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 6.

Tartalomjegyzék

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 6.

Fehér János
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 bevezető tartalmazza a diplomaterv-kiírás elemzését, történelmi előzményeit, a feladat indokoltságát (a motiváció leírását), az eddigi megoldásokat, és ennek tükrében a hallgató megoldásának összefoglalását.

A bevezető szokás szerint a diplomaterv felépítésével záródik, azaz annak rövid leírásával, hogy melyik fejezet mivel foglalkozik.

A Simonyi Károly Szakkollégiumban jelenleg üzemel egy könyvtár, ahonnan a hallgatók különféle tankönyveket és szórakoztató irodalmat van lehetőségük kölcsönözni. Ennek az adminisztrációja egy megosztott Google Docs táblázaton keresztül történik, ami azonban a publikum felé nem nyilvános, a hallgatók a könyvtárban lévő könyvek elérhetőségét csak személyesen, vagy az üzemeltetőknek írt emailen keresztül tudják ellenőrizni. Ez nagyban hátráltatja, hogy a hallgatókban tudatosuljon a könyvtár létezése, valamint megnehezíti annak karbantartását és a kölcsönzés menetét is.

Ezen helyzet adta a motivációt arra, hogy készítsek egy, a hallgatók által könnyen hozzáférhető, és az üzemeltetők által kényelmesen menedzselhető webes alkalmazást a szakkollégium számára. A cél egy reszponzív, web alapú felület tervezése és elkészítése a könyvtár könnyű használhatósága érdekében. (TODO: ezt talán szebben fogalmazni)

TODO: szebben fejezetek: - Használt technológiák kiválasztása - frontend, backend, db, rest/graphql - alkalmazás felépítése - adatbázis - db séma (dbdiagram export, prisma schema, prisma migrate) - prisma (schema file, migrations) - backend - nextjs file-based routing, middleware-ek - frontend - components/ mappa, file based routing - kódmegosztás - lib/ mappa, prisma típusok frontenden - alkalmazás működése - authentikáció, authorizáció - adatlekérés a backendtől: useSWR - prisma - kosár - order leadása, menedzselése - keresés: postgres full text search, kihívások, megoldások - fájlfeltöltés: amazon S3

- tesztek (???) - CI - github actions, eslint - hosting, jövőbeli deploy (kubernetes, docker) - jövőbeli bővítési lehetőségek - real-time chat websocket segítségével - hosted / dockerben futó fuzzy search (elastic, algolia) - authsch integráció a user-pass mellé

2. fejezet

Specifikáció

2.1. Felhasználókezelés

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

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 és darabszámukat szerkeszteni.

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.

2.5. Admin funkcionalitás

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

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

3. fejezet

Használt technológiák kiválasztása

3.1. Verziókezelés

A fejlesztés során fontos a változások átlátható, könnyű követése, a szoftver különböző verzóinak 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 funkcionalitásban lényegében megyegyeznek, azonban az elérhető könyvtárak mennyisége és minősége a React esetében a legnagyobb, emiatt végső soron erre esett a választásom.

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.

3.3. Backend

A backend keretrendszer kiválasztása során a legfontosabb szempont az volt, hogy minél jobban képes legyen integrálódni a frontend keretrendszerhez. Mindenképpen szerettem volna elkerülni a külön repository használtatát. 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-t (TODO: lábjegyzet) támogató keretrendszer, amely rendelkezik egy úgynevezett API routes nevű funkcióval.

Ennek segítségével a React alkalmazásunkon belül készíthető egy API réteg, amely a fordítás után szerveroldalon 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.

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, inegyenes és nyílt forráskódú megoldás.

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 hansúlyt a type safety-re helyezi. Az adatbázis sémát egy speciális .prisma kiterjesztésű fájlban tudjuk megírni, majd ezt a Prisma migrációs eszközével tudjuk az adatbázisunkba átvezetni. (TODO: lábjegyzet hogy a migráció még csak experimental) 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

todo: táblázat

ORM query builder paradigma oop funkcionális programozás séma definíció ts class+dekorátorok változó adatelérés példányosított query interface, chain-elt függvények objektumon keresztül

3.5. REST és GraphQL

A fentieken túl utolsó lépésként szükséges volt eldönteni a frontend és a backend közötti kommunikáció módját. A REST architektúra mellett ugyanis megjelent a GraphQl, ami az eddigi lehetőségekhez képest egy flexibilisebb megoldást kínál az adatok elérésére. A két technológia előnyeit és hátrányait az alábbi táblázatban foglaltam össze.

TODO: táblázat

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

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, 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 kifezezetten 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.

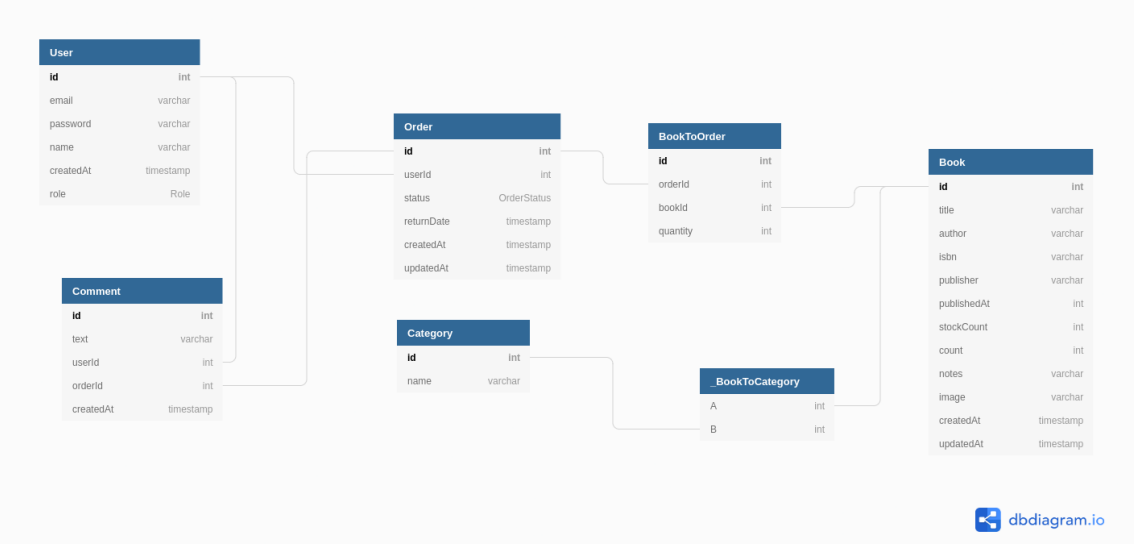
4. fejezet

Az alkalmazás felépítése

4.1. Adatbázisséma

4.1.1. Tervezés

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.



4.1. ábra. Az adatbázisséma ER diagramja.

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

4.1.2. Implementáció

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 migrate[1] használata. Ez esetben nekünk manuálisan kell létrehozni a Prisma schema fájlt a korábbi diagram alapján, majd a

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

```
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")
}

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
}
```

4.1. 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.

4.1.3. Kapcsolódás az adatbázishoz

A Prisma esetében az adatbáziskapcsolatot 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. A backend felépítése

4.2.1. Next.js API routes

A Next.js keretrendszer a 9-es verzió óta lehetővé teszi szerveroldali kód írását az alkalmazásunkhoz. Ennek segítségével a `pages/api` mappába helyetett fájljaink szolgálnak backendként. Minden ide helyezett fájl egyben a nevének megfelelő 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 },
  } = req

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

4.2. lista. Next.js dinamikus routing

4.2.2. 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 nezhézkessé teszi a különböző kérések feldolgozását (pl. GET, POST és PUT), valamint különbőző 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
  }
}
```

4.3. 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
```

4.4. lista. Kérés kezelése next-connect segítségével

4.2.2.1. 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(requreLogin)
  .post((req, res) => {
    // Process POST request
  })
  .use(requireAdmin)
  // Process other requests

export default handler
```

4.5. 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.

4.3. 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 fájlnévnek 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 újrafelhasznált, 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.

5. fejezet

Az alkalmazás működése

Lorem ipsum

6. fejezet

CI/CD

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.

6.1. 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.

6.1.1. 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 folyamataként automatikusan is futtatható.

6.2. 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 deploymentre 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.

Köszönetnyilvánítás

Ez nem kötelező, akár törölhető is. Ha a szerző szükségét érzi, itt lehet köszönetet nyilvánítani azoknak, akik hozzájárultak munkájukkal ahhoz, hogy a hallgató a szakdolgozatban vagy diplomamunkában leírt feladatokat sikeresen elvégezze. A konzulensnek való köszönetnyilvánítás sem kötelező, a konzulensnek hivatalosan is dolga, hogy a hallgatót konzultálja.

Irodalomjegyzék

[1] Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Kar: Diplomaterv portál (2011. február 26.). http://diplomaterv.vik.bme.hu/.

[2] James C. Candy: Decimation for sigma delta modulation. 34. évf. (1986. 01) 1. sz., 72–76. p. DOI: 10.1109/TCOM.1986.1096432.

[3] Gábor Jeney: Hogyan néz ki egy igényes dokumentum? Néhány szóban az alapvető tipográfiai szabályokról, 2014. http://www.mcl.hu/~jeneyg/kinezet.pdf.

[4] Peter Kiss: Adaptive digital compensation of analog circuit imperfections for cascaded delta-sigma analog-to-digital converters, 2000. 04.

[5] Wai L. Lee – Charles G. Sodini: A topology for higher order interpolative coders. In *Proc. of the IEEE International Symposium on Circuits and Systems* (konferenciaanyag). 1987. 4-7 05., 459–462. p.

[6] Alexey Mkrtychev: Models for the logic of proofs. In Sergei Adian – Anil Nerode (szerk.): *Logical Foundations of Computer Science*. Lecture Notes in Computer Science sorozat, 1234. köt. 1997, Springer Berlin Heidelberg, 266–275. p. ISBN 978-3-540-63045-6. URL http://dx.doi.org/10.1007/3-540-63045-7_27.

[7] Richard Schreier: *The Delta-Sigma Toolbox v5.2*. Oregon State University, 2000. 01. http://www.mathworks.com/matlabcentral/fileexchange/.

[8] Ferenc Wettl – Gyula Mayer – Péter Szabó: *LaTeX kézikönyv*. 2004, Panem Könyvkiadó.

Függelék