BMSZC Bláthy Ottó Titusz Informatikai Technikum

1032. Budapest, Bécsi út 134.

Vizsgaremek

Havi Munkaidő Beosztás Tervező

Készítette:

Kavarnyik Zsuzsanna

Kádár László Marcell

Vizsgaremek adatlap

A Vizsgaremek készítői:

Neve: Kavarnyik Zsuzsanna
E-mail címe: kavarnyikzs@gmail.com

Neve: Kádár László Marcell
E-mail címe: kadarlaszlomarcell@gmail.com

A Vizsgaremek témája:

Havi munkaidő beosztás tervező készítése, egy vállalat számára, amely segíti a boltvezető munkáját, a kódban meghatározott paramétereknek eleget téve automatikusan készíti el a beosztást, ezzel meggyorsítja az egyébként több órás folyamatot. Továbbá bármikor elérhetővé és átláthatóvá téve az egyes napokban dolgozó kollégák számára is, hogy melyik műszakban, melyik kollégákkal végzik munkájukat, mikor van szabadnapjuk.

A Vizsgaremek címe:

Havi Munkaidő Beosztás Tervező

Kelt: Budapest, 2026. Január 29.

....

Kavarnyik Zsuzsanna Kádár László Marcell

Eredetiség nyilatkozat

Alulírott tanuló kijelentem, hogy a vizsgaremek saját és csapattársa(i)m munkájának eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült vizsgaremekrészét képező anyagokat az intézmény archiválhatja és felhasználhatja.

Kelt: Budapest, 2026. Január .29.

.....
Kavarnyik Zsuzsanna

.....
Kádár László Marcell

Tartalomjegyzék

1. Bevezetés

1.1. Problémafelvetés

A mai rohanó világban létfontosságú, hogy minden feladatot a lehető leghatékonyabban végezzünk el. A modern és strukturált vállalati környezetben egyre nagyobb kihívást jelent az alkalmazottak hatékony beosztása különböző műszakokba, a szabadságok kezelése, azok jóváhagyása, valamint az információk eljuttatása az érintett dolgozókhoz. A manuális megoldások gyakran átláthatatlanok, időigényesek és hibalehetőségeket hordoznak. Ráadásul a változtatások nehezen követhetőek, sokszor bonyolultabb áttervezni, mint egy új beosztást elkészíteni. Figyelemmel kell lenni a munkavállalok egyéni igényeire, a Munka Törvénykönyvében szereplő szabályokra, a Shiftek közötti munimális pihenőidőre, a ledolgozandó optimális munkaórára. És ezeket az adatokat összesíteni is kell. Leadni a jelentéseket a felsővezetők és a bérszámfejtők felé. Egy központi rendszer, melyhez betekintő hozzáférése lehet szabadon azon dolgozóknak, akiknek erre szükségük lehet fontos támogatás egy boltvezető részére. Rendkívül sokrétűen lehet szabályozni. Kényelmesé és tetszetőssé lehet tenni. Továbbá a mai világban elengedhetetlenül fontos , hogy környezet tudatos életvitelhez igazodik azzal, hogy nem kell kinyomtatni minden egyes változtatás alkalmával, hanem bárki elérheti a nap bármely szakában, akár mobilon is.

1.2. Általános ismertető

A **Vizsgaremek** egy teljes értékű, konténerizált webalkalmazás, amely egy szervezet **felhasználókezelési és műszaktervezési** folyamatait valósítja meg. A rendszer célja egy olyan modern, biztonságos és skálázható megoldás létrehozása volt, amely a valós ipari környezetekben is alkalmazott technológiákat és tervezési elveket használja.

A projekt a következő fő területeket fedi le: - Backend fejlesztés (Node.js, Express) - Adatbázis-tervezés és kezelés (MongoDB) - Frontend fejlesztés (React) - Hitelesítés és jogosultságkezelés - Konténerizáció és telepítés (Docker, Docker Compose)

Ez a dokumentáció **kifejezetten védéshez készült**, célja a teljes rendszer működésének, felépítésének és tervezési döntéseinek részletes bemutatása.

Az általunk fejlesztett alkalmazás célja egy olyan egységes rendszer létrehozása, amely lehetőséget biztosít:

- az alkalmazottak munkaidőbeosztásaik kezelésére,
- a szabadságkérelmek felülbírálására,
- automatikus műszakbeosztás készítésére,

- valamint a beosztások alkalmazottak felé történő közzétételére,
- és a havi munkaidő elszámolások elkészítésének meggyorsítására.

A szoftver forradalmasítja a céges ügyintézés folyamatát azáltal, hogy gyors, digitális, átlátható és biztonságos megoldást kínál.

Segíti a boltvezető munkáját, átláthatóvá téve az egyes napokban dolgozó kollégák számára is, hogy melyik műszakban, melyik kollégákkal végzik munkájukat, mikor van szabadnapjuk.

Lehetővé teszi számukra, hogy bizonyos napokra betervezett program miatt ne legyenek beosztva a „Blocked Date” funkcióval.

A „Create Schedule” funkció a kódban meghatározott paramétereknek eleget téve autómatikusan készíti el a beosztást, ezzel meggyorsítja az egyébként több órás folyamatot.

Az alkalmazás támogatja:

- felhasználók hozzáadását és törlését,
- szabadnapok kezelését,
- az adatok adatbázisban történő tárolását,
- a későbbi megtekintést és ellenőrzést

1.3. Munkamegosztás

A tanulók a program elkészítése során a következő munkamegosztásban dolgoztak:

- Kádár László Marcell: Frontend felület kialakítsa
- Kavarnyik Zsuzsanna: Backend elkészítése

A projektmunka dokumentációját a Worldben egy megosztott dokumentum segítségével készítették el közösen.

2. Rendszerarchitektúra áttekintése

2.1. Réteges architektúra

A rendszer három, logikailag elkülönülő rétegből áll:

- Prezentációs réteg (Frontend – React)

Felhasználói felület megjelenítése:
Böngészőben fut
HTTP kéréseken keresztül kommunikál a Backenddel

- Alkalmazáslogikai réteg (Backend – Node.js / Express)
  REST API végpontok biztosítása
  Hitelesítés és jogosultságkezelés
  Üzleti logika megvalósítása
- Adatkezelési réteg (MongoDB)
  Tartós adatmegőrzés
  Felhasználók és műszakok adatainak tárolása

A rétegek közötti szétválasztás növeli a rendszer:

- átláthatóságát,
- karbantarthatóságát,
- továbbfejleszthetőségét.

3. Konténerizáció és indítás (Docker)

3.1. A konténerizáció fogalma

A konténerizáció lehetővé teszi, hogy az alkalmazás:

- minden környezetben azonos módon fusson,
- a függőségeitől elszigetelten működjön,
- könnyen telepíthető és skálázható legyen.

A projekt Docker-alapú, vagyis az alkalmazás nem közvetlenül a hoszt gépen fut, hanem konténerekben.

3.2 Használt konténerek

A rendszer az alábbi konténereket használja:

Konténer	Szerep
Backend	Node.js + Express API
MongoDB	Adatbázis
Frontend	React build kiszolgálása
NGINX	Reverse proxy

3.3. Docker Compose

A Docker Compose segítségével a teljes rendszer egyetlen paranccsal indítható.

A docker úgy működik, mint egy virtuális gép. De itt csak a Linux kerneltől felfele lévő dolgok vannak. Tehát nem kell az egész MongoDB, hanem elég a Compass, hogy be tudjuk szúrni az első felhasználót. És egyébként a konténrből használjuk.

Előnyei:

- egységes indítási folyamat,
- szolgáltatások közötti hálózatkezelés,
- egyszerű konfiguráció.

Ez a megoldás ipari környezetben is széles körben alkalmazott.

Szükséges hozzá a MongoDB Compass továbbá a Docker desktop letöltése és telepítése.

Ez után újra kell indítani a gépet.

Ezeket befejezvén meg kell nyitni a Github-ról letöltött projektet a Visual Studio Code ablakban, ahol a terminalban le kell futtatni a “docker compose up” parancsot.

A parancs futtatása után a MongoDB Compass programban csatlakozni kell a localhost:27018 -as porton elérhető adatbázishoz, ahol az users collection-be bekell illeszteni az alábbi kódot:

{

"_id": { "$oid": "6276688e8ac0526064fb3ade" },

"username": "admin",

"hash":
"864bcf999790114b894299fbc5899cd5c998b3166e1deacc69096fb10b8eda84d2ff1c36637
ad4679621b1c17ab3ee2260f6ef4f40c722a0528d645afaeeecce",

"salt": "1322927e381826e0f7a6fa6ccf4ea81660301f4956ef889eef60e43a2c32065b",

"admin": true,

"blockedDates": [],

"__v": 0

}

Így lesz egy admin felhasználónk, aminek a jelszava admin

4. Adatbázis-tervezés (MongoDB)

4.1. Miért MongoDB?

A MongoDB választásának okai:

- dokumentum-alapú adatmodell,
- rugalmas séma, szemben a MySQL-el, ami fix sémával dolgozik
- könnyű JSON-kezelés,
- az adatokat Json szerű formában Bson-ben tárolja
- JavaScripthez való kiváló illeszkedés,
- gyors fejlesztési lehetőség,
- jó skálázhatóság.

A MongoDB server, maga az adatbázis motor, a háttérben fut MongoDB Shell (mongosh) parancssoros kliens, ezzel beszélünk az adatbázissal.

A MongoDB Compass egy grafikus felület, kezdőknek, mint mi is, nagyon ajánlott.

Nem kell eltárolni semmit a gépen, hanem, csak a dockerre, és docker konténerekből telepítjük fel.

A Dockert úgy működik, mint egy virtuális gép. De itt csak a Linux kerneltől felfele lévő dolgok vannak. Tehát nem kell az egész MongoDB, hanem elég a Compass, hogy be tudjuk szúrni az első felhasználót. És egyébként a konténrből használjuk.

A docker telepítjük WSL2-vel, így sokkal gyorsabb lesz az alkalmazás hosszútávon, csak a Docker telepítése így több időt vesz igénybe.

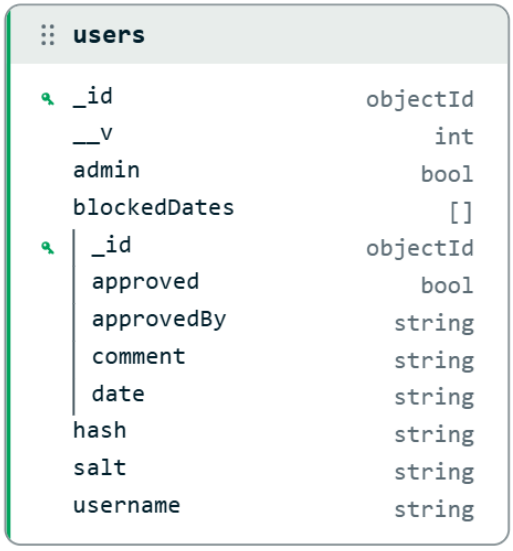
Szemben a relációs adatbázisokkal, ez egy dokumentum alalpú adatbázis, és nincsen fix séma meghatározva. Ami lehetővé teszi a fejlesztés közbeni szabad bővítést, hiszen a struktúra dinamikusan változtatható.

A Mongo DB rendelkezik egy nagyon komoly kiépített modulrendszerrel a React-hez. Nem lennének jók a relációs adatbázisok, a túl szigorú struktúra miatt, mert ez egy nagyon dinamikusan fejlődő cég, sok és hirtelen változással, amit így könnyeb lesz követni.

Későbbiekben szeretném bővíteni a programot egy leltározási szoftverrel. Illetve további boltok nyitásakor lehet külön egységekre bontani. Eddig felvettük a dolgozókat név-jelszóval. Később pedig meg fogjuk toldani a különböző értékesítési egységekhez csatolással, amik most még nem léteznek. Mivel ez egy nem szigorúan strukturált rendszer, így a későbbiekben nem kell módosítani a már feltöltött adatokat, hogy kompatibilis legyen az újonnan feltöltendő több mezővel rendelkező adatokkal, mint például egy új értékesítési egység.

4.2. Adatbázis diagram

1.,ábra



4.3. User kollekció séma (users)

Kétfélék. Admin vagy nem admin. Ezt a middleware, az isAdmin dönti el, ha admin, akkor true, és tovább enged a controllerhez, ha nem, akkor hibát dob. Azaz, nem enged pl. felhasználót törölni.

METÓDUS	ÚTVONAL	JOGOSULTSÁG	LEÍRÁS
GET	/api/user	User	Aktuális felhasználó
GET	/api/user	Admin	Összes elhasználó

A MongoDB user collection séma azt határozza meg, hogy egy felhasználót milyen mezőkkel (attribútumokkal) tárolunk az adatbázisban.

MongoDB-ben nincs kötelező, merev séma, de a gyakorlatban logikusan felépített struktúrát használunk az adatok egységessége miatt.

Egy tipikus **users collection** például így is kinézhet, mint nálunk az admin:


```
{
  "_id": { "$oid": "6276688e8ac0526064fb3ade" },
  "username": "admin",
  "hash":
"864bcf999790114b894299fbc5899cd5c998b3166e1deacc69096fb10b8eda84d2ff1c36637a
d4679621b1c17ab3ee2260f6ef4f40c722a0528d645afaeeecce",
  "salt": "1322927e381826e0f7a6fa6ccf4ea81660301f4956ef889eef60e43a2c32065b",
  "admin": true,
  "blockedDates": [],
  "__v": 0
}
```


_id: MongoDB által automatikusan generált egyedi azonosító

username: bejelentkezéshez használt adatok (érdemes egyedivé tenni indexszel)

hash: a jelszó *soha nem sima szövegként*, hanem hash-elve kerül tárolásra

salt: titkosítás koncepció,

admin: jogosultsági szint (pl. user, admin)

2., ábra

4.4. Shift kollekció séma (shifts)

Ez a struktúra heti bontásban tárolja a műszakokat.

Egy **beosztástervező rendszerben** a **shifts kollekció** célja, hogy **heti bontásban** tárolja a dolgozók műszakjait. Ez különösen hasznos, ha előre tervezett (pl. heti) beosztásról van szó, és gyakran kell lekérdezni egy adott hét adatait.

MEZŐ	TÍPUS	LEÍRÁS
_id	ObjectId	Egyedi azonosító
weekStart	Date	Hét kezdete
assignments	Array	Beosztások
assignments.user	ObjectId	Felhasználások
assignments.day	Number	Nap index

3., ábra

backend > models > JS Shift.js > ...

```
1   const mongoose = require('mongoose');
2   const Schema = mongoose.Schema;
3
4   const Shift = new Schema({
5     name: String,
6     date: Date,
7     data: Object,
8     savedBy: String,
9   });
10
11  module.exports = mongoose.model('Shift', Shift);
12
```

5. API végpontok teljes listája

5.1. Hitelesítés

```
1  // USER API
2  router.get('/api/user', (req, res) => {
3    if (req.isAuthenticated()) {
4      const { id, username, admin, blockedDates } = req.user;
5      res.json({ id, username, admin, blockedDates, isAuthenticated: true });
6    } else {
7      res.json({ isAuthenticated: false });
8    }
9  });
```

4., ábra

Az a felhasználó hitelesítése.

A jelszó ellenőrzése.

A felhasználó be van-e jelentkezve, admin-e?

METÓDUS	ÚTVONAL	JOGOSULTSÁG	LEÍRÁS
POST	/register	Admin	Felhasználó létrehozása
POST	/login	Public	Bejelentkezés
GET	/logout	User	Kijelentkezés

5.2. Műszakok

Két műszak van beállítva: mid és evening. A bolt működéséhez, kell egy nyitást és egy zárást végző személy.

METÓDUS	ÚTVONAL	JOGOSULTSÁG	LEÍRÁS
POST	/api/shifts	Admin	Műszak létrehozása
GET	/api/shifts	User	Műszak

6. Rendszerkövetelmények

6.1. Szerver oldali követelmények

- Node.js
- MongoDB
- Docker
- Docker Compose
- Minimum 8 GB RAM
- Minimum 4 magos processzor
- Minimum 10 GB tárhely

6.2. Kliens oldali követelmények

- Chromium alapú böngésző (Chrome, Edge)
- Windows 11 követelmények:

64 bites CPU
4 GB RAM (ajánlott: több)
64 GB tárhely

7. Telepítési útmutató

Szükséges hozzá a MongoDB Compass továbbá a Docker desktop letöltése és telepítése.

A docker telepítjük WSL2-vel, így sokkal gyorsabb lesz az alkalmazás hosszútávon, csak a Docker telepítése így több időt vesz igénybe.

Ez után újra kell indítani a gépet.

Ezeket befejezvén meg kell nyitni a Github-ról letöltött projektet a Visual Studio Code ablakban, ahol a terminalban le kell futtatni a "**docker compose up**" parancsot.

A parancs futtatása után a MongoDB Compass programban csatlakozni kell a **localhost:27018** -as porton elérhető adatbázishoz, ahol az users collection-be bekell illeszteni az alábbi kódot:

{

"_id": { "$oid": "6276688e8ac0526064fb3ade" },

"username": "admin",

"hash":
"864bcf999790114b894299fbc5899cd5c998b3166e1deacc69096fb10b8eda84d2ff1c36637
ad4679621b1c17ab3ee2260f6ef4f40c722a0528d645afaeeecce",

"salt": "1322927e381826e0f7a6fa6ccf4ea81660301f4956ef889eef60e43a2c32065b",

"admin": true,

"blockedDates": [],

"__v": 0

}

Így lesz egy admin felhasználónk, aminek a jelszava admin

A program két fő része a Frontend és a Backend. Ez két konténer.
Létrehozunk egy adatbázis konténert, ami letölti a MongoDB-nek a 6.0.27 verzióját.
Létrehozunk egy mappát a data/DB néven, amit a számítógépen tárolunk, és akkor létre tudja hozni az adatbázist.

Van egy frontend és egy backend konténer. Bild részen, ahol van a docker compose fájl, mellette van a backend mappa. Benne van, hogy a frontendhez átmásolja a package.jsont. Utána ezt telepíti. a Node 16-os verzióját telepítjük. Ebben benne van, hogy minden csomagok, amire szükség van a futásához. Frontend néven. **work.dir**-ként kijelöljük. Minden konténer linux mappa struktúrát használ. **src mappát** kijelöljük. Ebbe a mappába bemásoljuk a saját gépünkön levő **frontend/package.json**.

Majd lefutattjuk **RUN npm install –force**

A különböző package json csomagok újabb verziói miatt, a compatibilitás problémák elkerülése érdekében force-oljuk.

npm run bild-el elindítjuk a frontendet.

Az nginx egy web szerver motor. A node.js csak egy javascript alapu renderelést végez, szóval az nginx konténert arra használjuk, hogy a megjelenést kivetítsük, elérhető legyen az oldal, azaz ezzel juttatjuk el magához a klienshez.

Mint ahogy átmásoljuk a frontendet, ugyan úgy átmásoljuk a backendet is.

8. Frontend felhasználói felület és kliensoldali működés dokumentációja (React, Tailwind CSS és Mantine alapokon)

8.1. Frontend architektúra és technológiai alapok

8.1.1. Áttekintés és szerep a rendszerben

A frontend egy React alapú webalkalmazás, amely a műszaktervező rendszer teljes kliensoldali felületét valósítja meg. A felhasználói felület célja, hogy egyszerű, áttekinthető és reszponzív módon tegye elérhetővé a rendszer funkcióit mind a normál felhasználók, mind az adminisztrátorok számára. A frontend nem tartalmaz üzleti logikát vagy jogosultságkezelést biztonsági szempontból kritikus módon, ezek a backend felelősségi

körébe tartoznak, azonban kliensoldalon támogatja a felhasználói élményt logikus navigációval és állapotkezeléssel.

8.1.2. Miért React?

- komponens-alapú felépítés,
- gyors fejlesztés,
- újrafelhasználható kód,
- széles körű ipari használat.

A frontend a backend által biztosított REST API végpontokat Axios HTTP kéréseken keresztül éri el. A hitelesítés session-alapú, ezért a kliensoldalon nincs szükség token vagy jelszó tárolására, a böngésző automatikusan kezeli a session cookie-t.

Frontend működése

- React komponensek jelenítik meg az adatokat
- HTTP kérések a backend API-hoz
- Session cookie automatikusan kezelve

A frontend kizárólag megjelenítési feladatokat lát el.

8.1.3. Technológiai stack és használt könyvtárak

A kliensoldal React könyvtárra épül, funkcionális komponensekkel és hook-alapú állapotkezeléssel. A főbb technológiák és csomagok a következők:

- React – komponensalapú architektúra
- react-router-dom – kliensoldali routing
- axios – HTTP kommunikáció
- date-fns – dátumformázás és dátumlogika
- Tailwind CSS – utility-first stílusozás
- Mantine – UI komponensek (Button, Collapse, AppShell)
- Headless UI – navigációs elemek és animációk
- react-spinners – betöltési visszajelzések
- react-day-picker – dátumválasztó komponens

Ez a technológiai stack lehetővé teszi a gyors fejlesztést, a jó felhasználói élményt és a könnyű karbantarthatóságot.

8.1.4. Alkalmazásszerkezet és belépési pontok

Az alkalmazás belépési pontja az index.js fájl, amely létrehozza a React rootot, majd rendereli az App komponenst. Az App komponens a teljes alkalmazást UserContextProvider-be csomagolja, így a felhasználói adatok és az autentikációs állapot globálisan elérhetővé válnak.
A Main komponens tartalmazza az alkalmazás vázát és a routing konfigurációt. Az AppShell komponens biztosítja az egységes layoutot, míg a Footer minden oldalon megjelenik. Az alkalmazás indulásakor a refresh() függvény automatikusan lefut, amely a backend felől lekéri az aktuális felhasználói állapotot.

8.1.5. Konténerizáció

A frontend Docker segítségével futtatható fejlesztői és éles módban is. A konténer a 3000-es porton szolgálja ki az alkalmazást.

8.1.6. Fejlesztői és éles környezet elkülönítése

A Dockerfile és Dockerfile.dev lehetővé teszi a fejlesztői és éles környezet elkülönítését. Fejlesztés során a hot reload támogatott, míg éles környezetben optimalizált build kerül futtatásra. Ez a megoldás professzionális fejlesztési workflow-t biztosít.

8.2. Nacigáció, routing és felhasználói jogosultságok

8.2.1. Routing és navigáció

A frontend kliensoldali routingot használ, így az oldalak közötti váltás nem jár teljes oldalú újratöltéssel. A főbb útvonalak:

- /login – bejelentkezés
- / – aktuális heti beosztás
- /block – tiltott dátum kérése
- /requests – saját kérelmek
- /admin – admin felület
- /admin/requests – admin kérelmek
- /admin/schedule – beosztás szerkesztése
- /admin/schedule-history – korábbi beosztások
- /admin/users – felhasználók kezelése

A nem létező útvonalak fallbackként a login oldalra irányítanak.

8.2.2. Jogosultságkezelés kliensoldalon

A kliensoldali jogosultságellenőrzés célja elsősorban a felhasználói élmény javítása. A Navbar komponens csak admin felhasználó esetén jeleníti meg az admin menüpontot. Az admin oldalak ellenőrzik a felhasználó állapotát: ha nincs bejelentkezve, átirányítás történik a login oldalra, ha nem admin, akkor a főoldalra.
Ez a megoldás nem helyettesíti a backend oldali jogosultságellenőrzést, de csökkenti a hibás navigáció lehetőségét.

8.2.3. Bejelentkezés folyamata

A LoginPage kontrollált input mezőket használ. A felhasználónév és jelszó state-ben tárolódik. A form elküldésekor POST /login kérés indul a backend felé. Sikeres belépés esetén frissül a globális user állapot, majd a felhasználó a főoldalra kerül. Hibás adatnál egy Msg komponens jelenít meg hibaüzenetet.

8.2.4. Navigáció és kijelentkezés

A Navbar reszponzív kialakítású, mobilon összecsukható menüvel. A menüpontok Link komponensekkel navigálnak. A kijelentkezés POST /logout kéréssel történik, amely után a felhasználó visszairányításra kerül a login oldalra.

8.3. Funkcionális működés és adatkezelés

8.3.1. Tiltott dátum kérése

A tiltott dátum kérése az AvailabilityPage oldalon történik. A DateInput komponens react-day-picker alapú, amely nem engedi a múltbeli dátumok kiválasztását. A kiválasztott dátum dd-MM-yyyy formátumban kerül elküldésre a backendnek. A rendszer visszajelzést ad a kérelem sikerességéről vagy hibájáról.

8.3.2. Saját kérelmek kezelése

A MyRequests oldal a felhasználó saját blockedDates listáját jeleníti meg. Alapértelmezetten csak a jövőbeli kérelmek láthatók, de a „View All” gombbal az összes korábbi kérelem is megtekinthető. A RequestInfoModal részletes információkat ad, és lehetőséget biztosít a kérelem törlésére.

8.3.3. Admin kérelmek kezelése

Az admin felületen az összes felhasználó kérelmei megjelennek. A státuszok színkódoltak (jóváhagyott, függőben lévő, elutasított). Az admin egy kattintással módosíthatja egy kérelem állapotát, amely azonnal frissül a felületen.

8.3.4. Beosztás megjelenítése

A WeekSchedule komponens betöltéskor lekéri a legfrissebb beosztást. A megjelenítés reszponzív, asztali és mobil nézettel. A nem admin felhasználók számára elérhető az „Only me” gomb, amely csak a saját beosztást mutatja.

8.3.5. Állapotkezelés és React hookok használata

A frontend alkalmazás az állapotkezelést React hookok segítségével valósítja meg. A legtöbb komponens lokális state-et használ a useState hookkal, például a form mezők, modális ablakok, lenyitható elemek (Collapse), illetve betöltési állapotok kezelésére. A useEffect hook kulcsszerepet játszik az adatok inicializálásában, például amikor az alkalmazás indulásakor vagy egy komponens betöltésekor HTTP kérés indul a backend felé.
A globális állapotkezeléshez egyedi Context API megoldás került kialakításra. A UserContextProvider felel a bejelentkezett felhasználó adatainak, jogosultságának és autentikációs állapotának tárolásáért. A UsersContextProvider az admin felületen az összes felhasználó listáját kezeli, lehetővé téve az admin számára a kérelmek és felhasználói adatok hatékony kezelését.
Ez a megoldás elkerüli a külső state management könyvtárak (pl. Redux) szükségességét, miközben a projekt méretéhez mérten átlátható és jól karbantartható marad.

8.3.6. HTTP kommunikáció és adatfrissítés

A frontend minden adatlekérést és adatküldést az Axios könyvtár segítségével végez. A HTTP kérések aszinkron módon futnak, és a válaszok alapján frissítik a komponensek állapotát. A session-alapú hitelesítés miatt nincs szükség külön authorization headerre, a böngésző automatikusan csatolja a session cookie-t minden kéréshez.
Az adatfrissítés több helyen tudatosan újrahívással történik, például egy admin által jóváhagyott kérelem után a refreshAllUsers() függvény lefut, így a felület mindig a legfrissebb adatokat mutatja. Ez a megközelítés csökkenti az inkonzisztens UI állapot kialakulásának esélyét.

8.4. Felhasználói élmény, megjelenés és minőségi szempontok

8.4.1. Megjelenés és stílus

A stílusozás Tailwind CSS segítségével történik. A utility osztályok biztosítják az egységes megjelenést és a gyors fejlesztést. A Footer kiegészítő információkat és külső hivatkozást tartalmaz.

8.4.2. Felhasználói visszajelzések és UX megoldások

A frontend kiemelt figyelmet fordít a felhasználói visszajelzésekre. Betöltés közben animált spinner (HashLoader) jelenik meg, így a felhasználó mindig tudja, hogy a rendszer dolgozik. Sikeres vagy sikertelen műveletek esetén a Msg komponens jelenít meg egyértelmű üzeneteket.

A státuszok színkódolása (zöld, sárga, piros) gyors vizuális visszajelzést ad, így a felhasználó azonnal felismeri egy kérelem állapotát. Ez különösen fontos az admin felületen, ahol egyszerre sok adat jelenik meg.

8.4.3. Reszponzív kialakítás

A frontend teljes mértékben reszponzív. A Tailwind CSS utility osztályai lehetővé teszik, hogy az alkalmazás mobilon, tableten és asztali nézetben is jól használható legyen. A beosztás megjelenítésére külön DesktopView és MobileView komponensek készültek, így az adatok mindig az adott képernyőmérethez optimalizált formában jelennek meg.

A navigációs sáv mobilon összecsukható, asztali nézetben pedig teljes menüsor formájában érhető el.

8.4.4. Komponens-alapú tervezés előnyei

A frontend felépítése erősen komponens-alapú. Az újrafelhasználható komponensek, mint például gombok, táblázatsorok, modális ablakok és űrlap elemek csökkentik a kódismétlést. Ez növeli a karbantarthatóságot és megkönnyíti a későbbi bővítéseket.

A RequestsListTableRow komponens például több különböző nézetben is használható, attól függően, hogy admin vagy felhasználói oldalról jelenik meg.

8.4.5. Hibakezelés kliensoldalon

A frontend nemcsak sikeres válaszokra van felkészítve, hanem a hibás esetekre is. A try-catch blokkok és feltételes renderelés biztosítja, hogy egy sikertelen kérés ne törje meg az

alkalmazás működését. Hibás válasz esetén a felhasználó érthető üzenetet kap, nem technikai hibaüzenetet.

Ez a megközelítés különösen fontos vizsgaremek esetén, mert jól mutatja a felhasználóközpontú gondolkodást.

8.4.6. Biztonsági megfontolások frontend oldalon

Bár a tényleges biztonsági ellenőrzések backend oldalon történnek, a frontend tudatosan nem jelenít meg olyan funkciókat, amelyekhez a felhasználónak nincs jogosultsága. Az admin funkciók elrejtése nem biztonsági megoldás, de csökkenti a véletlen vagy félrevezető interakciók számát.
A frontend nem tárol érzékeny adatokat, nem ment jelszót vagy tokent localStorage-be, ami csökkenti az XSS jellegű támadások kockázatát.

8.4.7. Továbbfejlesztési lehetőségek

A rendszer továbbfejleszthető egységes hibakezeléssel, TypeScript teljes bevezetésével, fejlettebb admin szűrésekkel és egységes API-kezeléssel.

8.4.8. Összegzés

A frontend egy modern, jól strukturált React alkalmazás, amely hatékonyan támogatja a műszaktervező rendszer működését. A komponens-alapú felépítés, a reszponzív dizájn, a tiszta routing és az átgondolt állapotkezelés mind hozzájárulnak ahhoz, hogy az alkalmazás könnyen használható és hosszú távon is karbantartható legyen**.**

9. Backend bemutatása

9.1. Alkalmazott technológiák

- Node.js – JavaScript futtatókörnyezet
- Express – HTTP szerver és routing
- Passport.js – hitelesítés
- Mongoose – MongoDB objektumkezelés

Node.js, mint Backend egy teljesen általános, de nem olyan robosztus, mint a Laravel például. Remekül átlátható, könnyen szerkeszthető, full-stack élmény ad.

Támogatja a szerver oldali renderelést, tehát nem kell az egész kódot letölteni, hanem szerversen fut és tehermentesíti a kliens gépet a számolási kapacitás alól, és a felhasználó csak egy kész HTML oldalt kap, így gyorsabb a kliens gép futása.

Egy beosztástervezőnél gyakori, hogy:
több felhasználó (vezető, dolgozók) egyszerre nézi / módosítja a beosztást
változásoknak azonnal látszódniuk kell (pl. csere, szabadság, túlóra), jól kezeli az üzleti szabályokat.

A React egy javascript keret rendszer, ami jól illaszkedik a Node.js-hez mivel mindkettő javascript alapú, szintúgy mint a MongoDB, így tökéletes kiegészítői egymásnak.
Ideális valósidejű frissítéshez.

9.2. Hitelesítés és biztonság

9.2.1. Jelszókezelés

A rendszer nem tárol sima szöveges jelszavakat.

A jelszókezelés folyamata: a jelszó soha nem sima szövegként, hanem hash-elve kerül tárolásra. Salt felhasználásával kerül titkosjtásra heshelve.

A jelszókezelés folyamata:

- A felhasználó megadja a jelszót
- Egyedi salt generálása
- PBKDF2 algoritmus alkalmazása
- A hash és a salt mentése az adatbázisba

Ez a módszer megfelel a modern biztonsági elvárásoknak.

9.2.2. Session-alapú hitelesítés

- Bejelentkezéskor session jön létre
- A session MongoDB-ben kerül tárolásra
- A böngésző cookie segítségével azonosítja a felhasználót

Ez stabil és biztonságos megoldás webalkalmazások esetén.

9.3. Általános backend működés összefoglalása

9.3.1. Session kezelés részletes működése

A backend session-alapú autentikációt használ az express-session middleware segítségével. A session azonosító egy HTTP cookie-ban (connect.sid) kerül eltárolásra a kliens böngészőjében.
A session adatok MongoDB-ben kerülnek tárolásra a connect-mongo csomag használatával, így az alkalmazás újraindítása esetén sem vesznek el a bejelentkezett felhasználók adatai.

A session élettartama 24 óra, amelyet a cookie.maxAge paraméter határoz meg. Ez megfelelő kompromisszum a felhasználói kényelem és a biztonság között.

9.3.2. Middleware-ek szerepe és működése

A backend több middleware-t használ:

- Express beépített JSON és URL-encoded middleware-k a request body feldolgozására
- CORS middleware a frontend és backend közötti kommunikációhoz
- Passport middleware az autentikáció és session kezelés biztosítására

Ezek biztosítják, hogy minden kérés során elérhető legyen a req.user objektum.

9.3.3. Admin jogosultság ellenőrzésének elvi felépítése

Az admin jogosultság ellenőrzése middleware segítségével történik.
A middleware ellenőrzi:

- a felhasználó be van-e jelentkezve,
- rendelkezik-e admin jogosultsággal.

Nem megfelelő jogosultság esetén a szerver HTTP 403 (Forbidden) választ ad.

9.3.4. API végpontok szervezési elvei

Az API végpontok logikusan csoportosítva találhatók a routes fájlokban.
Az admin végpontok külön middleware mögött helyezkednek el, ezzel csökkentve a jogosultsági hibák lehetőségét.

A REST elvek részben érvényesülnek:

- GET – adatlekérés
- POST – adatváltoztatás

9.3.5. Dátumkezelés és üzleti logika

A backend a date-fns könyvtárat használja a dátumok számításához.
A rendszer automatikusan kiszámolja a hét kezdő dátumát, valamint az ISO hét számot, ezzel segítve a beosztások visszakeresését.

9.3.6. Adatintegritás és konzisztencia

A MongoDB rugalmassága miatt a backend felelőssége az adatok konzisztenciájának biztosítása.
A rendszer ellenőrzi például, hogy ugyanarra a napra ne lehessen többször szabadságot kérni.

9.3.7. Hibakezelési stratégia

A jelenlegi hibakezelés try-catch blokkokra és konzol logolásra épül.
Éles környezetben javasolt egy központi error middleware alkalmazása, amely egységes formátumban adja vissza a hibákat.

9.3.8. Skálázhatóság és továbbfejlesztési irányok

A rendszer alkalmas kis- és közepes méretű felhasználói bázis kiszolgálására egyaránt. Praktikus és gyors. Továbbá egyszerűsége miatt nem igényel különösebb számítástechnikai képzettséget a felhasználóktól, könnyen elsajátítható a használata.

Bővíthetősége és alakíthatósága miatt széles körben elterjedhet, mivel nincs termékkör és tevékenység specifikációja. Akár a manapság divatos home officehoz kapcsolódó “asztalfoglalások”-hoz is praktikus megoldás lehet, a szigorún irodában töltendő idők kiosztásához, amikor össze kell hangolni egy kis befogadóképességű irodát, az állandóan cserélődő és változó mértékű dolgozói létszámhoz.

További fejlesztési lehetőségek:
- részletesebb logolás,
- role-alapú jogosultságkezelés,
- finomhangolt session beállítások.
- adatok közlése felhasználónkként
- adatok összevetése és kiértékelése
- munkaidő, szabadság, betegállományban töltött idő nyilvántartása

- további munkahelyi egységek hozzáadása
- Soft delete használata, esetleges visszaállítási igény esetére

9.3.9. Biztonsági megfontolások összefoglalása

A rendszer:
- nem tárol plain text jelszavakat,
- nem ad vissza érzékeny adatokat,
- session-alapú hitelesítést alkalmaz.

Ez megfelelő alapot biztosít egy biztonságos webalkalmazás számára.

9.3.10. Összegzés

A backend egy jól strukturált, modern Node.js alapú alkalmazás, amely hatékonyan támogatja a műszaktervező rendszer működését.
A REST API végpontok, a MongoDB alapú adattárolás és a Passport által biztosított hitelesítés együtt egy stabil, bővíthető és védhető rendszert alkotnak.

A backend felelős az üzleti logika megvalósításáért, a hitelesítésért, az adatbázis-kezelésért és az API végpontok biztosításáért. A frontend és a backend szorosan együttműködik HTTP alapú kommunikáción keresztül.

9.4. Backend szerveralkalmazás dokumentációja (Node.js alapú REST API, Express és MongoDB)

9.4.1. Áttekintés és szerep a rendszerben

A backend feladata a kliens (frontend) kiszolgálása REST jellegű HTTP végpontokon keresztül. A rendszer központi funkciói:

- felhasználók kezelése (bejelentkezés, admin általi regisztráció),
- jogosultságkezelés (admin vs. user),
- műszakbeosztások mentése és lekérése,
- felhasználói tiltott dátumok (blockedDates) kérése, jóváhagyási metaadatokkal.

A szerver Express alapú, az adatkezelés MongoDB + Mongoose segítségével történik. A hitelesítést a Passport (LocalStrategy) valósítja meg, session-alapú beléptetéssel.

9.4.2. Technológiai stack és fő csomagok

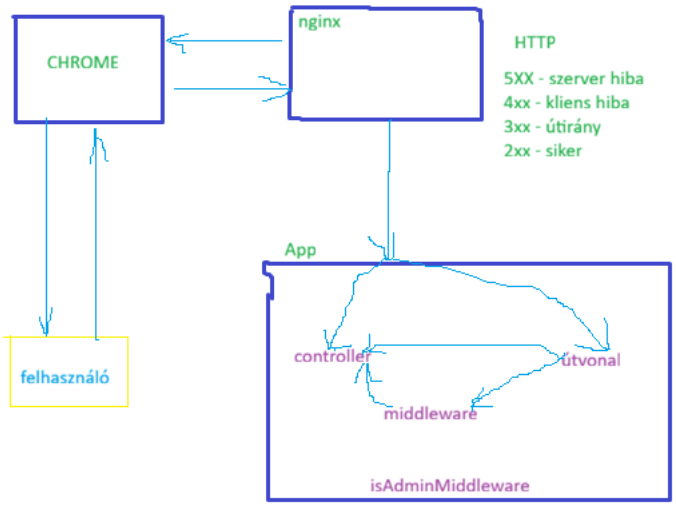
- Node.js + Express: HTTP szerver, routing és middleware lánc.
- MongoDB + Mongoose: perzisztencia és sémák.

- passport, passport-local: felhasználó hitelesítés.
- express-session + connect-mongo: session kezelés, session tárolása MongoDB-ben.
- crypto (PBKDF2): jelszóhash-elés salt-tal.
- cors: cross-origin kérések támogatása.
- date-fns: dátum számítások (pl. következő vasárnap, ISO hét).

9.4.3. Projektstruktúra (logikai egységek)

A backend kód szerkezete funkcionálisan elkülönülő részekre bontható:

- server.js: alkalmazás indítása, DB kapcsolat, session + passport inicializálás, route-ok bekötése.
- models/: MongoDB kollekciók sémái (User, Shift).
- passport/passwordFunctions.js: jelszó generálás és ellenőrzés (PBKDF2).
- routes/index.js: API végpontok (user + admin műveletek).
- routes/middleware/isAdmin.js: admin jogosultság ellenőrzése (a cél szerint; a beküldött fájlban jelenleg téves tartalom látszik – lásd lent).



5., ábra

9.4.4. Adatmodell (Mongoose sémák)

User modell (models/User.js)
A User kollekció a felhasználói azonosítást és jogosultsági adatokat tárolja:

- username: String – egyedi felhasználónév (a kódban így van keresve).
- hash: String – a jelszó PBKDF2 hash-e.
- salt: String – a hash-hez tartozó egyedi salt.
- memberSince: String – csatlakozás ideje (ISO stringként is használod).
- status: String – pl. „active”.
- admin: Boolean (default: false) – admin jogosultság.
- blockedDates: [{ date, comment, approved, approvedBy }] – tiltott napok és hozzájuk tartozó metaadatok:
    date: String – tiltott dátum
    comment: String – indoklás
    approved: Boolean – admin jóváhagyás állapota
    approvedBy: String – jóváhagyó admin neve/azonosítója

Megjegyzés: a blockedDates objektumok Mongoose-ban automatikusan kapnak _id mezőt, ezért törlésnél a $pull: { blockedDates: { _id: dateID } } működőképes.

Shift modell (models/Shift.js)
A Shift kollekció a beosztás mentésére szolgál, jellemzően heti mentések formájában:

- name: String – emberi olvasásra szánt címke (pl. hét sorszám + dátum).
- date: Date – mentés időpontja.
- data: Object – a teljes beosztás (rugalmas szerkezet, frontend által küldött objektum).
- savedBy: String – mentést végző felhasználó neve/azonosítója.

A data: Object mező előnye a gyors fejlesztés és rugalmasság; hátránya, hogy validáció/konzisztencia a kliens oldali struktúrától függ, ezért érdemes lehet később részletesebb sémára bontani.

9.4.5. Hitelesítés és biztonság

Jelszókezelés (PBKDF2 + salt)

A jelszavak nem kerülnek tárolásra sima szövegként. A passwordFunctions.js a következő elvet valósítja meg:

- regisztrációnál generálódik egy 32 byte-os salt (crypto.randomBytes(32)),
- a jelszó PBKDF2-vel hash-elve lesz (pbkdf2Sync(password, salt, 10000, 64, 'sha512')),

- mentésre kerül: salt + hash,
- belépéskor a megadott jelszó ugyanígy hash-elve összehasonlításra kerül a tárolttal (validPassword).

Ez ipari szinten is elterjedt megoldás, és megfelel annak, hogy a backend ne tároljon visszafejthető jelszót.

Passport LocalStrategy
A server.js-ben a Passport LocalStrategy a következő folyamatot használja:

- User.findOne({ username }) alapján kikeresi a felhasználót,
- validPassword(password, user.hash, user.salt) ellenőrzi a jelszót,
- siker esetén cb(null, user), hiba esetén cb(null, false).

Session alapú működés (express-session + connect-mongo)
A backend session-alapú autentikációt használ:

- a session cookie a kliensben tárolódik (connect.sid),
- a session tartalma MongoDB-ben kerül tárolásra a connect-mongo store segítségével,
- a cookie maxAge 1 nap.

Beléptetés után a frontend automatikusan „viszi" a session cookie-t, így a /api/user végponttal könnyen ellenőrizhető a bejelentkezés állapota.

Jogosultság:
A routing résznél van lekezelve a admin és a bejelentkezettségi státusz.

9.4.6. API végpontok (routes/index.js)

User státusz és belépés
GET /api/user
Cél: aktuális felhasználó és autentikációs állapot lekérése.

- Ha be van jelentkezve: { id, username, admin, blockedDates, isAuthenticated: true }
- Ha nincs: { isAuthenticated: false }

POST /login
Cél: belépés Passport segítségével.
Siker esetén: "loginSuccessful".

POST /logout
Cél: kijelentkezés, session megszüntetése.

- req.logout() + req.session.destroy()
- cookie törlése: clearCookie('connect.sid')
- válasz: { status: 'Success' }

Admin felhasználókezelés
GET /api/users (isAdmin)
Cél: összes felhasználó lekérése (admin).
Válasz: User[] vagy { isAuthenticated: false }.

POST /register (isAdmin)
Cél: új felhasználó létrehozása admin által.
Folyamat:

- ellenőrzi, létezik-e username,
- ha nem és nem üres: genPassword -> salt + hash,
- mentés new User({ username, salt, hash }).
  Válaszok:
  "UserAlreadyExists" ha foglalt,
  "Registered" siker esetén.

Tiltott dátumok kezelése (blockedDates)
POST /block-date
Cél: felhasználó tiltott dátum kérést küld.

- auth szükséges,
- duplikáció védelem: ha már van azonos dátum, BlockAlreadyRequested,
- különben push a blockedDates tömbbe approved:false.

Válasz:

- { msg: 'BlockAlreadyRequested' } vagy { msg: 'BlockRequestSuccess' }.

GET /api/request-info?employeeID=...&dateID=...
Cél: egy adott felhasználó adott request-jének adatai.

- User.findById(employeeID)
- _.filter(foundUser.blockedDates, { id: dateID })

POST /delete-request
Cél: felhasználó request törlése.

- User.findOneAndUpdate({ _id: employeeID }, { $pull: { blockedDates: { _id: dateID }}})

Válasz: { msg: 'RequestDeletionSuccess' } (hibánál Error).

Műszaktervezés és schedule kezelés

GET /getSchedule
Cél: legutóbb mentett beosztás lekérése.

- Shift.find().sort({ _id: -1 }).limit(1)
  Válasz: tömb (1 elemmel).

POST /postSchedule (isAdmin)
Cél: új heti beosztás mentése.

- bemenet: { savedSchedule, savedBy }
- currentDate alapján kiszámolja a következő vasárnapot (nextSunday),
- név képzés: (WN <ISO hét>) <dd-MM-yyyy>,
- mentés: new Shift({ name, data: savedSchedule, savedBy, date: currentDate })
  Válasz: "Success".

POST /removeSchedule (isAdmin)
Cél: beosztás törlése id alapján.

- Shift.findByIdAndDelete(id).

GET /getScheduleHistory (isAdmin)
Cél: összes mentett beosztás listázása.

- Shift.find({}).

9.4.7. Indítás, konfiguráció, konténerizáció

Szerverport és futtatás

A backend alapértelmezett portja: 4080 (PORT = 4080).

MongoDB kapcsolat:

- környezeti változóval: process.env.MONGODB_URI
- fallback: mongodb://localhost:27017/shift-scheduler

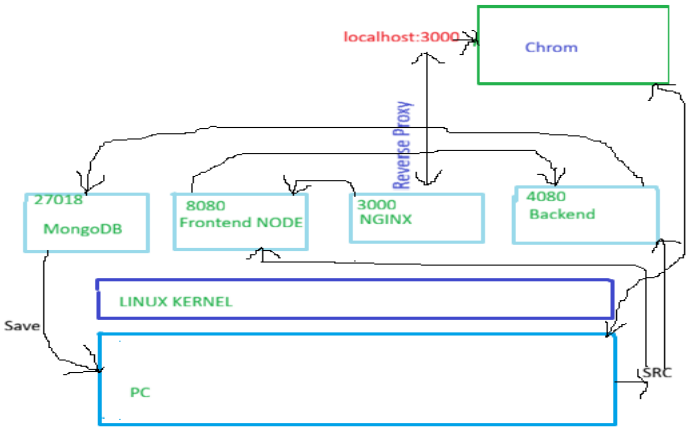
Dockerfile

A Dockerfile a backend konténeres futtatását támogatja:

- alap image: node:16
- WORKDIR /usr/src/app

- npm install
- port kinyitása: EXPOSE 4080
- indítás: npm start

Ez biztosítja, hogy a backend egységes környezetben fusson fejlesztői és éles rendszeren is.



6., ábra

9.4.8. Seed: alap admin létrehozása

A server.js tartalmaz egy egyszer futó seed logikát (ensureAdmin()), amely létrehoz egy alap admin felhasználót, ha még nem létezik:

- username: admin
- password: admin
- admin: true
- a default jelszó kötelezően megváltoztatandó,
- a seed kód opcionálisan eltávolítandó vagy környezeti flag mögé teendő.

9.4.9. Hibakezelés és továbbfejlesztési javaslatok (backend)

A kódban több helyen try/catch és konzol logolás van, de egységes HTTP hibakódok még nem mindenhol látszanak. Továbbfejlesztésre javasolt:

- egységes error response séma ({ errorCode, message }),
- helyes státuszkódok (401/403/404/500),
- isAdmin.js tényleges admin ellenőrzésének implementálása,

- /api/request-info szűrés javítása _id-ra,

- Shift.data validálása (séma vagy szerver oldali ellenőrzés).

9.4.10. Session kezelés részletes működése

A backend alkalmazás session-alapú autentikációt használ, amely az express-session middleware-re épül. A session azonosító egy HTTP cookie-ban (connect.sid) kerül eltárolásra a kliens böngészőjében. Maga a session tartalom nem a memóriában, hanem MongoDB-ben kerül tárolásra a connect-mongo csomag segítségével, így az alkalmazás újraindítása esetén sem vesznek el a bejelentkezett felhasználók adatai.

A session élettartama 24 óra, amelyet a cookie.maxAge paraméter határoz meg. Ez a megoldás kompromisszumot jelent a felhasználói kényelem és a biztonság között. A session-alapú megközelítés különösen jól illeszkedik a klasszikus webalkalmazásokhoz, ahol a frontend és backend szorosan együttműködik.

9.4.11. Middleware-ek szerepe és működése

A backend több middleware-t használ a kérésfeldolgozás során. Az Express beépített JSON és URL-encoded middleware-jei felelősek a request body feldolgozásáért. A CORS middleware biztosítja, hogy a frontend alkalmazás külön domain vagy port esetén is képes legyen kommunikálni a backenddel.

Az egyik legfontosabb middleware az autentikációhoz kapcsolódik: a Passport inicializálása és a session kezelés összekapcsolása. Ezek biztosítják, hogy minden kérés során elérhető legyen a req.user objektum, amely a bejelentkezett felhasználó adatait tartalmazza.

9.4.12. Admin jogosultság ellenőrzésének elvi felépítése

- a felhasználó be van-e jelentkezve,
- rendelkezik-e admin jogosultsággal.

Ha bármelyik feltétel nem teljesül, a szerver HTTP 403 (Forbidden) státuszkóddal válaszol. Ez a megoldás garantálja, hogy az admin funkciók kizárólag jogosult felhasználók számára legyenek elérhetők.

9.4.13. API végpontok szervezési elvei

Az API végpontok logikusan csoportosítva találhatók a routes/index.js fájlban. A felhasználói és admin funkciók elkülönítése növeli az átláthatóságot és megkönnyíti a karbantartást. Az admin végpontok következetesen middleware mögé vannak helyezve, ami csökkenti a jogosultsági hibák lehetőségét.

A REST elvek részben érvényesülnek: a GET kérések adatlekérésre, a POST kérések adatváltoztatásra szolgálnak. Ez a struktúra könnyen érthető és frontend oldalról egyszerűen használható.

9.4.14. Dátumkezelés és üzleti logika

A backend a date-fns könyvtárat használja a dátumok számításához. A beosztások mentésekor a rendszer automatikusan kiszámolja a következő vasárnap dátumát, majd az ISO hét számmal együtt generál egy emberileg olvasható elnevezést. Ez segíti az adminisztrátorokat a korábbi beosztások visszakeresésében.

A dátumkezelés során fontos szempont volt az időzónafüggetlenség, ezért a mentések egységes Date objektumként kerülnek eltárolásra.

9.4.15. Adatintegritás és konzisztencia

A MongoDB dokumentum-alapú felépítése rugalmas adatkezelést tesz lehetővé, ugyanakkor a backend felelőssége az adatok konzisztenciájának biztosítása. A tiltott dátumok esetén például a rendszer ellenőrzi, hogy ugyanarra a napra ne lehessen többször kérést benyújtani.

A műszakbeosztások esetében a teljes struktúra egy objektumban kerül mentésre, amely gyors fejlesztést tesz lehetővé, de jövőbeli bővítés esetén érdemes lehet részletesebb sémára bontani az adatokat.

9.4.16. Hibakezelési stratégia

A backend jelenlegi hibakezelése elsősorban try-catch blokkokra és konzol logolásra épül. Bár ez fejlesztés közben elegendő, éles környezetben ajánlott egységes hibakezelési stratégiát kialakítani. Ez magában foglalhat egy központi error middleware-t, amely egységes formátumban küldi vissza a hibákat a kliens felé.

Az egységes hibakezelés növeli a rendszer megbízhatóságát és megkönnyíti a frontend oldali feldolgozást.

9.4.17. Skálázhatóság és továbbfejlesztési irányok

A jelenlegi backend architektúra alkalmas kis és közepes méretű felhasználói bázis kiszolgálására. A MongoDB és az Express jól skálázható megoldást biztosít. Nagyobb terhelés esetén érdemes lehet bevezetni:

- környezeti változókkal konfigurálható session beállításokat,
- részletesebb logolást,
- role-alapú jogosultságkezelést (nem csak admin / user).

9.4.18. Biztonsági megfontolások összefoglalása

A backend nem tárol plain text jelszót, nem ad vissza érzékeny adatokat a frontendnek, és session-alapú hitelesítést alkalmaz. Ezek együtt megfelelő alapot biztosítanak egy biztonságos webalkalmazás számára. A rendszer felépítése megfelel a vizsgaremekkel szemben támasztott alapvető biztonsági elvárásoknak.

9.4.19. Összegzés

A backend egy jól strukturált, modern Node.js alapú alkalmazás, amely hatékonyan támogatja a műszaktervező rendszer működését. A REST API végpontok, a MongoDB alapú adattárolás és a Passport által biztosított hitelesítés együtt egy stabil és bővíthető rendszert alkotnak.

10. Ábrajegyzék

11. Felhasznált szakirodalom

MongoDB:

The Definitive Guide - Shannon Bradshaw, Eoin Brazil, Kristina Chodorow

Seven Databases in Seven Weeks - Eric Redmond, Jim R. Wilson

Node.js:

Node.js Design Patterns - Mario Casciaro, Luciano Mammino

Node.js in Action - Mike Cantelon, Marc Harter, T.J. Holowaychuk

Express.js:

Express.js Documentation - https://expressjs.com/

React:

React Official Documentation - https://react.dev/