FELADATKIÍRÁS

A feladatkiírást a tanszéki adminisztrációban lehet átvenni, és a leadott munkába eredeti, tanszéki pecséttel ellátott és a tanszékvezető által aláírt lapot kell belefűzni (ezen oldal *helyett*, ez az oldal csak útmutatás). Az elektronikusan feltöltött dolgozatban már nem kell beleszerkeszteni ezt a feladatkiírást.

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 5.

Tartalomjegyzék

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 5.

Király Bálint Martin
hallgató

Kivonat

Jelen dokumentum egy diplomaterv sablon, amely formai keretet ad a BME Villamosmérnöki és Informatikai Karán végző hallgatók által elkészítendő szakdolgozatnak és diplomatervnek. A sablon használata opcionális. Ez a sablon LaTeX alapú, a *TeXLive* TeX-implementációval és a PDF-LaTeX fordítóval működőképes.

Abstract

This document is a LaTeX-based skeleton for BSc/MSc theses of students at the Electrical Engineering and Informatics Faculty, Budapest University of Technology and Economics. The usage of this skeleton is optional. It has been tested with the *TeXLive* TeX implementation, and it requires the PDF-LaTeX compiler.

1. fejezet

Bevezetés

A félév 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 nemfunkcionális 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 arhitekturális 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. Ezeket később részletesen taglalom.

Az alkalmazás tervezésén és fejlesztésén kívül az üzemeltetés előkészítését is elvégeztem.

Végül a tesztelésre fektettem a hangsúlyt.

A dolgozat fejezeteivel is próbáltam ezt a sorrendet tartani, hogy az olvasó számára is könnyen követhető legyen a folyamat és a meghozott döntések.

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 felhasználót létrehozni egy regisztrációs oldalon. 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

2.1.2. Bejelentkezés

A regisztráció során megadott email cím és jelszó segítségével a látogatónak képesnek kell lennie authentikálni magát. A rendszer csak authentikált felhasználók számára legyen elérhető. Nem authentikált felhasználóknak csak a bejelentkezés és a regisztráció opciókat kínáljuk fel.

2.1.3. Kijelentkezés

A bejelentkezett felhasználónak biztosítsunk lehetőséget a munkamenet megszünetetésére. 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) kezelését. Ez alatt értjük a raktár létrehozását, és 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 helyezhetőek. A tárolókat a nevükkel, meretükkel és a raktáron belüli poziciójukkal együtt rögzíthetjuk. Amennyiben a felhasználó rendelkezik a megfelelő jogosultsággal az adott raktáron belül, legyen lehetősége a tárolók szerkesztésre, létrehozására és törlésére.

2.1.6. Eszközök kezelése

A hierarchia harmadik szintjén helyezkednek el az eszközökök. Minden eszköz rendelkezzen az alábbi tulajdonságokkal. Név, amely az egyszerű azonosítást és a kereshetőséget biztosítja. Érték, ami az eszköz raktárba vételekori értékét tartalmazza. Mérete és a tárolón belüli poziciója. Minden egyes leltárba vett eszköhöz legyen lehetőségünk a kiadások felvételére. Minden kiadáshoz egy összeg és egy leírás tartozik, amely a kiadás okának magyarázatára szolgál.

2.1.7. Raktár térképes nézettel

A raktárakban a tárolók elhelyezkedését jelenítsük meg egy felülnézeti, térképes nézet formájában is. A tárolók mozgatását 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 ball billentyűjét lenyomva 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 fontos, hogy képesek legyünk nyomonkövetni az adatbázisba történt változások okait, különösen igaz ez egy raktár rendszer esetén. A rendszer logoljon minden eszközzel kapcsolatos felhasználói interakciót, amely adatbázis művelethez vezet. Tehát az eszköz létrehozását, szerkesztését és törtlését. A megtekintések logolása nem szükséges.

2.2. Nem funkcionális követelmények

Az alkalmazással szemben természetesen nem csak funkcionális követelmények vannak. A nem funkcionális követelmények 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 egyidejüleg több felhasználó kiszolgálása webböngésző segítségével. 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 elgendőnek kell lennie egy átlagos személyi számítógép hardware készlete. A fejlesztést és üzemeltetést tegyük lehetővé Linux, MacOS vagy Windows operációs rendszereken is.

3. fejezet

Wireframe-ek

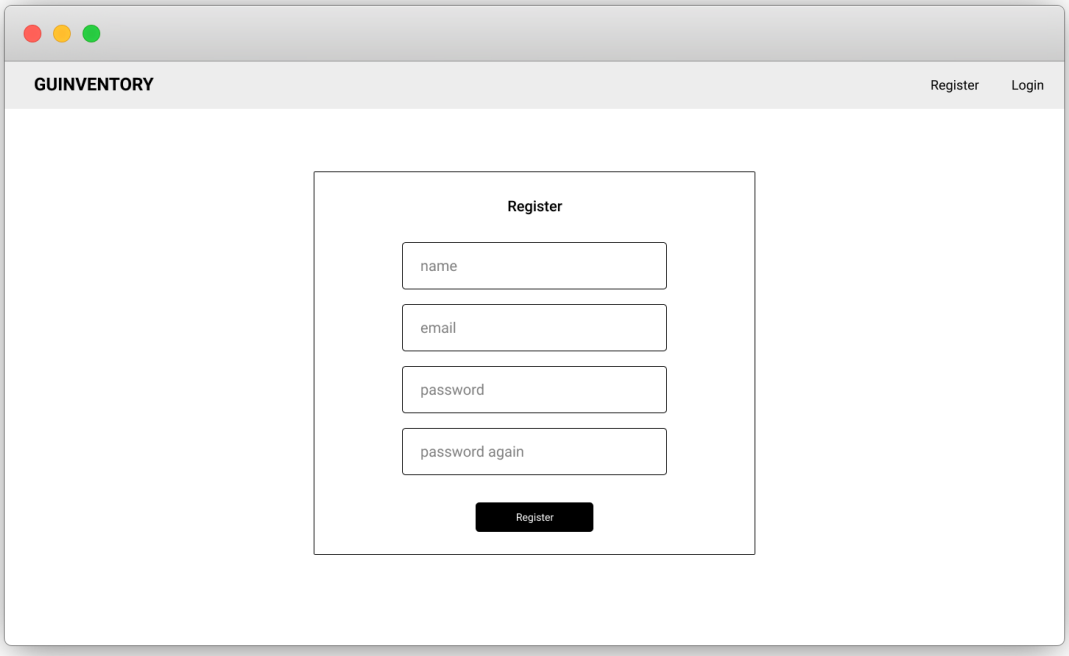
A wireframe-ek egy alkalmazás tervezésénél nagyon gyakran elmaradnak, pedig igenis fontos szerepük van. Rengeteg olyan dologra villágithatnak rá, amire egyébként nem gondolnánk és segíti a kommunikációt a fejlesztő(k) és a megrendelő között.

A dolgozatba csak a főbb képernyők wireframe-ét helyeztem el.

Ezeknek az elkészítéséhez a Figma névre keresztelt webes alkalmazást használtam. Ennek segítségével az egyszerű wireframe-ektől kezde komplex protoipizált 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.

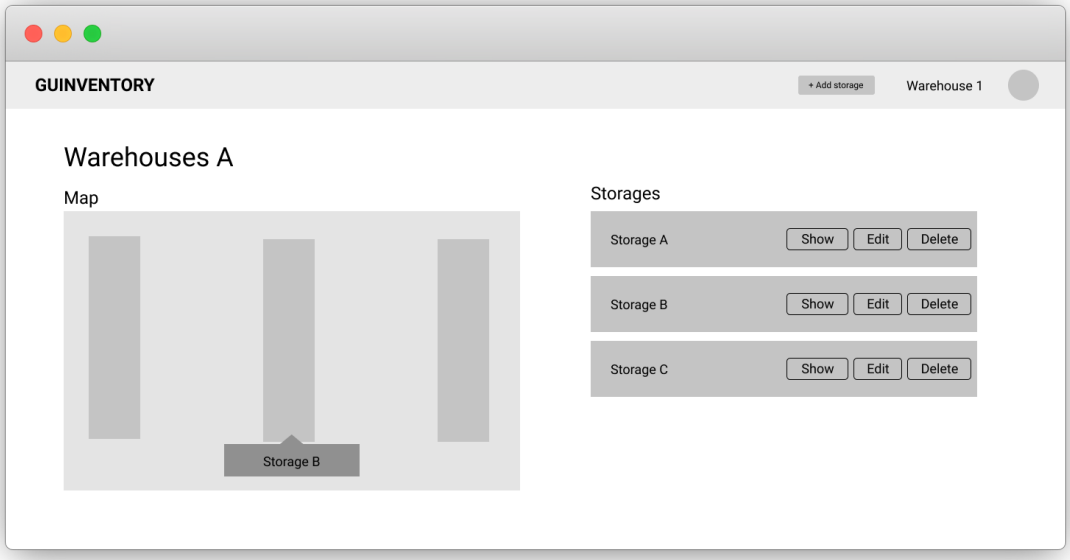


3.1. ábra. Regisztráció wireframe

3.2. Raktár nézet

A raktár oldalán (3.2. ábra) láthatunk egy térképes nézetet és egy listát is a tárololókról. A térképes nézeten a kurzort a tároló fülé mozgatva megjelnítjük annak nevét a könnyebb azonosítás érdeképben.

Ezen felül a navigációs bárban láthatjük az éppen kiválasztot 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.

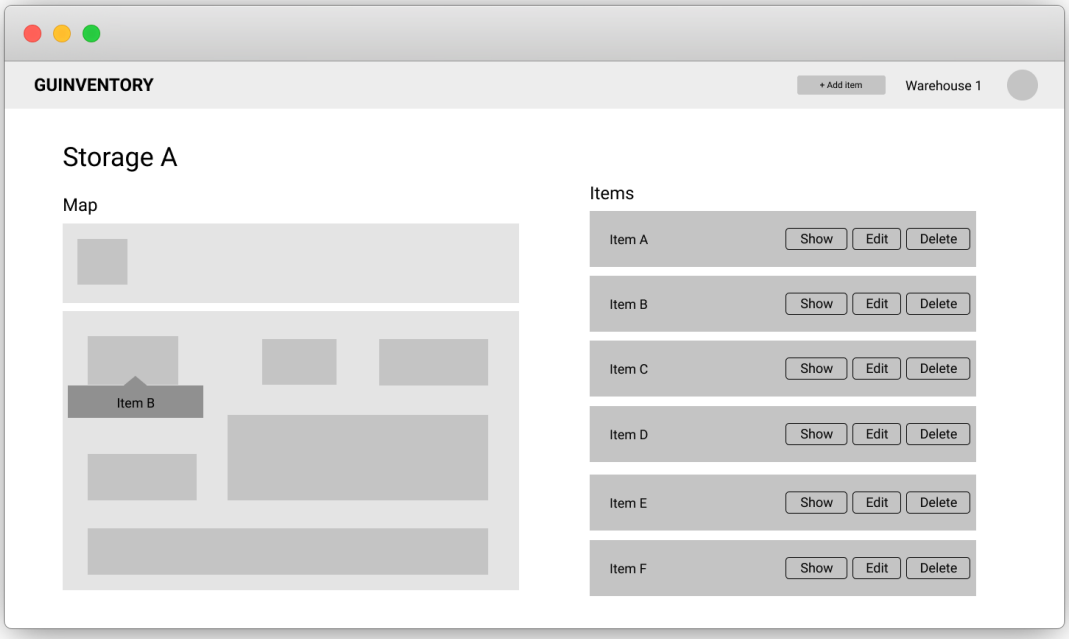


3.2. ábra. Raktár wireframe

3.3. Tároló nézet

A tároló nézete (3.3. ábra) nagyban hasonlít a raktáréhoz, azonban itt kiegészítésképt a térkép felett megjelenítünk egy másik tárolót. Ez a feladatspecifikációban megkövetelt tárolók közötti eszköz mozgatását teszi lehetővé.

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



3.3. ábra. Tároló wireframe

4. fejezet

Választott technológiák

Annek a fejezetnek a keretein belül a felhasznált technológiák kiválasztásának szempontjait szeretném bemutatni. Elöszőr a frontend technológiát választottam ki úgyanis az alkalmazásnak ez a része a bonyolultabb a raktár térképek 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 ugyan az mégis nagy eltéréseket tapasztalhatunk a kódbázisban, felépítésben és a fejlesztúk filozofiájában.

4.1.1. Angular

Az Angular a Google által 2010-ben elinditott és a mai napig általuk karbantartott keretrendszer. Filozofiája az, hogy egy általános felhasználásra felkészített alapot ad. Tartalmazza a formok validációját, az állapot kezelést, a routing-ot és a felhasználói input-ok kezelését és ezen kivül 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. Az Angularral szemben a React, filozofiája szerint csak egy könnyű sulyú 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 nyujtott megoldásokat.

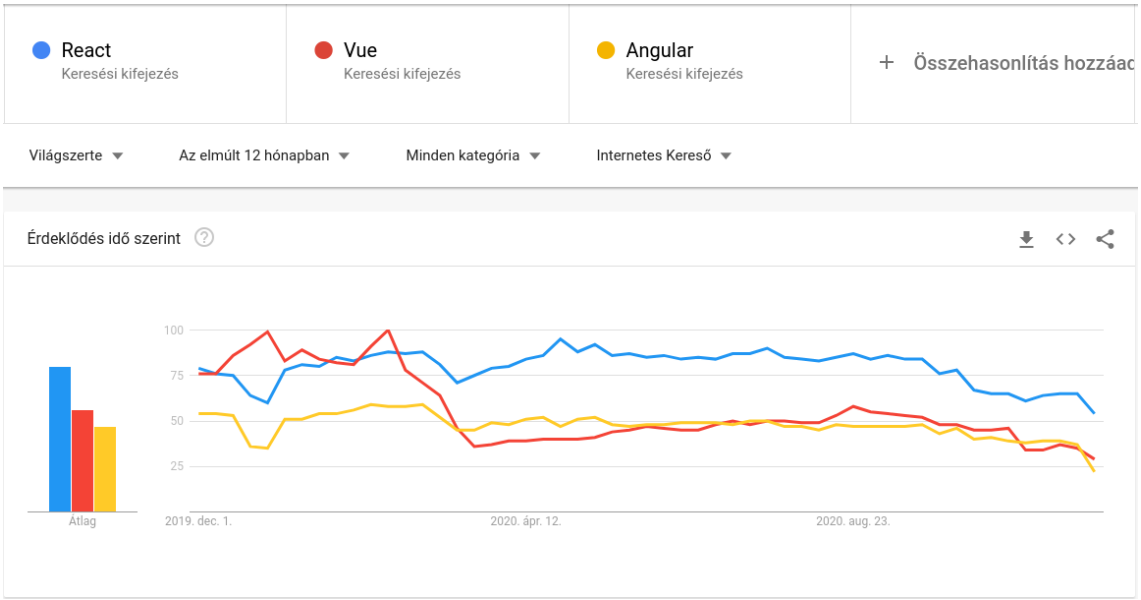
4.1.3. Vue

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 szerves részét képezi a projektnek. Filozofiáját 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 legkönnyeb elkezdeni a

fejlesztést, egy egyszerű alkalmazás elkészítése nem igényel sok ismeretet a HTML, CSS és JavaScript-en kívül. Természetesen komplex alkalmazásokat is készíthetünk vele, ipari környezetben is remekül megállja a helyét.

4.1.4. Trendek

A döntés meghozása elött végeztem egy kisebb kutatást a napjainkban tapasztalható trendekről. Ehhez a Google Trends és az NPM Trends szolgáltatásait vettem igénybe. Előbbi segitségével a Google keresések számát tudjuk összehasonlítani, míg utóbbival az NPM 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.

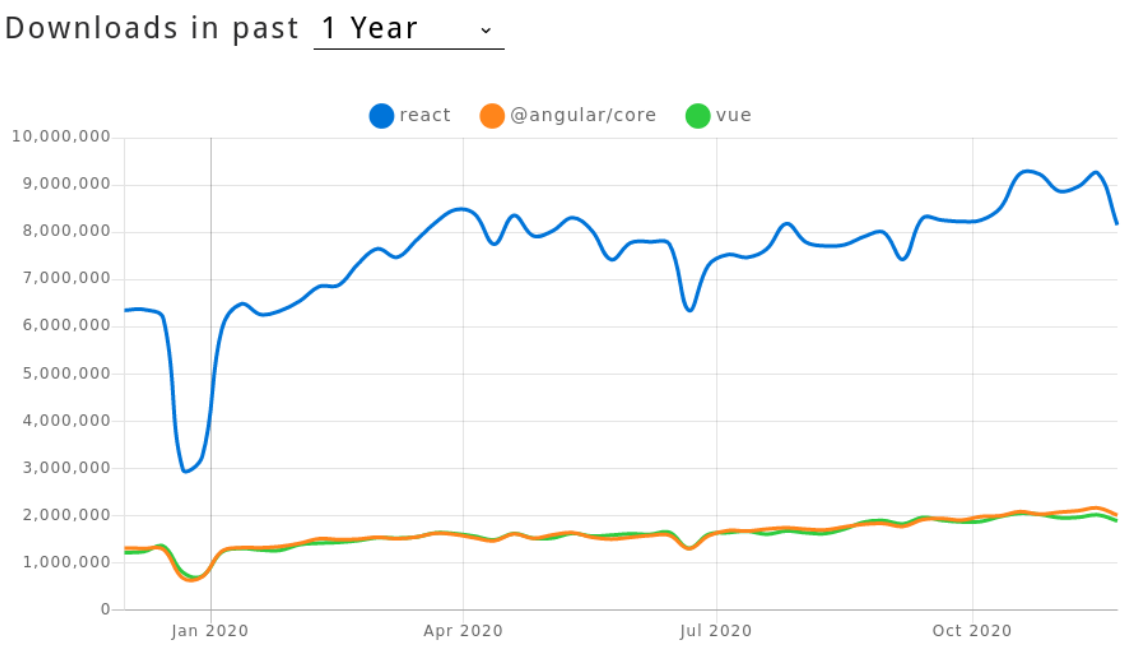
A Google keresések nem mutattak szignifikáns különbséget, azonban az NPM letöltések számában már jelentős eltéréseket tapasztalhatunk. A grafikonról (4.2. ábra) könnyedén leolvasható, hogy több, mint 4-szer annyi a letöltések száma a React esetén, mint a másik 2 keretrendszernél.

4.1.5. Konkluzió

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 során. A döntést segítette, hogy ezzel a keretrendszerrel már volt szerencsém dolgozni és pozitív élmény volt.

4.2. Backend

A kliens oldali technológia kiválasztása után a szerver oldali keretrendszer kiválasztására keürlt sor. Itt sokkal több népszerű és az iparban is használt keretrendszer érhető el. Ilyen például a DotNet, a Spring, a Laravel, a Ruby on Rails, a Django és rengeteg rajtuk kívűl. Ezek összehasonlítása nagyon nehéz, így megpróbáltam egyszerűsíteni a folyamatot. Az elsődleges szempont a kiválasztás során az, hogy a választott frontend technológiával



4.2. ábra. NPM Trends - letöltések összehasonlítása.

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 nyelv azonos frontend és backend oldalon is. Ez a NodeJS. 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 fejlesztet V8 JavaScript motor segítségével teszi lehetővé JavaScript futtatását web böngészőn kívül, így lehetővé téve a nyelv felhasználását backend oldalon is. A NodeJS gyors ütemben fejlödött, 2011-ben már a Microsoft is kivette a részét a fejlesztésből napjainkra pedig az egyik legnépszerűbb technológia webes környezetben.

4.3. Kommunikációs megoldások

A frontend és a backend technológiák kiválasztása után a következő lépés a köztük történő kommunikáció miként jének 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 Átvitel. Ez - ahogy a nevéből is következtetni lehet - probálja kifejező módon átvinni az adatot a kliens és a server 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

4.1. táblázat. Példa egy entitáson végezhető müveletekre 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 resolverekből 3 fő típust különböztetünk meg Query, Mutation és Subscription.

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 megvalósításák 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 választás szempont elsődlegesen az volt, hogy a már kiválasztott technológiákkal együtt a lehető legkényelmesebb és legjobb fejlesztési élményt nyújtsa.

Ennek követelménye az volt, hogy adatbázisok helyett ORM rendszerek összehasonlítását kezdtem 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.

4.4.1. TypeORM

4.4.2. Sequelize

4.4.3. Mongoose

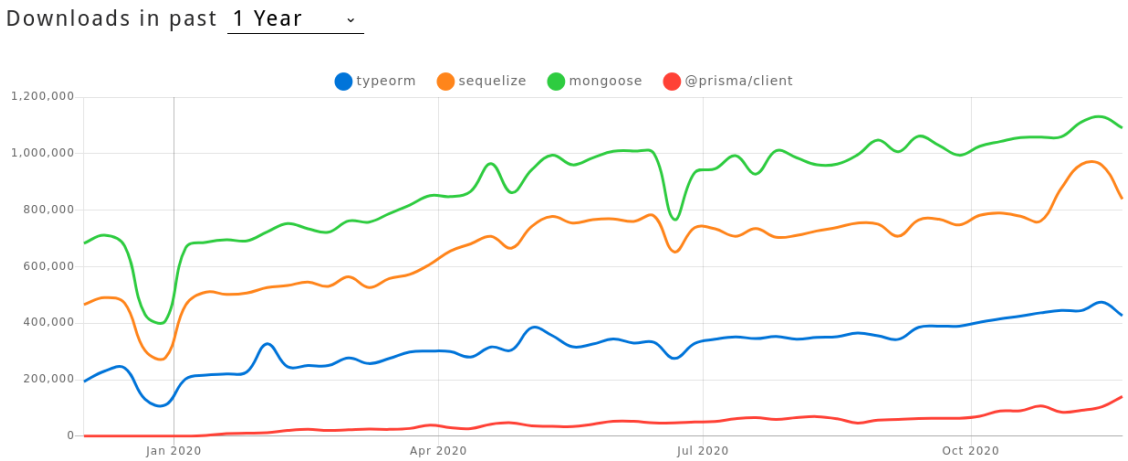
A Mongoose ORM a mongoDB kezeléséhez létrejhozott csomag. A

4.4.4. Prisma

4.4.5. Konkluzió

Habár a trendek (4.3. ábra) alapján a Prisma jócskán el van maradva népszerűségben az összehasonlításban részt vett többi ORM-től mégis erre esett a választásom. A korábban felsorolt elönyök és a növekvő népszerűség miatt úgy érzem, hogy a lemaradása egyedül az újdonságának köszönhető.

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



4.3. ábra. NPM Trends - ORM-ek letöltésének összehasonlítása.

4.5. Közös technológiák

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

4.5.1. TypeScript

A TypeScript egy - a Microsoft által fejlesztett - nyílt forráskódú nyelv, amely JavaScript-et 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 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 complier.

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

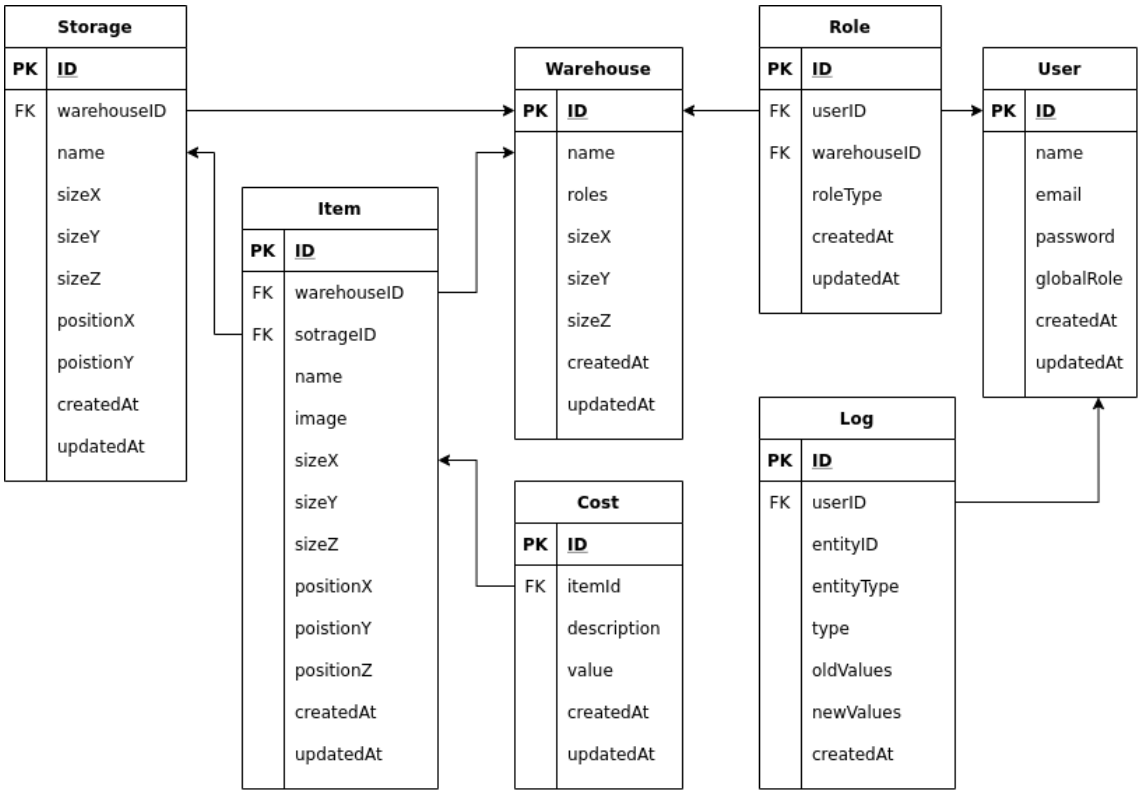
Architektúra

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

5.1. Adatbázis séma

Az adatbázis migrációját nem kellett manuálisan végrehajtanom hála a Prisma-nak. 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 elöl.

Ennek ellenére mégis relációs adatbázist terveztem, majd ezt ültettem át a Prisma által kíván sémába. Tanulmányaim során ezzel a tervezési metodikával találkoztam és olyannyira rögzült, hogy elöszőr nehéz volt kicsit más szemszögből vizsgálni a problémát.



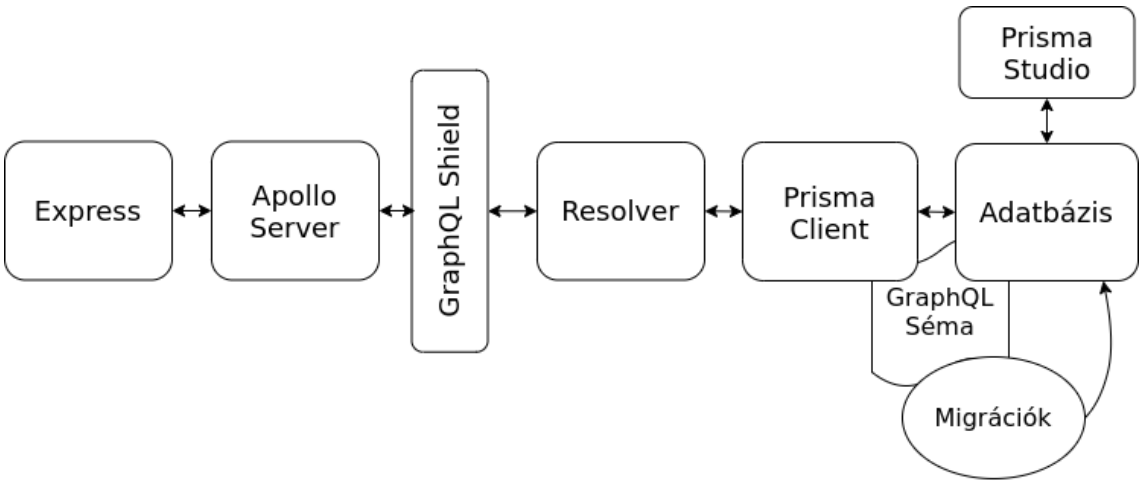
5.1. ábra. Adatbázis séma

Az adatbázis sémát a követelményeket figyelembe véve hoztam létre. Az ábrán (5.1. ábra) átható, 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 mozgatást megoldását szolgálja. Ennek megvalósítására új entitás bevezetése felesleges volt. Opciók között szerepelt még, hogy a tárolók entitásban egy kitüntetett tárolót használunk erre a célra, azonban ez rengeteg elágazást ereményezet volna a kódban. Így egyszerű tervezési döntés értelmében, ezek a mozgatás során nem tartoznak tárolóhoz, csak a raktárhoz, amely ezáltal betölti az edeiglenes tároló szerepét.

A naplózás megoldásánál igyekesztem egy generikus megoldást létrehozni. A napló (Log) tábla csak egy tényleges kapcsolatot tartalmaz a naplóbejegyzés létrejöttét kiváltó felhasználóra mutatva. A későbbi feljesztések 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 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 middelware-ek definiálsára, melyek minden kérés kiszolgálása elött lefutnak. Erre a lehetőségre épít a GraphQL Shield nevű könyvtár, aminek segítségével minden egyes GraphQL műveletre megadhatünk ahhoz szükséges előfeltételeket egyszerű szabályok segítségével. Ilyen szabályokkal valósítottam meg a teljes authorizációt és az authentikáció ellenörzését is.



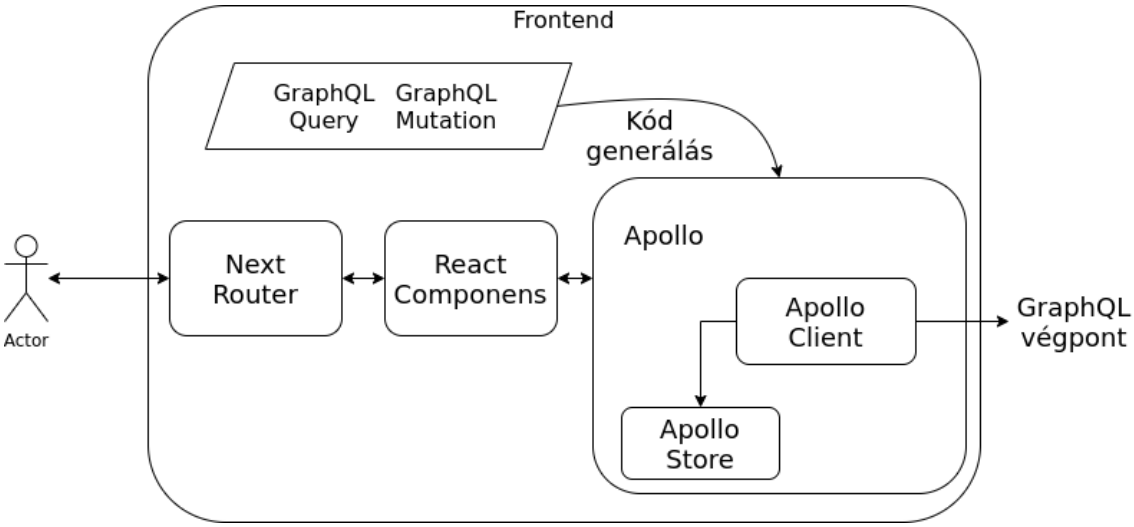
5.2. ábra. Backend felépítése

A megfelelő kódrészlet és a middelware-ek futtatása után, a Prismán keresztül az adatbázishoz fordulunk adat lekérés vagy modósítás 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 érvenyesülni. Fontos, hogy az itt végrehajtott modósítások nem várt működéshez is vezethetnek.

5.3. Frontend felépítése

Ahogy azt a korábbi fejezetekben taglaltam a kliens alkalmazás megvalósításához React-et azon belül pedig NextJS-t használtam. 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 tipus ellenörzéssel írhatjuk a lekérdezéseinket. Az Apollo Client a komm unikáció mellett a gyórsírótárazást 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 típusokkezelés is megvalósított



5.3. ábra. Frontend felépítése

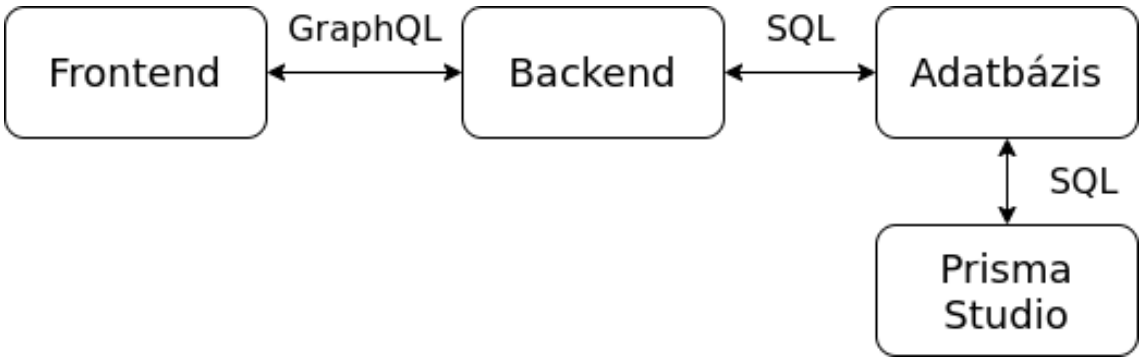
A React-ben a komponensek közötti logika megosztását a Context-ek segítségével valósítothatjuk meg. Ennek használaával nem szükséges minden kompononsnek átadni manuálisan a szükséges változókat, hanem elegendő csak egy vagy szükség esetén több provider-t az alkalmazásunk köré rakni. A provider-ek segítségével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

```
<ApolloProvider client={client}>
  <ChakraProvider>
    <AuthProvider>
      <Layout>
        <Component {...pageProps} />
      </Layout>
    </AuthProvider>
  </ChakraProvider>
</ApolloProvider>
```

5.1. lista. Frontend-hez használt provider-ek

5.4. Architektúra összefoglalása

Tehát 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 Cliens és az Apollo Server között. A backend és az adatbázis kommunikációja pedig SQL segítségével történik, azonban ezt a Prisma teljesen elfedi.



5.4. ábra. Teljes alkalmazás felépítése

6. fejezet

Fejlesztést segítő eszközök

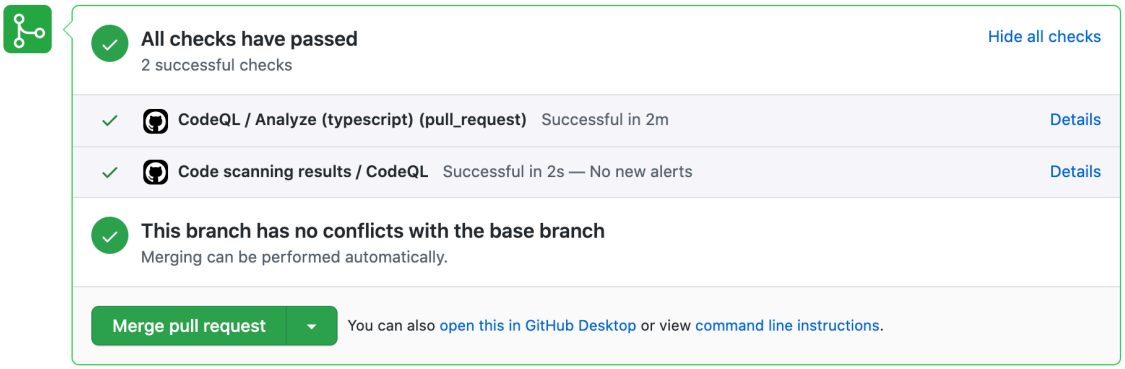
A fejlesztés során igyekeztem minél több olyan eszközt használni, ami elősegíti a munkát és javítja a kódminőséget és biztonságot.

6.1. Continuous Integration

A Continuous Integration (rövidítve CI) napjainkban már elengedhetetlen része a fejlesztési folyamatoknak. Rengetek megoldás létezik, azonban én a GitHub Action mellett tettem le a voksomat. Ennek oka, hogy a GitHub-ot használtam a verziókezelt kódom tárolására, így kézen fekvő 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 ugyan azon 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

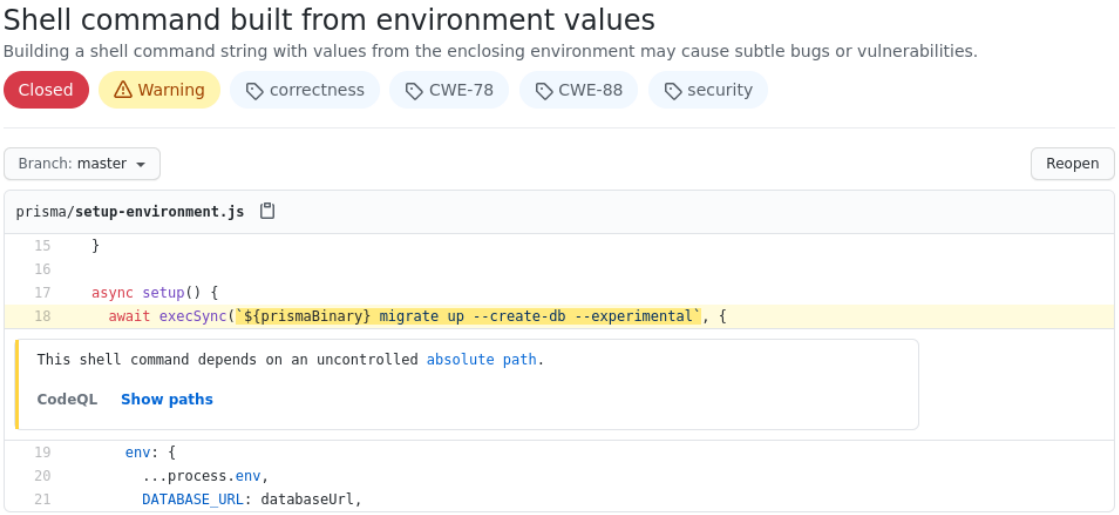
6.1.1. Linter

A linter egy viszonylag gyors lefutású és kevés erőforrást igénylő action, ezért beállítása szerint bármilyen commit kerül a GitHub repository-ba azonnal elindul.

6.1.2. Statikus kódellenörzés

A statikus kódellenörzést 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 a főágba történik a commit vagy ha Pull Request-et kezdeményezek, 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 sérülékny kódrészlet az alkalmazása futása közben nem érhető el, ugyanis a teszt környezet kiallakitására szolgál.



6.2. ábra. Code scanning figyelmeztetés

6.2. Git hook

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

```
"husky": {
  "hooks": {
    "pre-commit": "lint-staged"
  }
},
"lint-staged": {
  "*.ts": [
    "eslint --fix"
  ]
}
```

6.1. lista. Pre-commit hook beállításai

6.3. Continuous Deployment

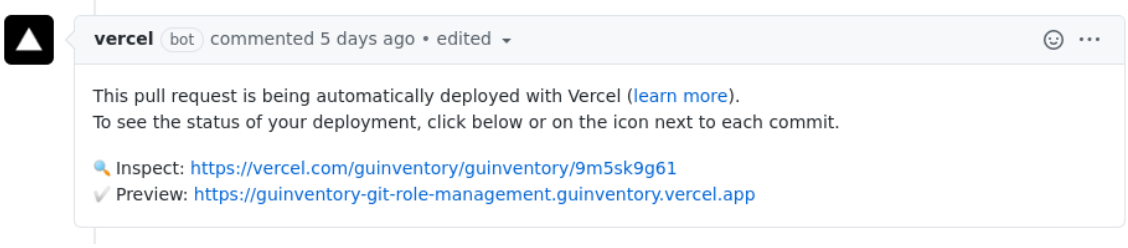
A Continuous Integration melett elengedhetetlen része a fejlesztésnek a Continuous Deployment (röviden CD). A CD az alkalmazás folyamatos kitelepítését jelenti, hogy a kód felöltése után szinte azonnal elérhető legyen az alkalmazás legfrissebb változata.

6.3.1. Heroku

A backend alkalmazás üzemeltetésére a Heroku szolgáltatásai 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 CI 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 ellenörzését. 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

7. fejezet

Alkalmazás fejlesztése és működése

7.1. Backend

A backend architekurális felépítéséről a korábbi fejezetben volt szó. Most térjünk át az egyes rétegek és egyes funkciók megvalósításának bemutatására.

7.1.1. GraphQL Playground

A legtöbb GraphQL server-hez lehetőségünk nyilik valamilyen interaktív GraphQL szerkesztő felület kiszolgálásra is. Az alkalmazásban én a GraphQL Playground-ot használtam, a konfigurációt úgy valosítottam meg, hogy éles környezetbe ne szolgálja ki ezt a felületet, csak feljesztői környezet estén. Ezt környezeti változók segítségével oldaottam meg.

GraphQL Playground es dokumentáció

7.1.2. Modellek és kapcsolatok

modellek és kapcsolatok

7.1.3. Resolver felépítése

resolver felépítése

7.1.4. Authentikáció és authorizáció

GraphQL Shield és JWT

shield szabályok

JWT Ábra, token kezelés, kód magyarázat

7.2. Frontend

7.2.1. Útvonalválasztás

NextJS routing mappa szerkezet lista/kép

7.2.2. Kódgenerálás

Apollo codegen

hook példa, és használata

7.2.3. Validáció

Yup validációs séma és react form hook

7.2.4. Felhasználói felület

ChakraUI
: screenshotok az alkalmazásból példa kód téma definiálás dark mode

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 máködő funkciók válnak használhatatlanná új funkciók bevezetése közben. Ennek a kockázatát megfelelő tesztlefedetséggel minimálisra csökkenthetjük.

8.1. Frontend tesztelés

A frontend teszteléséhez úgy nevezett end-to-end teszteket készítettem. Az end-to-end tesztek (röviden E2E) esetén a teszt a felhasználó viselkedését szimulálja. Egy szimulált böngészőben végzi el a tesztet és tényleges kattintás eseményt vált ki és figyeli a kirenderelt képet a tesztnek megfelelő egyezést keresve.

8.2. Backend tesztelés

A Prisma fejlesztú csapata csupűn pár hónappal ezelött jelentette be a 2.0-ás verziót. A folyamatos fejlesztés ellenére is még hiányos az eszközkészlete. Ennek köszönhetően teszteléshez sem kinál semmilyen megoldást.

A tesztelési környezet kialakításház 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 dolgozom. A tesztelés előtt egy scripttel automatikusan létrehozom a szükséges adatbázist, majd a tesztek végeztével törlöm azt, így biztosítva, hogy véletlenül se maradjon semmilyen adat az adatbázisba, ami esetleg fals ereményt váltana ki a tesztekből.

A tesztek során a GraphQL resolver-eket hívtam meg különböző query-kel és mutation-ökkel.

```
await execSync(`${prismaBinary} migrate up --create-db --experimental`, {
    env: {
    ...process.env,
    DATABASE_URL: databaseUrl,
    },
})
```

8.1. lista. Teszt atadbázis migrció

```
const client = new Client({
    connectionString: databaseUrl,
```

```
3 })
4 await client.connect()
5 await client.query(`DROP SCHEMA IF EXISTS "test" CASCADE`)
6 await client.end()
```

8.2. lista. Teszt adatbázis törlése

```
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 })
```

8.3. lista. Regisztráció első teszt eset

9. fejezet

Üzemeltetés

Korábbi fejezetben írtam arról, hogy az üzemeltetést Heroku és Vercel segítségével oldottam meg. Azonban ez az alkalmazás tényleges üzembehelyezése után nem feltétlenül a legkölcsékhatékényabb módja az üzemeltetésének. Ezért igyekeztem alternatívákkat kínálni ay 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 cachelésénel maximális kihasználása. Különösen nagy segítség lehet, ha úgyan azt a verziót több környezetbe is szeretnénk élesíteni.

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

WORKDIR /opt/app
COPY package.json yarn.lock ./
RUN yarn install --frozen-lockfile

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

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

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

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

9.1. lista. Frontend Dockerfile

A backend-hez tartozó konténer 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.

```
version: '3'
services:
  server:
    build: .
    links:
      - mysql
    depends_on:
      - mysql
    ports:
      - '${SERVER_PORT}:4000'
      - '${STUDIO_PORT}:5555'
    environment:
      NODE_ENV: ${NODE_ENV}
      DATABASE_URL: ${DATABASE_URL}
      APP_SECRET: ${APP_SECRET}
    networks:
      - server-network
  mysql:
    image: mysql:5.7
    restart: always
    environment:
      MYSQL_USER: prisma
      MYSQL_PASSWORD: prisma
      MYSQL_ROOT_PASSWORD: prisma
      MYSQL_DATABASE: prisma
    networks:
      server-network:
        aliases:
          - mysql.db
    volumes:
      - db_data:/var/lib/mysql
volumes:
  db_data:
networks:
  server-network:
```

9.2. lista. Backend docker compose

10. fejezet

Összefoglalás

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

10.1. Tovább fejlesztési lehetőségek

A félév során temérdek új fejlesztési lehetőség jutott eszembe, melyeket megvalósítva egy még jobb és méginkább a piaci igényeknek megfelelő alkalmazást kaphatunk. A szakdolgozat keretein belül igyekeztem az elengedhetetlen funkciókat a lehető legjobban megvalósítani és inkább a fejlesztői eszköz tár felépítését tartottam fontosnak, mint a rengeteg funkció belezsúfolását egy alkalmazásba. Ennek köszönhetően remélhetőleg egy hosszú távon is fejleszthető és fentartható alkalmazás születet. A félév végeztével folytatnám a munkát, hogy a lehető legtöbb piaci igényt legyen képes kiszolgálni az alkalmazásom.

10.1.1. Keresés

A jelenlegi keresés egy nagyon egyszerű string összehasonlításon alapul, már egyetlen karakter elgépelése esetén sem talál egyezést. Ennek a problémának a megoldására több lehetőség is kinálkozik, 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 félév elején felvetödött 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 az volt, hogy kategoriákat hozhattunk volna létre minden raktáron belül. A kategoriákhoz egy név megadása után felvehetőek lettek volna a tulajdonságok nevei és típusai. Ezután az eszköz felvételekor kiválasztjuk, hogy milyen kategoriába, kategoriákba tartozik. Így a kategoriák révén már tudjuk azt, hogy milyen egyedi tulajdonságai lehetnek.

Ábrák jegyzéke

Táblázatok jegyzéke

Irodalomjegyzék

Függelék