat lekérés, vagy módosítás miatt.

Az ábrán (5.2. ábra) világosan látszik, hogy a Prisma által nyújtott Studio az adatbázishoz csatlakozik, így az általunk írt üzleti logika nem fog érvényesülni. Fontos, hogy az itt végrehajtott módosítások nem várt működéshez is vezethetnek.

5.3. Frontend felépítése

Ahogy azt a korábbi fejezetekben taglaltam a kliens alkalmazás megvalósításához React-et használtam. Mivel a React önmagában egy nagyon könnyű keretet ad, ezért ezt egy NextJS nevű csomaggal egészítettem ki, amely még kényelmesebbé teszi a használatát. A backendhez csatlakozást az Apollo Client könyvtárral oldottam meg. Az Apollo Client és az Apollo Server együtt egy nagyon jól és könnyen használható rendszert alkotnak. A Client megkapja a Server-től a GraphQL sémát, így fejlesztés közben kódkiegészítéssel és típus ellenőrzéssel írhatjuk a lekérdezéseinket. Az Apollo Client a kommunikáció mellett a gyorsírotárazást is kezeli, így biztosítva az alkalmazás lehető leggyorsabb működését. Ezeken felül lehetőségünk nyílik kódgenerálásra is. A megírt GraphQL Query és GraphQL Mutation kódokból React hook-okat kapunk, melyekben az állapot- és a típusok kezelése is megvalósított.



5.3. ábra. Frontend felépítése

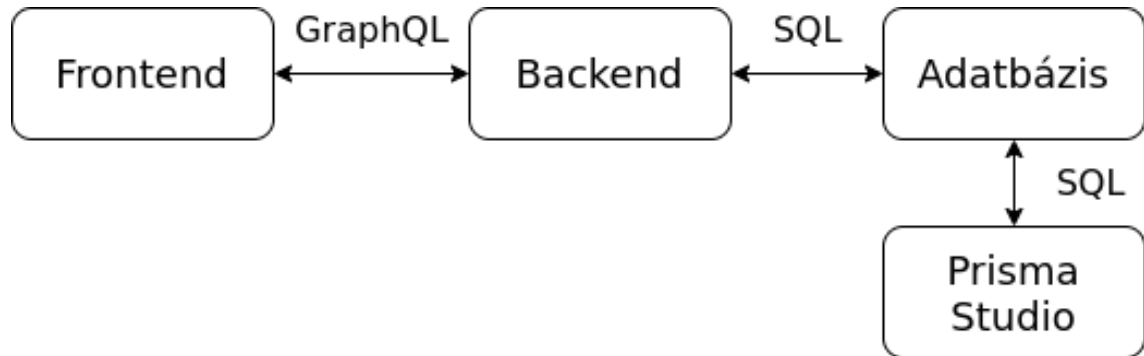
A React-ben a komponensek közötti logika megosztását a Context-ek segítségével valósítottam meg. Ennek használatával nem szükséges minden komponensnek átadni manuálisan a nélkülözhetetlen változókat, hanem elegendő csak egy, vagy szükség esetén több provider-t az alkalmazásunk köré helyezni. A provider-ek segítségével adatot szolgáltatunk az összes provider-en belüli komponens számára. Elegendő csak az alkalmazás belépési pontjában ezeket a tényleges alkalmazás köré helyeznünk, ahogy azt a mellékelt kód is mutatja.

```
1 <ApolloProvider client={client}>
2   <ChakraProvider>
3     <AuthProvider>
4       <Layout>
5         <Component {...pageProps} />
6       </Layout>
7     </AuthProvider>
8   </ChakraProvider>
9 </ApolloProvider>
```

5.1. kódrészlet. Frontendhez használt provider-ek

5.4. Architektúra összefoglalása

Összegezve az architektúrát, a három fő komponense az alkalmazásnak a frontend, a backend és az adatbázis. A frontend és a backend közötti kommunikáció GraphQL segítségével történik az Apollo Client és az Apollo Server között. A backend és az adatbázis kommunikációja pedig SQL segítségével, azonban ezt a Prisma teljesen elfedi a fejlesztő előtt.



5.4. ábra. Teljes alkalmazás felépítése

6. fejezet

Fejlesztést segítő eszközök

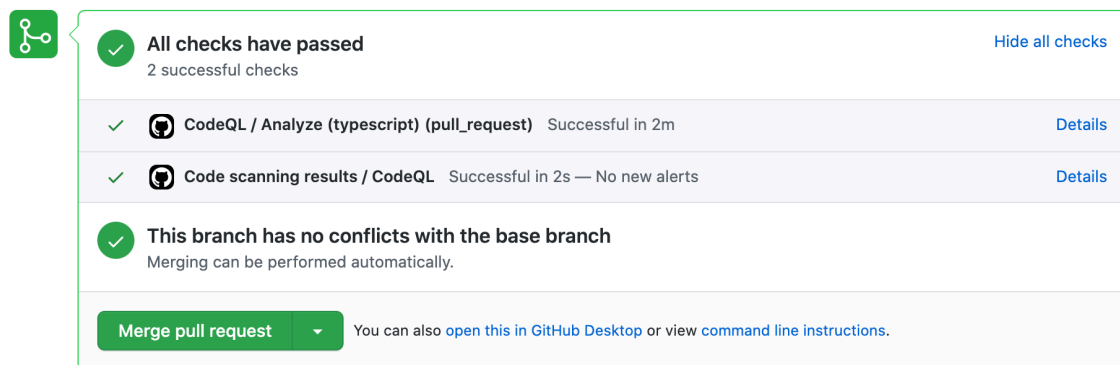
A fejlesztés során igyekeztem minél több olyan eszközt használni, ami elősegíti a munkát és javítja a kódminőséget és biztonságot.

6.1. Continuous Integration

A Continuous Integration napjainkban már elengedhetetlen része a fejlesztési folyamatoknak. Rengetek megoldás létezik, azonban én a GitHub Action mellett döntöttem. Ennek oka, hogy a GitHub–ot használtam a verziókezelte kód tárolására, így kézenfekvő volt ennek a megoldásnak az alkalmazása.

Az alkalmazást 2 külön repository–ban kezeltem, hogy jobban elkülönüljön a frontend és a backend kódja. Ennek a hátránya az volt, hogy bár ugyanazon nyelvet használja a két repository mégis kétszer kellett implementálnom a GitHub Action–öket.

Az action–ök létrehozását egyszerű szöveges formában tehetjük meg. A `.github/workflows` mappába létrehozott `github.yml` és `.yaml` kiterjesztésű fájlok automatikusan kiértékelődnek a GitHub Action által.



6.1. ábra. GitHub Action működés közben

6.1.1. Statikus kódellenőrzés

6.1.1.1. Linter

A linter egy viszonylag gyors lefutású és kevés erőforrást igénylő action, ezért beállítása szerint bármilyen commit kerül a GitHub repository–ba azonnal lefut és ellenőrzi a kód formázását és jelzi az esetleges hibákat statikus kódellenőrzés segítségével.

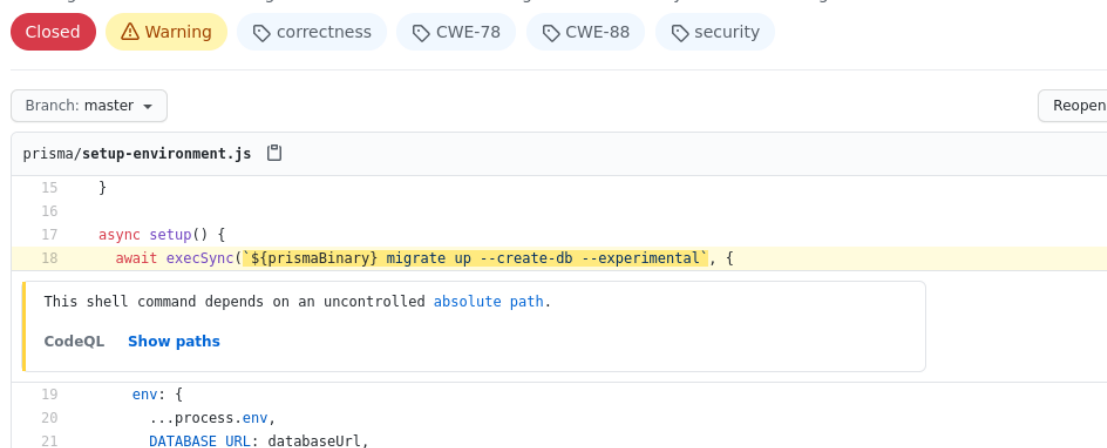
6.1.1.2. Sercurity check

A statikus kódellenőrzés egy másik lehetséges felhasználási módja a biztonsági rések keresése, ezt egy a GitHub által ajánlott megoldással valósítottam meg, a CodeQL-lel. Mivel ez egy több erőforrást igénylő folyamat, ezért a futtatása nem történik meg minden commitnál, csak ha az a főágba történik, vagy ha Pull Request-et nyit valaki, melynek a cél ága a főág.

A képen (6.2. ábra) egy a GitHub által észlelt lehetséges sebezhetőség detektálása látható. A kód analízis során a rendszer észlelte, hogy shell command futtatása történik úgy, hogy annak tartalma környezeti változóból származik. A figyelmeztetés jelen esetben egy fals riasztás volt, mert a sérülékeny kódrészlet az alkalmazása futása közben nem érhető el, ugyanis a teszt környezet kialakítására szolgál.

Shell command built from environment values

Building a shell command string with values from the enclosing environment may cause subtle bugs or vulnerabilities.



6.2. ábra. Code scanning figyelmeztetés

6.2. Git hook

Git használata esetén rengeteg eseményhez definiálhatunk úgynevezett hook-okat. Ezek segítségével bizonyos események után, előtt, vagy közben futtathatunk tetszőleges kódot. Az alkalmazásomban egy linter-t állítottam be a pre-commit hookra. Ennek értelmében, új commit elkészítése előtt minden alkalommal lefut a linter így biztosítva a kódminőségét.

```
1 "husky": {
2   "hooks": {
3     "pre-commit": "lint-staged"
4   }
5 },
6 "lint-staged": {
7   "*.ts": [
8     "eslint --fix"
9   ]
10 }
```

6.1. kódrészlet. Pre-commit hook beállításai

6.3. Continuous Delivery

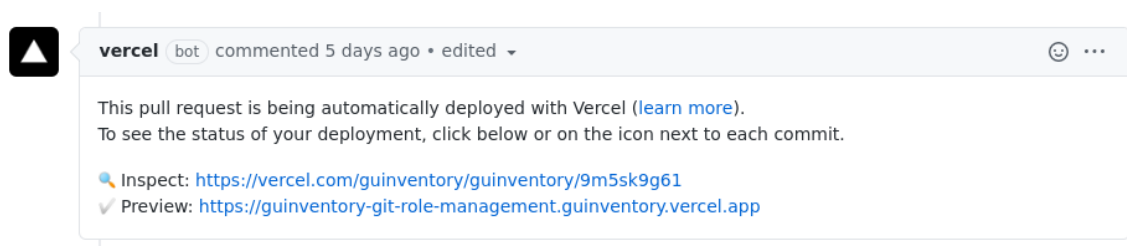
A Continuous Integration mellett elengedhetetlen része a fejlesztésnek a Continuous Delivery. A CD¹ az alkalmazás folyamatos kitelepítését jelenti, hogy a kód felöltése után szinte azonnal elérhető legyen az szoftverünk legfrissebb változata.

6.3.1. Heroku

A backend alkalmazás üzemeltetésére a Heroku szolgáltatásait vettem igénybe. Webes felületének és GitHub integrációjának köszönhetően nem igényel komoly szakértelmet az alkalmazás elindítása. Ingyenes keretek között csak egy ág automatikus kitelepítésére van lehetőség, azonban beállítható az is, hogy megvárja a CI² kimenetelét és csak sikeres futás után kezdje el a telepítést. Így elkerülhetjük hibás– vagy biztonsági réseket tartalmazó kódok éles környezetbe jutását.

6.3.2. Vercel

A frontend üzemeltetéséhez a Vercel-t használtam. A Herokuhoz hasonlóan remek GitHub integrációval és webes felülettel rendelkezik. Lehetőségünk nyílik a Pull Request-ekhez egy előnézeti alkalmazás kitelepítésére is, ezzel elősegítve a csapatmunkában egymás munkájának ellenőrzését. Ilyenkor a Vercel automatikusan hozzáad egy megjegyzést (6.3. ábra) a Pull Requesthez az előnézeti alkalmazás linkjével.



6.3. ábra. Vercel megjegyzése Pull Reques-nél

¹Continuous Delivery

²Continuous Integration

7. fejezet

Alkalmazás fejlesztése és működése

7.1. Backend

A backend architektúráis felépítését a korábbi fejezetben már részletesen taglaltam. Ebben a fejezetben az egyes rétegek és egyes funkciók megvalósításának bemutatására fektettem a hangsúlyt.

7.1.1. GraphQL Playground

A legtöbb GraphQL server-hez lehetőségünk nyílik valamilyen interaktív GraphQL szerkesztő felület kiszolgálásra is. Az alkalmazásban én a GraphQL Playground-ot használtam. Ennek a konfigurációját úgy valósítottam meg, hogy éles környezetbe ne szolgálja ki a felületet, csak fejlesztői környezet estén. Ezt környezeti változók segítségével oldattam meg.

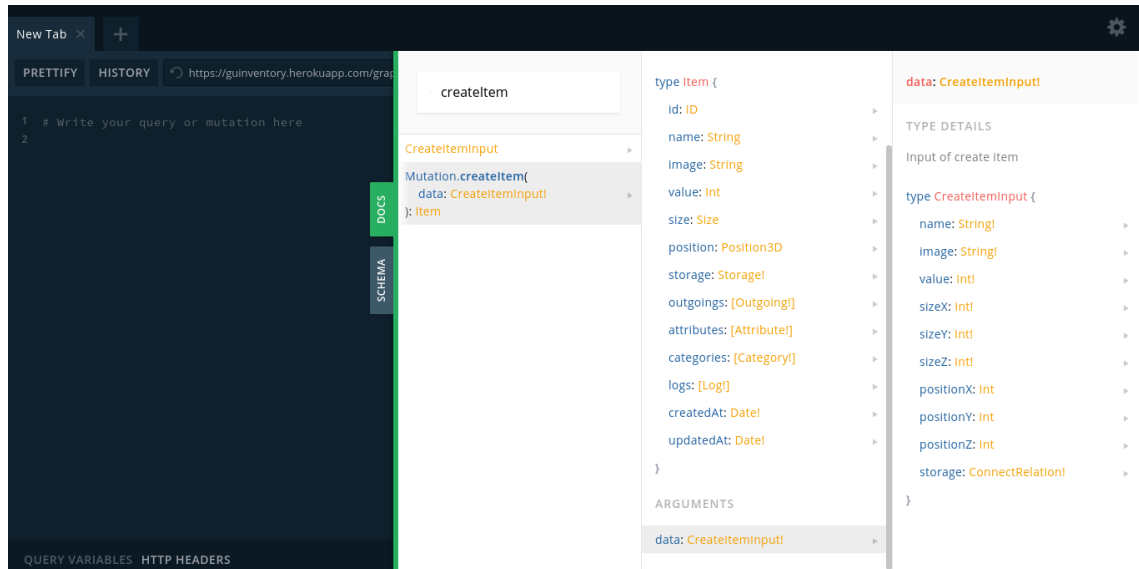
Azon felül, kódkiegészítéssel szerkeszthetjük a GraphQL kódunkat, kapunk egy dokumentációt is, amely tartalmazza az elérhető Query-k és Mutation-ök listáját a lehetséges paraméterekkel és a válaszok típusával együtt. Így a használatával még kényelmesebben készíthetjük el a lekérdezéseinket, melyeknek futtatására is lehetőségünk van.

7.1.2. Modellek és kapcsolatok

A modell definiálásakor megadhatjuk bármelyiket a sémában szereplő attribútumok közül, azonban fontos figyelni arra, hogy a típusok megegyezzenek a sémában és a modellben. A kapcsolatokat is egyszerű mezőként kezelhetjük. Beállítható továbbá az is, hogy a kapcsolat opcionális, vagy kötelező illetve, hogy egy, vagy több entitást tartalmazhat. Lehetőségünk van egyedi attribútumok létrehozására is, amely az adott modell bármely attribútumából származtatható, de tetszőleges kód futtatása is megengedett, így bármilyen származtatott értéket képesek vagyunk felvenni a modelljeinkhez.

Az egyszerűség és teljesség kedvéért a példában (7.1. szakasz) a `MyModel` felépítésén mutatom be egy modell definiálását. Az `id` és a `name` egyszerű, az adatbázisból származó attribútumok. A `user` egy kapcsolatot reprezentál, pontosan egy darab `User`-t tartalmaz. A `custom` mező egy egyedi attribútum, ami az `id` elejére egy kettős keresztet fűz, és úgy adja vissza azt.

```
1 import { objectType } from '@nexus/schema'
2
3 export const MyModel = objectType({
```



7.1. ábra. GraphQL Playground Docs

```

4  name: 'MyModel',
5  definition(t) {
6    t.id('id')
7    t.string('name')
8    t.field('user', {
9      type: 'User',
10     nullable: false,
11   }),
12   t.field('custom', {
13     type: 'User',
14     nullable: false,
15     resolve: ({id}) => `#${id}`
16   })
17 },
18 })

```

7.1. kódrészlet. Példa model

7.1.3. Resolver felépítése

A Mutation-ökhöz és Query-khez tartozó resolverek minden esetben tartalmaznak egy függvényt, amely eldönti, hogy meghívásukkor milyen kódrészlet fusson le. Ezen felül a használatukhoz szükségünk van metainformációkra is, ezért tartalmaz egy visszatérési típust, hogy pontosan tudjuk milyen típussal térhet vissza, valamint opcionálisan tartalmaz egy argumentum listát, ami meghatározza, hogy milyen paraméterek szükségesek és milyen paraméterek opcionálisak a meghíváshoz. A paraméter lista meghatározza a paraméterek típusát is.

Az alábbi példában (7.2. szakasz) egy egyszerű létrehozásra láthatunk egy lehetséges implementációt. A kódrészletben látható, hogy az argumentum lista helyett használhatunk külön definiált bemenetet is, így növelve a kód olvashatóságát. A resolve függvényben elvégezhetjük a szükséges adatbázis műveletet és meghívhatunk külső függvényeket is. A csatolt kódban az eszköz létrehozáson felül meghívunk egy függvényt, amely elkészíti az

eseményhez tartozó napló bejegyzést, valamint küldünk egy jelzést az elem létrehozásához. Ezekre a jelzésekre a GraphQL Playground – vagy bármilyen kliens segítségével – feliratkozhatunk GraphQL Subscription-ök használatával, mely egy egyszerű websocket kapcsolatot nyit a backend és a frontend között. A Subscription-nek köszönhetően ezeket a többi kommunikációhoz hasonlóan kezelhetjük, így egy egységesebb kódbázist kaphatunk.

```
1 t.field('createItem', {
2   type: 'Item',
3   args: { data: CreateItemInput.asArg({ required: true }) },
4   resolve: async (_, { data: { storage, ...rest } }, context: Context) => {
5     const item = await context.prisma.item.create({
6       data: {
7         storage: { connect: storage },
8         ...rest,
9       },
10    })
11    await log({
12      type: 'CREATE',
13      entityId: item.id,
14      entityName: 'Item',
15      newValues: { storage, ...rest },
16      context,
17    })
18    await publishItemEvent('itemCreated', item, context)
19    return item
20  },
21 })
```

7.2. kódrészlet. Eszköz létrehozás resolver

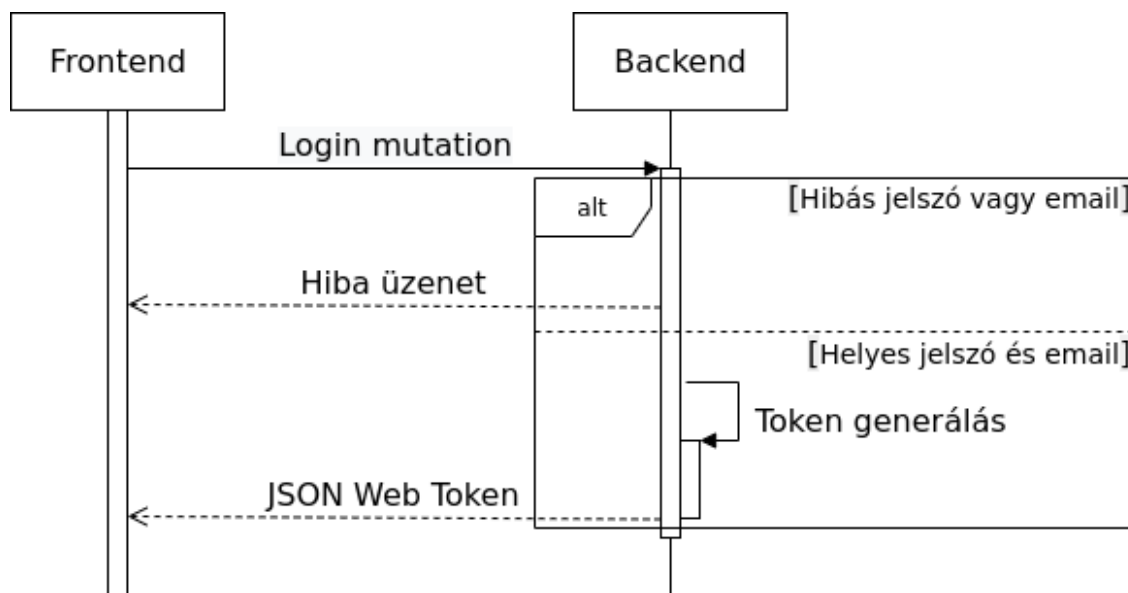
7.1.4. Autentikáció és autorizáció

Az autentikáció és autorizáció ellenőrzésére a GraphQL Shield-et használtam. Ennek segítségével egy egyszerű JavaScript object-tel megadható, hogy egyes Query-k és Mutation-ök esetén, milyen validáció fusson le. Lehetőséget biztosít egy úgynevezett fallback rule beállítására is, amely minden külön nem specifikált kérésnél fut le. Ezzel valósítottam meg a bejelentkezett felhasználó validálását (7.3. szakasz).

```
1 export const shield = GraphQLShield(
2   {
3     Query: {
4       warehouses: isGlobalAdmin,
5       logs: isGlobalAdmin,
6     },
7     Mutation: {
8       login: allow,
9       register: allow,
10    },
11  },
12  {
13    allowExternalErrors: true,
14    fallbackRule: fallbackRule,
15  },
16 )
```

7.3. kódrészlet. GraphQL Shield

Az autentikációt JSON Web Token segítségével végzem. Sikeres bejelentkezés esetén a backend egy tokent küld a frontend részére, melyet a böngészőbe elmentve a későbbiekben minden kéréshez csatolni tudunk. Az ábrán (7.2. ábra) a token létrehozásának folyamatát figyelhetjük meg.



7.2. ábra. JWT Bejelentkezés

7.1.5. Képek kezelése

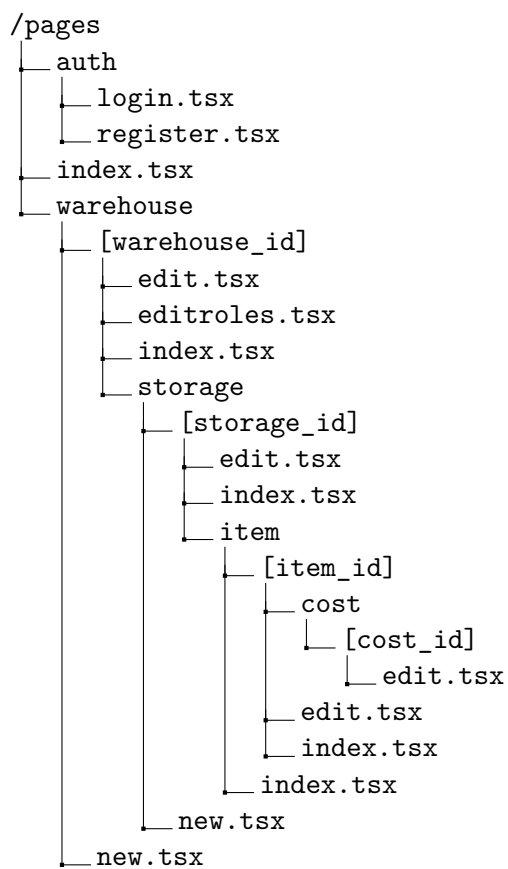
Az alkalmazáson belül lehetőségünk van kép feltöltésre a raktárba vett eszközökről. A képek tárolására a Google Cloud Storage szolgáltatását használtam. A képeket feltöltés előtt base64-es formátumra kódolom át és elküldöm a backend részére. A backend végzi a kép feltöltését a Google Cloud Storage rendszerébe és eltárolja a kép URL címét az adatbázisba a megfelelő eszköz attribútumai között.

7.2. Frontend

A frontend magas szintű architektúrájának bemutatása után ebben a fejezetben az egyes rétegek működésének részletes bemutatására kerül sor. Bizonyos komponensek bemutatása nem célja a szekciónak, ugyanis azok belső működése az alkalmazás és a fejlesztő elől rejtve vannak, így nem releváns a fejlesztés bemutatásának szempontjából. Ilyen például az Apollo Client, ami a kapcsolat felépítésért és az Apollo Storage, ami a gyorsírtárazásért felel.

7.2.1. Útvonalválasztás

NextJS használatával alapértelmezetten fájl alapú útvonalválasztást használhatunk. Ez azt jelenti, hogy az oldal URL-jét a fájl neve és a szülő mappák nevei határozzák meg. Lehetőségünk van paraméterek használatára is, ezt szögletes zárójel közé írt névvel jelezhetjük. A paramétert ezzel a névvel fogjuk elérni a kódbázison belül is. Ilyen paraméterrel jelzett nevet adhatunk bármelyik szinten lévő fájlnak, de akár mappának is.



7.3. ábra. Routing mappa szerkezete

7.2.2. Kódgenerálás

Ahogy az korábban már többször is szóba került az Apollo-nak köszönhetően remek kódgenerálási lehetőségeink vannak. Alább egy példa keretein belül szeretném bemutatni ennek használatát

A GraphQL Mutation-t (7.4. szakasz) paraméterekkel ellátva szükséges megírunk, hogy a kód generátor tudja a lehetséges paramétereket.

```
1 mutation Login($email: String!, $password: String!) {  
2   login(data: { email: $email, password: $password }) {  
3     token  
4   }  
5 }
```

7.4. kódrészlet. Login Mutation

A generálást futtatva azonnal használhatjuk az elkészült hook-okat, a csatolt példában (7.5. szakasz) ez a hook a useLoginMutation.

A hálózati kapcsolat kezelésén felül megkapjuk annak az állapotát is, így tudjuk a felhasználói felület kinézetét a kérés állapotához kötni. Például töltés esetén egy töltő képernyőt megjeleníteni, vagy a hibákat kezelni.

```
1 const { register, handleSubmit, errors } = useForm<Inputs>()  
2 const [login, { loading }] = useLoginMutation()  
3 const toast = useToast()  
4 const { setAuthToken } = useAuthToken()  
5  
6 const onSubmit = async (inputData) => {  
7   try {  
8     const {  
9       data: {  
10         login: { token },  
11       },  
12     } = await login({  
13       variables: inputData,  
14     })  
15     setAuthToken(token)  
16     window.location.href = '/'  
17   } catch (error) {  
18     toast({  
19       title: error.message,  
20       status: 'error',  
21       duration: 3000,  
22       isClosable: true,  
23     })  
24   }  
25 }
```

7.5. kódrészlet. Bejelentkezés kódrészlet

7.2.3. Validáció

A felhasználótól érkező adatok minden esetben validálva vannak, ehhez a yup csomagot használtam. A yup segítségével definiálhatunk egy sémát, amelyre a bemenetnek

illeszkedni kell. Hiba esetén egy hibaüzenetet is generál az egyes mezőkhöz, így visszacsatolást adhatunk a felhasználónak, hogy melyik adat hibás és miért. Amennyiben szeretnénk eltérni az alapértelmezett beállításoktól, a hibaüzeneteket szövegezését a séma definiálással együtt adhatjuk meg (7.6. szakasz).

```
1 import * as yup from 'yup'
2
3 export const itemSchema = yup.object().shape({
4   name: yup.string().required(),
5   image: yup.string().required(),
6   value: yup.number().required().typeError('Must be a number'),
7   positionX: yup.number().required().typeError('Must be a number'),
8   positionY: yup.number().required().typeError('Must be a number'),
9   positionZ: yup.number().required().typeError('Must be a number'),
10  sizeX: yup.number().required().typeError('Must be a number'),
11  sizeY: yup.number().required().typeError('Must be a number'),
12  sizeZ: yup.number().required().typeError('Must be a number'),
13 })
```

7.6. kódrészlet. Esköz validációs séma

A formok elküldését egy React hook-kal valósítottam meg. A hook visszaadja a hibákat és lehetőséget biztosít a form újra-beállítására is. Minden beviteli mezőt regisztrálni kell a form hook-ba, így biztosítva azok elérését.

```
1 const { register, handleSubmit, reset, errors } = useForm<Inputs>({
2   resolver: yupResolver(itemSchema),
3 })
```

7.7. kódrészlet. Regisztrációnál használt form hook

A hibákat egy JavaScript objektumban kapjuk meg, melynek a kulcsa minden esetben az adott beviteli mező neve.

```
1 <form onSubmit={handleSubmit(onSubmit)}>
2   <FormControl mb={4} isValid={!errors.name}>
3     <FormLabel htmlFor="name">Name</FormLabel>
4     <Input name="name" type="text" ref={register} />
5     <FormErrorMessage>{errors.name?.message}</FormErrorMessage>
6   </FormControl>
7   ...
8 </form>
```

7.8. kódrészlet. Form hibák megjelenítésével

7.2.4. Felhasználói felület

A felhasználói felületet a Chakra UI¹ könyvtár segítségével készítettem el. Ez kifejezetten React-hez készült és rengeteg konfigurálható beállítást tartalmaz, így teljes mértékben személyre szabható, egyedi design-t kaphatunk. Lehetőségünk van globális téma definiálására, de akár komponensenként is változtathatjuk a kinézetet a rengeteg paraméternek köszönhetően. A Chakra használatának köszönhetően az összes általános komponens egy szép és minden lehetséges állapotra felkészített változatban rendelkezésemre állt.

¹A Chakra UI hivatalos weboldala <https://chakra-ui.com/>

Register

Name
Teszt

E-mail address
john.doe@example.org
email is a required field

Password

password must be at least 8 characters

Password again

Passwords must match

Register

7.4. ábra. Regisztrációs oldal

A regisztrációs oldalon (7.4. ábra) láthatjuk a beviteli mezőket különböző állapotban. Hibásan kitöltött mező esetén felhasználó azonnali visszajelzést kap a hibás adatról. Javítás után a hibaüzenet automatikusan eltűnik, amint megfelelő formátumú adatot gépelt be a felhasználó.

GUINVENTORY Warehouses Logout

Search

Item	Storage	Warehouse
George Orwell - Animal farm	Konyvespolc	Teszt raktar
George Orwell - 1984	Konyvespolc	Teszt raktar

Konyvespolc

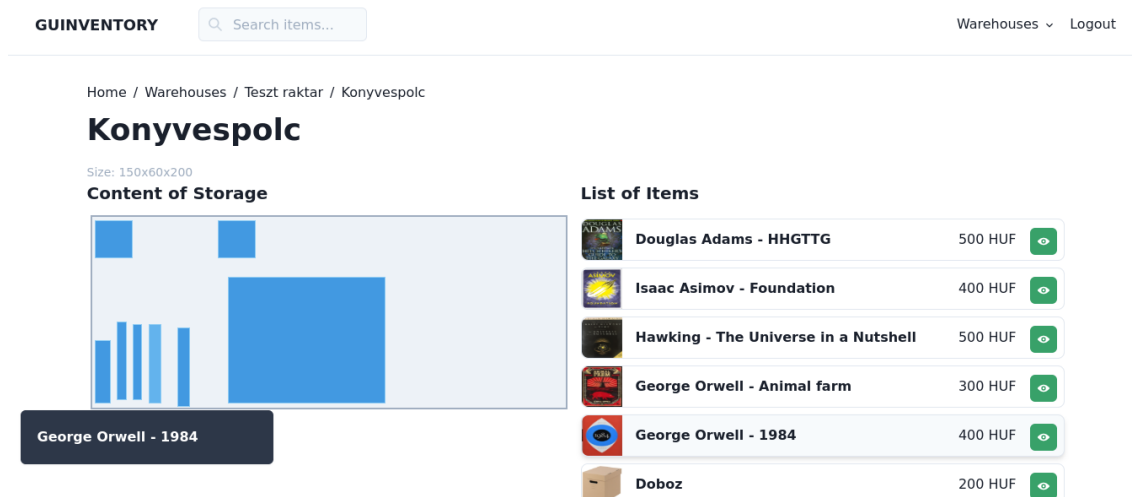
Ures szekreny

7.5. ábra. Keresés

A keresés eredményét egy 3 oszlopos táblázatban (7.5. ábra) jelenítjük meg. A három oszlop segítségével egyértelműen meghatározható a keresett eszköz holléte. Lehetőségünk van a keresett eszköz raktárára, tárolójára vagy magára az eszközre navigálnunk.

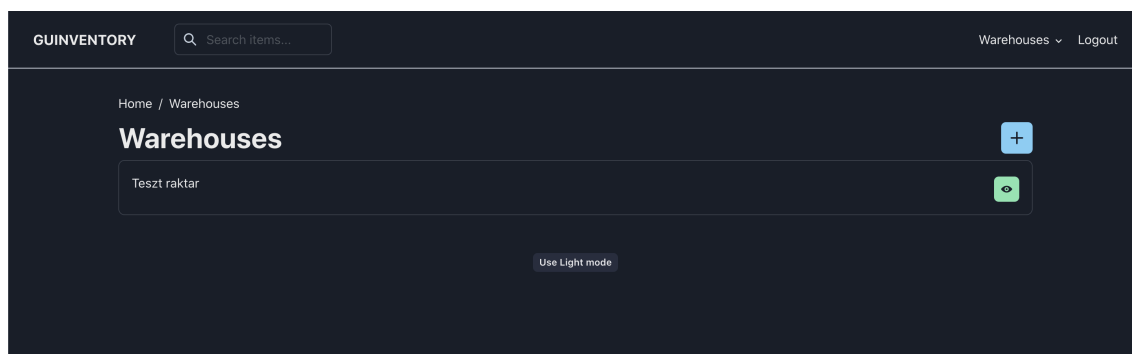
A tároló nézetén (7.6. ábra) a kurzort valamelyik eszköz fölé mozdgatva megjelenítjük annak a nevét és ezen felül kiemeljük a listában is, hogy elősegítsük az azonosítást. Természetesen ez a kiemelés a másik irányba is megvalósul, tehát ha a listában választjuk ki, akkor a térképes nézetben fogjuk kiemelten látni az éppen kiválasztott eszközt.

A raktár nézetén belül a tárolóhoz hasonlóan kiemeléssel jelezzük a kiválasztott tárolót. A tárolókról extra információként megjelenítjük az adott tároló becsült kihasználtságát. A becslés a tároló méretéből és a benne tárolt eszközök méreteiből számolt



7.6. ábra. Tároló oldal

értékét. Természetesen pontosan kihasználtságot nem tudunk adni, mivel a tárolókat általában nem lehet 100%-osan kitölteni.



7.7. ábra. Sötét téma

A Chakra UI segítségével egyszerűen megvalósítható a sötét és világos téma is az alkalmazáshoz, valamint az e kettő közötti váltás is. Ez jelen esetben egy sötétkék árnyalatú designt (7.7. ábra) eredményezett. A bejelentkezett felhasználó az oldal alján található gomb segítségével válthat témát. A választott témát az alkalmazás automatikusan elmenti a böngésző helyi tárolójába, így az oldal újbóli meglátogatásakor már a korábban kiválasztott téma lesz érvényes.

8. fejezet

Tesztelés

Az alkalmazás működésének validációjában elengedhetetlen lépés a tesztelés. A tesztelés ezen felül segíti a fejlesztő munkáját is, bármilyen aprónak tűnő változtatás olykor hatással lehet az alkalmazás más részeire is. Előfordulhat, hogy már meglévő és működő funkciók válnak használhatatlanná új funkciók bevezetése közben. Ennek a kockázatát megfelelő tesztlefedettséggel minimálisra csökkenthetjük.

8.1. Frontend tesztelés

A frontend teszteléséhez úgy nevezett end-to-end tesztek készítését. Az E2E¹ tesztek esetén a teszt a felhasználó viselkedését szimulálja. Egy szimulált böngészőben végzi el a tesztet és tényleges kattintás eseményt vált ki, majd figyeli a renderelt képet a tesztnek megfelelő egyezést keresve.

Ezt a Jest² és a Playwright³ package-ek segítségével valósítottam meg. A tesztelés során szükség van bizonyos adatbázis adatokra, ezeket környezeti változókba helyeztem. Jelenleg a tesztek csak lokális környezetben futnak, de a későbbi fejlesztések során így biztosítva lesz, hogy adatbázisból adatok ne kerüljenek harmadik fél kezébe.

Az alábbi példában (8.1. szakasz) a bejelentkezés E2E tesztje látható. A bejelentkezési oldalra navigálunk, ahol kitöltjük az email és jelszó mezőket majd elküldjük a formot. A teszt rövid várakozás után ellenőrzi, hogy megtörtént-e az átirányítás és a renderelt oldal tartalmazza-e a Warehouse szöveget egy h2 HTML tag-ben.

```
1 it('should allow users to sign in', async () => {
2   await page.goto('http://localhost:3000/auth/login')
3   await page.waitForTimeout(1000)
4   await page.fill('[name="email"]', process.env.TEST_USERNAME)
5   await page.fill('[name="password"]', process.env.TEST_PASSWORD)
6   await page.click('[type=submit]')
7
8   await page.waitForTimeout(1000)
9   await expect(page.url()).toBe('http://localhost:3000/')
10  await expect(page).toHaveText('h2', 'Warehouses')
11 })
```

8.1. kódrészlet. Bejelentkezés E2E teszt

¹end-to-end

²Jest NPM oldala <https://www.npmjs.com/package/jest>

³Playwright NPM oldala <https://www.npmjs.com/package/playwright>

8.2. Backend tesztelés

A Prisma fejlesztő csapata csupán pár hónappal ezelőtt jelentette be a 2.0-ás verziót. A folyamatos fejlesztés ellenére is még hiányos az eszközkészlete. Ennek köszönhetően teszteléshez sem kínál semmilyen megoldást.

A tesztelési környezet kialakításához egy SQLite adatbázist szerettem volna használni, hogy ne a fejlesztés közben használt adatbázist használjam, és ne teljen túl sok időbe a tesztek futtatása. Azonban a Prisma jelenlegi verziója nem támogatja az enumerációt SQLite adatbázis esetén. Emiatt itt is egy PostgreSQL adatbázissal kellett dolgoznom. A tesztelés előtt egy scripttel automatikusan létrehozom a szükséges adatbázist, majd a tesztek végeztével törölöm azt, így biztosítva, hogy véletlenül se maradjon semmilyen adat az adatbázisba, ami esetleg fals eredményt váltana ki a tesztekben.

A tesztek során a GraphQL végpontot hívtam meg különböző query-ek és mutation-ökkel, majd a választ vizsgálva megbizonyosodtam a helyes működésről.

```
1 test('successfully register a user', async () => {
2   const data = {
3     name: 'User Name',
4     email: 'test.user@example.org',
5     password: 'password',
6   }
7   const req: any = await request(config.url, register, data)
8
9   expect(req).toHaveProperty('register')
10  expect(req.register.user.email).toEqual(data.email)
11 })
```

8.2. kódrészlet. Regisztráció első teszt eset

9. fejezet

Üzemeltetés

Korábbi fejezetben már kifejtettem, hogy az üzemeltetést Heroku és Vercel segítségével oldottam meg. Azonban ez az alkalmazás tényleges üzembe helyezése után nem feltétlenül a legköltséghatékonyabb módja az üzemeltetésének. Ezért igyekeztem alternatívákat kínálni az esetleges üzembehelyezőknek.

9.1. Docker

Az üzemeltetés megkönnyítése érdekében minden komponenshez készítettem egy-egy dockerfile-t és docker-compose file-t.

A frontendhez tartozó Docker felépítése három lépésből áll. Első lépésben a szükséges függőségeket telepítjük a yarn csomagkezelő segítségével. Második lépésben a TypeScript kódot fordítjuk JavaScript kódra, a harmadik lépésben pedig egyszerűen elindítjuk az alkalmazást.

A három lépéses konténer készítés oka a docker cachelésének maximális kihasználása. Ez különösen nagy segítség lehet, ha ugyanazt a verziót több környezetbe is szeretnénk élesíteni.

```
# 1 - Install dependencies
FROM node:12-alpine AS dependencies

WORKDIR /app
COPY package.json yarn.lock ./
RUN yarn install

# 2 - Build
FROM node:12-alpine AS build

ENV NODE_ENV=production
WORKDIR /app
COPY . .
COPY --from=dependencies /app/node_modules ./node_modules
RUN yarn build

# 3 - Run
FROM node:12-alpine AS run

WORKDIR /app
ENV NODE_ENV=production
COPY --from=dependencies /app/node_modules ./node_modules
COPY --from=build /app/public ./public
COPY --from=build /app/next.config.js ./
COPY --from=build /app/.next ./next
CMD ["node_modules/.bin/next", "start"]
```

9.1. kódrészlet. Frontend Dockerfile

A backendhez tartozó container sokkalta egyszerűbb. A hozzá tartozó docker-compose file tartalmazza a backend és az adatbázis elindítását, valamint a közöttük lévő kapcsolat kiépítését és a Prisma Studio indítását is. A Prisma Studio és az alkalmazás konkurensen futnak egy containeren belül. Fontos megjegyezni, hogy a Studiot csak feltétlenül szükséges esetben használjuk, érdemes tiltani a hozzáférést ugyanis nem tartalmaz semmilyen beépített jelszavas védelmet.

```
version: '3'
services:
  server:
    build: .
    links:
      - postgres
    depends_on:
      - postgres
    ports:
      - '${SERVER_PORT}:4000'
      - '${STUDIO_PORT}:5555'
    environment:
      NODE_ENV: ${NODE_ENV}
      DATABASE_URL: ${DATABASE_URL}
      APP_SECRET: ${APP_SECRET}
    networks:
      - server-network
  postgres:
    image: postgres:12.5
    restart: always
    environment:
      POSTGRES_USER: prisma
      POSTGRES_PASSWORD: prisma
      POSTGRES_DB: prisma
    networks:
      server-network:
        aliases:
          - postgresql.db
    volumes:
      - db_data:/var/lib/postgresql/data
volumes:
  db_data:
networks:
  server-network:
```

9.2. kódrészlet. Backend docker compose

10. fejezet

Összefoglalás

A fejlesztés során rengeteg olyan problémát kellett megoldanom, melyekkel egyébként nem találkoztam volna. Elmélyültem olyan technológiákban, melyek napjainkban divatosak és piaci környezetben is megállják a helyüket. Ezeket a későbbiekben alkalmazhatom, mint tanulmányaimban, mint munkám során.

10.1. Tovább fejlesztési lehetőségek

A szakdolgozat elkészítése során temérdek új fejlesztési lehetőség jutott eszembe, melyeket megvalósítva egy még jobb és még inkább a piaci igényeknek megfelelő alkalmazást kaphatunk. A szakdolgozat keretein belül igyekeztem az elengedhetetlen funkciókat a lehető legjobban megvalósítani és inkább a fejlesztői eszköztár felépítését tartottam fontosnak, mint a rengeteg funkció belezsúfolását egy alkalmazásba. Ennek köszönhetően remélhetőleg egy hosszú távon is fejleszthető és fenntartható alkalmazás született. A későbbiekben szeretném folytatni az általam írt alkalmazás fejlesztését, annak érdekében, hogy a lehető legtöbb piaci igényt legyen képes kiszolgálni az alkalmazásom.

10.1.1. Keresés

A jelenlegi keresés egy nagyon egyszerű string összehasonlításra alapul, már egyetlen karakter elgépelése esetén sem talál egyezést. Ennek a problémának a megoldására több lehetőség is kínálkozik, ilyen például az Elastic Search vagy a PostgreSQL-be épített full-text search. Ezeknek az alapvető működési elve, hogy nem magában az adatbázisban keres hanem létrehoz egy index-szelt szótárat és ezt hasonlítja a keresési kifejezéshez. Ennek köszönhetően nagy adatbázisok esetén is gyors keresés érhető el.

10.1.2. Egyedi tulajdonságok kezelése

Már a tervezés elején megfogalmazódott az ötlet a raktárban tárolt eszközök egyedi tulajdonságainak kezelése, mint ötlet. A tervezés során kiderült, hogy ez jóval több időt venne igénybe, mint azt az elején gondoltam, így a megvalósítását kihagytam a szakdolgozatból.

A koncepció lényege, hogy kategóriákat hozhatunk létre minden raktáron belül. A kategóriákhoz egy név megadása után felvehetőek a tulajdonságok nevei és típusai. Ezután az eszköz felvételekor kiválasztjuk, hogy milyen kategóriába, kategóriákba tartozik. Így a kategóriák révén már tudjuk azt, hogy milyen egyedi tulajdonságai lehetnek. Természetesen ehhez szükséges előredefiniált attribútum típusok létrehozására is.

10.1.3. 3D-s térkép nézet

Az alkalmazás már a jelenlegi verziójában is tárolja a 3 dimenziós adatokat, azonban ez a megjelenítésben figyelmen kívül hagytam. A későbbi fejlesztések során a jelenlegi térkép helyett egy 3 dimenziós nézet segítségével lehetőség nyílna az eszközök a valósághoz még közelebb eső állapotának tárolására. Ennek megvalósítása azonban nagyon sok problémát vet fel, így az implementációja rengeteg időt emésztethet fel.

Ábrák és táblázatok jegyzéke

3.1.	Regisztráció wireframe	6
3.2.	Keresés wireframe	6
3.3.	Raktár wireframe	7
3.4.	Tároló wireframe	7
4.1.	Google Trends – keresések összehasonlítása.	9
4.2.	NPM Trends – letöltések összehasonlítása.	10
4.3.	Példa egy entitáson végezhető műveletekre a REST API elvei szerint	11
4.4.	NPM Trends – ORM-ek letöltésének összehasonlítása.	13
5.1.	Adatbázis séma	15
5.2.	Backend felépítése	15
5.3.	Frontend felépítése	16
5.4.	Teljes alkalmazás felépítése	17
6.1.	GitHub Action működés közben	18
6.2.	Code scanning figyelmeztetés	19
6.3.	Vercel megjegyzése Pull Reques-nél	20
7.1.	GraphQL Playground Docs	22
7.2.	JWT Bejelentkezés	24
7.3.	Routing mappa szerkezete	25
7.4.	Regisztrációs oldal	28
7.5.	Keresés	28
7.6.	Tároló oldal	29
7.7.	Sötét téma	29

Irodalomjegyzék

- [1] Youssef Nader: Angular vs react - detailed comparison. <https://hackr.io/blog/angular-vs-react>.
- [2] Node.js. <https://en.wikipedia.org/wiki/Node.js>.
- [3] Npm – mongoose. <https://www.npmjs.com/package/mongoose>.
- [4] Npm – sequelize. <https://www.npmjs.com/package/sequelize>.
- [5] Npm – typeorm. <https://www.npmjs.com/package/typeorm>.
- [6] React (web framework). [https://en.wikipedia.org/wiki/React_\(web_framework\)](https://en.wikipedia.org/wiki/React_(web_framework)).
- [7] Typescript. <https://www.typescriptlang.org/>.
- [8] Vue.js. <https://en.wikipedia.org/wiki/Vue.js>.