Web Technológiák 2

Féléves beadandó

Könyvtári nyilvántartó

Készítette:

Név: Borbély Ádám

Neptun: R41N5H

2025.05

Tartalom

[Backend: 3](#_Toc199356993)

[Adatbázis 3](#_Toc199356994)

[Express 3](#_Toc199356995)

[Routes 4](#_Toc199356996)

[Entitások 6](#_Toc199356997)

[Item 6](#_Toc199356998)

[Member 7](#_Toc199356999)

[Kontrollerek 7](#_Toc199357000)

[Base-controller 7](#_Toc199357001)

[Item-controller 10](#_Toc199357002)

[Member-controller 12](#_Toc199357003)

[Frontend: 15](#_Toc199357004)

[Item-form 15](#_Toc199357005)

[Member-form 18](#_Toc199357006)

[Item-list 18](#_Toc199357007)

[Member-list 19](#_Toc199357008)

[Item-service 20](#_Toc199357009)

[Rent-form 21](#_Toc199357010)

A feladatom lényegében egy könyvtári nyilvántartó rendszer ahol a tagok is nyilván vannak tartva, valamint a kivehető könyvek, CD-k, kazetták. Lehetséges törölni és módosítani, valamint keresni adott szempontok alapján. A backendhez typesrciptet és mongoose-t használtam és angular frontendel rendelkezik. Adatbázisnak a MongoDB-t használom.

# Backend:

## Adatbázis

A backend mint korábban említettem a mongoDB-t használja az adatbázis kezelésére, amit a 27017-es localhoston hívunk meg az alábbi módon(db.ts)

import mongoose from 'mongoose';

const connectToDatabase = async () => {

try {

await mongoose.connect('mongodb://localhost:27017/konyvtar');

console.log('✅ Sikeres kapcsolódás a MongoDB-hez');

} catch (err) {

console.error('❌ Hiba a MongoDB kapcsolat során:', err);

process.exit(1);

}

};

export default connectToDatabase;

## Express

Ez a kód egy Node.js alapú webszerver alkalmazást valósít meg az Express keretrendszer segítségével. Az Express egy népszerű, könnyen használható webalkalmazás-fejlesztő könyvtár, amely lehetővé teszi HTTP szerverek gyors és hatékony létrehozását. A kód célja, hogy egy olyan szerveroldali alkalmazást hozzon létre, amely képes HTTP kéréseket fogadni és feldolgozni, valamint egy adatbázishoz kapcsolódva dinamikus API végpontokat szolgáltasson:

import express from 'express';

import connectToDatabase from './db';

import { getRouter } from './routes';

const app = express();

app.use(express.json());

app.use('/api', getRouter());

const PORT = 3000;

connectToDatabase().then(() => {

app.listen(PORT, () => {

console.log(`🚀 Szerver fut: http://localhost:${PORT}`);

});

}).catch(err => {

console.error('❌ Nem sikerült csatlakozni az adatbázishoz:', err);

});

## Routes

A bemutatott kód az Express keretrendszer segítségével definiál egy routert, amely az alkalmazás REST API végpontjait kezeli, két fő entitásra fókuszálva: az „items” (tárgyak) és „members” (tagok) erőforrásokra.

1. **Router létrehozása és kontrollerek példányosítása**   
   A kód az Express Router objektumát hozza létre, amely az API útvonalainak szervezésére szolgál. Ezt követően példányosítja az ItemController és MemberController osztályokat, amelyek az adott entitások kezeléséért felelnek. Ezek az osztályok tartalmazzák a szükséges üzleti logikát, például adatbázis műveleteket, adatok lekérését, létrehozását, módosítását és törlését.
2. **API végpontok definiálása**   
   A router az alábbi HTTP módszerekkel és útvonalakkal szolgáltat hozzáférést az „items” és „members” erőforrásokhoz:
   * **Items végpontok:**
     + GET /items: Az összes tárgy lekérése.
     + GET /items/free: Azoknak a tárgyaknak a lekérése, amelyek valószínűleg „szabadok” (például nem foglaltak vagy nem használatban vannak).
     + GET /items/:id: Egy adott azonosítóval rendelkező tárgy lekérése.
     + POST /items: Új tárgy létrehozása.
     + PUT /items/:id: Egy adott azonosítóval rendelkező tárgy adatainak frissítése.
     + DELETE /items/:id: Egy adott azonosítóval rendelkező tárgy törlése.
   * **Members végpontok:**
     + GET /members: Az összes tag lekérése.
     + GET /members/:id: Egy adott azonosítóval rendelkező tag lekérése.
     + POST /members: Új tag létrehozása.
     + PUT /members/:id: Egy adott azonosítóval rendelkező tag adatainak frissítése.
     + DELETE /members/:id: Egy adott azonosítóval rendelkező tag törlése.
3. **Router visszaadása**   
   A függvény a beállított útvonalakkal rendelkező router objektumot visszaadja, amely az alkalmazás fő szerver fájljában integrálható, így az /api útvonal alatti HTTP kérések ezen végpontokhoz irányulnak.

Ez a megközelítés strukturált, moduláris és jól áttekinthető API felépítést biztosít, amely elősegíti az alkalmazás későbbi bővíthetőségét és karbantarthatóságát.

import express from 'express';

import { ItemController } from './controllers/item-controller';

import { MemberController } from './controllers/member-controller';

export function getRouter(){

const router = express.Router();

const itemController = new ItemController();

const memberController = new MemberController();

router.get('/items', itemController.getAll);

router.get('/items/free', itemController.getFree);

router.get('/items/:id', itemController.getOne);

router.post('/items', itemController.create);

router.put('/items/:id', itemController.update)

router.delete('/items/:id', itemController.delete)

router.get('/members', memberController.getAll);

router.get('/members/:id', memberController.getOne);

router.post('/members', memberController.create);

router.put('/members/:id', memberController.update)

router.delete('/members/:id', memberController.delete)

return router;

}

## Entitások

### Item

Itt mongoose modellt definiálok, az elemeit adatbázisban tároljuk(pl. cím) és kapcsolódhat egy taghoz(másik entitás, member)

import { Schema, model, Document, Types } from 'mongoose';

export interface IItem extends Document {

item\_type: string;

author: string;

title: string;

in\_date: Date;

status: string;

rent\_from?: Date;

member?: Types.ObjectId;

}

const ItemSchema = new Schema<IItem>({

item\_type: { type: String, required: true },

author: { type: String, required: true },

title: { type: String, required: true },

in\_date: { type: Date, required: true },

status: { type: String, required: true },

rent\_from: { type: Date },

member: { type: Schema.Types.ObjectId, ref: 'Member' }

});

export const ItemModel = model<IItem>('Item', ItemSchema);

### Member

Az item entitáshoz hasonlóan mongoose modell kerül definiálásra, kapcsolódhat a másik entitással.

## Kontrollerek

### Base-controller

import { Request, Response } from 'express';

import { Model, Document, Types } from 'mongoose';

Importálja az Express HTTP kérés-válasz típusait és a Mongoose modell és dokumentum típusait.

export class Controller<T extends Document> {

private model: Model<T>;

Létrehoz egy általános (generikus) vezérlőosztályt, amely egy Mongoose modellt kezel.

constructor(model: Model<T>) {

this.model = model;

}

Konstruktor, amely paraméterként megkapja a kezelendő Mongoose modellt, és elmenti.

create = async (req: Request, res: Response): Promise<void> => {

try {

const newEntity = new this.model(req.body);

const savedEntity = await newEntity.save();

res.status(201).json(savedEntity);

} catch (err: any) {

res.status(500).json({ message: err.message });

}

};

Új adat létrehozása az adatbázisban a kérés törzséből, mentés, és az eredmény visszaküldése 201-es státusszal.

Hibakezelés esetén 500-as státuszt küld.

getAll = async (\_req: Request, res: Response): Promise<void> => {

try {

const entities = await this.model.find();

res.json(entities);

} catch (err: any) {

res.status(500).json({ message: err.message });

}

};

Az összes adat lekérdezése az adatbázisból és visszaküldése JSON formátumban.

getOne = async (req: Request, res: Response): Promise<void> => {

try {

const entity = await this.model.findById(req.params.id);

if (!entity) {

res.status(404).json({ message: 'Nem található' });

return;

}

res.json(entity);

} catch (err: any) {

res.status(500).json({ message: err.message });

}

};

Egy adott elem lekérdezése azonosító alapján.

Ha nincs találat, 404-es státusszal válaszol.

update = async (req: Request, res: Response): Promise<void> => {

try {

const updatedEntity = await this.model.findByIdAndUpdate(req.params.id, req.body, {

new: true,

});

if (!updatedEntity) {

res.status(404).json({ message: 'Nem található' });

return;

}

res.json(updatedEntity);

} catch (err: any) {

res.status(500).json({ message: err.message });

}

};

Egy meglévő elem frissítése azonosító alapján a kérés törzsének adataival.

Ha nincs találat, 404-es státusz, különben a frissített adat visszaküldése.

delete = async (req: Request, res: Response): Promise<void> => {

try {

const deletedEntity = await this.model.findByIdAndDelete(req.params.id);

if (!deletedEntity) {

res.status(404).json({ message: 'Nem található' });

return;

}

res.status(200).json({ message: 'Törölve' });

} catch (err: any) {

res.status(500).json({ message: err.message });

}

};

}

Egy adott elem törlése azonosító alapján.

Ha nincs találat, 404-es válasz, sikeres törlés esetén 200-as státusz és visszajelzés.

### Item-controller

import { Request, Response } from 'express';

import { Controller } from './base-controller';

import { IItem, ItemModel } from '../entity/Item';

Importálja az Express HTTP kérés-válasz típusait, az általános vezérlőt (Controller), valamint az Item entitás típusát és adatmodelljét.

export class ItemController extends Controller<IItem> {

constructor() {

super(ItemModel);

}

Definiálja az ItemController osztályt, amely az általános vezérlőt specializálja az ItemModel kezelésére.

getAll = async (\_req: Request, res: Response): Promise<void> => {

try {

const products = await ItemModel.find().populate('member');

res.json(products);

} catch (err: any) {

res.status(500).json({ message: err.message });

}

};

Lekéri az összes Item rekordot az adatbázisból, kiegészítve a kapcsolódó member adatokkal, majd JSON-ként visszaküldi.

Hibakezelés 500-as státusszal.

getFree = async (\_req: Request, res: Response): Promise<void> => {

try {

const freeItems = await ItemModel.find({ status: 'Szabad' }).populate('member');

res.json(freeItems);

} catch (err: any) {

res.status(500).json({ message: err.message });

}

};

Lekéri azokat az Item rekordokat, amelyek státusza 'Szabad', szintén a kapcsolódó member adatokkal, és visszaküldi JSON formátumban.

getOne = async (req: Request, res: Response): Promise<void> => {

try {

const entity = await ItemModel.findById(req.params.id).populate('member');

if (!entity) {

res.status(404).json({ message: 'Nem található' });

return;

}

res.json(entity);

} catch (err: any) {

res.status(500).json({ message: err.message });

}

};

}

Egy adott Item lekérdezése azonosító alapján, kiegészítve a kapcsolódó member adatokkal.

Ha nincs találat, 404-es státusszal válaszol.

### Member-controller

import { Request, Response } from 'express';

import { Controller } from './base-controller';

import { IMember, MemberModel } from '../entity/Member';

import { ItemModel } from '../entity/Item';

import mongoose from 'mongoose';

Importálja az Express kérés-válasz típusait, az általános vezérlőt, a Member és Item entitások típusait és modelljeit, valamint a Mongoose-t.

export class MemberController extends Controller<IMember> {

constructor() {

super(MemberModel);

}

Meghatározza a MemberController osztályt, amely a generikus vezérlőből származik, és a MemberModel-t használja.

delete = async (req: Request, res: Response): Promise<void> => {

try {

const member = await MemberModel.findById(req.params.id);

if (!member) {

res.status(404).json({ message: 'Nem található' });

return;

}

member.deleted = true;

await member.save();

res.status(200).json({ message: 'Törölve' });

} catch (err: any) {

res.status(500).json({ message: err.message });

}

};

Tag „soft” törlése: az adott tagot megkeresi az azonosító alapján, majd a deleted mezőt true-ra állítja, az adatot nem véglegesen törli, megmarad a DB-ben.

Válaszol 200-as státusszal törlés után vagy 404-essel, ha nincs találat.

getAll = async (\_req: Request, res: Response): Promise<void> => {

try {

const members = await MemberModel.find({ deleted: false }).populate('items');

res.json(members);

} catch (err: any) {

res.status(500).json({ message: err.message });

}

};

Lekéri az összes törlés nélkül (deleted: false) jelölt tagot, kiegészítve a hozzájuk tartozó items adatokkal.

getOne = async (req: Request, res: Response): Promise<void> => {

try {

const member = await MemberModel.findById(req.params.id).populate('items');

if (!member) {

res.status(404).json({ message: 'Nem található' });

return;

}

res.json(member);

} catch (err: any) {

res.status(500).json({ message: err.message });

}

};

Egy tag lekérdezése azonosító alapján, az ő „items” adataival kiegészítve.

Ha nincs találat, 404-es válasz.

update = async (req: Request, res: Response): Promise<void> => {

try {

const updatedData = req.body;

const existingMember = await MemberModel.findById(updatedData.\_id).populate('items');

if (!existingMember) {

res.status(404).json({ message: 'Nem található' });

return;

}

Tag adatainak frissítése a kérésből érkező adatok alapján.

Megkeresi a frissítendő tagot és kiegészíti a kapcsolódó tételekkel.

const updatedItemIds = updatedData.items?.map((item: any) => new mongoose.Types.ObjectId(item.\_id)) || [];

for (const item of existingMember.items) {

if (!updatedItemIds.includes(item.\_id)) {

updatedItemIds.push(item.\_id);

}

}

A frissített „items” lista összeállítása úgy, hogy megőrzi a régi elemeket is, ha nem törölték őket.

existingMember.name = updatedData.name;

existingMember.phone = updatedData.phone;

existingMember.pid = updatedData.pid;

existingMember.address = updatedData.address;

existingMember.items = updatedItemIds;

await existingMember.save();

const populated = await MemberModel.findById(existingMember.\_id).populate('items');

res.json(populated);

} catch (err: any) {

res.status(500).json({ message: err.message });

}

};

}

A tag mezőinek frissítése, mentése, majd a frissített tag visszaküldése a kapcsolódó tételekkel együtt.

Hibakezelés 500-as státusszal.

# Frontend:

## Item-form

Az ItemFormComponent egy Angular-alapú űrlapkomponens, amely egy könyvtári nyilvántartó rendszerben könyvek vagy más tételek hozzáadására, szerkesztésére és törlésére szolgál. Az űrlapot a FormBuilder segítségével hozza létre. Az ngOnInit metódus alapján megkülönbözteti, hogy egy új tételről vagy meglévő szerkesztéséről van-e szó (URL paraméter alapján). Az addItem() metódus elvégzi az új tétel mentését vagy egy meglévő frissítését, míg a deleteItem() törli az aktuálisan szerkesztett tételt.

FormGroup definiálása:

itemForm: FormGroup = this.formBuilder.group({

\_id: [],

item\_type: [],

author: [],

title: [],

in\_date: [],

status: []

})

Ez az űrlap mezőit deklarálja: az \_id azonosító, a title a cím, az author a szerző, az item\_type a típus (pl. könyv, DVD), az in\_date a beérkezési dátum, és a status az elérhetőségi állapot (Szabad vagy más).

Getterek:

get id() { return this.itemForm.controls['\_id'].value }

get title() { return this.itemForm.controls['title'].value }

Kényelmes hozzáférés biztosítása az \_id és title mezőkhöz.

Komponens inicializálása (ngOnInit):

async ngOnInit(): Promise<void> {

const id = this.activatedRoute.snapshot.queryParams['id'];

if(id !== undefined) {

const item = await this.itemService.loadOne(id);

this.itemForm.patchValue(item);

} else {

this.itemForm.patchValue({

type: 'Könyv',

in\_date: new Date().toISOString(),

status: 'Szabad'

})

}

}

Ha az URL-ben van id paraméter, akkor meglévő tételt tölt be szerkesztésre.

Ha nincs id, akkor új tétel felvitele történik — alapértékekkel: típus = „Könyv”, státusz = „Szabad”, dátum = aktuális időpont.

Tétel mentése (létrehozás vagy frissítés):

async addItem() {

const data = this.itemForm.value;

if(data.\_id !== null) {

await this.itemService.updateItem(data);

} else {

await this.itemService.addItem(data);

}

this.router.navigateByUrl(`item\_list`);

}

Ha van \_id, akkor már meglévő tétel, így frissíti azt.

Ha nincs \_id, akkor új tételt ad hozzá.

A művelet után visszairányít az item\_list oldalra.

Tétel törlése:

async deleteItem() {

const data = this.itemForm.value;

await this.itemService.deleteItem(data.\_id);

this.router.navigateByUrl(`item\_list`);

}

A jelenlegi tétel törlését végzi, majd visszanavigál a tétellistához.

## Member-form

Alapvetően megegyezik az item-form-al, csak pár dologban tér el:

* A ngOnInit() csak akkor tölt adatot, ha id van az URL-ben.
* Nincs alapértelmezett érték beállítás új tagnál (pl. mint in\_date vagy status az itemnél).

## Item-list

ngOnInit() – Inicializálás

this.all\_item = await this.itemService.loadAll('');

this.items = this.all\_item;

A komponens betöltésekor minden tételt betölt az adatbázisból az ItemService segítségével.

A szűrés nélküli lista (all\_item) másolata az aktuálisan megjelenített items.

Keresési logika

A keresés működése a pattern\_type értékétől függ:

1. DELAY – Késés alapján:

Számítás: a tétel rent\_from dátuma alapján meghatározza, hány napot késett.

Csak azokat a tételeket listázza, amelyek a megadott napnál régebben vannak kikölcsönözve.

2. TITLE / AUTHOR – Szöveges keresés:

Az item.title vagy item.author tartalmazza-e a pattern értékét.

Csak azok jelennek meg, amelyek megfelelnek a keresésnek.

3. Üres mező esetén:

Minden tétel megjelenik (this.items = this.all\_item).

Késés kiszámítása

getDelay(rent\_from: Date): number

Meghatározza, hány napos késésben van egy tétel.

A késés akkor pozitív, ha a rent\_from óta eltelt napok száma meghaladja a keresési mezőben megadott napokat.

Ha nincs késés, vagy nincs rent\_from, akkor 0 a visszatérési érték.

Tétel visszavétele

item.status = "Szabad";

item.member = null;

item.rent\_from = null;

this.itemService.updateItem(item);

A tétel státuszát "Szabad"-ra állítja.

Törli a kölcsönző tagot és a kölcsönzés dátumát.

Frissíti az adatokat az adatbázisban az ItemService segítségével.

## Member-list

Keresés logikája

Egyszerűbb keresés:

ID alapján pontos egyezést vizsgál (member.\_id === pattern)

NAME és PID alapján részleges egyezést keres (indexOf)

Nincs késésszámítás vagy összetettebb logika.

Tételek visszavétele (back() metódus)

ItemListComponent tartalmaz back(item: Item) metódust, amely egy tétel státuszát visszaállítja "Szabad"-ra.

MemberListComponent nem tartalmaz hasonló funkciót.

Késésszámítás metódus

MemberListComponent nem végez semmilyen dátum alapú számítást.

## Item-service

Az ItemService osztály egy Angular-szolgáltatás, amely az Item típusú adatok kezelését végzi REST API hívásokon keresztül. Ez az osztály biztosítja az aszinkron kommunikációt a frontend és a backend között.

Dekorátor és injektálhatóság

@Injectable({

providedIn: 'root'

})

Ez az annotáció azt jelenti, hogy a szolgáltatás globálisan elérhető, nem kell külön modulként regisztrálni.

root szintű injektálás biztosítja, hogy singleton példány jöjjön létre.

HTTP kliens használata

constructor(private http: HttpClient) {}

A HttpClient segítségével történnek az HTTP-alapú kérés-küldések (GET, POST, PUT, DELETE).

Az osztály konstruktoron keresztül injektálja az Angular HttpClient-jét.

Aszinkron API-hívások kezelése

Az async függvények a firstValueFrom() függvénnyel kiegészítve várják meg az Observable értékének első kibocsátását.

Ez lehetővé teszi az await szintaxis használatát, így egyszerűbb és olvashatóbb az aszinkron kód

## Rent-form

A RentFormComponent célja, hogy lehetővé tegye a könyvtári rendszerben egy tag számára eszközök kölcsönzését. A komponens Reactive Forms alapú űrlapot használ, ahol a maximális kölcsönözhető mennyiség dinamikusan szabályozva van. A kölcsönzés során csak a valós, érvényes és egyedi eszközök kerülnek feldolgozásra, majd ezek frissítése történik az adatbázisban a MemberService segítségével. A szabad eszközök listája a művelet után automatikusan frissül.

Űrlap létrehozása – Reactive Form

rentForm: FormGroup = this.formBuilder.group({

id: [], // tag azonosítója

items: this.formBuilder.array([]) // kölcsönözni kívánt tételek tömbje

})

Ez a FormGroup tartalmazza a tag azonosítóját és a kölcsönözni kívánt eszközöket. A FormArray dinamikusan bővíthető űrlapmezőket jelent.

Inicializálás – adatok betöltése

async ngOnInit(): Promise<void> {

const id = this.activatedRoute.snapshot.queryParams['id'];

if(id == undefined) {

this.router.navigateByUrl(``);

}

this.member = await this.memberService.loadOne(id); // tag adatainak betöltése

this.rentForm.patchValue({ id: this.member.\_id }); // tag ID beállítása az űrlapba

// max. 6 eszközt lehet kölcsönözni – annyit enged, amennyit még lehet

for(let i = 0; i < 6 - this.member.items.length; i++) {

this.itemsFormArray.push(this.formBuilder.group({

id: [], // kiválasztott eszköz ID-je

status: ['Kikölcsönzött'], // alapértelmezett státusz

rent\_from: [new Date().toISOString()] // mai nap dátuma

}))

}

this.items = await this.itemService.loadFree(); // szabad eszközök betöltése

}

Ez a rész betölti a tagot, aki kölcsönöz, majd legenerálja a form mezőket – annyit, ahány eszközt még kölcsönözhet.

Kölcsönzés logikája – rent() függvény

async rent() {

const data = this.rentForm.value;

const tmp\_data: Item[] = [];

// csak a ténylegesen kiválasztott, egyedi eszközöket vesszük figyelembe

for(let item of data.items) {

if(item.id == '-1' || item.id == null) continue;

if(tmp\_data.filter(i => i.\_id == item.id).length > 0) continue;

tmp\_data.push(item);

}

data.items = tmp\_data;

// tag frissítése (új kölcsönzések hozzáadása)

this.member = await this.memberService.updateMember(data);

// szabad eszközök újratöltése

this.items = await this.itemService.loadFree();

// form tömb mezőinek ürítése

for(let tmp in tmp\_data) {

this.itemsFormArray.removeAt(0);

}

}

A rent() végzi el a tényleges kölcsönzést:

Először kiszűri az érvénytelen vagy ismétlődő eszközöket.

Meghívja a backend szolgáltatást (updateMember), amely menti a kölcsönzést.

Ezután frissíti a szabad eszközök listáját, és törli a már használt form mezőket.

Segédfüggvény – toGroup()

toGroup(item: AbstractControl) {

return item as FormGroup;

}

Ez egy technikai segédfüggvény arra, hogy a HTML sablonban egyszerűen tudjuk elérni a FormGroup mezőit a FormArray-on belül.