



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

Online könyvkölcsönző alkalmazás készítése a Simonyi Károly Szakkollégium számára

SZAKDOLGOZAT

Készítette
Fehér János

Konzulens
dr. Ekler Péter

2020. december 10.

Tartalomjegyzék

Kivonat	i
Abstract	ii
1. Bevezetés	1
2. Specifikáció	2
2.1. Felhasználókezelés	2
2.2. Könyvek listázása	2
2.3. Kosár	2
2.4. Kölcsönzés kezelése	2
2.5. Admin funkcionalitás	2
2.6. Use-case diagram	3
3. Használt technológiák kiválasztása	4
3.1. Verziókezelés	4
3.2. Frontend	4
3.2.1. JavaScript keretrendszer	4
3.2.2. CSS keretrendszer	4
3.3. Backend	5
3.4. Adatbázis	5
3.4.1. Az adatbázis elérése	5
3.5. REST és GraphQL	6
3.5.1. Kommunikáció megvalósítása	6
3.6. TypeScript	6
4. Az alkalmazás felépítése	8
4.1. Adatbázisséma	8
4.2. A backend felépítése	8
4.2.1. next-connect	9
4.2.1.1. Middleware támogatás	10
4.2.2. Kommunikáció az adatbázissal	10
4.2.3. Kapcsolódás az adatbázishoz	12
4.2.4. Az adatok elérése	12
4.3. A frontend felépítése	13
4.4. Kódmegosztás	13
4.5. Képfeltöltés	13
5. Az alkalmazás funkcióinak megvalósítása	14
5.1. Authentikáció	14
5.1.1. Megvalósítás	14

5.2.	Authorizáció	14
5.2.1.	Megvalósítás	15
5.3.	Könyvek listázása	16
5.3.1.	Részletes nézet	16
5.3.2.	Keresés a könyvek között	16
5.4.	Kosár	18
5.5.	Foglalási folyamat	19
5.5.1.	Foglalás kezelése	20
5.6.	Admin funkciók	21
5.6.1.	Könyvek kezelése	21
5.6.2.	Kategóriák kezelése	22
5.6.3.	Foglalások kezelése	22
5.7.	Reszponzív UI	22
5.8.	Dark Mode	23
5.9.	Validáció	23
6.	Tesztelés	27
7.	Fejlesztést segítő eszközök	29
7.1.	Continuous Integration	29
7.1.1.	Statikus kódanalízis	29
7.2.	Continuous Delivery	29
7.3.	Prisma Studio	30
7.4.	Dependabot	30
8.	Összefoglalás	31
8.1.	Továbbfejlesztési lehetőségek	31
8.1.1.	WebSocket	31
8.1.2.	Értesítés rendszer	31
8.1.3.	Keresés továbbfejlesztése	31
8.1.4.	Auth.SCH integráció	31
	Köszönetnyilvánítás	33
	Ábrák jegyzéke	34
	Irodalomjegyzék	34
	Függelék	36
F.1.	Telepítési útmutató	36
F.2.	Az alkalmazás elérhetősége	36

HALLGATÓI NYILATKOZAT

Alulírott *Fehér János*, 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 10.

Fehér János
hallgató

Kivonat

Manapság ha egy többfelhasználós, dinamikus tartalmat megjelenítő szolgáltatásra van szükségünk, önkéntelenül is a webes technológiák felé fordulunk. Ez nem véletlen, ugyanis a rendkívül gyorsan fejlődő webes technológiák lehetőséget nyújtanak arra, hogy kényelmes, gyors, megbízható és jó felhasználói élményt nyújtó megoldásokat készítsünk általuk.

Szakkollegzatomban a Simonyi Károly Szakkollégiumnak elkészített könyvkölcsönző alkalmazáson keresztül szeretném bemutatni ezeknek a technológiáknak egy szeletét és betekintést nyújtani a fejlesztés menetébe.

A szakkollégiumhoz tartozó könyvtár régóta nyitva áll a hallgatók előtt hasznos illetve szórakoztató irodalmat kínálva. Ennek elérése illetve karbantartása azonban ezidáig nehézkes és bonyolult volt, azonban szerintem nagy potenciál rejlik benne, ha ezt mind a hallgatók, mind az üzemeltetők egy kényelmesen használható felületek keresztül érhetik el.

A dolgozat ennek a szoftvernek a létrejöttét mutatja be. Ebbe beletartozik a megfelelő technológia kiválasztása – külön kitérve a Next.js által szolgáltatott funkciókra –, valamint az igények pontosítása a specifikáció által. Ezután a szoftver felépítéséről valamint a megvalósított funkciók működéséről lesz szó.

Végül az elkészült alkalmazás tesztelése, a fejlesztés segítő eszközök bemutatása, valamint a potenciális továbbfejlesztési lehetőségek kerülnek bemutatásra.

Abstract

Nowadays if someone wanted to make a multi-user solution that can provide dynamic data for the users, can't look away from the web. This is because today's web technologies allow the developers to create easy-to-use, fast and reliable applications that provide a great user experience.

In my thesis I'd like to demonstrate a portion of these technologies and the development process through a web-based book renting application created for Simonyi Károly College for Advanced Studies.

The library of Simonyi Károly College for Advanced Studies is open to students for renting books that can help them in their studies or provide other kinds of literature. In its current form the handling of the library proved to be difficult and complicated.

The goal of my thesis is to provide an easy-to-use and accessible web-based application to ease the access to the library for end-users and make managing the books and orders simpler and easier for college members. With this system using the library will be more straightforward and more students could be potentially reached about this service.

The goal of this thesis is to guide the Reader through how this application was made from scratch. This includes choosing the most appropriate frameworks and libraries – with extra details about Next.js and its provided services – and defining the exact demands through specification.

Next the dissertation will show the application's structure and how the functionalities defined in the specification work.

Lastly the thesis presents how the app was tested, what other technologies were used to make the development phase easier and how the application can be further optimized and developed.

1. fejezet

Bevezetés

A Simonyi Károly Szakkollégiumban jelenleg üzemel egy könyvtár, ahonnan a hallgatók különféle tankönyveket és szórakoztató irodalmat van lehetőségük kölcsönözni.

A jelenlegi megoldás azonban nehézkes és nehezen fenntartható. Az adminisztráció egy megosztott Google Docs táblázaton keresztül történik. Itt ömlesztve találhatóak a könyvek, azoknak a darabszáma és aktuális állapota (kölcsönözhető, kiadva). Ez a táblázat továbbá nem nyilvános, tehát a végfelhasználók nem tudják ellenőrizni egy adott könyv elérhetőségét, illetve foglalást is csak email-en keresztül tudnak leadni.

Ez nagyban hátráltatja mind a könyvtár hirdetését a hallgatók számára, valamint lassúvá és bonyolulttá teszi a könyvtár menedzselését.

A fentebbi szituáció ösztönzött arra, hogy a jelenleginél optimálisabb megoldást keressek a könyvtár üzemeltetésére.

Önálló laboratóriumom és szakkollégiumi tagságom során megismerkedtem több webes technológiával, különös figyelmet szentelve a manapság népszerű frontend technológiákra.

Ennek hatására kifejezetten frondend-fókuszú megoldást szerettem volna készíteni a minél jobb felhasználói élmény érdekében. Ennek megfelelően a választásom a Next.js keretrendszerre és a serverless backend megoldásra esett. Mivel a kezdetektől TypeScript nyelvvel ismerkedtem meg és ezt használtam evidens volt, hogy a szakdolgozatomban is a TypeScript-et és az azt támogató megoldásokat részesítem előnyben. Emiatt a backend és az adatbázis kapcsolatát biztosító könyvtárak közül a Prisma került ki nyertesnek.

A dolgozatomban a továbbiakban ennek a szoftvernek a megtervezéséről és működésének bemutatásáról fog szólni.

A második fejezetben a feladat által megkövetelt funkcionalitásokat. Ezek között szerepel a felhasználók kezelése, kölcsönzés leadása és állapotának követése, valamint a könyvek, kategóriák és kölcsönzések adminisztrátor oldalról történő menedzselése.

A harmadik fejezetben bemutatom az általam az alkalmazás egyes rétegeihez használt technológiák, valamint az egyes rétegeket összekötő könyvtárak kiválasztási folyamatait és a végső soron kiválasztott technológiákat.

A negyedik fejezet a weboldal magasszintű architektúráis felépítéséről értekezik. Bemutatom az egyes rétegek felépítését, illetve hogy az előző fejezetben kiválasztott könyvtárak hogyan működnek együtt az alkalmazás különböző részeivel.

Az ötödik fejezet az alkalmazás által megvalósított funkciókat és az ehhez szükséges lépéseket és használt technikákat tartalmazza.

A hatodik fejezetben az alkalmazás tesztelése, a kiválasztott keretrendszer bemutatása és a tesztek felépítése szerepel.

Ezek után röviden összefoglalom az elkészült alkalmazást és kitérek a további fejlesztési lehetőségekre.

2. fejezet

Specifikáció

Az alábbi fejezet az alkalmazás alapvető működéséhez szükséges funkciókkal foglalkozik. Ezeknek korai lefektetése és tisztázása fontos, hogy az implementáció során ne essünk overengineering[5] vagy a hibásan implementált funkcionalitás csapdájába.

2.1. Felhasználókezelés

Az alkalmazásra legyen lehetőség regisztrálni, illetve a regisztrált felhasználóknak bejelentkezni, valamint a már bejelentkezett felhasználóknak kijelentkezni. A regisztráció során a név, email cím és jelszó megadása kötelező, egy email címmel csak egy fiók hozható létre. Bejelentkezni a név és email cím megadásával lehet.

2.2. Könyvek listázása

A rendszer biztosítja az elérhető könyvek listázását, a könyvek közötti kulcsszavas keresést, valamint egy kiválasztott könyv részletes nézetét.

2.3. Kosár

A könyveket a webshopokban megszokott módon kosárba helyezhetünk (egy könyvből akár többet, de maximum az elérhető mennyiségnek megfelelőt). A kosár tartalmát lehetőségünk van listázni, a benne lévő elemeket szerkeszteni (darabszámot növelni és csökkenteni, könyvet eltávolítani, vagy a teljes kosarat kiüríteni).

2.4. Kölcsönzés kezelése

A kosárba helyezett könyveket le tudjuk adni a visszavitel időpontjának megadásával, ekkor a kölcsönzés „Függőben” állapotba kerül.

Ezután a kölcsönzés oldalán az alkalmazás adminisztrátoraival lehetőség van átvételi időpont egyeztetésére a kölcsönzéshez fűzhető kommenteken keresztül. A kölcsönzést illetve a hozzá tartozó kommenteket csak az kölcsönző, illetve megfelelő jogosultsággal rendelkező adminisztrátor láthatja

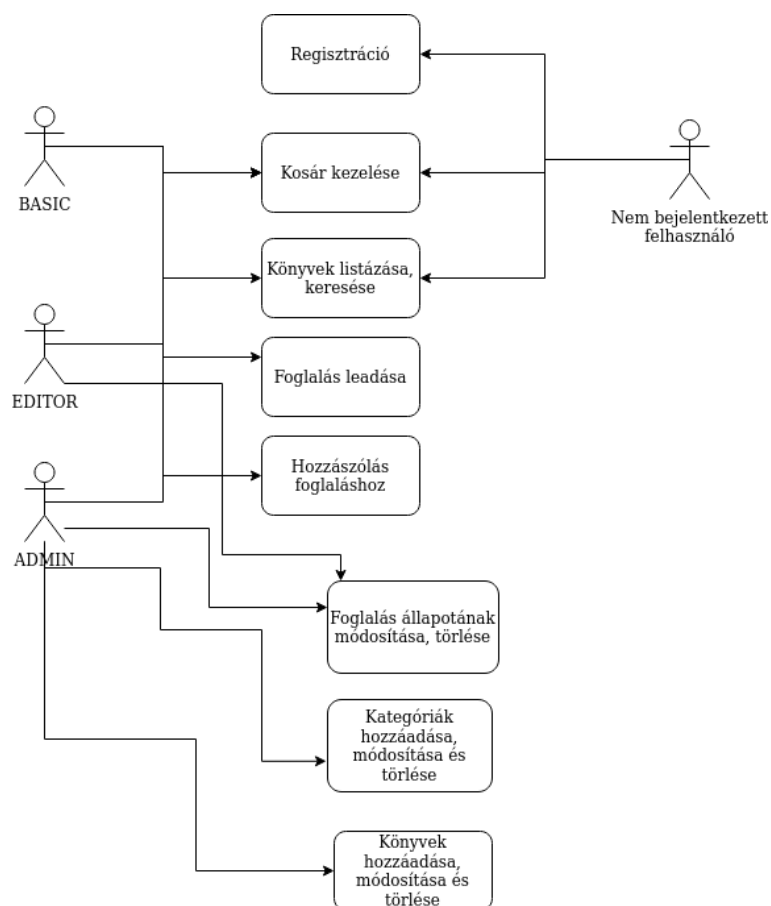
2.5. Admin funkcionalitás

Az alkalmazás adminjainak lehetősége van könyvek illetve kategóriák hozzáadására, szerkesztésére és törlésére.

Ezen kívül frissíthetik a kölcsönzés állapotát, törölhetik azt, valamint hozzászólást fűzhetnek a kölcsönzésekhez.

2.6. Use-case diagram

A fenti követelményeket az alábbi use-case diagramban foglaltam össze. Ebben megkülönböztetek nem bejelentkezett illetve bejelentkezett felhasználót (utóbbi mindenképp rendelkezik egy szerepkörrel).



2.1. ábra. Az alkalmazáshoz tartozó use-case diagram

3. fejezet

Használt technológiák kiválasztása

Az alábbi fejezetben az alkalmazás egyes részeihez elérhető és végső soron kiválasztott technológiákat mutatom be.

A technológiák kiválasztásakor fontos szempont volt a Developer Experience (DX) [2], azaz hogy a használt eszköz minél kevésbé legyen megterhelő a használója számára.

Ez takarhatja például a kiterjedt és jó minőségű dokumentációt, a megfelelő IDE integrációt, vagy hogy nem kell órákat tölteni az egyes funkciók konfigurálásával.

Fontos szempont volt ezen kívül az adott technológia időtállósága. Mivel a tervek szerint az alkalmazás a jövőben aktívan használva és fejlesztve lesz akár több fejlesztő által kulcsfontosságú volt a követhető és karbantartható kódbázis és az alkalmazás életciklusának nyomonkövethetősége.

3.1. Verziókezelés

A fejlesztés során fontos a változások átlátható, könnyű követése, a szoftver különböző verzióinak követése.

Erre a mára már kvázi ipari sztenderdnek számító `git` nevű szoftvert vettem igénybe. A forráskód felhőben történő tárolására a GitHub szolgáltatását használtam.

3.2. Frontend

3.2.1. JavaScript keretrendszer

A frontendhez használt technológia kiválasztásánál két fő szempontot tudunk megkülönböztetni. Az egyik az egyes backend rendszerek által támogatott, szerveroldalon renderlet, template engine-t használó megoldás, a másik a külön frontend framework használata. Utóbbi jóval nagyobb szabadságot és funkcionalitást ad a fejlesztő kezébe, és lehetőséget ad a legújabb technikák és könyvtárak használatára.

Emiatt úgy döntöttem, hogy az alkalmazás ezen részét a négy legnépszerűbb keretrendszer (React, Angular, Vue és Svelte) egyikével fogom megvalósítani. Ezek a keretrendszerek alapvető filozófiában és funkcionalításban lényegében megegyeznek, azonban az elérhető könyvtárak mennyisége és minősége a React esetében a legnagyobb, emiatt végső soron erre esett a választásom. [8]

3.2.2. CSS keretrendszer

A CSS keretrendszer kiválasztása során két fő irányvonal jött szóba: az ún. atomic CSS, illetve az előre már elkészített komponenseket kínáló könyvtárak. Míg előbbi lehetővé teszi

a teljesen egyedi design írását, utóbbi jóval nagyobb fokú kényelmet és a felület gyorsabb elkészítését teszi lehetővé.

Mivel jelen alkalmazás esetében nem volt szempont egy egyedi design elkészítése, ezért az utóbbi megoldás használata mellett döntöttem. A kiválasztott framework végül a Chakra UI lett, mely nagy mennyiségű kész, egyszerűen bővíthető és magas fokú accessibility-t kínáló komponensekkel rendelkezik. [1]

3.3. Backend

A backend keretrendszer kiválasztása során a legfontosabb szempont az volt, hogy minél jobban képes legyen integrálódni a frontend keretrendszerhez, emiatt mindenképpen szerettem volna elkerülni a külön repository használatát. Ennek fő okai, hogy mind a fejlesztés, mind a deployment jelentősen egyszerűsödik, valamint közös könyvtár esetén a közös kódrészletek megosztása is triviálissá válik.

A fentiek miatt a lehetőségeim a NodeJS alapú megoldásokra szűkítettem le. Egy népszerű, és általam már kipróbált megoldás az Express keretrendszer, azonban a React világában létezik egy, az igényeimnek méginkább megfelelő framework: a NextJS. Ez egy React-ra épülő, SSR-t¹ támogató keretrendszer, amely rendelkezik egy úgynevezett API routes nevű funkcióval.

Ennek segítségével a React alkalmazásunkon belül készíthető egy API réteg, amely a fordítás után serveroldalon futó függvényekké lesz átalakítva. Ezzel lényegében megspóroljuk, hogy külön API-t és hozzávaló elérési, illetve deployment környezetet kelljen létrehozni, miközben egy Express-hez hasonló interfészt tudunk használni. Előnye továbbá, hogy rendkívül egyszerűvé teszi a frontend és a backend közötti kódmegosztást, csökkentve ezzel a duplikációkat és erősítve a type safety-t.[4]

3.4. Adatbázis

Az adatbázis architektúra kiválasztása esetén két fő irányvonal volt a meghatározó: a hagyományos relációs adatbázis (pl. PostgreSQL, MySQL) és a NoSQL megoldások (pl. MongoDB, Google Firestore, AWS DynamoDB).

A technológia kiválasztása során fontos szempont volt az ACID elvek követése, a relációk egyszerű kezelése és a minél nagyobb típusbiztosság elérése a backend kódjában.

Ezeknek a szempontoknak a hagyományos relációs adatbázisok felelnek meg, ezen belül a PostgreSQL-re esett a választásom, mivel ez egy általam már ismert, sok funkciót támogató, ingyenes és nyílt forráskódú megoldás. [6]

3.4.1. Az adatbázis elérése

Miután a backend és adatbázis technológiák és könyvtárak kiválasztásra kerültek, szükséges a kettő közötti kommunikációt biztosító könyvtár meghatározására. NodeJS környezetben rendkívül nagy választék áll rendelkezésre, ezeket két nagy csoportra lehet bontani: ORM és query builder.

Míg az előbbi megoldás egy erős absztrakciós réteget képez az adatbázis szerkezete és a programkód között, addig a query builder egy, az SQL-hez közelebbi interfészt kínál a felhasználónak. Ez utóbbi előnye, hogy a felhasználó által írt kód közelebb van a végső soron lefutó SQL-hez, így átláthatóbbak az egyes lekérések végeredményei.

A query builder megoldások közül is kiemelkedik a Prisma, amely egy NodeJS környezetben működő adatbázis kliens, ami a hangsúlyt a type safety-re helyezi. Az

¹Server-Side Rendering

adatbázis sémát egy speciális `.prisma` kiterjesztésű fájlban tudjuk megírni, majd ezt a Prisma migrációs eszközzel tudjuk az adatbázisunkba átvezetni. Ezután lehetőségünk van a séma alapján a sémából TypeScript típusokat generálni, melyek használata nagyban megkönnyíti és felgyorsítja a fejlesztés menetét.

```
const user = new User()
user.firstName = "Bob"
user.lastName = "Smith"
user.age = 25
await repository.save(user)
```

3.1. lista. Adatbázis beillesztés TypeORM környezetben

```
const user = await prisma.user.create({
  data: {
    firstName: "Bob",
    lastName: "Smith",
    age: 25
  }
})
```

3.2. lista. Adatbázis beillesztés Prisma segítségével

A fentiek alapján a Prisma könyvtárat használtam az adatbázissal történő kommunikációra. [7]

3.5. REST és GraphQL

A fentiekén túl utolsó lépésként szükséges volt eldönteni a frontend és a backend közötti kommunikáció módját.

A REST architektúra mellett ugyanis a közelmúltban megjelent a GraphQL. Ez egy, a REST API-nál flexibilisebb megoldást kínál az adatok elérésére. Hátránya azonban, hogy sok extra kód és konfiguráció szükséges a használatához. Ezzel szemben – Next.js API routes használatával – egy REST API végpont létrehozásához csupán a kiválasztott névvel kell egy fájlt helyezni az `src/pages/api` könyvtárba.

A fenti szempontokat figyelembe véve végül a hagyományos REST architektúra alkalmazása mellett döntöttem. [9]

3.5.1. Kommunikáció megvalósítása

A HTTP kérések küldéséhez rengeteg megoldás közül tudunk válogatni. A legegyszerűbb a böngészőbe beépített `fetch` API[3], ami egy egyszerű interfészt biztosít, azonban funkciókat tekintve erősen korlátozott.

A választásom végül a Vercel által fejlesztett SWR-re[10] esett. Ez egy React könyvtár, ami a `stale-while-revalidate` stratégiát alkalmazza. Segítségével könnyedén tudjuk a változtatásokat gyorsan és reaktívan követni a frontenden, miközben fejlett `cache` funkcionalitást is nyújt.

3.6. TypeScript

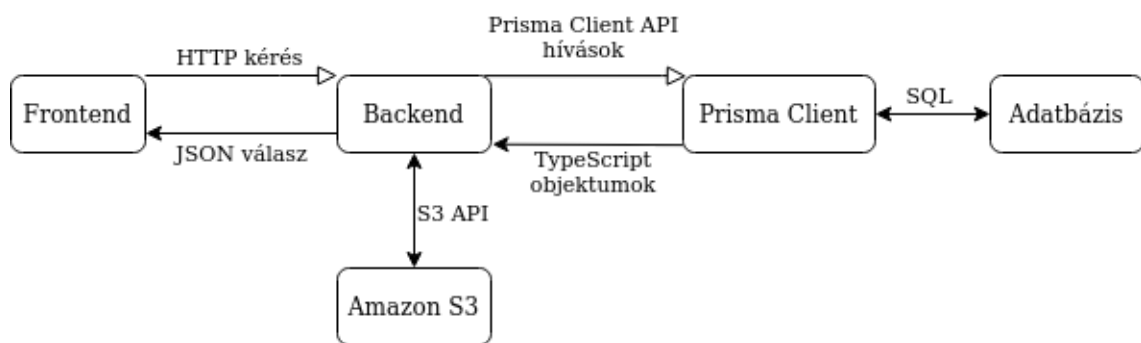
A frontend illetve a Node.js világában lehetőségünk van arra, hogy a JavaScript helyett egy arra lefordítható egyéb nyelvet használjunk fejlesztés közben. Az egyik ilyen lehetőség a TypeScript[11], amely egy már több éve jelenlévő, gyorsan fejlődő alternatíva többek között olyan funkciókkal, mint a `static type check`, `type inference` és magas szintű IDE integráció.

Az általam választott technológiák közül a Prisma kifejezetten épít a TypeScript nyújtotta előnyökre, illetve a Next.js integrációhoz elegendő egy konfigurációs fájlt létrehozni, így evidens volt, hogy az alkalmazást TypeScript segítségével írjam meg.

4. fejezet

Az alkalmazás felépítése

Az alkalmazás három fő rétege az adatbázis, a backend és a frontend. Az alábbi fejezet ezen három réteg architektúráis felépítéséről valamint az egyes rétegek összeköttetéséről foglalkozik.



4.1. ábra. Az alkalmazás high-level architektúrája

4.1. Adatbázisséma

Az adatbázisséma tervezéséhez a dbdiagram.io nevű platformfüggetlen, webes ER diagram tervező szoftvert használtam. Ez egy saját fejlesztésű, DBML nevű DSL nyelvet használ a séma leírására, és lehetővé teszi ennek exportálását különféle formátumokba.

A séma tervezése során a Prisma által használt elnevezési konvenciókat használtam megkönnyítve a két technológia közötti átjárhatóságot.

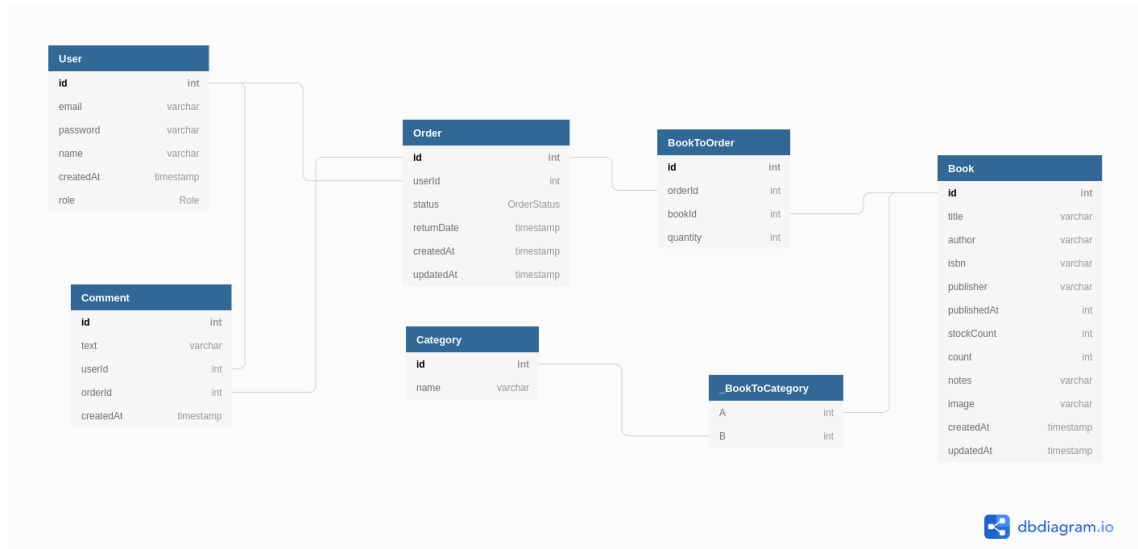
4.2. A backend felépítése

Az alkalmazás backendje eltér a hagyományos backend API-tól. A Next.js API routes által a backendünket úgynevezett cloud function-ök alkotják.

Ennek segítségével a `pages/api` mappába helyezett fájljaink szolgálnak backendként. Minden ide helyezett fájl egyben a nevének megfelelő REST API végpont lesz, tehát például a `pages/api/books.ts`-ben lévő kódot a `pages/api/books` URL-en keresztül tudjuk elérni.

Támogatja továbbá a backend-oldali dinamikus routing-ot, azaz például a `pages/api/books/[id].tsx` fájl a `pages/api/<id>` URL-nek felel meg, ahol az `id` változót alábbi módon érhetjük el:

```
export default function handler(req, res) {
  const {
    query: { id },
```



4.2. ábra. Az adatbázisséma ER diagramja.

```

    } = req
    res.end(`Book: ${id}`)
  }

```

4.1. lista. Next.js dinamikus routing

4.2.1. next-connect

Alapesetben a Next.js csak egy egyszerű interface-t biztosít nekünk, amin keresztül elérhetjük a HTTP kérés request és response objektumokat annak kezeléséhez. Ez azonban nehézkessé teszi a különböző kérések feldolgozását (pl. GET, POST és PUT), valamint különböző kódrészletek egyszerű újrafelhasználását.

Ennek kényelmesebbé tételére döntöttem a next-connect könyvtár használata mellett, amellyel a fenti igények könnyedén megvalósíthatóak. Az alábbi két kódrészletben szeretném bemutatni a főbb különbségeket.

```

export default function handler(req: NextApiRequest, res: NextApiResponse) {
  if (req.method === 'GET') {
    res.statusCode = 200
    res.setHeader('Content-Type', 'application/json')
    res.end(JSON.stringify({ name: 'John Doe' }))
  } else if (req.method === 'POST') {
    // Process a POST request
  }
}

```

4.2. lista. Default Next.js API routes

```

import nextConnect from "next-connect"

const handler = nextConnect<NextApiRequest, NextApiResponse>()

handler
  .get((req, res) => {
    res.json({ name: 'John Doe' })
  })
  .post((req, res) => {
    // Process POST request
  })

export default handler

```

4.3. lista. Kérés kezelése next-connect segítségével

4.2.1.1. Middleware támogatás

A Next.js alapesetben nem rendelkezik beépített middleware támogatással, emiatt bizonyos kódrészletek újrahasználása körülményes lehet. A next-connect azonban ezt a folyamatot rendkívül egyszerűvé teszi, így könnyen lehet védett útvonalakat létrehozni például csak bejelentkezett felhasználók számára.

```
import nextConnect from "next-connect"
import requireLogin from "middleware/requireLogin"
import requireAdmin from "middleware/requireAdmin"
const handler = nextConnect<NextApiRequest, NextApiResponse>()

handler
  .get((req, res) => {
    res.json({ name: 'John Doe' })
  })
  .use(requireLogin)
  .post((req, res) => {
    // Process POST request
  })
  .use(requireAdmin)
  // Process other requests

export default handler
```

4.4. lista. Middleware kezelés next-connect segítségével

A fenti kódrészletben a POST kérés csak bejelentkezett felhasználók számára elérhető. Ezek egymás után is fűzhetőek, így komplex igények is rendkívül egyszerűen megvalósíthatóak.

4.2.2. Kommunikáció az adatbázissal

A backend az adatbázisból a Prisma segítségével szolgáltatja az adatokat. Ahhoz hogy ezt meg tudjuk tenni, első körben szükséges az adatbázisséma és a Prisma séma közötti kapcsolatot felépíteni.

A séma adatbázisba történő átvezetésére két lehetőségünk van. Az egyik, hogy a dbdiagram oldalról lehetőségünk van .sql kiterjesztésű fájlt letölteni, ezt a létrehozott adatbázisunkon futtatni, majd a Prisma introspect funkcióját használva legenerálni hozzá a Prisma schema fájlt a backendünk számára. A másik, egyszerűbb megoldás a Prisma migrate¹ használata. Ez esetben nekünk manuálisan kell létrehozni a Prisma schema fájlt a korábbi diagram alapján, majd a

```
prisma migrate save --experimental
prisma migrate up --experimental
```

parancsokat kiadva létrehozzuk és futtatjuk a Prisma migrációt. Ez utóbbi megoldás előnyei, hogy egy központi helyen tudjuk kezelni a séma változásait, valamint ezt adatbázis-agnosztikus módon tehetjük meg.

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}
```

¹A dolgozat írása idejében ez a funkció még experimental státuszban volt, de a használata során nem ütköztem problémákba.


```

}

model Book {
  id          Int          @id @default(autoincrement())
  title       String
  author      String?
  isbn        String?
  publisher   String?
  publishedAt Int?
  stockCount  Int?         @default(1)
  count       Int?         @default(1)
  notes       String?
  image       String?
  createdAt   DateTime?    @default(now())
  updatedAt   DateTime?    @default(now())
  orders      BookToOrder[]
  categories  Category[]
}

model BookToOrder {
  id          Int          @id @default(autoincrement())
  orderId     Int
  bookId      Int
  quantity    Int?        @default(1)
  books       Book         @relation(fields: [bookId], references: [id])
  orders      Order        @relation(fields: [orderId], references: [id])

  @@unique([bookId, orderId], name: "BookToOrder_book_order_unique")
}

model Category {
  id          Int          @id @default(autoincrement())
  name        String
  books       Book[]
}

model Comment {
  id          Int          @id @default(autoincrement())
  text        String?
  createdAt   DateTime?    @default(now())
  userId      Int
  orderId     Int
  order       Order        @relation(fields: [orderId], references: [id])
  user        User         @relation(fields: [userId], references: [id])
}

model Order {
  id          Int          @id @default(autoincrement())
  userId      Int
  returnDate  DateTime?
  status      orderstatus? @default(PENDING)
  createdAt   DateTime?    @default(now())
  updatedAt   DateTime?    @default(now())
  user        User         @relation(fields: [userId], references: [id])
  books       BookToOrder[]
  comments    Comment[]
}

model User {
  id          Int          @id @default(autoincrement())
  email       String        @unique
  password    String
  name        String?
  createdAt   DateTime?    @default(now())
  role        userrole?    @default(BASIC)
  comments    Comment[]
  orders      Order[]
}

enum orderstatus {
  PENDING
  RENTED
}

```

```

    RETURNED
    LATE
  }

enum userrole {
  BASIC
  ADMIN
  EDITOR
}

```

4.5. lista. Séma leírása a Prisma DSL-ben

A Prisma DSL-ben a dbdiagram.io-hoz hasonló módon tudjuk felvenni a relációkat. Ennek nagy előnye, hogy a táblák közötti kapcsolatokat egyszerűen tudjuk modellezni, amit a `prisma migrate` át tud vezetni az adatbázisunkba.

Ahogy a fenti kódrészletben is látszik, a `Category` és a `Book` modellek közötti kapcsolatot biztosító kapcsolótábla nem jelenik meg a Prisma schema fájlban. Ez egy úgynevezett implicit kapcsolat, melyet a Prisma motorja automatikusan kezel és a migráció során létrehozza a megfelelő táblát a két entitás között.

4.2.3. Kapcsolódás az adatbázishoz

A Prisma esetében az adatbázis-kapcsolatot egy connection string segítségével tudjuk megadni, amit alapesetben a megadott környezeti változóból (`DATABASE_URL`) olvas ki. Ennek nagy előnye, hogy könnyen tudunk lokális és remote adatbázisok között váltani, illetve egyszerűvé teszi a production-ben lévő kód és az adatbázis közötti kapcsolat létrehozását.

4.2.4. Az adatok elérése

Miután minden szükséges konfigurációt elvégeztünk, el tudjuk érni az adatainkat a Prisma Client-en keresztül.

Ahhoz hogy az adatbázis-kapcsolatok számát minimalizáljuk, készítettem egy központi fájlt, ami a Prisma Client-et exportálja. Ennek köszönhetően csak egy példány fog élni az alkalmazáson belül, optimalizálva ezzel annak működését.

```

import { PrismaClient } from "@prisma/client"

const prisma = new PrismaClient()

export default prisma

```

4.6. lista. Prisma Client export

Ezután az adatbázis eléréséhez elegendő ezt az egy fájlt importálni a serverless függvényünkben.

```

const handler = nextConnect<NextApiRequest, NextApiResponse>()

handler
  .use(auth())
  .use(requireLogin())
  .use(requireRole(userrole.ADMIN))
  .get(async (req, res) => {
    try {
      const categories = await db.category.findMany()
      res.json(categories)
    } catch (error) {
      console.error(error)
      res.status(500).send(error.message)
    }
  })
}

```

4.7. lista. Adatelérés next-connect middleware és Prisma segítségével

4.3. A frontend felépítése

A Next.js framework a frontenden is alkalmazza a file-based routing koncepcióját. Ennek megfelelően elegendő az `src/pages/` mappában elhelyezett `.tsx` fájlban definiálnunk egy React komponenst, és a keretrendszer a megadott fájlnevnék megfelelő URL-en fogja kirenderelni az oldalunkat.

Az adminoknak elérhető oldalakat egy külön `admin` mappába helyeztem a könnyebb elkülöníthetőség érdekében.

A `components/` mappába kerültek az újr felhasznált, illetve kiszervezett React komponensek. Az általam írt saját React hook-ok pedig az `src/lib/hooks.tsx` fájlba kerültek.

4.4. Kódmegosztás

Mivel a backend és frontend kódja egyazon git repository-ban van elhelyezve, a kettő között kódmegosztás szinte triviális.

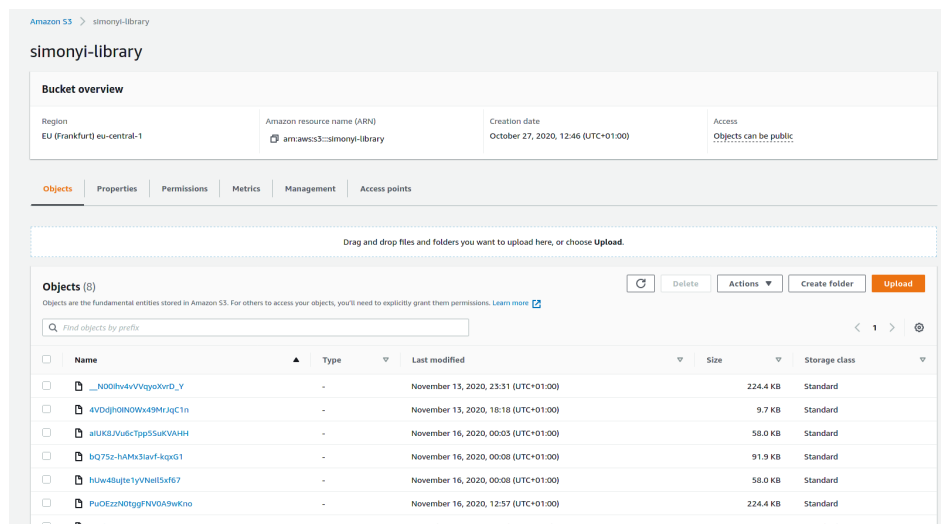
Bármely, az `src` mappán belül elhelyezett fájl felhasználható a szerver- és kliensoldali kód számára. Ennek megfelelően az `src/lib/interfaces.ts` és `src/lib/constants.ts` fájlokban találhatóak a közösen használt interfészek és konstans változók.

Az általam használt Prisma ORM egy nagy előnye továbbá, hogy az általa generált interfészek és típusdefiníciók is használhatóak kliens- és szerveroldalon egyaránt, ezáltal csökken a bugok mennyisége, és egy esetleges sémaváltozás esetén jóval könnyebb lekövetni az okozott változásokat.

4.5. Képfeltöltés

A könyvekhez lehetőség van borítóképet feltölteni, ezeket azonban valahol tárolni is kell.

Ennek a tárolására az Amazon S3 szolgáltatását választottam. A csatolt képet a frontend először elküldi a backendnek, majd az az `aws-sdk` könyvtárat használva feltölti a képet az S3 bucket-be, az adatbázisba csak a képet azonosító generált kerül be. Így lehetőségünk van a képek egyszerű és adatbázis-független kezelésére.



4.3. ábra. Amazon S3 konzol

5. fejezet

Az alkalmazás funkcióinak megvalósítása

Az alábbi fejezetben a weboldal korábban ismertetett követelményeinek implementációjáról, az azokhoz használt megoldásokról lesz szó.

5.1. Authentikáció

A felhasználók magukat egy email-jelszó párossal tudják azonosítani, melyet regisztrációkor adhatnak meg.

A regisztráció kezelésének két dologra kellett figyelmet fordítanom: egy email cím csak egyszer szerepelhessen az adatbázisban, valamint a jelszót megfelelő módon tároljuk az adatbázisban.

Az első kritériumra megoldásként szolgál az adatbázisban az email mező egyedivé tétele, valamint a regisztráció során a megadott email cím ellenőrzése.

A jelszó biztonságos tárolása érdekében azt regisztráció során hash-elve mentem el az adatbázisba, erre az `argon2` könyvtárat használtam fel.

5.1.1. Megvalósítás

A session kezeléshez a `passport` könyvtárat és cookie alapú megoldást használtam. Ennek során a felhasználót a böngészőben tárolt cookie információ azonosítja, amit backenden ellenőrizni tudunk.

A frontenden történő ellenőrzéshez egy hook-ot készítettem, ennek segítségével ellenőrizhető, hogy a felhasználó be van-e jelentkezve.

```
export function useUser() {
  const { data, mutate } = useSWR<{ user: User }>("/api/user", fetcher)
  const loading = !data
  const user = data?.user
  return [user, { mutate, loading }] as const
}
```

5.1. lista. Authentikáció hook

5.2. Autorizáció

Az alkalmazás megfelelő használata érdekében szükséges volt bizonyos funkciók elérésének szűkítésére. Ehhez a role-based access control megoldást választottam, mely alapján a felhasználókat különböző kategóriákba tudjuk besorolni, majd ezeknek a kategóriáknak adunk jogosultságokat.

Ennek megfelelően három jogosultság-kategóriát hoztam létre: BASIC, ADMIN és EDITOR.

A BASIC felhasználók képesek foglalást leadni és a sajátjaikhoz megjegyzést fűzni, valamint a hozzájuk tartozó foglalásokat listázni.

Az EDITOR jogosultsággal rendelkezők ezen felül képesek az egyes foglalások állapotát állítani, valamint bármely foglaláshoz megjegyzést fűzni.

Az ADMIN joggal rendelkezők a fentiekén kívül képesek könyvek és kategóriák hozzáadására, törlésére és szerkesztésére.

5.2.1. Megvalósítás

A kódbázison belül a frontenden és backenden is szükséges ellenőrizni a megfelelő jogosultságokat.

A backenden ehhez létrehoztam egy middleware-t, amely a már bejelentkezett emberek jogosultságát ellenőrzi.

```
const requireRole = (...roles: userrole[]) => {
  return (req: NextApiRequest, res: NextApiResponse, next: NextHandler) => {
    if (roles.some(it => it === req.user.role)) {
      return next()
    } else {
      return res.status(401).json({ message: "Nincs ömegfelel jogosultságod" })
    }
  }
}
```

5.2. lista. Authorizáció middleware

A frontenden történő validáció esetén két forgatókönyv lehetséges: egy teljes oldal vagy az oldalon belül bizonyos komponensek elrejtése a felhasználó előtt.

Az előbbi kezelésére létrehoztam egy React hook-ot, amivel az oldalak megjelenítését tudjuk kontrollálni.

```
// src/lib/hooks.tsx
export function useRequireRoles(roles: userrole[] = []) {
  const [user] = useUser()

  return roles.some((it) => it === user?.role)
}

// src/pages/admin/index.tsx
const hasAccess = useRequireRoles([userrole.ADMIN, userrole.EDITOR])
if (!hasAccess) {
  return <ErrorPage statusCode={401} message="Nincs ömegfelel jogosultságod!" />
}
```

5.3. lista. Authorizációs hook és használata

Az oldalon belüli bizonyos tartalmak elrejtésére készítettem egy komponenst, ami a tartalmát a felhasználó jogosultsági szintjének megfelelően jeleníti csak meg.

```
// src/components/HasRole.tsx
export default function HasRole({ roles, children }: Props) {
  const [user] = useUser()
  const hasRole = roles.some((it) => it === user?.role)

  return <>{hasRole && children}</>
}

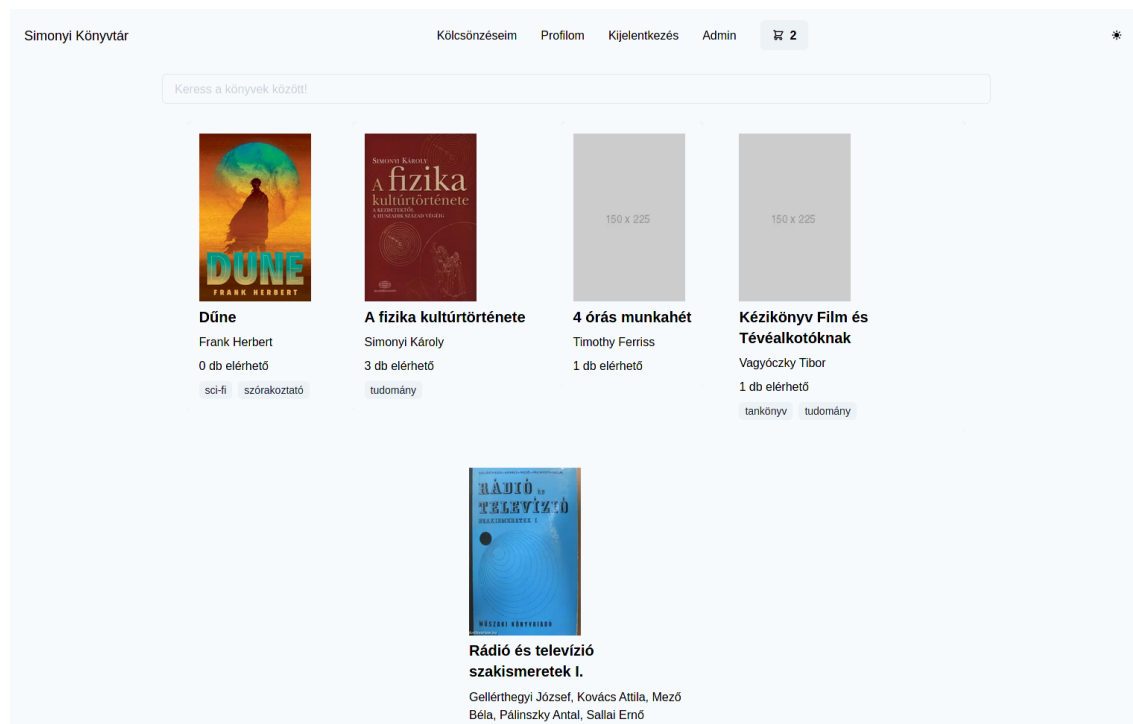
// src/components/Navbar.tsx
<HasRole roles={[userrole.ADMIN, userrole.EDITOR]}>
  <NextLink href="/admin">
    <Link>Admin</Link>
  </NextLink>
</HasRole>
```

5.4. lista. Authorizáció komponens és használata

A fenti módszerek alkalmazásával egy robusztus jogosultság-kezelő megoldást sikerült implementálnom a szoftverbe.

5.3. Könyvek listázása

A kezdőoldalon az elérhető könyveket tudjuk listázni egy rácsszerkezetben. Itt látható a könyvhöz kapcsolt borítókép, a könyv címe, szerzője és kategóriái.



5.1. ábra. Az alkalmazás kezdőlapja

5.3.1. Részletes nézet

A főoldalon egy könyvre kattintva megkapjuk annak részletes nézetét. Itt láthatjuk az összes hozzá tartozó információt, valamint lehetőségünk van a könyv kosárba helyezésére (feltéve, hogy van belőle elérhető példány).

5.3.2. Keresés a könyvek között

A főoldalon tudunk a meglévő könyvek közötti kulcsszavas keresésre. Ezt a PostgreSQL Full Text Search funkciójával implementáltam.

Ehhez szükséges volt létrehozni az indexeléshez szükséges oszlopot a `Book` táblán, ami alapján az adatbázismotor a keresést el tudja végezni.

```
ALTER TABLE "Book"
ADD COLUMN document tsvector;
update "Book"
set document = to_tsvector(title || ' ' || author || ' ' || publisher || ' ' || notes);

ALTER TABLE "Book"
ADD COLUMN document_with_idx tsvector;
update "Book"
set document_with_idx = to_tsvector(title || ' ' || coalesce(author, '') || ' ' || coalesce(
    publisher, '') || ' ' || coalesce(notes, ''));
CREATE INDEX document_idx
```



5.2. ábra. Könyv részletes nézete

```
ON "Book"
USING GIN(document_with_idx);

ALTER TABLE "Book"
  ADD COLUMN document_with_weights tsvector;
update "Book"
set document_with_weights = setweight(to_tsvector(title), 'A') ||
  setweight(to_tsvector(coalesce(author, '')), 'B') ||
  setweight(to_tsvector(coalesce(publisher, '')), 'C') ||
  setweight(to_tsvector(coalesce(notes, '')), 'D');
CREATE INDEX document_weights_idx
  ON "Book"
  USING GIN (document_with_weights);

CREATE FUNCTION book_tsvector_trigger() RETURNS trigger AS $$
begin
  new.document :=
    setweight(to_tsvector('english', coalesce(new.title, '')), 'A')
    || setweight(to_tsvector('english', coalesce(new.author, '')), 'B')
    || setweight(to_tsvector('english', coalesce(new.publisher, '')), 'C')
    || setweight(to_tsvector('english', coalesce(new.notes, '')), 'D');
  return new;
end
$$ LANGUAGE plpgsql;

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
  ON "Book" FOR EACH ROW EXECUTE PROCEDURE book_tsvector_trigger();
```

5.5. lista. A kereséshez szükséges SQL utasítások

A keresés implementálásához szükséges volt még egy egyedi lekérdezés írása, ugyanis a Prisma jelenleg nem támogatja a tsvector alapú keresést. Ehhez az alábbi megoldást használtam a backenden.

```
const sql = escape(`
select id, title, author, "stockCount", "updatedAt", image
from "Book"
where document_with_idx @@ plainto_tsquery('%s')
order by ts_rank(document_with_idx, plainto_tsquery('%s')) desc;`, term)
const books = await db.$queryRaw(sql)
```

5.6. lista. Könyvek közti keresés megvalósítása

Alapesetben ha elkezdünk a keresőmezőbe gépelni, a frontend minden egyes leütés után kérést intéz a backend felé. Ez azonban felesleges forgalmat és adatbázis-elérést okoz, ezért az input késleltetésére a use-debounce package-et használtam.

```
const [term, setTerm] = useState("")
```

```

const { data, error } = useSWR<BookWithCategories[]>(`/api/books?q=${term}`, fetcher)
const debounced = useDebounceCallback((value) => setTerm(value), 500)

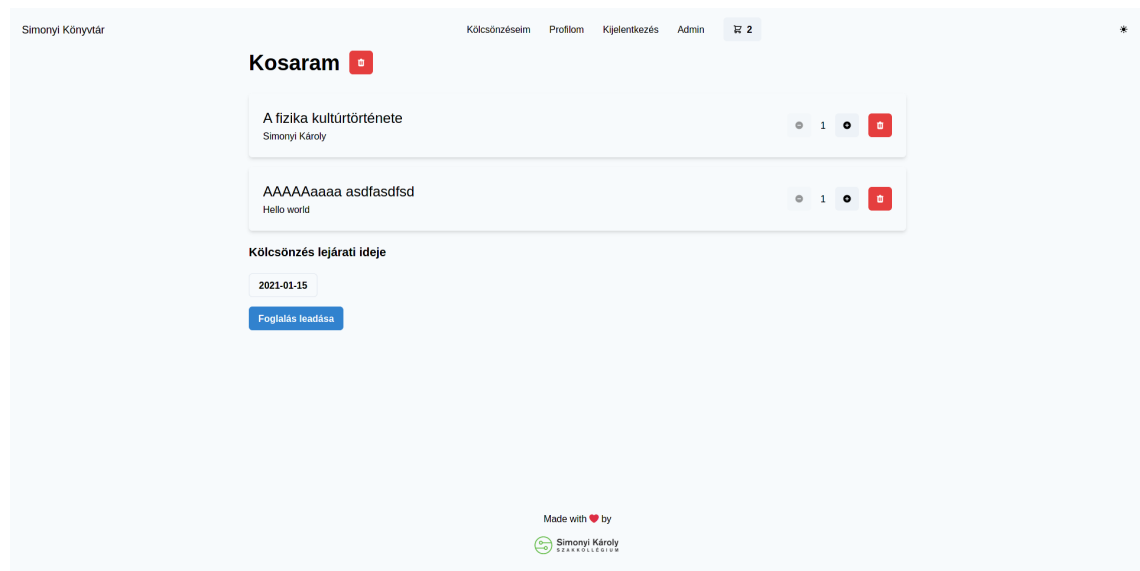
return (
  <Input
    placeholder="Keress a könyvek között!"
    mt="1rem"
    onChange={(e) => debounced.callback(e.target.value)}
  />
  { /* ... */ }
)

```

5.7. lista. A keresést megvalósító kódrészlet a frontenden

5.4. Kosár

A könyv részletes nézetében lehetőség van a kosárba helyezésre. Ennek a tartalmát a felső navigációs sávon lévő bevásárlókosár ikonra kattintva lehet megtekinteni.



5.3. ábra. Kosár oldal

Itt lehetőség van az egyes könyvek darabszámának állítására, illetve a kosár tartalmának módosítására.

Ezután a felhasználó ki tudja választani, hogy meddig szeretné kiválasztani a könyveket, majd a „Foglalás leadása” gombra kattintva véglegesítheti azt.

A kosár adatainak tárolására a böngésző Local Storage funkcióját használtam.

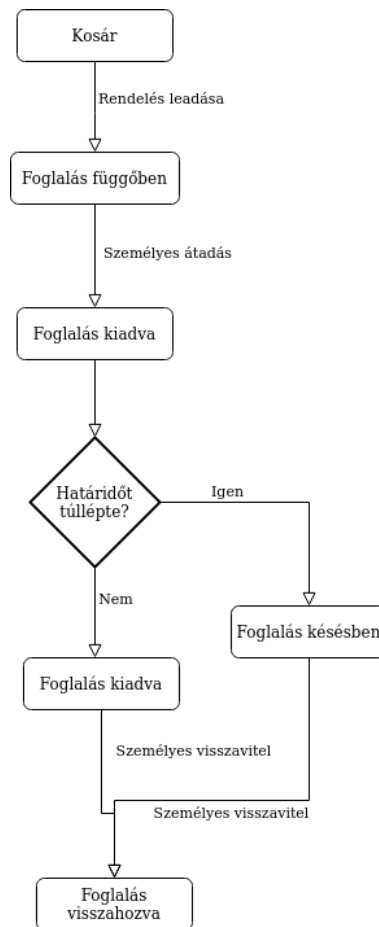
Ezen megoldás előnye, hogy bejelentkezés nélkül is hozzáadhatóak könyvek és egyszerűbbé teszi az adatbázis-struktúrát. Hátránya azonban, hogy nem kezeli a különböző böngészők (pl. asztali és mobil kliens) közötti szinkronizációt. Mivel ez utóbbi a követelmények kidolgozása közben nem merült fel, így az egyszerűbb megoldás alkalmazása mellett döntöttem.

A kosár React-ban történő eléréséhez a `use-persisted-state` könyvtárat vettem igénybe.

Segítségével több böngészőablakon keresztül is szinkronban tartható a kosárba helyezett könyvek, valamint könyv hozzáadásakor illetve törlésekor automatikusan frissül minden megjelenített, kosárhoz kapcsolódó tartalom.

5.5. Foglalási folyamat

Az alkalmazás legfontosabb eleme a könyvek foglalásának nyomonkövetése. Az alábbi ábrán ennek a folyamatát foglaltam össze.



5.4. ábra. Rendelés leadásának menete

Egy foglalás leadása esetén a könyvből elérhető darabszámot is frissíteni kell. Ez több tábla szimultán módosításával jár, viszont az inkonzisztencia elkerülése érdekében ennek teljesítenie kell az ACID elveket.

Mivel a Prisma jelenleg nem támogatja ilyen komplex módosítások egy utasításban történő végrehajtását, annak Transaction API funkcióját használtam. Ennek segítségével több, egymástól független adatbázis-módosítást lehet egyetlen tranzakcióba csomagolni, így vagy minden kívánt módosítás érvényesül, vagy egyik sem.

```
const createOrder = db.order.create({
  data: {
    returnDate,
    user: {
      connect: { id: userId },
    },
    books: {
      create: books.map(it => ({
        quantity: it.quantity,
        books: {
          connect: { id: it.id },
        }
      })),
    },
  },
});
```

```

const bookUpdates = books.map(book => {
  const bookUpdate = db.book.update({
    where: { id: book.id },
    data: {
      stockCount: { decrement: book.quantity }
    }
  })
  return bookUpdate
})

const [order] = await db.$transaction([createOrder, ...bookUpdates])

```

5.8. lista. Transaction API használata kölcsönzés létrehozásakor

A fenti kódrészlet három fő utasításra bontható: létrehozza az `Order` táblában az új bejegyzést a foglalásra, beszúrja az `Order` és `Book` táblák közti kapcsolatot megteremtő kapcsolótáblába a megfelelő rekordokat, valamint frissíti a `Book` táblában lévő könyvek elérhető darabszámát.

5.5.1. Foglálás kezelése

A foglalás leadás után a „Kölcsönzéseim” linkre kattintva tudjuk listázni azokat. Itt egy adott kölcsönzésre kattintva léphetünk a részletes nézetre, ahol megjegyzéseket tudunk fűzni hozzá. Ez a funkció szolgál az átvételi időpont egyeztetésére, problémák és egyéb igények megbeszélésére.

The screenshot shows the 'Kölcsönzéseim' (My Loans) page in the Simonyi Könyvtár application. The page has a navigation bar at the top with links: 'Simonyi Könyvtár', 'Kölcsönzéseim', 'Profilom', 'Kijelentkezés', 'Admin', and a shopping cart icon with '2' items. The main heading is 'Kölcsönzéseim'. Below it, there are four loan entries, each with a status label, a date range, and a list of books.

Status	Dates	Books
Függőben	2020. november 16. → 2020. november 16.	Dűne 1 db A fizika kultúrtörténete 1 db
Kiadva	2020. november 16. → 2020. november 21.	A fizika kultúrtörténete 1 db
Függőben	2020. november 15. → 2020. november 19.	Rádió és televízió szakismeretek I. 1 db
Kiadva	2020. október 27. → 2020. október 27.	A fizika kultúrtörténete 1 db

5.5. ábra. Kölcsönzések listázása



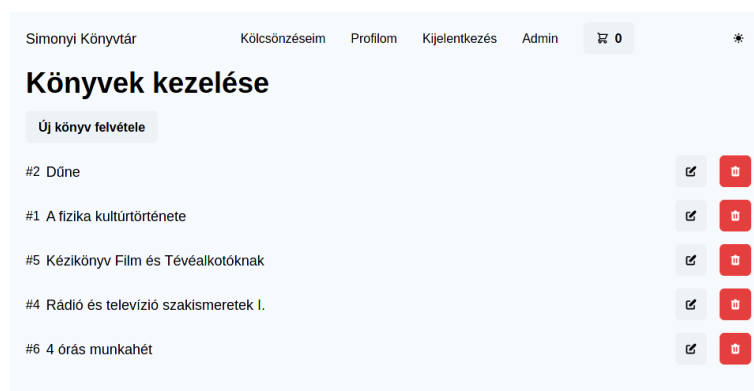
5.6. ábra. Kölcsönzés részletei kommentekkel

5.6. Admin funkciók

Az ADMIN illetve EDITOR jogosultsággal rendelkező felhasználóknak lehetőségük van a rendszerben lévő adatokat a webes felületről szerkeszteni. Ehhez a `pages/admin` mappában külön oldalakat hoztam létre, a backenden pedig a védett middleware-ekben valósítottam meg a funkciókat.

5.6.1. Könyvek kezelése

A könyveket egy listanézet foglalja össze. Itt lehetőség van az egyes elemek törlésére és szerkesztésére, valamint új könyv hozzáadására. A szerkesztéshez és létrehozáshoz ugyanazt a komponenst használtam a kód duplikáció elkerülése érdekében.



5.7. ábra. Könyvek listája az admin panelen

Könyv szerkesztése

Cím

Dűne

Szerző

Frank Herbert

ISBN

9789634069034

Darabszám

1

Kiadó

GABO KÖNYVKIADÓ ÉS KERESK.KFT.

Kiadás éve

2019

Készleten

0

Megjegyzés

Teljesen új állapot

Kategóriák

sci-fi

szépirodalom

szórakoztató

tankönyv

tudomány

tech

adatbáziskezelés

fizika

informatika

hálózatok

matematika

Kép

Mentés

5.8. ábra. Könyv adatainak szerkesztése

5.6.2. Kategóriák kezelése

A kategóriákat egy összefoglaló oldalon lehet szerkeszteni és törölni. Egy elem eltávolítása esetén minden hozzá kapcsolódó könyvből is törlődik az adott kategória. Szerkesztés esetén egy felugró párbeszédablakban lehet megadni az új nevet.

5.6.3. Foglalások kezelése

A foglalásokat egy központi oldalon lehet listázni a fontosabb információkkal.

Egy kölcsönzésre kattintva a részletes nézet jelenik meg. Itt az arra jogosultaknak lehetősége van az állapot módosítására vagy a foglalás törlésére. Szintén itt van lehetőség minden arra jogosultnak a foglaláshoz hozzászólást fűzni.

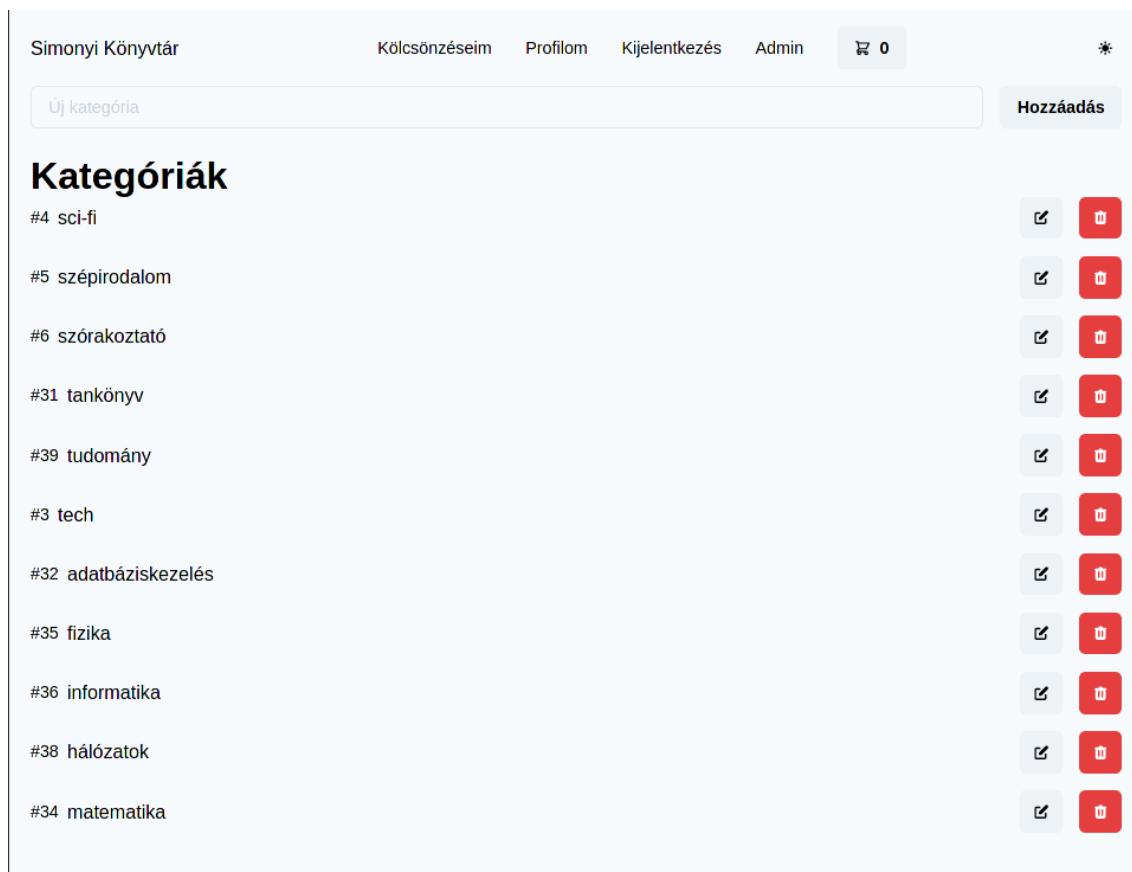
Ha az állapot „Visszahozva” státuszra módosul, a foglalás leadásához hasonlóan frissülnek a könyvhöz tartozó elérhetőségi adatok.

5.7. Reszponzív UI

A felület fejlesztése közben fontos volt, hogy minden oldal megfelelően működjön mobil képernyőkön is. A Chakra UI szerencsére első kézből támogatja ezt a Responsive Styles funkciójának köszönhetően.

Ennek segítségével egy Chakra komponens tulajdonságait egy tömbben tudjuk megadni, ahol minden elem egy adott képernyőmérethez fog társulni.

```
<Flex direction={['column', null, 'row']}>
  <Box mr={4}>
    <NextImage
      src={
        book.image
        ? `${process.env.NEXT_PUBLIC_S3_URL}/${book.image}`
```



5.9. ábra. Kategóriák kezelése oldal

```

: "https://via.placeholder.com/200x300"
}
width={300}
height={450}
/>
</Box>
{/* Other content */}
</Flex>

```

5.9. lista. Chakra UI Responsive Styles használata

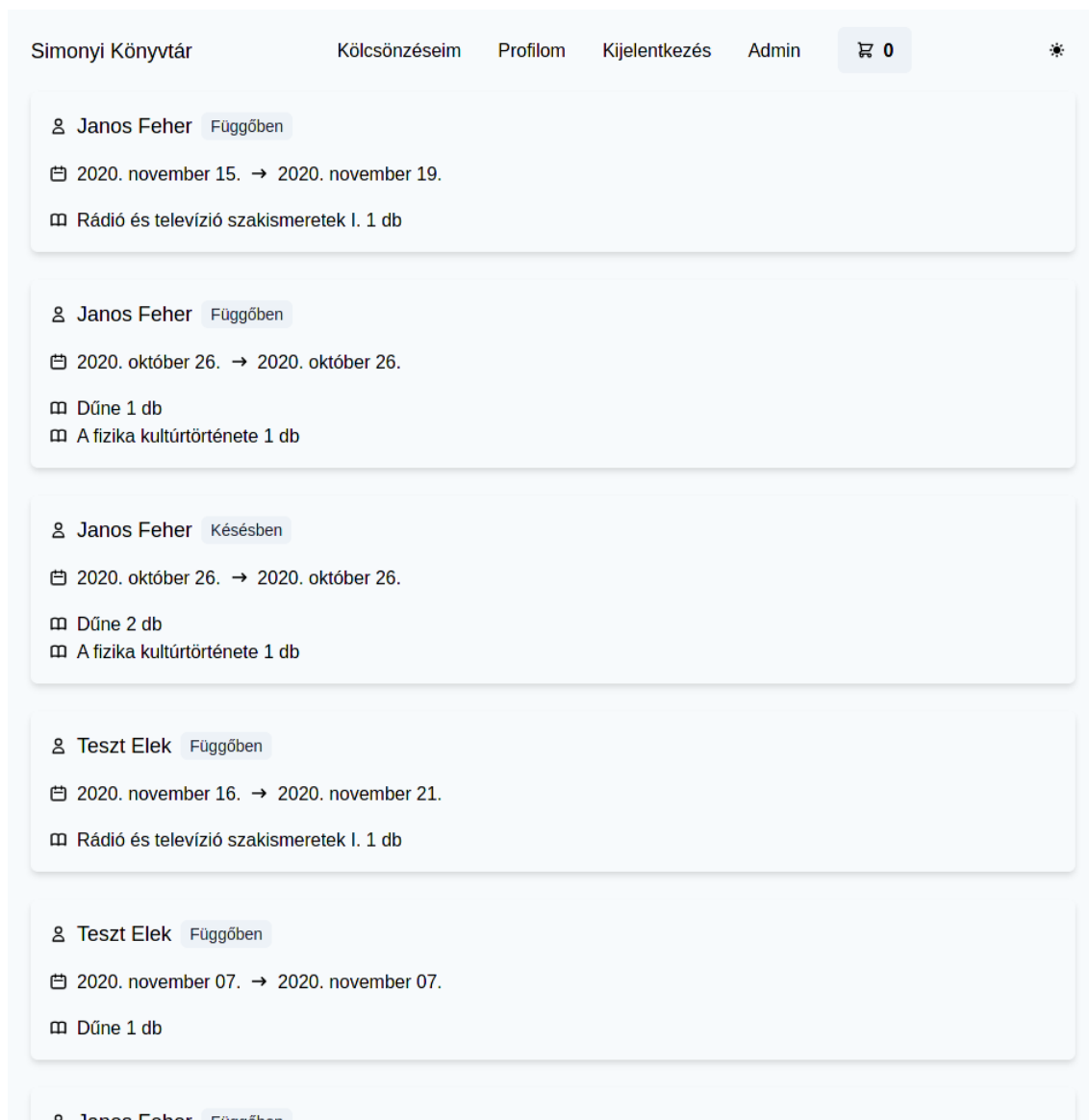
5.8. Dark Mode

A Chakra UI egyik rendkívül hasznos tulajdonsága, hogy elsőrendű dark mode támogatással rendelkezik, és valamennyi komponensnek létezik sötét módú variánsa. Ennek segítségével gombnyomásra válthatunk a világos és sötét módok között.

5.9. Validáció

A felhasználó által szolgáltatott adatok minden esetben potenciális veszélyforrást jelenthetnek, legyen szó XSS támadásról vagy az adatbázis struktúrájának integritásáról. Emiatt különösen fontos a backenden történő adatok megfelelő validációja.

Ezen adatok ellenőrzésére a yup könyvtárat használtam. Segítségével definiálhatunk egy sémát, ami ellen a felhasználó által megadott adatot validálhatjuk. Ezáltal a



5.10. ábra. Összes foglalás listázása az adminok számára

potenciális inkonzisztenciákat még az adatbázisba írás előtt kiszűrhetjük és tudathatjuk a felhasználóval.

```
export const BookSchema = yup.object().shape({
  title: yup.string().required(),
  author: yup.string(),
  count: yup.number(),
  stockCount: yup.number(),
  isbn: yup.string(),
  publisher: yup.string(),
  publishedAt: yup.number(),
  notes: yup.string(),
  image: yup.string(),
})
```

5.10. lista. yup validációs séma a könyvekre

Simonyi Könyvtár

Kölcsönzéseim

Profilom

Kijelentkezés

Admin

0

Kölcsönzés kezelése

Függőben

Kiadva

Visszahozva

Kölcsönzés részletei

Függőben

Teszt Elek

2020. november 16. → 2020. november 21.

Rádió és televízió szakismeretek I.
Gellérthegyi József, Kovács Attila, Mező Béla, Pálinszky Antal, Sallai Ernő

1 db

Foglalás törlése

Janos Feher

szia

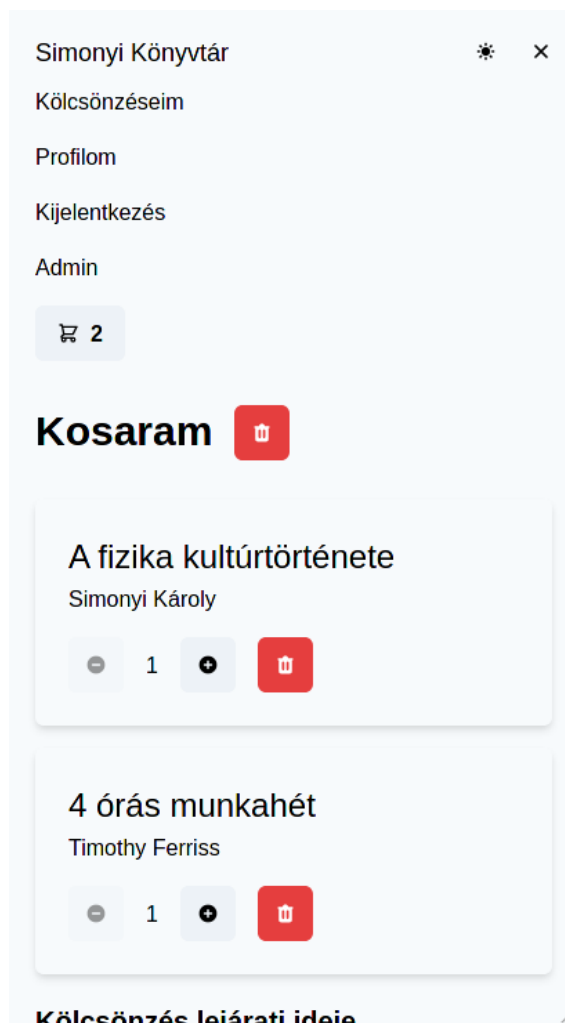
16 nappal ezelőtt

Küldés

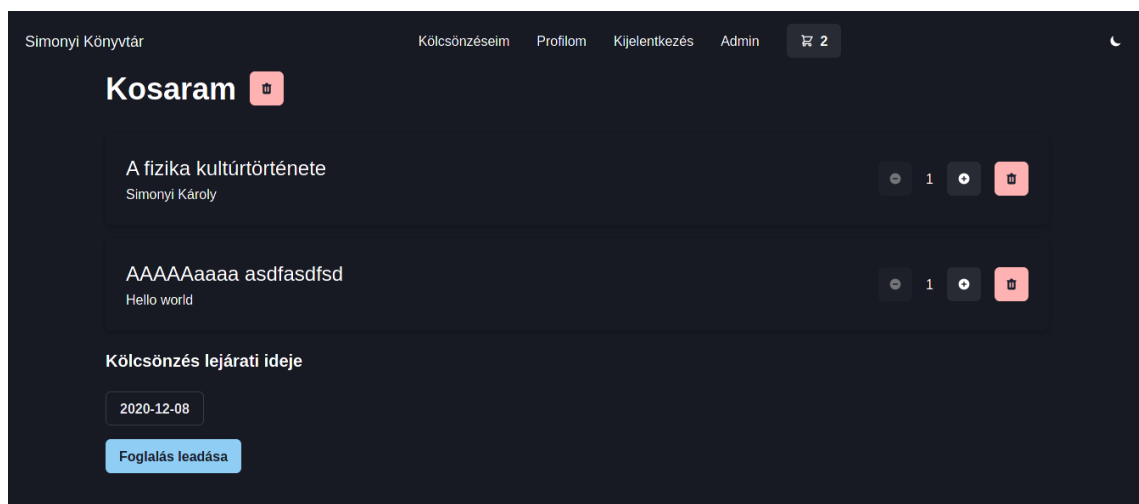
Made with by

Simonyi Károly
SZAKKOLLEGIUM

5.11. ábra. Foglalás részletes nézete az adminok számára



5.12. ábra. Kosár oldal mobil képernyőn, nyitott menüvel



5.13. ábra. Sötét téma

6. fejezet

Tesztelés

Az alkalmazás megfelelő működésének érdekében szükséges annak tesztelése. Ezt manapság mindenhol igyekszenek automatizálni, mivel a manuális tesztelés nem csak időigényes, de sok esetben repetitív és nagy hibalehetőséggel jár.

Mivel az itt bemutatott szoftver monolitikus, külön a frontendet és backendet nem tartottam célszerűnek tesztelni.

Emiatt az alkalmazáshoz end-to-end (röviden e2e) teszteket írtam, ami a szoftver egészét veszi górcső alá, így akár komplex folyamatok szimulálását is lehetővé teszi.

A következő lépés a megfelelő tesztelési keretrendszer kiválasztása. Az e2e tesztek futtatásához szükség van egy test runner környezetre, egy assertion library-re, valamint egy böngésző emulátorra.

Ezekre a követelményekre sok, egymástól független megoldás létezik, azonban ezek konfigurációja és összekötése nehézkes lehet. Emiatt döntöttem végül a `cypress` könyvtár használatára mellett, mely a fent említett funkciókat egyetlen szoftverként tudja biztosítani.

```
describe("Login flow", () => {
  it("sets auth cookie when logging in via form submission", () => {
    // destructuring assignment of the this.currentUser object
    const user = {
      email: "john@doe.com",
      password: "asdfghjkl",
    }

    cy.visit("/login")

    cy.get("input[name=email]").type(user.email)

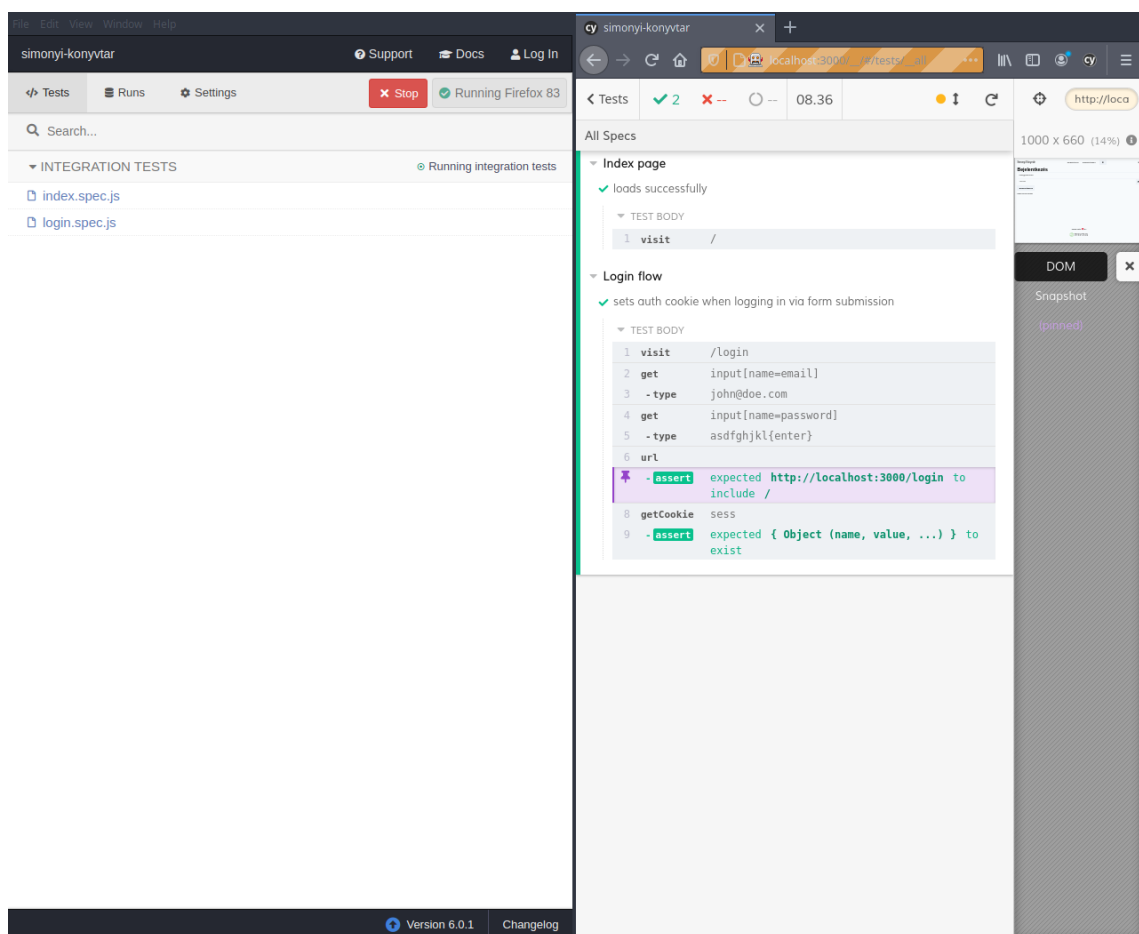
    // {enter} causes the form to submit
    cy.get("input[name=password]").type(`${user.password}{enter}`)

    // we should be redirected to /dashboard
    cy.url().should("include", "/")

    // our auth cookie should be present
    cy.getCookie("sess").should("exist")
  })
})
```

6.1. lista. Bejelentkezés tesztelése

A tesztek futtatásához először is futnia kell az alkalmazásunknak. Ezután a `cypress open` parancs kiadásával tudjuk megnyitni a futtatókörnyezetet. Itt akár egyesével, akár egymás után sorban tudjuk indítani a tesztjeinket.



6.1. ábra. Tesztek futtatása Cypress segítségével

7. fejezet

Fejlesztést segítő eszközök

Az alkalmazás készítése során igénybe vettem néhány olyan szolgáltatást, amelyek bár extra funkcionalitást nem adnak hozzá a kódbázishoz, de nagyban megkönnyítik a fejlesztést végzők munkáját.

A Continuous Integration / Continuous Delivery egy manapság már általánosan használt módszer az alkalmazások folyamatos tesztelésére és publikálására.

A Simonyi Könyvtár esetében igyekeztem ezeket a megközelítéseket alkalmazni a fejlesztés és deploy esetében is.

7.1. Continuous Integration

A forráskód tárolására a GitHub-ot használtam, így CI megoldásnak evidens volt a GitHub Actions használata.

Ez esetben elegendő egy `.yaml` fájlt elhelyezni a `.github/workflows/` mappába elhelyezni, és egy `git push` parancs kiadása után automatikusan lefut a szkriptünk. Ennek az állapotát a GitHub repository webes felületén tudjuk követni.

7.1.1. Statikus kódanalízis

A fejlesztés során nagy segítséget nyújtanak azok az eszközök, amelyek segítségével a forráskódunkat még fordítás vagy futtatás előtt ellenőrizni tudjuk.

A JavaScript világában ennek egyik szinte sztenderddé vált eszköze az `eslint`, amely egy TypeScript-hez is használható linter. Ennek segítségével ki tudjuk szűrni a nem használt kódrészleteket, valamint egységes formázást írhatunk elő a forráskódra, nagyban segítve ezzel a közös munkát.

Az alkalmazásomban én is ezt a megoldást használtam, mely lokálist a `yarn lint` parancs kiadásával lokálisan, valamint a GitHub Actions build folyamatként automatikusan is futtatható.

7.2. Continuous Delivery

Az alkalmazás hostolására két szolgáltatást használtam.

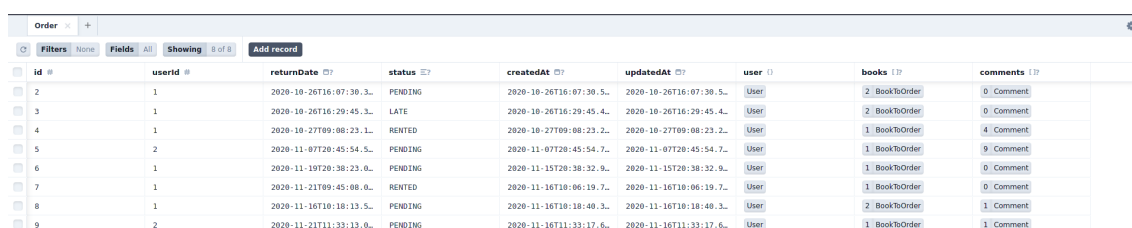
A frontend és backend közös deploymentjéhez a Vercel-t választottam. Ez rendkívül egyszerűen integrálható a Next.js keretrendszerrel, elegendő a GitHub repository-val összekötni és bármiféle extra konfiguráció nélkül elérhető lesz az alkalmazásunk egy `git push` parancs kiadása után.

A Vercel felületén lehetőségünk van a deployment státuszát ellenőrizni, korábbi deploymentra visszaállni, illetve a Next.js 10-es verziójától kezdve már különböző analitikák monitorozására is.

Az adatbázishoz a Heroku ingyenes PostgreSQL szolgáltatását vettem igénybe. Ebben az esetben elegendő az adatbázishoz kapott connection string-et a Vercel felületén a környezeti változók között megadni, és az alkalmazásunk hozzáfér az adatbázisunkhoz.

7.3. Prisma Studio

A Prisma Studio egy grafikus felület az adatbázis kezeléséhez. Ennek használatával adatbázismotortól függetlenül tudjuk az adatokat módosítani, vagy új rekordot hozzáadni. Ezzel rendkívül kényelmessé tudjuk tenni a teszteléshez szükséges adatok felvitelét illetve kezelését.



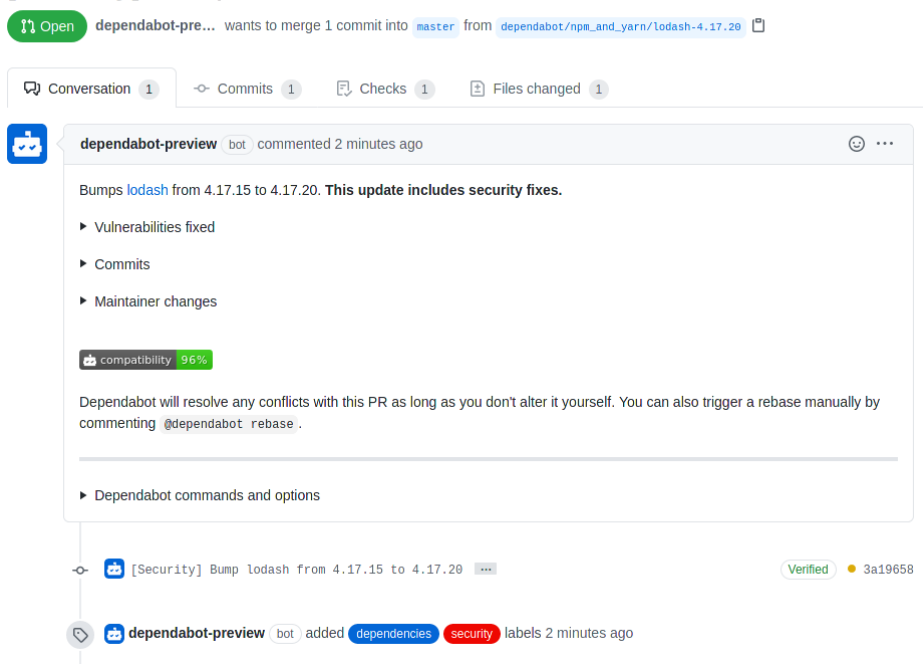
id	userId	returnDate	status	createdAt	updatedAt	user	books	comments
2	1	2020-10-26T16:07:30.3...	PENDING	2020-10-26T16:07:30.5...	2020-10-26T16:07:30.5...	User	2 BookToOrder	0 Comment
3	1	2020-10-26T16:29:45.3...	LATE	2020-10-26T16:29:45.4...	2020-10-26T16:29:45.4...	User	2 BookToOrder	0 Comment
4	1	2020-10-27T09:08:23.1...	RENTED	2020-10-27T09:08:23.2...	2020-10-27T09:08:23.2...	User	1 BookToOrder	4 Comment
5	2	2020-11-07T20:45:54.5...	PENDING	2020-11-07T20:45:54.7...	2020-11-07T20:45:54.7...	User	1 BookToOrder	9 Comment
6	1	2020-11-19T20:38:23.0...	PENDING	2020-11-15T20:38:32.9...	2020-11-15T20:38:32.9...	User	1 BookToOrder	0 Comment
7	1	2020-11-21T09:45:08.0...	RENTED	2020-11-16T10:06:19.7...	2020-11-16T10:06:19.7...	User	1 BookToOrder	0 Comment
8	1	2020-11-16T10:18:13.5...	PENDING	2020-11-16T10:18:40.3...	2020-11-16T10:18:40.3...	User	2 BookToOrder	1 Comment
9	2	2020-11-21T11:33:13.0...	PENDING	2020-11-16T11:33:17.6...	2020-11-16T11:33:17.6...	User	1 BookToOrder	1 Comment

7.1. ábra. A Prisma Studio felülete

7.4. Dependabot

A Dependabot segítségével automatizálhatjuk a függőségeinknek frissítését. Ennek köszönhetően azonnal értesülhetünk arról, ha egy package-nek új verziója érhető el.

[Security] Bump lodash from 4.17.15 to 4.17.20 #3



dependabot-preview bot commented 2 minutes ago

Bumps [lodash](#) from 4.17.15 to 4.17.20. This update includes security fixes.

- Vulnerabilities fixed
- Commits
- Maintainer changes

compatibility 96%

Dependabot will resolve any conflicts with this PR as long as you don't alter it yourself. You can also trigger a rebase manually by commenting `@dependabot rebase`.

► Dependabot commands and options

[Security] Bump lodash from 4.17.15 to 4.17.20

dependabot-preview bot added dependencies security labels 2 minutes ago

7.2. ábra. Dependabot által nyitott Pull Request

8. fejezet

Összefoglalás

Az alkalmazás elkészítése során sok eddig ismeretlen problémát kellett megoldanom, valamint megismertem általam eddig kevésbé használt technológiákat és módszereket. A dolgozat elkészítése sok tapasztalattal látott el, amiket kamatoztathatok a későbbiekben.

8.1. Továbbfejlesztési lehetőségek

8.1.1. WebSocket

A végfelhasználók és az oldal adminjai közötti kommunikáció jelenleg egyszerű HTTP kérésekkel történik, ami bár az SWR könyvtárnak köszönhetően képes az oldal újratöltése nélkül frissíteni az üzeneteket tab refocus esetén, de nem nyújt valós idejű kommunikációs élményt.

Ennek egyik fejlesztési lehetősége a WebSocket használata, mely segítségével a szerver is tud direkt üzenetet küldeni a frontend felé, ezzel valódi real-time kommunikációt lehetővé téve.

8.1.2. Értesítés rendszer

A felhasználói élmény javításának egyik módja lehet egy értesítés rendszer implementálása. Ennek segítségével az oldalon regisztráltak elsőkézből értesülnének, ha a foglalásuk állapota módosult vagy ahhoz új komment érkezett.

Ez megvalósítható csak a weboldalba épített módon, vagy email küldésével is, vagy akár a kettő kombinálásával.

8.1.3. Keresés továbbfejlesztése

A jelenleg megvalósított megoldás bár kulcsszavas keresésre jól alkalmazható, de elgépeléseket, illetve szavak részleteire keresést nem támogat.

Egy fejlettebb, robusztusabb keresést biztosító rendszert kínál például az Algolia¹ vagy az Elastic Search² szolgáltatás, melyek működhetnek cloud vagy self-hosted módban is.

8.1.4. Auth.SCH integráció

A kollégisták által igénybe vett weboldalaknál gyakran használt SSO megoldás az Auth.SCH³, amivel a hallgatók egy központosított azonosítóval tudnak bejelentkezni.

¹<https://www.algolia.com/>

²<https://www.elastic.co/>

³<https://auth.sch.bme.hu/>

Ezt kihasználva a felhasználóknak nem kell külön jelszót megjegyezniük az alkalmazás használatához, illetve az itt tárolt adatok alkalmazhatóak a felhasználó jogkörének megállapítására.

Köszönetnyilvánítás

Szeretném megköszönni a segítséget és együttműködést a Simonyi Károly Szakkollégium elnökségének, akik támogattak a projekt elkészítésében és megvalósításában.

Ábrák jegyzéke

2.1. Az alkalmazáshoz tartozó use-case diagram	3
4.1. Az alkalmazás high-level architektúrája	8
4.2. Az adatbázisséma ER diagramja.	9
4.3. Amazon S3 konzol	13
5.1. Az alkalmazás kezdőlapja	16
5.2. Könyv részletes nézete	17
5.3. Kosár oldal	18
5.4. Rendelés leadásának menete	19
5.5. Kölcsönzések listázása	20
5.6. Kölcsönzés részletei kommentekkel	21
5.7. Könyvek listája az admin panelen	21
5.8. Könyv adatainak szerkesztése	22
5.9. Kategóriák kezelése oldal	23
5.10. Összes foglalás listázása az adminok számára	24
5.11. Foglalás részletes nézete az adminok számára	25
5.12. Kosár oldal mobil képernyőn, nyitott menüvel	26
5.13. Sötét téma	26
6.1. Tesztek futtatása Cypress segítségével	28
7.1. A Prisma Studio felülete	30
7.2. Dependabot által nyitott Pull Request	30

Irodalomjegyzék

- [1] Chakra ui. <https://chakra-ui.com>.
- [2] Chris Coyier: What is developer experience (dx)? <https://css-tricks.com/what-is-developer-experience-dx/>, 2020.
- [3] Fetch api. https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.
- [4] Next.js. <https://nextjs.org/>.
- [5] Overengineering. <https://en.wikipedia.org/wiki/Overengineering>.
- [6] PostgreSQL. <https://www.postgresql.org/>.
- [7] Prisma. <https://www.prisma.io/>.
- [8] React (web framework). <https://reactjs.org/>.
- [9] Rest api. https://en.wikipedia.org/wiki/Representational_state_transfer.
- [10] Swr. <https://swr.vercel.app/>.
- [11] Typescript. <https://www.typescriptlang.org/>.

Függelék

F.1. Telepítési útmutató

Az alkalmazás lokális futtatásához az alábbiak szükségesek:

- Node.js 12
- yarn
- PostgreSQL
- Amazon S3 bucket létrehozása

PostgreSQL-ben hozzunk létre egy új adatbázist, majd a hozzá tartozó connection string-et a `prisma/.env` fájlba mint környezeti változó vegyük fel.

Szükséges továbbá az S3-hoz tartozó környezeti változók beállítása a `.env` fájlban, ezt az alkalmazás gyökérmappájában helyezzük el. Ezután a `yarn install` parancs kiadásával tudjuk a kellő függőségeket letölteni, majd a `yarn dev` utasítással tudjuk az alkalmazást futtatni.

F.2. Az alkalmazás elérhetősége

A program forráskódja a GitHub-on megtalálható. A legfrissebb build pedig megtekinthető a <https://simonyi-konyvtar.vercel.app> oldalon.