Szakdolgozat

Miskolci Egyetem

Szabadság nyilvántartó rendszer

Készítette:

Tőzsér Zétény Csaba

Programtervező informatikus

Témavezető:

Dr. Agárdi Anita

Miskolc, 2024

Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Alkalmazott Matematikai Intézeti Tanszék **Szám:**

Szakdolgozat Feladat

Tőzsér Zétény Csaba (QGNLD2) programtervező informatikus jelölt részére.

A szakdolgozat tárgyköre:

A szakdolgozat címe: Szabadság nyilvántartó rendszer

A feladat részletezése:

A szakdolgozat célja egy webes nyilvántartó rendszer készítése. A szoftver nyilvántartja egy cég dolgozóinak szabadságait. Számításba veszi a különböző szabadságtípusokat. A program képes új szabadság létrehozására, törlésére és módosítására. Szabadságot lehet kérni, elfogadni, kért szabadságot visszautasítani.

Témavezető: Dr. Agárdi Anita (Adjunktus)

A feladat kiadásának ideje: 2023 szeptember 28

. . . . . .
szakfelelős

Eredetiségi Nyilatkozat

Alulírott **Tőzsér Zétény Csaba**; Neptun-kód: `QGNLD2` a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős Programtervező informatikus szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy *Szabadság nyilvántartó rendszer* című szakdolgozatom saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, ...év ..... .hó ..... .nap

.....

Hallgató

1.

szükséges (módosítás külön lapon)

A szakdolgozat feladat módosítása

nem szükséges

....

dátum témavezető(k)

2. A feladat kidolgozását ellenőriztem:

témavezető (dátum, aláírás): konzulens (dátum, aláírás):

....

....

....

3. A szakdolgozat beadható:

....

dátum témavezető(k)

4. A szakdolgozat ..... szövegoldalt

..... program protokollt (listát, felhasználói leírást)

..... elektronikus adathordozót (részletezve)

.....

..... egyéb mellékletet (részletezve)

.....

tartalmaz.

....

dátum témavezető(k)

5.

bocsátható

A szakdolgozat bírálatra

nem bocsátható

A bíráló neve: .....

....

dátum szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata: .....

a bíráló javaslata: .....

a szakdolgozat végleges eredménye: .....

Miskolc, ...

a Záróvizsga Bizottság Elnöke

Tartalomjegyzék

1. fejezet

Bevezetés

Egy elektronikus rendszer nélkül igen csak idő és költség igényes egy vállalat alkalmazottai számára számon tartani az igényelt, kivett és kivehető szabadságokat. Minél nagyobb a vállalat, annál több embert kell alkalmazni erre és efféle feladatokra, amelyek hasznot nem termelnek.

Számon kell tartani az alkalmazottak korát, hisz attól függően is kaphatnak pótszabadságot. Gyermekeik számát, gyermekeik eseteleges fogyatékosságát. Szabadságtípustól függően távolléti díjukat. A különböző tényezőket még bőven lehetne sorolni, már ennyiből is látható, hogy mindez nem kis feladat.

Ha valamelyik vállalt nem papíron akarja számon tartani az alkalmazottai szabadságait és egyéb adatait, de nem szánta el magát egy dedikált szabadság nyilvántartó rendszerre való beruházásba, akkor a legtöbbször a Microsoft Excelhez fordulnak. Habár ez valamelyest segít az adminisztratív feladatok terhének csökkentésében, a szoftver csak a feladatok egy kis részét végzi el.

A dolgozat célja egy ezt a terhet, ha nem is teljes mértékben eltüntetni, de legalább nagyban könnyíteni hivatott szabadság nyilvántartó rendszer létrehozása. Minden alkalmazott meg tudja tekinteni saját szabadságait és új szabadságokat kérni. Az arra jogosult személy vagy személyek pedig elbírálni kérelmeiket.

Ezzel a rendszerrel az emberi munka csökkenni fog, mivel a szoftver el tudja végezni a szükséges számításokat. Az alkalmazottak pedig időt spórolnak azzal, hogy pár kattintással kérhetnek szabadságot, egy adminisztrátor személyes vagy telefonos felkeresése helyett. Az alkalmazás interneten keresztül érhető el, így bárhonnan, bármikor intézhetők ezek az ügyek.

Minden alkalmazottnak lesz egy saját profilja, melybe a nevével és email címével tud majd bejelentkezni. Így mindenki csak saját magának tud majd szabadságot kérni, kérést törölni. Ezeket a profilokat egy adminisztrátori jogosultsággal rendelkező személy fogja létrehozni minden új alkalmazott számára.

Az alkalmazás egyes részeit csak adminisztrátori jogosultsággal lehet majd elérni. Nem lenne szerencsés ha minden alkalmazott tudná bírálni saját, vagy más szabadság kérelmeit.

2. fejezet

Felhasznált technológiák

2.1. HTML

A HTML [1], teljes nevén Hyper Text Markup Language. Magyarra fordítva hiperszöveges jelölőnyelv. A Hyper Text a weboldalakat összekötő hiperlinkeket jelenti. Jelenlegi verziója a HTML 5.

A jelölőnyelvek dokumentumformázás értelmezését teszik lehetővé számítógépek számára. Tehát a szövegek, képek elrendezését, formázását. A HTML az alapértelmezett jelölőnyelv a világháló oldalai számára.

A HTML elemek, tagok, < és > szimbólumok közt helyezkednek el. Lehetnek önzáróak <tagnév />, külön nyitó és záró tag esetén <tagnév>tartalom</tagnév>.

Egy HTML dokumentum felépítése a következő:

- Dokumentum típus megadása: <!DOCTYPE html>. Ez definiálja a dokumentumot HTML dokumentumként. Nincs hozzá záró tag.
- Ezután jön a gyökér elem: <html></html>. Ebben található minden másik elem.
- Fejléc: <head></head>. Ebben meta adatok találhatók, melyeket a böngésző közvetlenül nem jelenít meg.
- Törzs elem: <body></body>. A megjelenítendő szöveget, egyéb elemeket tartalmazza.

2.2. CSS

Teljes nevén Cascading Style Sheets [2], magyarul lépcsőzetes stíluslapoknak hívják. Stílusleíró nyelv HTML dokumentumok számára. Jócskán megkönnyíti egy-egy oldal kinézetének formázását.

Egy oldal által használt stílusokat általában egy külön, .css, fájlban tároljuk, de magán a HTML dokumentumon belül is meg lehet adni az adott taghoz tartozó stílust. <tagnév style="stílus ide">. Az utóbbi módszernél a megadott stílus CSS érték.

A CSS megjelenése előtt a stílust meghatározó elemek a HTML kódba kerültek, a böngészőkhöz pedig egyre több új tagot adtak hozzá, hogy bővítsék a formázási lehetőségeket. Így a weblapok HTML kódja egyre nehezebben olvashatóvá vált.

Egy CSS stíluslapban megadhatjuk az egyes HTML tagok formázását általános érvényességgel. Például minden gomb legyen zöld hátterű. A tagoknak adhatunk saját

osztályneveket, melyek segítségével az azonos típusú, de más osztálynevű tagok különkülön formázhatók.

2.3. JavaScript

A JavaScript [3], röviden JS, a HTML és CSS mellett a mai weboldalak alapját képezi. Segítségével megváltoztathatjuk egy weboldal HTML tartalmát, az oldal újratöltése nélkül. Lecserélhet képeket, új szöveget jeleníthet meg stb.

Kliens oldali nyelv, ami azt jelenti, hogy csak a felhasználó eszközén lévő feladatokat tud végrehajtani. Azonban ma már ez nem igaz, szervereken is használható a Node.js segítségével. Mobil és asztali eszközökön is használható megfelelő környezettel.

A böngészőknek van JavaScript motorja, ami értelmezi a JavaScript kódot és futtatja. Eredetileg interpretálták a kódot, vagyis nem volt szükség előzetes fordításra gépi kódra, mint például a C nyelv esetében. A modern JavaScript motorok futásidejű fordítást alkalmaznak, amivel hatékonyabban dolgoznak.

A JavaScript magas szintű, biztonságos, mivel nem ad hozzáférést a memóriához és processzorhoz, webes eredete révén.

Gyengén típusos nyelv, vagyis nem kell megadni a változók típusait. Futási időben kapnak értéket, az alapján pedig típust. Ez azt jelenti, hogy sokminden csak futási időben derül ki, a problémák keresését is megnehezíti.

2.4. React

A React [4], vagy ReactJS, egy frontend JavaScript könyvtár, melyet a Facebook adott ki 2013-ban. Nem az egész oldalt tölti újra, és JSX-et használ. Nyílt forráskódú, bárki készíthet hozzá csomagokat saját igényeik kielégítésére, problémáik megoldására.

Összetett felhasználói felületeket több kisebb, újrahasználható komponensre bontva lehet elkészíteni vele. Több rétegnyi komponensből épül fel egy React alkalmazás. Komponensek lehetnek osztály, vagy függvény komponensek. Mindkettő HTML-t ad vissza.

A JSX, JavaScript XML rövidítése, lehetővé teszi a HTML elemek használatát JavaScript környezetben. A HTML tagokat React elemekké alakítja. Könnyebbé teszi az elemek megírását, de hosszú kódnál rontja az olvashatóságot.

A virtuális DOM, dokumentumobjektum-modell, a valós DOM reprezentációja, amit jóval olcsóbb változtatni. A React benne végzi el a módosításokat. Ezután megnézi, hogy mik a különbségek a DOM és virtuális DOM között, majd csak a szükséges módosításokat hatja végre a DOM-on.

Az SSR, magyarul szerveroldali renderelés, ami segítségével a kliensoldali JavaScript alkalmazást a szerveren rendereljük. Ez nagyban gyorsítja egy-egy oldal betöltését, hisz a kliens készen kapja a felhasználói felületet.

2.5. Node.js

A Node.js [5] segítségével a szervert is JavaScriptben írhatjuk. Így nem kell egy új nyelvet megtanulni a backend fejlesztéshez. Nyílt forráskódú, több platformon futtatható, eseményalapú programozást használ.

Csak egy szálon fut, nem blokkoló aszinkron módon. Nem kell megvárnia a választ egy kérésre új létrehozása előtt. Visszahívásokkal jelzi egy feladat befejeztét.

Népszerűsége miatt hatalmas közösség épült köré. Bárki készíthet hozzá egy csomagot, valamilyen probléma megoldására, amit aztán megoszthat a világgal. Saját csomagkezelője, a telepítést egyszerűvé teszi.

2.6. Express

Az Express [6] egy keretrendszer Node.js alkalmazások számára, egyszerű dizájnával megkönnyíti az alkalmazásfejlesztést. Remekül skálázható, hasznos mind kis és nagy projektek esetén.

2.7. MongoDB

A MongoDB [7] egy NoSQL, tehát nem relációs adatbázis. Felépítése ehelyett dokumentum orientált, BSON formátumot használ hozzá.

A BSON a JSON bináris reprezentációját jelenti, vagyis a kapott JSON adatokat átalakítja, majd ezeket tárolja.

Az adatbázis kollekciókat tartalmaz, ezeken belül a találhatók dokumentumok. Sémamentessége miatt nem kell definiálnunk sémát. Egy kollekció több, különböző dokumentumot is tartalmazhat. Ez nagy flexibilitást kölcsönöz számára.

Ezzel bemutattam az úgynevezett MERN stack minden részét. Ez MongoDB adatbázist, Express web keretrendszert, React frontendet és Node webszervert jelent.

2.8. Visual Studio Code

A Visual Studio Code [8] egy a Microsoft által fejlesztett kódszerkesztő. Az egyik legnépszerűbb ilyen program a világon. Egyszerűsége miatt kezdő programozók kedvence, de emiatt nem szabad alábecsülni.

Beépített támogatást tartalmaz egyes népszerű programozás nyelvekhez, de szabadon bővíthető. Bármilyen nyelvhez lehet találni bővítményeket, ha az alap támogatás nem elégséges.

Git támogatása is beépített, még egy-két GitHub bővítmény telepítése után elérhetjük GitHub weboldalát, onnan repozitorikat klónozhatunk a számítógépünkre.

2.9. Postman

A Postman [9] egy tesztelő program a backend api teszteléséhez fejlesztés során. HTTP kérések küldésére használható, ezek elmentésére későbbi ellenőrzéshez.

Nem csak HTTP kérések küldésére képes, hanem az API-t kódot több nyelvre is képes átalakítani. A teszteket pedig lehet automatizálni, hogy adott időközönként megismétlődjenek.

2.10. Git és Github

A Git [10] egy elosztott verziókövető és forráskód kezelő rendszer. Nyomon követi a változásokat a forráskódban, lehetőséget ad kollaborálni más fejlesztőkkel. Ingyenes és nyílt forráskódú. Fejlesztését Linus Torvalds, a Linux atyja kezdte meg 2005-ben.

Egy Gittel létrehozott mappát repozitorinak nevezünk. Ebben tartja számon az összes változtatást a fájlokhoz. Ha mentjük a változtatásainkat, létrehozunk egy pillanatképet a programunkról. Ezeket a pillanatképeket lokálisan tárolja a Git, hogy bármikor elővehessük őket szükség esetén.

Egy Git repozitoriban több ágat is létrehozhatunk, ez ajánlott is. Így nem kell mindent a fő ágba mentenünk. A különböző ágakban például minden fejlesztő saját funkciókon dolgozhat, majd a projekt vezető egyesítheti ezeket az ágakat.

A GitHub [10] egy internetes hosting szolgáltatás a Git repozitoriknak. Lehetőséget ad dokumentáció létrehozására. Hiba követésre, nyílt projektek esetében hozzászólásra és kód javaslatok adására. Használata szintén ingyenes.

3. fejezet

Elvárások és fejlesztői környezet

3.1. Elvárások

Ebben a szekcióban az alkalmazással szembeni elvárásokat fogom összegyűjteni.

3.1.1. Egyszerű felhasználó felől

- legyen saját fiókja
- tudjon szabadságot kérni
- lássa szabadságait
- lássa fennmaradó szabadnapjait
- kért szabadságait tudja törölni, szerkeszteni

3.1.2. Munkáltató felől

- legyen bővebb jogkörű fiók
- lássa a kért szabadságokat
- tudja jóváhagyni, elutasítani
- lássa a jóváhagyottakat alkalmazottanként
- alkalmazottak hozzáadása, törlése
- alkalmazottak adatainak szerkesztése

3.1.3. Használati feltételek

Az alkalmazás internet böngészőből lesz elérhető, arra alkalmas eszközről (például asztali számítógép vagy okostelefon). Az alábbiak pedig a használatához, üzemeltetéséhez szükséges feltételek.

- weblap üzemeltetéséhez szerver
- szervergép

- adatbázis számára elegendő tárhely
- állandó elérhetőség

3.1.4. Összegezve

- bejelentkezés lap
- bejelentkezés nélkül ne legyen elérhető
- lap saját szabadságokra
- lap szabadságok jóváhagyására
- lap alkalmazottak kezelésére
- különböző hozzáférés adminisztrátornak

Az előzőeken felül még a teljesítmény felé is van elvárás. Mégpedig lehetőleg gyors legyen az alkalmazás. Ennek érdekében például ne legyen túl mély az egy oldalról kiinduló gyermek komponensek egyik ága sem. Az sok időt hozzátehet egy-egy oldal megjelenítéséhez.

A felhasználói felület pedig lehetőleg egyszerű legyen és mindent egyértelműen közöljön, amit kell.

3.2. Alkalmazás Mern Stackkel

Előzőleg már említettem a MERN Stacket, most részletesebben fogok beszélni róla. Neve a MongoDB, az Express.js, a React és a Node.js kezdőbetűiből áll össze. Egy alkalmazáshoz szükségek összes réteg megtalálható benne, egymásra helyezve, innen jön a stack elnevezés.

A következőkben a MERN Stack-beli alkalmazás fejlesztést fogom bemutatni.

3.2.1. MongoDB

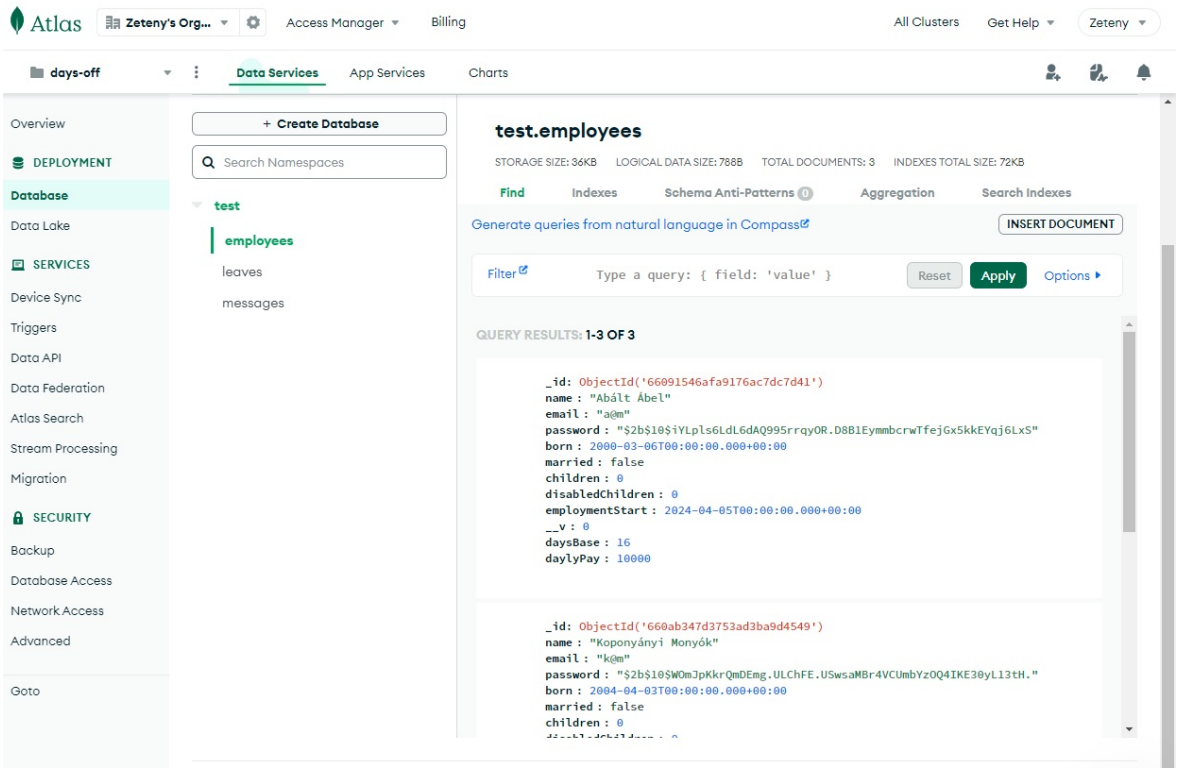
MongoDB Atlas

Ehhez a projekthez a MongoDB Atlas [11] szolgáltatását használtam. Ez egy felhő alapú adatbázis szolgáltatás. Ingyenes lehetőséget is biztosít adatbázis létrehozására, noha igencsak korlátozott megengedett adatmennyiséggel. Ami szerencsére nem okoz gondot ebben az esetben.

A MongoDB Atlas használatához először is kell egy MongoDB Felhő fiókot regisztrálni. A regisztráció után egy projektet indítani, melynek saját klasztere lesz. Egy klaszterben több adatbázis is helyet foglalhat.

Az új klaszteren belül létre kell hozni egy adatbázist, amelyben már eleve kell hogy legyen egy kollekció. Egy kollekció összetartozó dokumentumok gyűjteménye. Egy adatbázis pedig összetartozó kollekciók gyűjteménye, bár tartozhat hozzá csupán egy kollekció is.

A klaszterekhez meg kell adnunk a hozzáférési lehetőségeket. Lehet csak a saját IP-címről hozzáférni, de akár bárhonnan is engedélyezhető a hozzáférés. Ehhez a projekthez az utóbbi lehetőséget választottam, az egyszerűség kedvéért.



3.1. ábra. MongoDB Atlas felhasználói felülete

Az alkalmazás számára szükséges egy kapcsolódási string, hogy az adatbázishoz hozzáférhessen. Ezt célszerű titokként őrizni, bár ennél az alkalmazásnál ez nem számít. A program fájlok közé fog kerülni.

Dokumentumok felépítése

A MongoDB egy noSQL adatbázis, más néven nem relációs. Tehát az adatok nem táblázatokba vannak szervezve. A MongoDB dokumentum alapú, ezek a dokumentumok flexibilisek. A React frontendből küldött JSON formátumba gyűjtött adatokat tárolj, BSON formátumra alakítva.

```
{
        _id: ObjectId('66111c07e620b129128df246'),
        name: "Kalapos Karcsi",
        email: "k.k@mail.com"
}
```

Minden dokumentumnak kell egyedi kulcs, hogy biztosan meg lehessen különböztetni bármely mástól. Erre való az id, amit automatikusan hoz létre az adatbázis egy új dokumentumban.

Mellette még lehet más is funkcionálhat kulcsként a dokumentumok lekérdezésekor. Például az email címeket, feltéve hogy minden email cím egyedi.

Sémák

Mint előbb is említettem, egy dokumentum flexibilis. Valamiféle szabványra mégis szükség van azonban. A mongo sémák arra szolgálnak, hogy a létrehozni kívánt dokumen-

tum helyességét ellenőrizzük velük. Erre a mongoose csomagot fogom használni.

Megadhatjuk, hogy milyen mezők lehetnek a dokumentumban. Például név, születési dátum. Ezek közül melyek megadása kötelező. A mezők adattípusait is meghatározhatjuk.

```
const leaveSchema = new Schema({
        employeeName: {type: String, required: true},
        startDate: {type: Date, required: true},
        endDate: {type: Date, required: true},
        employeeID: {type: String, required: true},
        employeeEmail: {type: String, required: true},
        approved: {type: String, required: true},
        leaveType: {type: String, required: true}
})
```

3.2.2. Node.js

Itt nem a MERN-ben szereplő kezdőbetűk alapján haladok. Az Express a Node-ra épül, így jobbnak látom a Node-ot előrébb venni.

A Node.js [5] egy JavaScript futási idejű környezet. Vele a JavaScript kód böngészőn kívül futhat. Az alkalmazás backendjének alapja.

A telepítése szükséges előfeltétel egy React alkalmazás fejlesztésének megkezdése előtt, mivel a segítségével tudjuk csak létrehozni azt.

npm csomagkezelő

Az npm [12] a Node.js alapértelmezett csomagkezelője. Új projektnél a szükséges csomagokat egyesével telepíthetjük, ahogy haladunk fejlesztés közben.

```
        npm install csomag—neve
```

Ha már meglévő projekten kezdünk el dolgozni, akkor

```
        npm install
```

segítségével minden függőséget telepíteni fog számunkra az npm. Amennyiben a projekthez tartozik egy package.json fájl, mely tartalmazza a függőségeket. Az install parancsszó egyszerűen i betűvel helyettesíthető.

A package.json

A package.json fájl mondhatni a Node.js rendszer szíve. A projektről számos információt tartalmaz, az előbb említett csomagok mellett. Például a a projekt címét.

Példának itt van a backend package.json fájlja.

```
    {
      "name": "backend",
      "version": "1.0.0",
      "description": "",
      "main": "index.js",
      "scripts": {
```

```
    "start": "node server.js",
    "dev": "nodemon server.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "bcrypt": "^5.1.1",
    "body-parser": "^1.20.2",
    "cors": "^2.8.5",
    "dotenv": "^16.4.5",
    "express": "^4.18.3",
    "jsonwebtoken": "^9.0.2",
    "mongoose": "^8.2.1"
  }
}
```

Egymás után rendre a projekt neve, verziója és leírása. Utána a main az alkalmazás kiinduló pontja, ahonnan a futás elindul. A szkriptek az alkalmazás futtatásához. A kulcsszavak egy sztring tömb, az alkalmazás jellemzői kulcsszavakban. Alatta a szerző neve, adatai szerepelhetnek. Azalatt pedig a licenc típusa. A függőségek pedig a szükséges csomagok futtatáshoz.

Ezeken felül még több része is lehet a package.jsonnak, de ezek leírását most mellőzöm.

3.2.3. Express.js

Az Express [6] bemutatására most kerül csak sor. Ez egy kis súlyú, minimalista keretrendszer. Az egyik legnépszerűbb vetélytársai közül. A Node backend és a React frontend közt helyezkedik el, egyszerűsíti a kettő közti kommunikációt. Az alkalmazásban a server és a router fájlokban használom.

A server fájlban először is egy Express alkalmazást kell létrehozni. Ehhez importálni kell az Express-t, az alkalmazást csak utána lehet létrehozni.

Használt metódusai:

- express.json() Elemzi a JSON tartalmú kéréseket. Az elemzett adatokat tartalmazó objektuumot hoz létre. Ez üres, ha a kérés body részében nem volt semmi.
- expresss.Router() Ez hozza létre a routert„ melyhez hozzáadhatjuk a különböző kérésekhez tartozó utakat.

3.2.4. React

Végül, de nem utolsó sorban következik a React [4]. Mivel a felhasználók csak a frontendet fogják látni, így fontosságát nem lehet alábecsülni.

Komponensek

Egy React projekt legkisebb építő kövei. Amikor lehetőség adódik rá, ajánlott újrahasználhatóra írni őket, hiszen az alkalmazás bővülése közben egyszerűbb újra felhasználni eg már meglévő komponenst, más bemeneti adatokkal, mint egy teljesen újat írni.

A React komponensek eredetileg JavaScript osztályok voltak, ezek az osztálykomponensek még a mostani verziókban is támogatottak, bár használatuk megszűnt és nem is ajánlott.

Helyettük függvény komponenseket használunk, melyek JavaScript függvények. A megjelenítendő HTML elemek a visszatérési érték.

```
const Component = () => {
  return (
    <div>
      <h4>Hello World</h4>
    </div>
  )
}
```

JSX

A fenti mintán látszik, hogy a JavaScript kód keveredik a HTML-el. Ez a JSX miatt van így, ami a JavaScript Syntax Extention [13] rövidítése.

Segítségével együtt lehet tartani egy oldal megjelenését és az azt vezérlő logikát. Itt nincs szükség az HTML és JavaScript külön tárolására, esetleg két külön fájlban.

Nem pontos HTML-nek hívni a HTML-szerű kódot. Társulnak hozzá új szabályok.

A visszatérési értéknek egy szülő eleme kell hogy legyen.

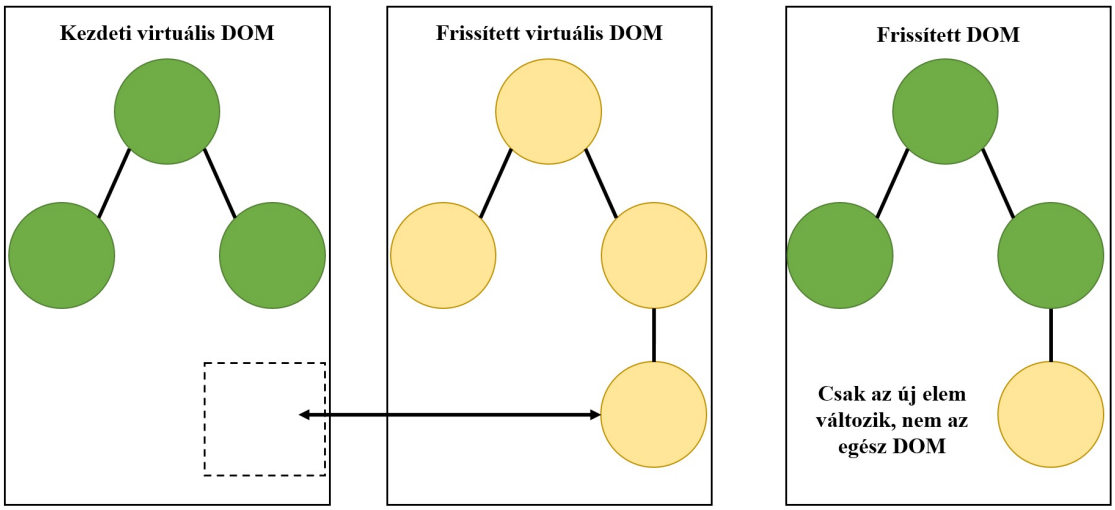
```
// helytelen
return (
  <h4>Hello World</h4>
  <h1>have a nice day</h1>
)

//helyes
return (
  <div>
    <h4>Hello World</h4>
    <h1>have a nice day</h1>
  </div>
)
```

Az önzáró tagokat expliciten le kell zárni.

```
//helytelen
<img src="image.jpg">

//helyes
<img src="image.jpg" />
```



3.2. ábra. A virtuális DOM használata

Végül az attribútum neveket camelCase-szel kell írni, vagyis kisbetűvel kezdődik, új szavakat nem választjuk el kötőjellel, csupán nagybetűvel kezdjük őket.

```
<img
  src="image.jpg"
  className="image"
/>
```

Virtuális DOM

A React előnye még a virtuális DOM használata is. Ezt memóriában tárolja, változás esetén pedig először a virtuális DOM-ot frissíti. A frissített verziót összehasonlítja az előzővel, így megtalálja a kettő közti különbségeket. A böngésző DOM-jában csak a változott elemeket módosítja és csak azokat rendereli újra. Ez sokkal hatékonyabb, mint mindent a böngésző DOM-jában csinálni, mivel a virtuális DOM manipulálása jóval kisebb költségű.

Egyirányú adatkötés

Az adatkötés a megjelenített elem és az őt benépesítő adatok összekapcsolásának folyamatát jelenti. Az egyirányú adatkötés lényege bele van foglalva a nevébe. Iránya fentről lefelé halad, vagyis szülőből gyerek komponens felé.

A gyerek nem tud visszatérni adattal a szülőbe, de a tőle kapott adatokat tudja módosítani.

4. fejezet

Tervezés

Az előző fejezetben rövid pontokban megfogalmaztam az elvárásokat, utána pedig bemutattam a MERN stacket, amiben a megvalósítás fog történni.

Következőkben részletesen fogom leírni az alkalmazás tervét.

4.1. Navigációs sáv

Ez minden oldalon látható a képernyő tetején. Bejelentkezés nélkül csak a bejelentkezési oldal látható. Nem akarjuk, hogy anélkül lássa bárki az alkalmazás többi részét, hiába nem tudna mit kezdeni velük.

Sikeres bejelentkezés után láthatóvá válnak az adott felhasználó számára elérhető oldalak, vagy oldal. Továbbá a kijelentkezés gomb is.

4.2. Bejelentkezés oldal

A felhasználók szemszögéből haladva, az első oldal amivel találkozni fognak, az a bejelentkezési oldal.

Itt az alkalmazás többi része nem elérhető, ha valaki bejelentkezés előtt más oldalra próbál lépni az URL címével, akkor az ne jelenjen meg neki. Átirányítás a bejelentkezési oldalra még jobb lesz.

A bejelentkezéshez email cím és jelszó szükséges. Ha mindkettő érvényes, a bejelentkezés sikeres.

Regisztrációs oldalra nem lesz szükség. A dolgozók profiljait egy adminisztrátor fogja létrehozni. Erről később lesz szó.

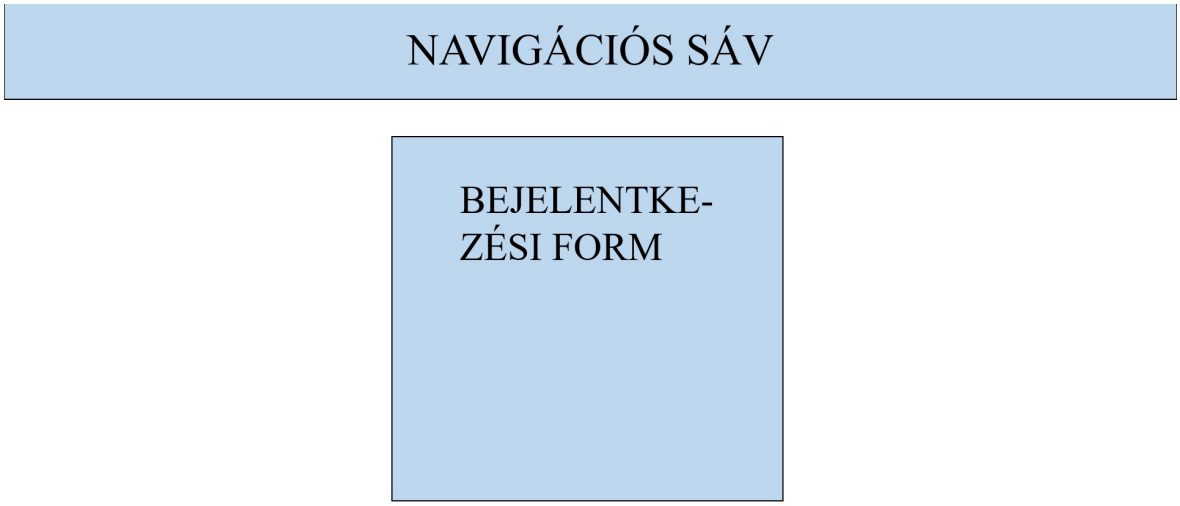
4.3. Saját szabadságok

Sikeres bejelentkezés után ide lesz átirányítva a felhasználó. Adminisztrátori jogosultság nélkül csupán ezt az oldalt tudja elérni az egyszerű felhasználó, a bejelentkezésein kívül.

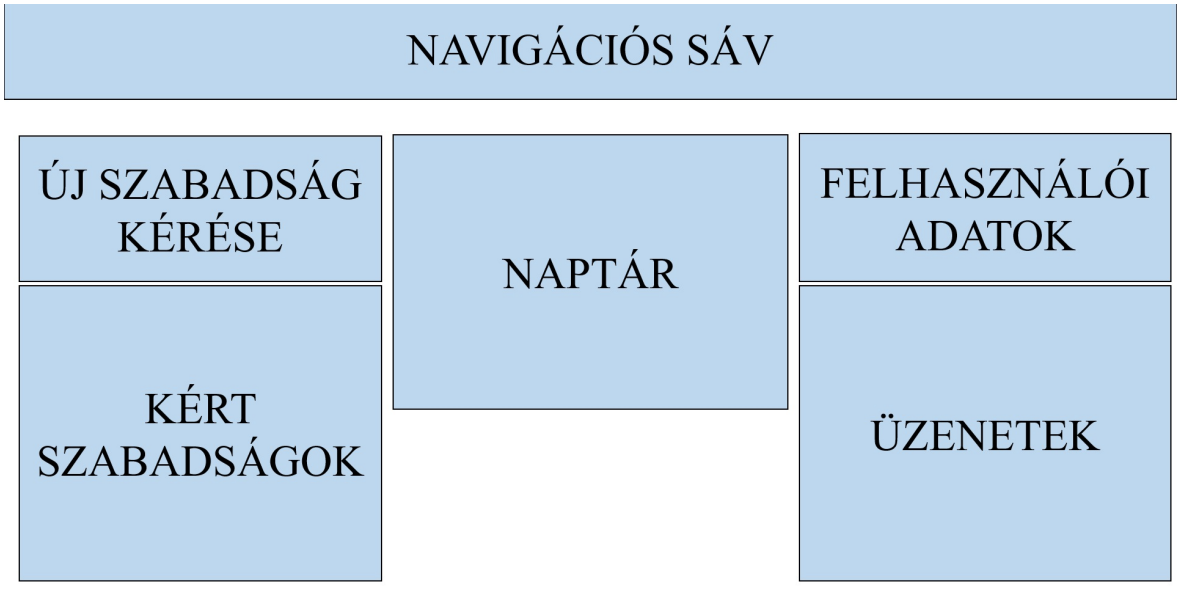
Itt van összegyűjtve minden információ és funkció amire az "alap" felhasználóknak szüksége lesz.

Kérhet új szabadságot, láthatja kért szabadságainak állapotát. Szerkesztheti a még jóváhagyásra váróakat. Naptárban láthatja már jóváhagyottak napjait.

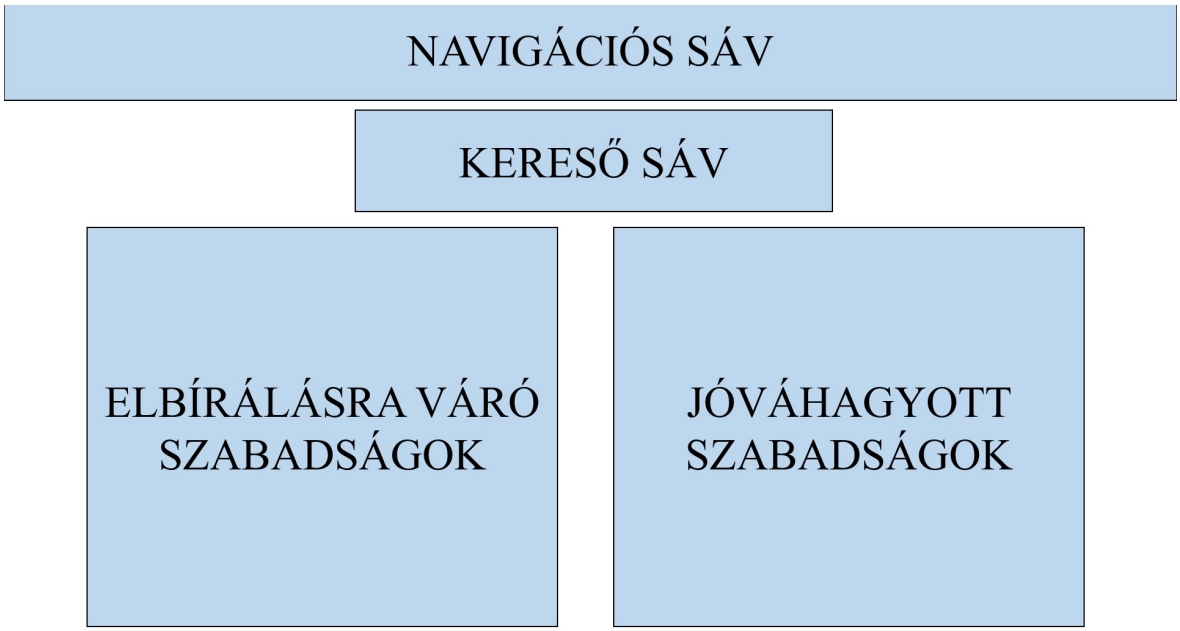
Egy szabadság jóváhagyásáról, elutasításáról üzenetet kap, ezek is itt jelennek meg.



4.1. ábra. A bejelentkezési oldal terve.



4.2. ábra. Saját oldal terve.



4.3. ábra. Szabadság kezelő oldal terve terve.

4.4. Szabadság kezelő oldal

Ez az oldal már csak adminisztrátor számára érhető el.

Itt láthatja az összes jóváhagyásra váró szabadságot. Külön szedve pedig a már jóváhagyottakat.

Itt végezheti el a jóváhagyást és elutasítást. Az elutasítás törölni fogja a kért szabadságot.

Kereső funkció jó lenne, hiszen sok alkalmazott esetén időigényessé válhatna legörgetni a szabadságaik között.

4.5. Alkalmazottak kezelése

Ez az oldal is szintén csak adminisztrátori jogosultsággal érhető el.

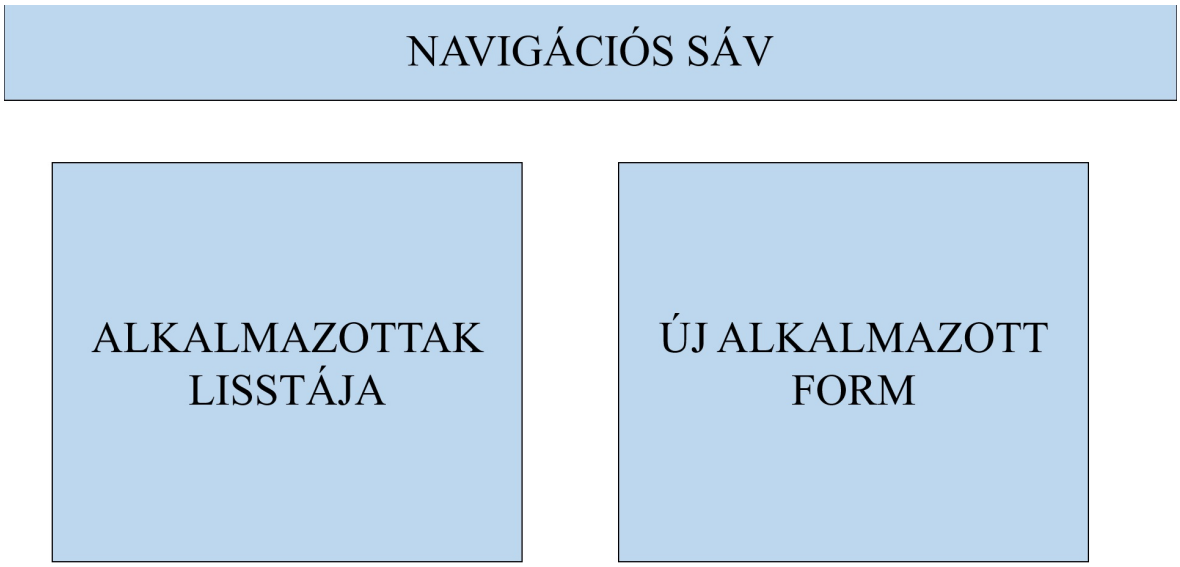
Itt látható minden alkalmazott, profiljaik szerkeszthetőek és törölhetőek. Új alkalmazottnak itt hozható létre profil.

4.6. Felhasználói profil

Egy alkalmazottnak szükséges a neve, email címe, jelszava. A szabadságainak számát több tényező is befolyásolja, ezeket célszerű tárolni.

Ezek az adatok a kor, gyermekek száma, fogyatékkal élő gyermekek száma, veszélyes munkakörülmények (például sugárzásnak van kitéve).

A munkaviszony kezdete is szerepel a profilban, hiszen a munkaviszony hosszától függően kaphat egy alkalmazott több szabadnapot, ösztönözve a céghez való hűséget.



4.4. ábra. Alkalmazott kezelő oldal terve terve.

4.7. Szabadságok adatai

Egy szabadságnak kell hogy legyen első és utolsó napja. Ez a kettő lehet ugyanaz a nap is. Tartalmaznia kell valamit az őt kérő alkalmazottról, hogy beazonosítható legyen.

Erre célszerű egy egyedi alkalmazott azonosítót használni, de nem az alapján fogja ismerni az adminisztrátor. Így benne lesz a neve és email címe is, mivel az emailnek egyedinek kell lennie, a névnek pedig nem.

4.8. Üzenetek

Üzenet akkor jön létre, amikor egy kért szabadságot jóváhagy, vagy elutasít egy adminisztrátor.

Az adott alkalmazottnak kell küldeni, szóval tartalmaznia kell az alkalmazott azonosítóját és a szabadság adatait. Legalább a kezdő és záró napokat.

4.9. Adatbázis

A MongoDB Atlas-szal egyszerű dolgozni. Egy adatbázist kell vele létrehozni, abban pedig külön kollekciókat az alkalmazottak, szabadságok és üzenetek számára.

5. fejezet

Megvalósítás

5.1. Backend

Ebben a szekcióban a backendet nem minden részletében fogom bemutatni. Itt csak az általános működéséről, felépítéséről lesz szó.

Majd a frontend egyes komponenseinél fognak megjelenni a hozzájuk tartozó fájlok vagy részletek.

5.1.1. Modellek

Mivel a MongoDB önmagában nem tartalmaz sémákat a dokumentumok számára, szükséges sajátot létrehozni. A sémák megadják az adott dokumentum elemeit, azoknak típusait.

Ebben áll segítségünkre a mongoose csomag, mely segítségével létrehozhatjuk a szükséges sémákat. Ezekből a sémákból modelleket fog készíteni a mongoose a MongoDB számára.

Példaként itt szerepel az alkalmazott séma. Nem teljességében.

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema

const employeeSchema = new Schema({
  name: {type: String, required: true},
  email: {type: String, required: true, unique: true},
  password: {type: String, required: true},
  born:{type: Date, required: true},
  children:{type: Number},
  employmentStart: {type: Date, required: true},
  daysBase: {type: Number}
})

module.exports = mongoose.model('Employee', employeeSchema)
```

Az első lépés a mongoose megkövetelése. A séma elemei egy alkalmazott adatai. A nem kötelező elemek alapértelmezett értéket kapnak, ha üresen maradnak új alkalmazott létrehozásakor. Például nulla a gyermekek számánál (children). A szabadnapok száma (daysBase) pedig a többi adat alapján lesz megállapítva.

Külön azonosító nincs a sémában, azt az adatbázisba bekerüléskor automatikusan megkapja.

Most pedig a modell következik. A modul exportja az alkalmazott (Employee) modell, ami az előbb megadott sémát használja. Létrehozáskor meg az új alkalmazott adatai össze lesznek vetve a modellel.

5.1.2. Kontrollerek

Az alkalmazottakhoz és a többi modellhez tartozó kérések leírása. GET, POST, DELETE és PATCH mindegyikéhez nem biztos, hogy fog tartozni függvény, de az is előfordul, hogy egyhez több is társul.

Az üzeneteknél nincs PATCH, az alkalmazottaknál pedig van GET az összesre és csak egyre is, azonosító alapján.

```
const Employee = require('../models/employeeModel')
const mongoose = require('mongoose')

const getEmployees = async (req, res) => {...}

const getEmployee = async (req, res) => {
  const {id} = req.params
    if (!mongoose.Types.ObjectId.isValid(id)) {
      return res.status(404).json({error: 'Ervenytelen azonosito'})
        }

  const employee = await Employee.findById(id)
    if (!employee) {
      return res.status(404).json({error: 'Nincs ilyen dolgozo'})
        }

  res.status(200).json(employee)
}

const createEmployee = async (req, res) => {...}

const deleteEmployee = async (req, res) => {...}

const updateEmployee = async (req, res) => {...}

module.exports = {
  getEmployees,
  getEmployee,
  createEmployee,
  deleteEmployee,
  updateEmployee
}
```

Itt csak a getEmployee függvényt hagytam meg teljes formájában, mivel rajta be lehet mutatni amit kell.

Aszinkron, mert a válasz nem biztos, hogy azonnali lesz. Várakozás közben pedig csinálhat mást az alkalmazás, anélkül, hogy hiba lenne a függvény válaszával.

A getEmployee egy alkalmazottat talál meg azonosító alapján. A kérésben érkezik az azonosító, id. A mongoose segítségével ellenőrizzük a kapott azonosító érvényességét.

A válasz az azonosító alapján megtalált alkalmazott lesz, az employee. Itt az await, várj, operátorral fog várni a program az alkalmazott megtalálására.

Ha megtalálta az alkalmazottat, a válasz az alkalmazott lesz. Ellenkező esetben hibát jelez.

5.1.3. Routerek

A kontrollerek tartalmát a routerekben is meg lehetett volna írni, de ezek célja az utak és a hozzájuk tartozó függvények összegyűjtése. Így ha azokat itt definiálnám, az rontaná a kód átláthatóságát.

```
const express = require('express')
const {
    getEmployees,
    ...
} =  require('../controllers/employeeController')
const router = express.Router()

router.get('/', getEmployees)

router.get('/:id', getEmployee)

router.post('/', createEmployee)

router.delete('/:id', deleteEmployee)

router.patch('/:id', updateEmployee)

module.exports = router
```

Az Express-szel létrehozok egy routert, utána jön mindegyik kérés a hozzájuk tartozó utakkal és függvényekkel.

Ha egy kéréshez több függvény is társul, akkor más útvonalon kell lenniük. Az összes alkalmazott lekérdezésénél csak egy / van útnak megadva. Egy alkalmazott lekérdezésénél pedig /:id, hiszen kell hozzá az id.

5.1.4. Szerver

Az Express alkalmazás ebben a fájlban lesz létrehozva. Ide vannak összegyűjtve az utak a router fájlokból, de a bemutatott kódból kihagytam az importálásokat.

```
const app = express()
app.use(express.json())

app.use((req, res, next) => {
  console.log(req.path, req.method)
```

```
  next()
})

app.use('/api/leaves', leaveRouter)
app.use('/api/employees', employeeRouter)
app.use('/api/messages', messageRouter)

mongoose.connect(process.env.DB_URI)
  .then(() => {
    const port = process.env.PORT
    app.listen(port, () => {
      console.log('listening at port', process.env.PORT)
      })
    })
    .catch((error) => {
        console.log(error)
    })
```

Az Express alkalmazás létrehozása után következő sorban megadom, hogy használja az express.json() függvényt. Ez elemezni fogja a bejövő JSON adatokat HTTP kérésekből.

Az express.json() köztes szoftver. Itt röviden olyan kódot jelent, ami egy kérés érkezése és a válaszadás között fut.

Ezután jön még egy köztes szoftver. Ebben a kérés útja és a fajtája kerül naplózásra. Ezután jön a next() függvény, amivel a program a következő köztes szoftverré léphet tovább.

Ezalatt vannak megadva a különböző routerekhez tartozó utak. Mindegyik routerhez egyedi társul, hogy elkülönüljenek egymástól.

A fájl végén jön az adatbázishoz való csatlakozás. A mongoose.connect()-nek az adatbázishoz tartozó kapcsolódási sztring az argumentuma. Amikor sikeresen csatlakozik, naplózza azt a tényt, a porttal együtt.

5.1.5. Kérések hitelesítése

A kérések kezelésekor ellenőrizni kell, hogy küldő felhasználó valóban érvényes jogosultságokkal rendelkezik-e.

Erre szolgál a JSON webtoken. Ezek a tokenek az információ biztonságos küldésére használatosak. Tartalmuk kódolt, itt egy titokkal van aláírva, ami egy sztring. Célszerű véletlenszerűen generált karaktereket használni hozzá. A .env fájlban van tárolva az adatbázis kapcsolódási sztringjével együtt.

A token bejelentkezéskor jön létre a felhasználó számára. Ez csatolva lesz a küldött kéréseihez, így ellenőrizni lehet adatait, jogosultságait.

Három részre tagolt, azok ponttal vannak elválasztva. Ezek a header, a payload és a signature. A headernek két része van. Az aláíráshoz használt algoritmus és a token típusa. A payload, hasznos teher, amiben a felhasználó adatai vannak tárolva. A signature-ben található a titok.

```
        // a token
        eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
```

```
        eyJfaWQiOiI2NjBhZjFjMDdiMDRkY2E0NTc0ODFiMTY
        iLCJlbWFpbCI6ImJAbSIsImlzQWRtaW4iOnRydWUsIm
        lhdCI6MTcxNDIzMzUxOCwiZXhwIjoxNzE0Mjk4MzE4fQ.
        XoGIr—MFQNgt1l68kN_WlCUpnZ8Gv_o7D96cDUR6XbU

        // payload
        {
          "_id": "660af1c07b04dca457481b16",
          "email": "b@m",
          "isAdmin": true,
          "iat": 1714233518,
          "exp": 1714298318
        }
```

A felhasználó adatain kívül még két elem látható a payloadban. Az iat, Issued At Time rövidítése. Magyarul a létrehozás ideje. Az exp pedig az Expiration Time, lejárati idő rövidítése.

A lejárati időt nem kötelező megadni. Ha nincs megadva a token egyszerűen nem jár le, érvényességét soha nem veszíti el. Ha felhasználó kijelentkezik az alkalmazásból, a tokenje törlésre kerül, így a lejárati dátum nélküli token sem biztos, hogy örök éltű lenne.

A token ellenőrzését a requireAuth köztes szoftver fogja elvégezni. Ez megkapja a tokent a kérés headerben, majd dekódolja és kigyűjti belőle a felhasználó adatait.

A requireAuth-ot a routerek használják fel, a védett útvonalakon, vagyis a bejelentkezés oldalon kívül mindenhol.

```
const requireAuth = async (req, res, next) => {
    const { authorization } = req.headers
    const token = authorization.split(' ')[1]

    try {
        const user = jwt.verify(token, process.env.SECRET)
        req.employee = user._id
        // ...
        next()
    } catch (error) {
    // ...
     }
}
```

Ez csupán részlet a requireAuth fájlból, amin a legfontosabb részeit bemutathatom.

A kérés header részéből leválasztja a JSON webtokent, a try blokkban hitelesíti a titkos sztring felhasználásával. Ez visszaadja a dekódolt tartalmat. Itt csak az alkalmazott azonosítót hagytam benne.

A routerekben egyszerű a felhasználása. A router létrehozása után meg kell adni, hogy a router használja.

```
const router = express.Router()

router.use(requireAuth)
```

```
router.get('/', getEmployees)
// ...
```

5.2. Frontend

Ebben a szekcióban néhány részlet fog szerepelni a frontendről, melyek nem illetek egy oldal leírásához sem, vagy többre is vonatkoznak.

5.2.1. Hitelesítési kontextus

Mivel a backendnél a hitelesítéssel zártam, ezért itt azzal kezdem.

Az authContext, hitelesítési kontextus, egy React kontextus fájl. A React kontextusok a komponensek közti adatküldésre használatosak. A bennük lévő állapot változókat közvetlenül tudja az alkalmazáson belül bármelyik komponens változtatni, amibe a kontextust importáljuk.

Kontextus nélkül az egyirányú adatkötés miatt körülményes lenne sok állapotváltozó kezelése.

A kontextusokat kontextus szolgáltatóban kell használni (context provider). Ennek ellenőrzésére társul, hitelesítési kontextus esetén, a hitelesítési kontextus használó (useAuthContext) horog. Ezen keresztül lesz felhasználva a hitelesítési kontextus, hogy az ellenőrzést végre tudja hajtani.

A kontextus ellátók az index.jsx fájlban körbefogják az alkalmazást, így ellátva azt.

Az index.jsx render metódusa argumentumaival:

```
root.render(
  <React.StrictMode>
    <AuthContextProvider>
      <EmployeeContextProvider>
        <LeaveContextProvider>
          <MessageContextProvider>
            <App />
          </MessageContextProvider>
        </LeaveContextProvider>
      </EmployeeContextProvider>
    </AuthContextProvider>
  </React.StrictMode>
)
```

A React.StrictMode (szigorú mód) a fejlesztés során extra ellenőrzést biztosít hibák ellen. Ez minden mást körbe ölel.

Ezen belül találhatók a különböző kontextusok, a kontextusokon belül pedig maga az alkalmazás.

5.2.2. Navigációs sáv

Ez a komponens minden oldalt érint, hisz mindegyiken megjelenik. Bár tartalma változik bejelentkezéstől és felhasználói szerepkörtől függően.

A sáv elemei kapcsos zárójelben van egy div taggal. Ez azt jelenti, hogy a div csak akkor lesz látható, ha az zárójelbeli kifejezés bal oldalán szereplő feltétel teljesül. Ezt feltételes megjelenítésnek hívják.

```
<nav>
  {!employee && (
    <div>
      <Link className='nav-link' to="/">Bejelentkezes</Link>
    </div>
            )}
  {employee && (
    <Link className='nav-link' to="/my-holidays">Szabadsagaim</Link>
  )}
  {admin && (
    <Link className='nav-link' to="/approve">Jovahagyas</Link>
   )}
  </nav>
```

Ez nem a teljes sáv, csak mindegyik feltételhez egy példa. Az employee azt jelenti, hogy be van-e jelentkezve. A bejelentkezés oldal csak akkor jelenik meg, ha nincsen bejelentkezve. A saját szabadságok esetében pont az ellenkező. A jóváhagyásnál pedig nem elég a bejelentkezés, még adminisztrátornak is kell lenni.

5.2.3. Utak regisztrálása

A frontenden regisztrálni kell az oldalakhoz tartozó utakat. Először is minden oldalhoz létre kell hozni egy út (route) fájlt ami egyszerűen csak visszatér az oldallal.

Ezt még meg kell tenni a navigációs sávval és érdemes utat létrehozni arra az esetre, ha a felhasználó nem létező oldalra navigálna.

Az utakat az App.jsx fájlban kell regisztrálni. Itt belőle következik egy részlet.

```
return (
  <LocalizationProvider dateAdapter={AdapterDayjs} adapterLocale='hu'>
    <BrowserRouter>
      <Navbar />
      <Routes>
        <Route path='/'
          element=
            {!employee? <Auth /> : <Navigate to={"my-holidays"} />}
        />
        <Route path='my-holidays'
            element={employee? <MyHRoute /> : <Navigate to={'/'} />}
        />
        <Route path='*' element={<NoMatch />} />
      </Routes>
    </BrowserRouter>
  </LocalizationProvider>
)
```

A lokalizáció a dátumok és miatt fontos, hogy magyarul jelenjenek meg a hét napjai, hónapok nevei és hétfővel kezdődjön a hét.

A BrowserRouteren belül találhatók az utak, itt is feltételes megjelenítést használva. Amint bejelentkezik egy felhasználó, át lesz irányítva a bejelentkezési oldalról. Ha a többi oldalra menne bejelentkezés nélkül, akkor a bejelentkezésire lesz átirányítva. Ha nem létező oldalra menne, akkor pedig a NoMatch-re fog kerülni.

5.3. Bejelentkezési oldal

Az első oldal mellyel egy felhasználó találkozni fog. Nem található rajta sok minden, csupán a bejelentkezési űrlap és a navigációs sáv. A sávon csak a bejelentkezési oldal látszik bejelentkezés előtt.

Regisztrációs űrlapra nincsen szükség, mivel az adminisztrátorok fogják létrehozni az új felhasználói profilokat.

A példa felhasználók bejelentkezési adatai ott szerepelnek az oldal alján, hogy könnyebb legyen a tesztelés.

5.3.1. Bejelentkezési űrlap

A bejelentkezési oldal rövid és a bejelentkezési űrlap máshol nincs felhasználva, ezért nem készítettem belőle önálló komponenst, mint a többi űrlapból.

```
const [email, setEmail] = useState('')
const [password, setPassword] = useState('')

<form className="default—form" onSubmit={handleSubmit}>
  <h3>Bejelentkezes</h3>

  <label className="default—label">email</label>
  <input
    className="default—input"
    type="email"
    onChange={(e) => setEmail(e.target.value)}
    value={email}
  />
  <label className="default—label">jelszo</label>
  <input
    className="default—input"
    type="password"
    onChange={(e) => setPassword(e.target.value)}
    value={password}
  />

  <button className="default—button" disabled={isLoading}>
    Bejelentkezes
  </button>
  {error && <div className="error">{error}</div>}
</form>
```

Az osztálynevek (className) a stílusok meghatározására szolgálnak. A több helyen is felhasznált stílusok az index.css fájlban vannak összegyűjtve, míg a csak egy komponenshez tartozó stílusok külön CSS fájlokba vannak gyűjtve, a hozzájuk tartozó komponensekkel egy mappában.

Az űrlapon két input mező található, az email és a jelszó. A mezők típusai ennek megfelelőek. Eszerint az email mezőbe csak emailt lehet megadni, vagyis kell benne egy @ szimbólumnak lennie. A jelszó mezőbe bevitt karakterek pedig el vannak rejtve.

Az "változáskor" (onChange) attribútumok az input mezőben történő változás esetén meghívják a hozzájuk tartozó függvényt. Itt az email és jelszó értékei változhatnak.

Az email és jelszó értékei állapot (state) változókban vannak tárolva. A React ezekkel oldja meg az oldalon változó adatok kezelését. Tartozik minden állapot változóhoz egy setter függvény, amellyel az értékét lehet változtatni.

5.4. Bejelentkezés menete

5.4.1. A bejelentkezés függvény

A bejelentkezési űrlapnak, mint minden űrlapnak, van egy leadáskor (onSubmit) attribútuma. Ez értelemszerűen leadáskor meghívja a benne található függvényt.

Jelen esetben ez a függvény a handleSubmit függvény. Ez aszinkron, mivel a szervernek elküldeni az adatokat és onnan választ kapni nem azonnali.

A preventDefault megakadályozza az alapértelmezett cselekvést, ami űrlap leadáskor az oldal újratöltése lenne.

Ezután jön a bejelentkezés (login) függvény meghívása, ami kezelni fogja a bejelentkezést. Ez fogja megadni a tölt-e (isLoading) változót a bejelentkezési űrlap leadás gombjának. Amíg nem érkezik válasz a bejelentkezési próbálkozásra, addig nem lehet újra megnyomni a gombot ezáltal.

A bejelentkezés függvény fogja elküldeni a beütött emailt és jelszót a szervernek, ahol ellenőrizve lesz a helyességük.

Sikeres bejelentkezés esetén a lokális böngésző tárolóba helyezi a kapott JSON webtokent, az authContexttől.

```
const login = async (email, password) => {
  setIsloading(true)
  setError(null)

  const response = await fetch('/api/auth/login', {
    method: 'POST',
    headers: {'Content-Type': 'application/json'},
    body: JSON.stringify({email, password})
  })
  const json = await response.json()

  if (!response.ok) {
    setIsloading(false)
    setError(json.error)
  }
  if (response.ok) {
```

```
        localStorage.setItem('employee', JSON.stringify(json))
        dispatch({type: 'LOGIN', payload: json})
        setIsloading(false)
      }
    }
```

Itt megragadom az alkalmat, hogy bemutassam hogyan néz ki egy kérés küldése és a válasz kezelése frontenden.

A válasz (response) változó megvárja a kérésre jövő választ. A kérést a megfelelő úton kell küldeni, a fetch() első argumentuma. A második argumentumban a kérés típusa és tartalma szerepelnek.

Amint a válasz megérkezik, a benne lévő JSON fájlt megkapja egy változó. Sikeres válasz estén itt a kapott token a lokális tárolóba kerül. Máshol például a lekérdezett szabadságok kerülnek átadásra az oldalnak.

5.4.2. Bejelentkezés-kontroller

A kontrollerek szerepét már leírtam az előzőekben, így azt nem ismétlem itt. A bejelentkezés kontrollerében (loginControllerben) csupán két függvény található.

Az egyik a loginEmployee. Ez megkapja az emailt és jelszót a kérésből.

A loginEmployee az alkalmazott modell statikus bejelentkezés (login) metódust hívja meg, ami ellenőrzi a megadott adatokat.

```
employeeSchema.statics.login = async function(email, password) {
    if (!email || !password) {
        throw Error('Toltse ki az osszes mezot!')
    }

    const employee = await this.findOne({email})
    if (!employee) {
        throw Error('Az emailel nincs felhasznalo.')
    }

    const match = await bcrypt.compare(password, employee.password)
    if (!match) {
        throw Error('Teves jelszo.')
    }

    return employee
}
```

A statikus metódusokat a modellben lehet definiálni. Az alkalmazott séma létrehozása után szerepel a fájlban, hiszen annak lesz metódusa.

Először azt ellenőrzi, hogy van-e üres mező. Utána megnézi, hogy a megadott email címmel létezik-e felhasználó. Végezetül ellenőrzi a jelszó helyességét.

Itt a bcrypt csomagot használja fel. Ez jelszavak titkosítására használatos. Nem csupán a jelszó karaktereit változtatja meg, hanem hozzá ad egy véletlenszerű sztringet titkosítás előtt, így még biztonságosabbá téve azt.

A loginEmployee-ba visszatér az email alapján megtalált alkalmazottal. Megnézi, hogy az alkalmazott adminisztrátor-e. Az alkalmazott adataival létrehoz egy JSON webtokent, amit a válaszban elküld.

A JSON webtokent a másik függvény, a createToken, hozza létre.

```
const createToken = (empData) => {
    return jwt.sign(empData, process.env.SECRET, {expiresIn: '18h'})
}
```

5.5. Saját szabadságok

Sikeres bejelentkezés után ide lesz átirányítva a felhasználó. Ezen az oldalon található minden amire az egyszerű felhasználónak szüksége lesz.

Az oldalon megjelennek az felhasználó saját szabadságai, üzenetei és egyéb adatok a kivett szabadságairól. Ezekhez le kell kérdezni a szabadságait és felhasználó adatait.

A lekérdezett adatokat az oldal tovább adja a megfelelő gyermek elemeinek.

5.5.1. Új szabadság

Talán a legfontosabb része az egész oldalnak. Az űrlap alapvető működése nem tér el a bejelentkezési űrlaptól, ezért azt nem fogom részletezni.

Az új szabadság első és utolsó napját kell megadni. Ez a kettő lehet egy és ugyan azon nap, de az utolsó nap nem lehet hamarabbi, mint az első.

Emellett ki kell választani a szabadság típusát. Alapértelmezetten alapszabadság lesz, de lehet még betegszabadság és táppénz is. Betegszabadság egyszerre legfeljebb tizenöt nap lehet.

A szabadságtípus változásával változik a távolléti díj is. Ez látható a típus mellett, napi bérben megadva. A napi bért a szülőtől kapja meg, az alapján számolja ki a távolléti díjakat.

Mivel itt már bejelentkezett felhasználóval van dolgunk, leadáskor a kérés header részében benne van a JSON webtoken, amivel igazolni tudja magát.

Egy szabadság adatai a szabadság sémában a következők: alkalmazott név, első nap, utolsó nap, alkalmazott azonosító, alkalmazott email, állapot és típus.

Az alkalmazott azonosító alapján társítjuk egy alkalmazotthoz. A név és email az elbíráló adminisztrátornak szól, hiszen ő ezek alapján ismeri az alkalmazottat.

5.5.2. Szabadság kérelmek megjelenítése

Amikor egy új szabadság kérelem létrejön, akkor még elbírálásra vár. Ezeket jeleníti meg a displayLeaves komponens.

```
useEffect(() => {
  const waiting = leaves.filter((leave) => {
    return leave.approved == 'dontesre var'
  })

  setPending(waiting)
}, [leaves])
```

```
const DisplayLeaves = ({leaves}) => {

  return (
    <div className="column">
      <div className="leave-container">
        {pending && pending.map((leave) => (
          <LeaveDetails key={leave._id} leave={leave}/>
        ))}
      </div>
    </div>
  )
}
```

A useEffectben kiválogatja az alkalmazott elbírálásra váró szabadságait, csak ezeket adja tovább a következő komponensnek.

A map (térkép) segítségével megadott szabadságok mindegyikére meghív egy függvényt. Itt minden szabadságra megjelenít egy LeaveDetails (szabadság részletek) komponenst, az adott szabadság adataival.

5.5.3. Szabadság részletek

A szabdság részletek (leaveDetails) komponens kap egy szabadságot, aminek egy kártyán megjeleníti a egyes adatait. Az alkalmazott nevét, email címét és a két dátumot.

Két műveletet tud itt elvégezni a felhasználó a kért szabadságaival: törlés és szerkesztés. A törlés egyszerű, a szabadság azonosító alapján törlésre kerül. A szerkesztés már bonyolultabb.

A szerkesztés űrlap megjelenítése feltételhez kötött, ami nem más mint az editFormOpen (szerkesztés űrlap megnyitva) állapot változó értéke.

```
{editFormOpen && (
  <EditLeaveForm
    closeForm={() => {
      setEditFormOpen(false)
    }}
    onSubmit={handleEdit}
    defaultValue={leave}
  />
)}
```

Innen megkapja néhány metódusát és az alapértelmezett értékeit, melyek az adott szabadság jelenlegi értékei.

A szerkesztés gombra kattintva megnyílik a szerkesztés űrlap. Ezen lehet változtatni a kért szabadság első és utolsó napjait, a típust nem.

Leadáskor bezáródik a szerkesztési űrlap, a vezérlést átadja a szerkesztés kezelő függvénynek. Ez egy PATCH kérést küld, ami lecseréli a kiindulási szabadság kérelem adatait a szerkesztettével.

5.1. ábra. Egy jóváhagyott szabadság a naptáron

5.5.4. Jóváhagyottak naptárban

Ha egy szabadság kérelmet jóváhagy egy adminisztrátor, akkor az állapota annak megfelelően változik.

Ez alapján lesznek kiválogatva és napjaik megjelenítve egy naptárban. Ehhez a naptárhoz a react-calendar-t használtam, amely egy ingyenes naptár komponens.

A naptár megkapja a szabadságokat, kiválogatja a jóváhagyottakat, majd azoknak az első és utolsó napjait.

Ha a naptáron éppen látszó hónapban van szabadság, akkor végignézi a napokat. Ha olyan napot talál, ami beleesik egy szabadságba, akkor zöldre változtatja színét. A kék pedig az aktuális napot jelöli, ha éppen nem fedi egy szabadság.

A naptárban egy napra kattintva megjelenik az alkalmazottnak arra napra kivett szabadsága típusával együtt. Ezzel mindig látni fogja, hogy milyen szabadsága lesz vagy volt adott napon, még ha törli a rá vonatkozó üzenetet is.

5.5.5. Felhasználó adatai és üzenetei

A saját szabadságok oldal utolsó gyermek eleme. Ez felelős a felhasználó adatainak és üzeneteinek megjelenítéséért.

Először is lekérdezi a felhasználóhoz tartozó üzeneteket. Az üzenetek létrehozásáról és felépítésükről részletekbe később megyek csak bele, itt elég annyit tudni, hogy szabadság kérelem elfogadásáról vagy elutasításáról értesítenek.

Felül a felhasználó adatai jelennek meg. A maradék kivehető alapszabadságok. Ha egy szabadságtípusból már vett ki legalább egy napot, akkor az is megjelenik, hogy abból hány napot vett ki.

Alatta térképpel jelennek meg a felhasználó üzenetei, hasonlóan mint a szabadságai.

5.6. Jóváhagyás oldal

Ezt az oldalt már csak adminisztrátori jogosultsággal lehet megtekinteni.

Itt is szükség lesz a szabadságokra, de most mindegyik felhasználóét le kell kérdezni. Amint megvannak a szabadságok, továbbadja őket a gyermek elemének.

5.6.1. Szabadság kérelmek megjelenítése

Ezen az oldalon van keresési funkció, az alkalmazottak nevei alapján. A keresett nevet egy kereső doboz típusú inputba kell begépelni, a doboz változáskor hasonlóan jár el, mint az űrlapok input mezői.

```
const SearchBox = ({placeholder, onChangeHandler}) => (
  <div className='search-container'>
    <input
      className='search-box'
      type='search'
      placeholder={placeholder}
      onChange={onChangeHandler}
    />
  </div>
)
```

A helykitöltő szöveget, ami akkor jelenik meg, ha üres a kereső doboz, és a változáskezelőt a szülő elemtől kapja meg.

A kereső dobozban történő változáskor, a változáskezelő azonnal frissíti a keresés tartalmát tároló állapotváltozót. Ezután a szabadságokból kiszűri azokat, amelyekben az alkalmazott nevében megtalálható a kereső doboz tartalma. Kis és nagy betűkre nem érzékeny.

Innen a kereséssel leszűrt szabadságok kerülnek a két gyermek elembe.

A két gyermek elemben rendre a jóváhagyásra várók és a jóváhagyottak lesznek megjelenítve. Mindkettő kiválogatja a neki kellő szabadságokat, és csak azokat jeleníti meg.

5.6.2. Szabadság részletek

A jóváhagyott szabadságokkal nem tud semmit kezdeni az adminisztrátor, csak megtekinteni tudja őket. Az elbírálásra váró szabadságokat az adminisztrátor nem tudja szerkeszteni, csak megtagadni és jóváhagyni.

Éppen ezért külön szabadság részletek fájlok vannak az adminisztrátor számára. Egy az elbírálásra váróak megjelenítésére, a másik pedig a jóváhagyottakéra.

A megtagadás az egyszerűbb, törli a szabadság kérelmet, utána létrehoz egy üzenetet a törlésről.

A jóváhagyás megváltoztatja a szabadság állapotát jóváhagyottra, utána létrehoz egy üzenetet a jóváhagyásról. A szabadságot kérő felhasználó adatait megváltoztatja a szabadság típusa alapján.

5.6.3. Üzenet és jóváhagyás/megtagadás

A jóváhagyás és megtagadás is átállítja a szabadság állapotát a megfelelő értékre, utána meghívja a sendMessage (üzenetküldés) függvényt.

Ez a függvény létrehoz egy üzentet. Az üzenetet részei: szabadság állapota, alkalmazott azonosítója, első nap, utolsó nap és a szabadság típusa.

```
const sendMessage = async (approved) => {
        const message = {status: approved,
```

```
        employeeID: leave.employeeID, startDate: leave.startDate,
        endDate: leave.endDate, leaveType: leave.leaveType}

        const response = /* ... */
        const json = await response.json()

        if (!response.ok) {/* ... */}

        if (response.ok) {
            dispatchMessage(/* ... */)
            if (approved == 'jovahagyva') {
                let edited = leave
                edited.approved = approved
                handleEdit(edited)

                let leaveLength = 0
                leaveLength =
                    new Date(message.endDate) —
                    new Date(message.startDate)
                leaveLength = (leaveLength / 86400000) + 1

                let toEdit = await getEmployee(message.employeeID)
                updateDays(toEdit, leaveLength, leave.leaveType)
            } else if (approved == 'megtagadva') {
                handleDelete()
            }
        }
    }
```

Ha jóváhagyás hívta meg a függvényt, a szabadság hossza kiszámításra kerül. Ez a megfelelő típusú szabadnapokhoz lesz hozzáadva. Alapszabadság esetén levonja a hosszt a maradék alapszabadságok közül.

Megtagadás esetén az üzenetküldés után törlésre kerül a szabadság. Jóváhagyásnál szerkesztésre.

5.6.4. Összes jóváhagyott naptárban

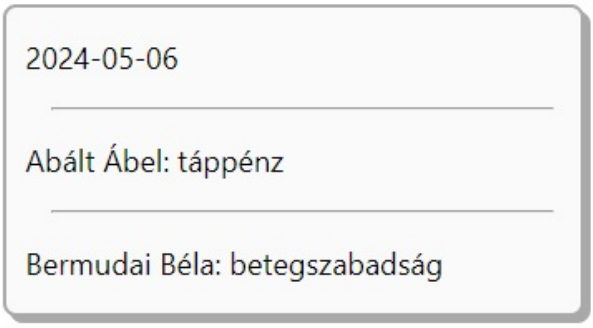
Ezen az oldalon is található egy naptár, de itt már az összes alkalmazott jóváhagyott szabadságai szerepelnek benne. Egy adott napra kattintáskor megjelennek az akkor szabadnapos alkalmazottak, a szabadságaik típusaival együtt.

A kérelmeket elbíráló adminisztrátor így könnyedén fel tudja mérni, hogy egyes napokon milyen munkaerő hiányt jelentene egy kérelem elfogadása.

A keresés használatával a naptárban csak a keresett alkalmazott szabadságai fognak megjelenni.

5.7. Alkalmazottak kezelése

Ezen az oldalon lehet létrehozni új alkalmazottakat, meglévőket szerkeszteni, törölni.



5.2. ábra. Egy nap a szabadnapos alkalmazottakkal

Itt az alkalmazottak listájára lesz szükség, ezt kérdezi le az oldal és adja tovább a megfelelő gyermeknek.

5.7.1. Alkalmazottak megjelenítése

Ebben az elemben is van keresési funkció, az alkalmazottak között név alapján. Ez ugyan úgy működik mint az előző oldalon leírt keresés.

Az alkalmazottakat átadja az alkalmazott részletek komponensnek, ami megjeleníti őket, mint a szabadság részletek a szabadságokat.

5.7.2. Alkalmazott részletek és szerkesztés

Az alkalmazott részletek kártyán a név és email cím van megjelenítve. Emellett két gomb van rajta. A szerkesztés és törlés.

A szerkesztés úgy működik, mint a szabadság szerkesztés. A törlésben már van különbség a szabadság törléshez képest.

Ez nem csak az alkalmazottat törli, hanem a hozzá tartozó szabadságokat és üzeneteket is. Erre külön függvényeket hív meg, melyek az alkalmazott azonosító alapján megtalálják és törlik a szabadságokat és üzeneteket.

5.7.3. Új alkalmazott létrehozása

Az új alkalmazott űrlappal hozható létre új felhasználói profil alkalmazott számára. Működése olyan, mint a többi űrlapé.

Annyi különbség van benne a többihez képest, hogy a nem kötelező mezők alapértelmezett értéket kapnak, abban az esetben ha üresen hagyja őket az adminisztrátor. Például a családi állapot nem, gyerekek száma pedig nulla értéket fog kapni.

A megadott adatokból lesz kiszámítva, hogy hány nap alapszabadságra jogosult az alkalmazott. Húsz az alapértelmezett, de ehhez járhat még pótszabadság is a következő esetekben:

- Életkor alapján pótszabadság. A maximum 45 éves kortól 10 nap
- Minden gyermek után 2 nap. 3 vagy több esetén ez 7 nap.
- Minden fogyatékkal élő gyermek után 2 nap.

6. fejezet

Összefoglalás

Az alkalmazás elkészítése során sok új dolgot tanultam, hiába részben ismerős alapokból indultam ki. A Reacttal dolgoztam már a szakdolgozat elkezdése előtt, nyári gyakorlatom során. Ahhoz a projekthez MongoDB adatbázis tartozott. Innen a MERN stack egyértelmű választásnak tűnt.

Olyan témát akartam, amivel backenden pótolhatom hiányosságaimat, saját felhasználó hitelesítést készíthetek, mivel javarészt frontend tapasztalattal rendelkeztem. A szabadság nyilvántartáshoz kellenek profilok, különböző jogosultságok, így tökéletesnek ígérkezett.

Érdekes volt megtanulni hogyan működik az adatbázisban egy dokumentum létrehozása és a felhasználói adatok ellenőrzése. Utólag nem is tűnik bonyolultnak. A legtöbb időt mégis a frontenddel töltöttem, a fejlesztés ezen része jobban is érdekel.

Az oldalakon való munka közben jöttem rá több mindenre a végleges kinézettel és funkciókkal kapcsolatban. Például a jóváhagyás oldali keresés első elgondoláskor nem is merült fel, de tesztelés közben annyi szabadság kérelem gyűlt össze, hogy egyértelműen hasznos lett.

Az alkalmazás az eredeti célkitűzéseket elérte. Mindenki tud kérni szabadságot, látja az elfogadott szabadságkérelmeit. Az adminisztrátorok tudnak jóváhagyni és megtagadni szabadságot, alkalmazottakat kezelni. Bővítésre pedig mindig bőven lesz lehetőség.

Például ha nagyobb cég használná az alkalmazást, akkor különböző adminisztrátori szintekre lehetne szükség, hogy csak a saját részlegük dolgozóit lássák. Telefonon pedig jelenleg nem jó élmény a használata, bár ott is működik.

Összességében ez egy jó kiinduló pont. Hasznos készségekre tettem szert a megírásával, a manapság leggyakrabban használt technológiákkal. További fejlesztéssel nem csak ezeket a készségeket fogom tovább fejleszteni, hanem munkakereséskor jobban fog mutatni az önéletrajzomban.

7. fejezet

Summary

During the development of the application, I learned many new things, despite building on somewhat familiar foundations. I had worked with React before, during my summer practice. The database for that project was a MongoDB one, so the MERN stack seemed like the obvious choice.

I wanted a topic that would allow me to remedy my shortcomings on the backend and require user authentication, since I mostly had frontend experience. Leave management requires profiles and different permissions; thus, it promised to be perfect.

Learning how a document is created in the database, the process of validating user data was interesting. Looking back, it doesn't seem that complicated. However, the most time-consuming part was the frontend. That part of development interests me more as well.

The final appearance and functions of the pages only took form during development, as some things hadn't even occured to me while planning. The search bar on the approve page was one of these. Only after filling the page completely with leave requests did I think of it.

The application meets the criteria laid out for it. Everyone can ask for leaves, see their own approved leaves. Administrators can approve or deny leaves, handle employees. There will always be room for expansion.

For example, if a larger corporation were to use the application they could require different levels of administrators, who could only see their own department's employees. Using the application on a smartphone isn't a pleasant experience, despite it working there too.

All in all, this is a good starting point. I acquired useful skills in writing it using some of the most widespread technologies of today. With further development, I will not only hone these skills, but it will look even better on my resume.

Irodalomjegyzék

[1] HTML
Nagy Gusztáv. Web programozás alapismeretek, 2011

[2] CSS
Nagy Gusztáv. Web programozás alapismeretek, 2011

[3] JavaScript
Szabó Bálint. Webprogramozás I., 2015

[4] React
`https://react.dev/`

[5] Node.js
`https://nodejs.org/docs/latest/api/`

[6] Express
`https://expressjs.com/en/5x/api.html`

[7] MongoDB
`https://www.mongodb.com/docs/`

[8] Visual Studio Code
`https://code.visualstudio.com/Docs`

[9] Postman
`https://learning.postman.com/docs/introduction/overview/`

[10] Git és GitHub
`https://www.w3schools.com/git/git_intro.asp?remote=github`

[11] MongoDB Atlas
`https://www.mongodb.com/docs/atlas`

[12] npm
`https://docs.npmjs.com/`

[13] JSX
`https://react.dev/learn/writing-markup-with-jsx`

Linkek ellenőrizve: 2024. május 1.

CD Használati útmutató

A CD tartalma a következők:

- Tőzsér-Zétény-Csaba-dolgozat.pdf
- A Tőzsér-Zétény-Csaba-dolgozat-tex mappában a LaTex fájlok és hozzájuk tartozó képek
- A days-off mappában pedig a forráskód
- Használati útmutató, ennek a fejezetek másolata külön pdf fájlban

Interneten elérés

A `render.com` révén a `https://days-off-8avj.onrender.com/` címen elérhető bárki számára. Ez a `render.com` szolgáltatásának ingyenes verzióján fut, így a legelső betöltés hosszabb időbe is telhet, de utána már gond nélkül válaszol mindenre.

Az alkalmazás futtatása saját számítógépen

Ehhez szükség lesz a days-off nevű mappára, a Visual Studio Code-ra és internet elérésre az adatbázishoz való csatlakozásra.

1. Visual Studio Code-ban megnyitni a days-off mappát
2. Visual Studio Code-ban megnyitni két terminált
3. Az első terminálban a frontend mappát kiválasztani a *cd frontend* paranccsal
4. Kiadni az *npm i* parancsot függőségek telepítésére
5. A második terminálban kiválasztani a backend mappát a *cd backend* paranccsal.
6. Függőségeket telepíteni az *npm i* paranccsal.
7. Kiadni az *npm run dev* parancsot
8. Az első terminálban pedig az *npm run start* parancsot

Az utolsó lépés után egy darabig várni kell amíg az alkalmazás elindul. Ez számítógép függő.

Az alapértelmezett böngészőben automatikusan meg kell hogy nyíljon az alkalmazás. Ha ez mégsem történne meg, akkor a `http://localhost:3000/` címet felkeresve lesz megtalálható.

A megjelenő bejelentkezési oldalon, a bejelentkezési űrlap alatt találhatóak néhány példa felhasználó bejelentkezési adatai.