**BiblioSys – Fejlesztői kézikönyv**

**Projekt áttekintése – cél, célközönség, fő funkciók**

A **BiblioSys** egy könyvkölcsönző rendszer, amelynek célja a könyvtári könyvek nyilvántartása és a kölcsönzések egyszerű kezelése egy webes alkalmazás formájában. A célközönség elsősorban kisebb könyvtárak vagy iskolai könyvtárak könyvtárosai és adminisztrátorai, akik egy könnyen használható eszközzel szeretnék követni, hogy mely könyvek vannak kikölcsönözve, ki kölcsönözte őket és mikor esedékes a visszahozataluk. A rendszer emellett több felhasználós bejelentkezést is támogat, így szükség esetén több könyvtári alkalmazott is használhatja.

**Főbb funkciók:**

* **Felhasználó hitelesítés:** Bejelentkezés biztonságos módon, jelszóval és JWT token alapú munkamenet-kezeléssel a jogosulatlan hozzáférés elkerülésére.
* **Könyvnyilvántartás:** A könyvtár összes könyvének nyilvántartása, listázása. Lehetőség új könyvek rögzítésére, meglévők adatainak megtekintésére, esetleges módosítására vagy törlésére.
* **Kölcsönzés rögzítése:** Egy adott könyv kölcsönadásának folyamata támogatott – a könyvtáros kiválasztja a könyvet, megadja a kölcsönvevő (olvasó) adatait és a kölcsönzés időtartamát. Ezzel rögzíti a rendszerben, hogy a könyv ki van kölcsönözve.
* **Visszavétel kezelése:** Amikor egy könyvet visszahoznak, a könyvtáros jelölheti a rendszerben a visszavételt. Ezzel a könyv újra elérhetővé válik kölcsönzésre, és a kölcsönzési rekord lezárul (vagy törlődik).
* **Aktuális kölcsönzések áttekintése:** A rendszer listázza az éppen kikölcsönzött könyveket, feltüntetve, hogy ki kölcsönözte ki és meddig (a kölcsönzés határideje). Ez segít a könyvtárosnak nyomon követni a kint lévő könyveket és az esetleges késedelmeket.
* **Célzott értesítés (jövőbeni bővítés lehetősége):** A rendszer alapjaiban támogatja, hogy a kölcsönzési határidők alapján a kölcsönzők értesítést kaphassanak (pl. e-mailben), ha egy könyvet vissza kell hozni – ez a funkció igény szerint továbbfejleszthető.

Összességében a BiblioSys célja, hogy digitalizálja és leegyszerűsítse a hagyományos kölcsönzési napló vezetését, csökkentve a papírmunkát és növelve a naprakész információkat a könyvtár könyveiről.

**Felhasznált technológiák – frontend és backend áttekintése**

**Frontend – Angular:** A kliens oldali alkalmazás az Angular keretrendszerre épül. Az Angular egy a Google által fejlesztett, **TypeScript alapú, nyílt forráskódú** front-end keretrendszer egyoldalas webalkalmazások fejlesztéséhez[en.wikipedia.org](https://en.wikipedia.org/wiki/Angular_(web_framework)#:~:text=Angular%20,6). Az Angular moduláris felépítésű, **komponens-alapú** architektúrát alkalmaz, amely jól skálázható webalkalmazások készítését teszi lehetővé. A BiblioSys frontenje Angular 15-ös verzióra épül (TypeScript nyelven íródott), és használja az Angular által biztosított szolgáltatásokat, pl. komponenseket, szervizeket, útvonalkezelést (routing) stb. A dizájn és interakciók kialakításához a projekt az **Angular Material** komponenstárat alkalmazza – ennek köszönhetően szabványosított, reszponzív UI elemeket (pl. gombok, űrlapok, dialógusablakok) használunk. Az Angular egy **egypámos alkalmazás** (SPA) formájában fut, azaz a szükséges HTML/JS/CSS erőforrásokat egyszer töltjük be, és a felhasználói navigáció az oldalon belül, kliens oldalon történik (nincs minden navigációnál teljes lapbetöltés).

**Backend – Spring Boot + Spring Security + JPA:** A szerver oldali komponens egy Java nyelven íródott REST API, amely a Spring Boot keretrendszerre épül. A **Spring Boot** egy nyílt forráskódú Java keretrendszer, mely leegyszerűsíti a **stand-alone**, éles üzemi Spring-alapú alkalmazások fejlesztését azáltal, hogy előre definiált beállításokat és beépülő könyvtárakat biztosít (a “konvenció az konfiguráció felett” elvet követve)[en.wikipedia.org](https://en.wikipedia.org/wiki/Spring_Boot#:~:text=Spring%20Boot%20is%20an%20open,7). A backend fő feladata a kérések kiszolgálása: a könyvek és kölcsönzések adatainak tárolása és visszakeresése a frontendtől érkező HTTP kérések alapján, valamint az üzleti logika érvényesítése (pl. ne engedje egy már kikölcsönzött könyv újbóli kiadását visszavétel előtt). A Spring Boot keretében a következő főbb modulokat használja a BiblioSys:

* **Spring Web (MVC):** REST kontrollerek létrehozásához, melyek az egyes API végpontokat megvalósítják (pl. könyvek listázása, kölcsönzés létrehozása stb.).
* **Spring Data JPA:** az adatbázis műveletek egyszerű kezeléséhez. A JPA (Java Persistence API) segítségével a Java osztályok (entitások) és az adatbázis táblák között ORM kapcsolat van, így a kódban objektumokkal dolgozhatunk az SQL lekérdezések helyett.
* **Spring Security (JWT):** a biztonságos bejelentkezéshez és a végpontok védelméhez. A rendszer JSON Web Token (JWT) alapú autentikációt alkalmaz: a felhasználó bejelentkezés után kap egy JWT tokent, amit a további kérések headerébe csatolva igazolja magát. A backend oldalon egy JWT szűrő komponens ellenőrzi a bejövő kérések Authorization headerében kapott tokent, és csak akkor engedi a védett erőforrásokat elérni, ha a token érvényes. A JWT használata lehetővé teszi, hogy a szerver állapotmentes módon (stateless) kezelje a munkameneteket – nincs szükség szerver oldali session tárolásra, minden kérés magában hordozza a szükséges hitelesítési információt.
* **Adatbázis – PostgreSQL:** Az alkalmazás adatainak tartós tárolására **PostgreSQL** adatbázist használunk. A PostgreSQL egy nagy teljesítményű, nyílt forráskódú, relációs adatbázis-kezelő rendszer, amely vállalati szintű szolgáltatásokat nyújt és több mint 20 éve folyamatosan fejlesztik[aws.amazon.com](https://aws.amazon.com/rds/postgresql/what-is-postgresql/#:~:text=PostgreSQL%20is%20an%20advanced%2C%20enterprise,mobile%2C%20geospatial%2C%20and%20analytics%20applications). A BiblioSys esetében a PostgreSQL tárolja a könyvek, a kölcsönzési nyilvántartások, valamint a felhasználói adatok (bejelentkező felhasználók) adatait. A Spring Data JPA segítségével a Java entitásokból a keretrendszer automatikusan leképezi az adatbázis táblákat, és támogatja az SQL műveletek végrehajtását Java metódus hívásokkal (repository interface-eken keresztül).

A frontend és backend HTTP-n keresztül JSON formátumban kommunikál egymással egy jól meghatározott REST API szerint. A következő szekciókban részletesen ismertetjük a projekt felépítését, a fő komponenseket, a működési logikát és a használat módját.

**Könyvtárstruktúra és felépítés**

A projekt két fő részből áll, melyek különálló mappákban helyezkednek el:

* **library-frontend:** az Angular alapú kliens alkalmazás forráskódja.
* **library-management:** a Spring Boot alapú szerver alkalmazás (backend) forráskódja.

Mindkét rész önállóan buildelhető és futtatható. A következőkben bemutatjuk mind a frontend, mind a backend projekt struktúráját, fontosabb fájljait és mappáit.

**Frontend könyvtárstruktúrája (Angular)**

*A front-end (Angular) projekt főbb könyvtárai és fájljai.* Az Angular projektben a kód legnagyobb része a src/app mappában található, komponensekre és szolgáltatásokra bontva. A fenti képernyőkép bemutatja a fontosabb részeket:

* **Komponensek (components):** A felhasználói felület elemei külön komponensekbe szervezve. A projektben megtalálható többek között a login komponens (bejelentkező oldal), a dashboard komponens (főoldali vezérlőpult), a borrow-dialog komponens (kölcsönzési adatbeviteli dialógus). Minden komponens saját TypeScript (.ts), HTML sablon (.html) és stílus (.scss) fájlokkal rendelkezik.
  + *Login komponens:* felelős a bejelentkezési űrlap megjelenítéséért és a hitelesítési folyamat elindításáért.
  + *Dashboard komponens:* a bejelentkezés utáni főoldal, amely összefoglalja a könyvtár adatait (pl. könyvlista, kölcsönzött könyvek listája).
  + *BorrowDialog komponens:* egy modális dialógusablak a kölcsönzési adatok felvételéhez (kölcsönző neve, elérhetőségei, kölcsönzés időtartama).
  + *BorrowedBooks komponens:* (ha külön komponensként van megvalósítva) a jelenleg kölcsönzött könyvek listáját jeleníti meg, visszavétel (return) gombokkal.
* **Szolgáltatások (services):** Ide tartoznak azok a singleton osztályok, amelyek az alkalmazás logikáját támogatják és adatokat töltenek be. Pl. auth.service.ts az autentikációval kapcsolatos HTTP kéréseket intézi (login, esetleg regisztráció), book.service.ts a könyvek lekérdezését végzi a backend API-tól, esetleg borrower.service.ts (ha van külön) a kölcsönzési műveletekhez.
* **Modellek (models):** Azok a TypeScript interfészek vagy osztályok, melyek a backend által küldött adatokat reprezentálják a kliens oldalon. Pl. Book modell (id, title, author stb. mezőkkel), Borrower vagy kölcsönzés modell (adott esetben a kölcsönzési rekord mezőivel).
* **Interceptorok (interceptors):** Az Angular interceptort használ a HTTP kérésekhez fűződő közös logika megvalósítására. A auth.interceptor.ts feladata például, hogy minden kimenő kérés fejlécéhez hozzáadja a JWT tokent (Authorization: Bearer ...), amennyiben a felhasználó be van jelentkezve. Így a védett végpontok hívásakor a szerver ellenőrizni tudja a jogosultságot.
* **Egyéb fájlok:** Az app.routes.ts vagy app-routing.module.ts (Angular verziótól függően) definiálja az ügyféloldali útvonalakat: pl. a gyökér URL a login komponenst jeleníti meg, sikeres bejelentkezés után a /dashboard útvonal a Dashboard komponenst stb. Az app.component.ts a fő gyökérkomponens, ami a komponensek nézetét összefogja. Továbbá, a gyökér index.html tartalmazza az alkalmazás indításához szükséges alapszerkezetet, a main.ts pedig bootstrapeli az Angular alkalmazást. Az environment.ts fájlban találhatók a környezeti beállítások (pl. a backend szerver URL-je); jelen projekt esetén a fejlesztői környezetben tipikusan http://localhost:8080 van beállítva API szerverként.

Összességében a frontend projekt jól strukturáltan, funkcionális egységenként tagolt. Ez megkönnyíti a kód karbantartását és bővítését, hiszen egy új funkció hozzáadásakor elegendő új komponenst vagy szolgáltatást létrehozni, illetve a meglévőket bővíteni a tiszta felelősségi körök megtartásával.

**Backend könyvtárstruktúrája (Spring Boot)**

*A backend (Spring Boot) projekt fő csomagjai és osztályai.* A backend egy tipikus Spring Boot alkalmazás felépítését követi, rétegekre bontva a funkciókat:

* **Config csomag (config):** Itt találhatók a konfigurációs osztályok, pl. CorsConfig (amely beállítja a CORS szabályokat, hogy a Angular frontend elérhesse a szervert) és SecurityConfig (a Spring Security beállításait tartalmazza, pl. mely URL-ek védettek, milyen jelszó kódolás és JWT filter van használatban). A CorsConfig jellemzően megengedő módon minden originről enged kéréseket a /api/\*\* végpontokra, mivel a fejlesztés alatt a http://localhost:4200 originről jönnek a kérések. A SecurityConfig kikapcsolja a Spring alapértelmezett form alapú beléptetését, helyette JWT alapú ellenőrzést használ, és meghatározza, hogy pl. a /api/auth/\*\* végpontok nyilvánosak (nem igényelnek tokent), míg a többi /api/\*\* végpont csak érvényes tokennel érhető el.
* **Controller csomag (controller):** A REST API végpontokat megvalósító vezérlő osztályok. A képen látható például:
  + AuthController: a bejelentkezéshez (és esetleg regisztrációhoz) tartozó végpontokat kezeli. Pl. POST /api/auth/login a belépéshez, POST /api/auth/register új felhasználó regisztrálásához. A bejelentkezés során a kapott felhasználónév/jelszó ellenőrzése után JWT tokent generál és ad vissza.
  + BookController: a könyvek kezeléséért felelős végpontokat tartalmazza. Tipikusan GET /api/books (összes könyv lekérdezése), POST /api/books (új könyv hozzáadása), GET /api/books/{id} (könyv részleteinek lekérdezése), DELETE vagy PUT a könyv törléséhez/módosításához.
  + BorrowerController: a kölcsönzések (kölcsönző adatok) kezelését végzi. Pl. GET /api/borrowers (jelenlegi kölcsönzések listája), POST /api/borrowers (új kölcsönzési rekord létrehozása egy könyvhöz), DELETE /api/borrowers/{id} (kölcsönzés törlése – ezzel a könyv visszavételét regisztráljuk).
* **DTO csomag (dto):** Data Transfer Object-ek, azaz olyan egyszerű objektumok, amelyek a külvilág felé történő adatkommunikációt szolgálják. A projektben például LoginRequest (a bejelentkezési adatok – felhasználónév, jelszó – befogadására), RegisterRequest (új felhasználó regisztrációs adatai), BorrowerDTO (egy kölcsönzési rekord adatait összefoglaló objektum, amit a frontendnek küldünk). A DTO-k használatával elérjük, hogy ne közvetlenül az entitásokat exponáljuk a REST API-n, így rugalmasabban alakítható, milyen adatot adunk át (és pl. elkerüljük a végtelen ciklusokat, érzékeny adatok kiszivárgását).
* **Entity (modell) csomag (entity):** Az üzleti modell entitásai, amik megfelelnek az adatbázis tábláinak. Itt található pl. Book, Borrower és User osztály:
  + Book: a könyveket reprezentáló entitás (id, cím, szerző, és esetleg további attribútumok, pl. év, kategória stb. – a projekt jelenlegi verziójában a legfontosabb a cím és szerző).
  + Borrower: a kölcsönzési eseményeket reprezentáló entitás. Tartalmazza a kölcsönvevő nevét és elérhetőségeit, a kölcsönzés kezdő dátumát, határidejét, és hivatkozást a kölcsönzött könyvre. (Az elnevezése megtévesztő lehet, de ebben az esetben nem csak egy személyt jelöl, hanem egy konkrét kölcsönzési aktust – lásd alább az adatbázismodell magyarázatát.)
  + User: a bejelentkező felhasználót reprezentáló entitás (id, felhasználónév, jelszó, szerepkör). A jelszó jellemzően titkosítva van tárolva (pl. BCrypt hash).
* **Exception csomag (exception):** Ide kerülhetnek a kivételkezeléssel kapcsolatos osztályok, pl. ValidationExceptionHandler amely globálisan elkapja az esetleges validációs hibákat (pl. ha egy @Valid annotációval ellátott bemenő DTO nem felel meg a feltételeknek) és egységes választ ad a kliensnek hibakódokkal.
* **JWT csomag (jwt):** A JWT token kezeléséhez tartozó segédosztályok. Pl. JwtUtil a token generálásához és érvényesítéséhez (ellenőrzi az aláírást, lejárati időt stb.), valamint JWTAuthFilter (vagy hasonló nevű filter), amely a bejövő kérések előtt lefutva ellenőrzi az Authorization fejlécben kapott JWT-t, és ha érvényes, beállítja a SecurityContext-ben a felhasználói adatokat, hogy a kérések során elérhető legyen a felhasználó.
* **Repository csomag (repository):** Az adatbázis műveletekért felelős interface-ek, amelyek a Spring Data JPA segítségével futtatnak le műveleteket. Például BookRepository (automatikusan leszármazik a JpaRepository<Book, Long>-ból, ezzel alap műveletek – mentés, törlés, keresés id alapján – már adottak), BorrowerRepository, UserRepository. Gyakran definiálnak itt további lekérdezéseket is. A projektben például lehet egy olyan metódus, hogy Optional<Book> findByTitleAndAuthor(String title, String author), amelyet a könyv egyedi beazonosítására használhat a kölcsönzés logikája (a könyv cím+szerző alapján keresése) – ezt a Spring automatikusan implementálja a metódusnév alapján.
* **Service csomag (service):** Az üzleti logika szolgáltatás rétege. Itt általában a kontrollerek hívják meg a service osztályokat, amelyek elvégzik a szükséges műveleteket a repository-kon keresztül. Például BookService (könyvek lekérdezése, hozzáadása; ide tartozhat logika pl. könyv törlésekor ellenőrizni, hogy nincs-e kint kölcsönben), UserService (felhasználó regisztráció, bejelentkezéshez szükséges ellenőrzések). A BorrowerService szerepét lehet, hogy a projektben maga a BookService vagy BorrowerController látta el üzleti logikával (ha nem hoztak létre külön service osztályt a kölcsönzésekhez). A service réteg segít abban, hogy a kontrollerekben ne legyen túl sok logika: itt történhetnek az olyan műveletek, mint pl. új kölcsönzés létrehozásakor a könyv megkeresése az adatbázisban, ellenőrzése, hogy éppen nem kölcsönzött-e, a Borrower entitás feltöltése adatokkal és mentése, stb.
* **Futtatható alkalmazás osztály:** A LibraryManagementApplication (vagy hasonló nevű main osztály) tipikusan a gyökér csomagban található, tartalmazza a public static void main metódust és a @SpringBootApplication annotációt. Ez indítja el a beágyazott szervert (Tomcat) és a Spring kontextust.

A backend projekt struktúrája tehát követi a bevált rétegzett architektúrát, ami átláthatóvá teszi a kódot és elősegíti a kód újrafelhasználhatóságát és bővíthetőségét. Az elkülönített rétegek (controller, service, repository) lehetővé teszik, hogy például az üzleti logikát könnyen módosítsuk anélkül, hogy a vezérlők vagy a perzisztencia réteg részleteit bolygatnunk kellene.

**Fő komponensek és szolgáltatások részletesen**

Ebben a részben bemutatjuk a rendszer néhány kiemelt komponensét és azok működését: a bejelentkezés folyamatát, a vezérlőpult (dashboard) felépítését, a kölcsönzési dialógus működését, a kölcsönzött könyvek listáját, valamint a könyvek visszavételének (visszahozatalának) logikáját.

**Bejelentkezés (Login komponens és folyamat)**

A bejelentkezéshez tartozó frontend komponens a **LoginComponent** (login komponens). Ez egy egyszerű űrlapot jelenít meg, ahol a felhasználó megadja a felhasználónevét és jelszavát. A komponens a AuthService segítségével végzi el a bejelentkeztetést:

* Az űrlap elküldésekor meghív egy authService.login() metódust, amely egy HTTP POST kérést indít a backend felé a következő végpontra: POST /api/auth/login. A kérés törzsében a felhasználónév és jelszó (tipikusan JSON formátumban, pl. { "username": "admin", "password": "titkos" }) szerepel.
* A backend AuthController-e fogadja a kérést. A LoginRequest DTO-ban kapott adatokat összeveti az adatbázisban tárolt felhasználóval (a UserRepository segítségével kikeresi a felhasználót felhasználónév alapján, majd ellenőrzi a jelszót a tárolt hash alapján). Ha az adatok helyesek, a backend egy JWT tokent generál. Ezt a tokent elküldi a válaszban (általában a response body-ban egy mezőként, pl. { "token": "eyJh...<JWT>..."} formában).
* A frontend AuthService megkapja a választ, kimenti a JWT tokent a böngésző tárolójába (pl. **localStorage** vagy sessionStorage), hogy a későbbi kéréseknél használni tudja. Ezt követően az Angular router átirányítja a felhasználót a védett főoldalra (Dashboard).
* Az AuthService beállít egy állapotot is, hogy a felhasználó be van jelentkezve, és a AuthInterceptor onnantól minden HTTP kéréshez hozzáfűzi az Authorization fejlécet: "Authorization": "Bearer <JWT token>". Ennek köszönhetően a további API hívások (könyv lekérdezés, kölcsönzés létrehozás stb.) a backend oldalon már autentikáltan érkeznek.
* A Spring Security a backend oldalon a JWT token alapján azonosítja a felhasználót minden kérésnél. Ha a token hiányzik vagy érvénytelen, a szerver **HTTP 401 Unauthorized** választ ad, és a frontend megfelelően reagál (pl. kilépteti a felhasználót, visszairányít a login oldalra).

A Login komponens kezelheti a sikertelen bejelentkezési próbálkozást is – pl. ha a szerver 401-et ad vissza hibás jelszó esetén, a komponens egy hibaüzenetet jelenít meg (pl. “Hibás felhasználónév vagy jelszó”). Ezenkívül – ha a rendszer támogatja – a bejelentkező oldalon lehetőség lehet **új fiók regisztrációjára** is. Jelen projektben külön regisztrációs UI nem készült; a regisztrációt a backend támogatja ugyan egy /api/auth/register végponttal, de ez tipikusan posztman vagy egy admin felület révén használható. A fejlesztés során az első felhasználót (adminisztrátort) érdemes manuálisan felvenni az adatbázisba vagy a data.sql inicializációs fájlban, illetve a fenti végpont meghívásával létrehozni.

**Összefoglalva:** a bejelentkezés komponens felelős a felhasználói hitelesítés indításáért és a JWT token kezeléséért. E komponens és a hozzá tartozó AuthService nélkül a többi funkció (amely védett végpontokat hív) nem elérhető.

**Vezérlőpult (Dashboard komponens)**

A bejelentkezés utáni első oldal a **DashboardComponent** (vezérlőpult). Ez a komponens összefogja a könyvtár-kezelés fő funkcióit egy nézetbe. A Dashboard megjeleníti:

* **Könyvlista:** a könyvtárban nyilvántartott összes könyv felsorolását (általában cím és szerző megjelenítésével, opcionálisan további adatokkal, pl. kategória, stb.). Ezt a listát a komponens inicializálásakor a frontend lekéri a backendről a BookService segítségével. A BookService.getBooks() metódus egy GET kérést küld a GET /api/books végpontra, mely válaszként visszaadja az összes könyvet JSON tömb formájában. A komponens elmenti ezt az adatot (pl. this.books) és a HTML sablonban \*ngFor direktívával listázza.
  + Minden könyvnél jelzi, hogy elérhető-e vagy már kikölcsönözték. Ezt többféleképpen lehet megoldani. Egyszerű esetben a frontend megnézi, van-e a könyvhöz tartozó aktív kölcsönzési bejegyzés (ehhez vagy a backend ad vissza jelzést, vagy a frontend külön lekérdezi a borrowers listát és összeveti). Ha a könyv szabad, akkor egy **“Kölcsönzés”** gomb jelenik meg mellette; ha már ki van kölcsönözve, akkor esetleg egy piros felirat jelzi pl. “Kölcsönözve – Határidő: 2023-12-01” és/vagy egy “Visszavétel” gomb, attól függően, hogy a visszavételt hol kezeljük (a Dashboardon belül vagy külön listában).
  + A **“Kölcsönzés”** gomb megnyomására a BorrowDialog (lásd következő pont) nyílik meg, hogy rögzíthessük a kölcsönzés részleteit.
* **Jelenleg kölcsönzött könyvek:** a Dashboard tartalmazhat egy külön szekciót vagy listát a pillanatnyilag kikölcsönzött könyvekről. Ez lehet egy külön komponens (BorrowedBooks) beágyazva a Dashboard nézetébe, vagy egyszerűen a Dashboard HTML sablonban egy második lista. Mindenesetre itt azok a könyvek szerepelnek, amelyeknél van aktív kölcsönzési rekord. Az egyes elemek tartalmazzák a könyv címét, szerzőjét, a kölcsönvevő nevét, elérhetőségeit (pl. telefon, email) és a **kölcsönzés határidejét** (end date). E lista elemei mellett található a **“Visszavétel”** gomb, amellyel a könyvtáros jelezheti, hogy az adott könyv visszaérkezett (ezzel törölve lesz a kölcsönzési rekord – erről bővebben később). Ha a visszavételt nem itt, hanem a könyvlistánál kezeljük, akkor a logika hasonló: pl. a kölcsönzött könyv sorában “Visszavétel” gomb jelenik meg.
* **Navigáció és egyéb elemek:** A Dashboard tetején tipikusan van egy üdvözlő felirat (pl. “Üdv, [felhasználónév]!”) és egy **Kijelentkezés** gomb. A kijelentkezés a AuthService logout() metódusát hívja, ami törli a tárolt JWT tokent és visszairányít a bejelentkező oldalra. Esetleg lehetőség van egy egyszerű menüre is (pl. navigáció külön oldalakra, ha lennének ilyenek, de jelen projekt fókuszában a könyvkezelés áll, így külön oldalak nélkül, egy oldalon belül történik minden).

A Dashboard komponens tehát egy áttekintő felület, ahol a könyvtáros egyből látja a fontos információkat: mely könyvek vannak kint, meddig, és innen érhet el minden műveletet (új kölcsönzés indítása vagy visszavétel regisztrálása). Ez a komponens leginkább megjelenítési logikát tartalmaz; az üzleti műveleteket a dialógusokon vagy szolgáltatásokon keresztül éri el.

**Kölcsönzés felvétele – BorrowDialog komponens**

A **BorrowDialogComponent** egy modális párbeszédablak (dialógus), amely akkor nyílik meg, amikor a könyvtáros elindít egy kölcsönzést (pl. a Dashboardon rákattint egy könyv melletti “Kölcsönzés” gombra). Ennek a komponensnek a feladata, hogy begyűjtse a kölcsönzés adatait:

* **Kapcsolódó könyv:** A dialógus tudja, hogy melyik könyvre nyitották meg (ezt paraméterként kapja, pl. Input property-n keresztül a kiválasztott book objektumot). A dialógus fejlécében megjeleníthetjük a könyv címét (pl. *“Kölcsönzés – A Gyűrűk Ura”*). Az űrlapon is szerepelhet a könyv címe és szerzője, de ezeket csak kijelzésre (és megerősítésre) használjuk, a könyvtáros nem módosíthatja őket – épp ezért tipikusan **le vannak tiltva** (disabled) ezek a mezők.
* **Kölcsönvevő adatai:** Az űrlap kéri a kölcsönző személy nevét, elérhetőségeit és a kölcsönzés időtartamát. Kötelező mezők:
  + Név (a kölcsönző személy neve)
  + Telefonszám
  + Email cím (így szükség esetén értesíteni lehet)
  + Kezdés dátuma (a kölcsönzés dátuma – alapértelmezésben a mai nap beállítható)
  + Visszahozatal dátuma (határidő – pl. 2 hét múlva; ezt a könyvtáros állíthatja be, a könyvtár szabályzata szerint)
* **Működés:** A BorrowDialog egy egyszerű HTML form, amely validációkat is tartalmaz: minden mező kötelező (required), az email mezőnek formailag e-mailnek kell lennie, a dátum mezők pedig dátum típusú inputok. Miután a könyvtáros kitöltötte az adatokat, a **“OK”** vagy **“Mentés”** gombra kattint. Ekkor a komponens meghívja a saját onSubmit() függvényét, ami ellenőrzi, hogy minden kötelező mező ki van-e töltve. Ha igen, akkor egy HTTP POST kérést indít a backend felé:
  + URL: POST /api/borrowers – ez a kölcsönzési rekord létrehozásának végpontja.
  + Törzs (body): a kölcsönzés adatai JSON formában. Példa küldött adatstruktúrára:

json

{

"name": "Kiss Péter",

"phone": "06201234567",

"email": "kiss.peter@example.com",

"startDate": "2025-07-01",

"endDate": "2025-07-15",

"bookTitle": "A Gyűrűk Ura",

"bookAuthor": "J. R. R. Tolkien"

}

Itt látható, hogy a kliens a könyv címét és szerzőjét is elküldi a biztonság kedvéért – a backend valójában a könyv egyedi azonosítóját is azonosíthatná, de jelen implementációban (a simplicity kedvéért) cím+szerző kombináció alapján azonosítjuk a könyvet. Ez feltételezi, hogy egy szerzőnek adott címmel csak egy könyve van a rendszerben (vagy legalábbis kicsi a kollízió esélye).

* + A komponens az Angular HttpClient modulját használja a kéréshez. Kód szinten így néz ki az elküldés (részlet a BorrowDialog komponensből):

typescript

this.http.post('http://localhost:8080/api/borrowers', this.borrower).subscribe({

next: () => {

this.close.emit(); // sikeres mentés után bezárjuk a dialógust

},

error: err => {

console.error('Hiba történt a mentés során:', err);

}

});

A sikeres válasz esetén (next ág) a dialógus bezárásra kerül – pl. a BorrowDialog a close eseményen keresztül jelzi a szülő komponensnek, hogy bezárható (vagy használhatja a MatDialogRef.close() metódust is, ha Angular Material dialog-ként van megnyitva). Sikertelenség esetén a hiba a konzolra kerül (esetleg a UI-ban is jelezhetnénk felugró értesítéssel, de jelen verzióban ez nem lett implementálva).

* **Backend oldali kezelés:** A BorrowerController a /api/borrowers végpontot feldolgozza. A beérkező JSON-t (lásd fent) felhasználva létrehoz egy új kölcsönzés (Borrower entitás) rekordot:
  + Lekérdezi az adatbázisból a szóban forgó könyvet: például a BookRepository.findByTitleAndAuthor(title, author) segítségével megkeresi a könyvet. Ha megtalálja, ellenőrizheti, hogy éppen nincs-e kölcsönadva (pl. megnézheti a BorrowerRepository-ban, hogy létezik-e aktív rekord ehhez a könyvhöz). Ha már ki van kölcsönözve, akkor hibát küld vissza (pl. 400 Bad Request vagy 409 Conflict státusszal).
  + Ha a könyv szabad, létrehoz egy Borrower entitást, beállítja annak mezőit: név, email, telefon, kezdés, határidő, valamint beállítja a hivatkozást a könyvre (pl. borrower.setBook(konyv)).
  + Esetleg beállíthatja a könyv állapotát is (pl. book.setBorrowed(true)), ha van ilyen mező. A mi esetünkben nincs külön flag, így ezt nem szükséges.
  + Elmenti a BorrowerRepository.save(borrower) hívással az új kölcsönzést. Ezzel az adatbázisban létrejön egy új rekord a Borrower (kölcsönző) táblában, ami jelzi, hogy XY személynek kölcsön van adva a könyv Y időpontig.
  + A válaszban visszaküldheti a létrehozott erőforrást vagy egy egyszerű üzenetet. REST konvenció szerint 201 Created státusz kóddal tér vissza, a body-ban az új erőforrás adataival (pl. egy BorrowerDTO formájában). A mi egyszerűbb megközelítésünkben akár 200 OK is lehet válaszként, hiszen a kliensnek nem feltétlenül kell újra feldolgoznia az adatot – elegendő, hogy tudja, sikerült a művelet.
* **Frontend folytatás:** A dialógus bezárása után a Dashboard komponensnek frissítenie kell a megjelenített adatokat: a könyvlista és a kölcsönzött könyvek listája is változik. Például az új kölcsönzés hatására az adott könyv már ne legyen kölcsönözhető (átkerül a “kint lévő” státuszba). Ezt megtehetjük úgy, hogy a BorrowDialog bezárását figyeljük (a close EventEmitter segítségével), és annak hatására meghívjuk újra a könyvlista és a kölcsönzött lista lekérdezését a szervertől. Így a Dashboard mindig naprakész adatokat fog mutatni.

**Megjegyzés:** a BorrowDialog komponens standalone (független) Angular komponensként lett megvalósítva (a @Component dekorátorban standalone: true szerepel). Ez az Angular 14+ egy új lehetősége, mellyel modul nélkül is definiálhatunk komponenst és közvetlenül betölthetjük. A modulok importálását a imports: [...] rész biztosítja a komponensen belül (pl. CommonModule, FormsModule, Angular Material modulok). Ennek köszönhetően a dialógus könnyen használható más komponensekből és modulárisabb.

**Kölcsönzött könyvek listája (BorrowedBooks komponens)**

A **BorrowedBooks** komponens (vagy szekció) azt a célt szolgálja, hogy a könyvtáros egy helyen lássa az összes folyamatban lévő kölcsönzést. Amennyiben a Dashboard részeként van megvalósítva, funkcionálisan akkor is elkülöníthető, így itt külön tárgyaljuk:

* **Adatok beszerzése:** A komponens inicializálásakor a frontend meghívja a backend GET /api/borrowers végpontját, tipikusan egy BorrowerService.getAll() vagy a BookService egy metódusa által. A szerver egy listát ad vissza a jelenleg aktív kölcsönzési rekordokról. Ezek a rekordok JSON objektumok, például így:

json

[

{

"id": 7,

"name": "Kiss Péter",

"email": "kiss.peter@example.com",

"phone": "06201234567",

"startDate": "2025-07-01",

"endDate": "2025-07-15",

"bookTitle": "A Gyűrűk Ura",

"bookAuthor": "J. R. R. Tolkien"

},

...

]

Látható, hogy a backend **BorrowerDTO** formában adja vissza az adatokat: az entitás kapcsolat helyett a könyv adatait (cím, szerző) is beletettük a DTO-ba, így a kliens egyetlen listában mindent megkap, ami a megjelenítéshez kell. (Alternatív megoldás: a Borrower objektum tartalmazhatna Book beágyazott objektumot is, de az komplikálhatja a JSON-t és a feldolgozást, ezért a DTO-k kényelmesebbek.)

* **Megjelenítés:** A BorrowedBooks listát a komponens táblázatos formában jelenítheti meg. Minden sor egy kölcsönzést reprezentál, oszlopokkal például:
  + **Könyv címe** – *Szerző* (egy oszlopban akár együtt: pl. “A Gyűrűk Ura – J. R. R. Tolkien”)
  + **Kölcsönvevő neve**
  + **Elérhetőség (telefon, email)** – ezt lehet két oszlopba is szedni vagy egy cellában felsorolni.
  + **Kölcsönzés kezdete** (formázott dátum)
  + **Határidő** (formázott dátum, ami jelzi, meddig maradhat a könyv)
  + **Műveletek** – ebben egy **“Visszavétel”** gomb.
* **Visszavétel művelet:** Ha a könyvtáros rákattint egy sorban a Visszavétel gombra, a rendszer végrehajtja a visszavétel logikát (ld. következő pont). Frontend oldalon ez annyit tesz, hogy meghív egy szolgáltatás metódust, ami törli a megfelelő kölcsönzési rekordot a szerveren:
  + Pl. BorrowerService.returnBook(borrowerId) hívódik, ami küld egy HTTP DELETE kérést a DELETE /api/borrowers/{id} végpontra.
  + A szerver sikeres törlés esetén 204 No Content választ ad (vagy 200 OK egy üzenettel). A frontend ezután frissíti a listát (lehívja újra a kölcsönzött könyvek listáját a szervertől, illetve a könyvlistát is, hogy az immár visszaérkezett könyv státusza frissüljön).
* **Üres lista kezelése:** Ha nincs éppen kölcsönzött könyv (üres a lista), a felület jelenítsen meg valami tájékoztatót, pl. “Jelenleg nincs kölcsönzött könyv.”. Ez a felhasználói élmény szempontjából hasznos.

A BorrowedBooks komponens így segít a könyvtárosnak gyorsan áttekinteni a kint lévő könyveket, és egyszerű felületen intézni a visszavételt. Mindez real-time-nak tűnik a felhasználó számára, hiszen minden művelet után rögtön frissítjük a listákat. Természetesen a “valós idejű” szinkronizációhoz kliens oldali polling vagy websockets is megvalósítható lenne, de ebben a projektben a kölcsönzések frekvenciája alacsony, így elegendő a műveletek utáni manuális frissítés.

**Könyv visszavételének logikája (Return process)**

A **visszavétel** azt jelenti, hogy egy könyv kölcsönzési időszaka lezárul, a könyvet visszajuttatták a könyvtárba. Ennek kezelése a rendszerben két fő lépésből áll:

1. **Kölcsönzési rekord lezárása/törlése a backendben:** A legegyszerűbb megoldás, hogy a visszavételkor töröljük a hozzátartozó Borrower (kölcsönzési) rekordot az adatbázisból. A BorrowerController DELETE /api/borrowers/{id} végpontja ezt teszi:
   * Megkeresi az adatbázisban az adott azonosítójú kölcsönzést (pl. borrowerRepository.findById(id)).
   * Ha nem találja, 404 Not Found választ ad (pl. ha már törölték vagy rossz id-t kap).
   * Ha megtalálja, egyszerűen meghívja a borrowerRepository.delete(borrower) metódust, ami eltávolítja az entitást. Ezzel az adott könyvhöz tartozó aktív kölcsönzés megszűnik.
   * (Opcionális lépés: ha a Book entitásnak volt valamilyen állapota, pl. isBorrowed flag, azt itt **false**-ra állíthatnánk és mentenénk a Book-ot. Mivel a mi modellünkben ilyen nincs, erre nincs szükség. A könyv elérhetőségét az határozza meg, hogy létezik-e aktív Borrower rekord rá vonatkozóan.)
   * Válaszként 204 No Content küldése (üres törzs), jelezve, hogy a törlés sikerült.
2. **Frontend frissítés:** Amikor a kliens megkapja a sikeres választ a törlésre, frissíti a felületet:
   * Eltávolítja a törölt kölcsönzést a BorrowedBooks listából (ha lokálisan tartotta nyilván, vagy újra kéri a listát).
   * A könyvlistában az érintett könyvnél ismét elérhetővé teszi a “Kölcsönzés” gombot, hiszen már nincs aktív kölcsönzés rá. (Ha újra lekérjük a /api/books listát, az is megoldja – mivel most már a backend sem látja kölcsönzöttként a könyvet. Alternatív megoldás: a Borrower törlése után a backend a könyv adatait is visszaküldhetné frissítve, de ez most nincs implementálva.)
   * Esetleg felhasználói üzenetet jelenít meg: pl. “A könyv visszavétele rögzítésre került.” (ez nem kötelező, de UX szempontból kedves).

**Üzleti szabályok:** A visszavétel logikánál figyelni kell arra, hogy csak akkor engedjük a műveletet, ha valóban van mit visszavenni. Praktikusan a UI-ban sem jelenik meg a Visszavétel gomb olyan könyvnél, ami nincs kikölcsönözve. Ha valahogy mégis rossz kérést küldene a kliens (pl. olyan ID-t törölne, ami nem létezik vagy már törölve lett), a szerver 404-et ad, amit a kliensnek le kell kezelnie (esetleg frissíteni az adatokat, mert lehet, hogy elavult információi vannak).

**Könyv újbóli kölcsönzése:** Miután a könyvet visszavettük, újra kölcsönadható. A rendszerben ez azt jelenti, hogy mivel a korábbi Borrower rekord már nincs, egy új kölcsönzés felvétele (BorrowDialog) ugyanarra a könyvre ismét lehetséges lesz. Az adatbázisban a régi kölcsönzés nyoma (ha töröltük) nincs meg – így a rendszer nem vezet teljes történeti naplót. Amennyiben fontos lenne a kölcsönzési előzmények megőrzése, más megközelítés kellene: pl. a Borrower entitásnak lenne egy returnedDate mező, és nem törölnénk a rekordot, csak beállítanánk, hogy visszahozva ekkor, és a lekérdezésekben csak a returnedDate == null (aktív) rekordokat kérdeznénk le. A jelenlegi implementáció egyszerűség kedvéért nem tárolja a múltbéli kölcsönzéseket – de ezt dokumentáljuk és a jövőbeni fejlesztésnél figyelembe lehet venni.

**Összefoglalás:** A visszavétel logikája egyszerű törlésként lett megvalósítva. Ez hatékony és könnyen kezelhető módja annak, hogy a könyvet “felszabadítsuk”. A rendszer a mindenkori aktuális állapotot tartja nyilván (mely könyvek vannak éppen kölcsönözve), de nem listázza a múltbéli kölcsönzéseket. Ez megfelel a célközönség igényeinek a jelen pillanatban – elsősorban azt akarják tudni, **mi van kint és mi van bent**.

**Adatbázismodell ismertetése**

Az alkalmazás relációs adatbázist használ (PostgreSQL), az objektum-orientált domain modell és az adatbázis táblák között a JPA teremt kapcsolatot. Az alábbiakban ismertetjük a fő entitásokat és azok kapcsolatát, különös tekintettel a **Book** (Könyv) és **Borrower** (Kölcsönzés/Kölcsönvevő) közötti összefüggésre.

* **Book (Könyv) entitás:** A book tábla minden rekordja egy-egy könyvet képvisel a könyvtárban. Főbb mezők:
  + id – egyedi azonosító (primer kulcs, generált érték).
  + title – a könyv címe.
  + author – a könyv szerzője.
  + További mezők lehetnek (a projekt követelményeitől függően) pl. year (kiadás éve), publisher, isbn, category stb., de a minimális rendszerhez ezek nem feltétlen tartoznak hozzá. Jelen implementációban a cím és a szerző a két lényeges adat.
  + A Book entitásban nincs közvetlenül olyan mező, hogy “kikölcsönözve” vagy “elérhető” – ezt dinamikusan a Borrower entitással való kapcsolat határozza meg.
* **Borrower (Kölcsönzési rekord) entitás:** A borrower tábla a könyvkölcsönzéseket tartalmazza. Fontos megérteni, hogy itt nem (csak) egy személy adatairól van szó, hanem **egy kölcsönzési eseményről**. Tartalmazza:
  + id – egyedi azonosító minden kölcsönzési rekordhoz.
  + name – a kölcsönző személy neve.
  + phone – a kölcsönző telefonja.
  + email – a kölcsönző email címe.
  + start\_date – a kölcsönzés dátuma (mettől vitte el a könyvet).
  + end\_date – a kölcsönzés határideje (meddig kell visszahozni).
  + **Kapcsolat a Book entitáshoz:** Itt jelenik meg a reláció. A Borrower entitás tipikusan tartalmaz egy book\_id idegen kulcsot, amely annak a könyvnek az azonosítójára mutat, amelyet kikölcsönöztek. JPA szinten ezt úgy valósítjuk meg, hogy a Borrower Java osztályban van egy mező pl. @ManyToOne private Book book;. Ez azt jelzi, hogy több kölcsönzési rekord is tartozhat egy könyvhöz (hiszen egy könyvet az idők során sokszor kikölcsönözhetnek, de **egy időben** általában csak egy aktív kölcsönzés van rá).
  + A kapcsolat másik oldalán, a Book entitásban definiálhatunk egy @OneToMany(mappedBy="book") private List<Borrower> borrowers; kapcsolatot, ami megmutatja, hogy egy könyvhöz tartozó összes kölcsönzést lekérdezhetjük. Ezt azonban érdemes óvatosan használni, mert ha nagyon sok rekord van vagy nem szűrjük státusz szerint, fölösleges adatot hozhat. Emiatt – és a JSON szerialization bonyodalmai miatt – sokszor kihagyjuk a Book oldalán a kollekciós kapcsolatot, és inkább lekérdezzük a BorrowerRepository segítségével, ha kell.
* **Kapcsolat jellege (Book – Borrower):** A kapcsolat **1-to-N (egy-több)**. Egy könyvnek több kölcsönzési rekordja lehet (időben egymás után). A rendszer biztosítja, hogy egy könyvhöz egy időben legfeljebb egy aktív Borrower rekord tartozzon (mégpedig a visszavétel logika miatt: visszavételkor töröljük a rekordot, így egy időben maximum egy létezik). Tehát a **Book és Borrower között logikailag egy-egy kapcsolat áll fenn aktív állapotban**, de történetileg nézve egy könyvhöz tartozhat sok Borrower (ha sokszor kölcsönözték ki egymás után).
* **User entitás:** Bár a kérdés főleg a Book–Borrower kapcsolatra fókuszál, említésre méltó, hogy van egy User entitás is, ami a felhasználói fiókokat (könyvtárosokat) tárolja. A User nincs közvetlen kapcsolatban a Book vagy Borrower entitással a modellben – legfeljebb annyi, hogy a kölcsönzéseket mindig egy bejelentkezett user indítja, de ezt a rendszer nem naplózza (ha szeretnénk, hozzáadhatnánk a Borrower entitáshoz egy user\_id mezőt, hogy ki adta ki a könyvet).
* **Adatbázis integritás:** A borrower.book\_id idegen kulcs hivatkozás meg van erősítve adatbázis szinten is (FOREIGN KEY), így nem létezhet olyan kölcsönzési rekord, ami nem létező könyvre mutat. Könyv törlése esetén az adatbázis integritási hiba lenne, ha van rá aktív kölcsönzés – ezt a alkalmazás logikájában is kezeljük (pl. nem engedjük törölni a könyvet, amíg ki van kölcsönözve). Ha mégis szükség lenne könyv törlésére (pl. hibásan vitték fel), előbb a kölcsönzési rekordokat kell rendezni (törölni vagy módosítani).
* **Tranzakciók:** A Spring JPA alapértelmezetten tranzakciósan kezeli a repository műveleteket a service rétegben (pl. a service metóduson szerepelhet @Transactional). A kölcsönzés létrehozása vagy törlése egy tranzakcióban történik, így biztosítva, hogy az adatbázisban konzisztens állapot maradjon. Például, ha kölcsönzés létrehozásakor valamilyen lépés közben hiba történik (pl. a könyv keresése nem talál semmit), akkor az egész művelet megszakad és nem jön létre csonka rekord.

**Összegezve az adatmodellt:** A Könyv és Kölcsönzés kapcsolat egy klasszikus példája a **master-detail** viszonynak: a könyv a fő entitás, a kölcsönzés pedig egy esemény ami a könyvhöz kapcsolódik. Egy másik szemlélet szerint a Borrower entitást tekinthetjük a “kölcsönzési napló” bejegyzéseinek. A BiblioSys jelen formájában az aktuális naplót tartja nyilván. A későbbiekben könnyen bővíthető a modell, hogy a történelem is megmaradjon (pl. returned flag vagy külön History tábla), de erre most nem volt kritikus igény.

**Az alkalmazás futtatása lokálisan**

A BiblioSys projekt futtatásához szükség van a megfelelő fejlesztői környezetre mind frontend, mind backend oldalon. Feltételezzük, hogy a forráskód már a gépünkön van két mappában (library-frontend és library-management). Az alábbiakban lépésről lépésre bemutatjuk, hogyan lehet a rendszert elindítani lokális környezetben:

**Előfeltételek telepítése:**

1. **Node.js és Angular CLI:** Győződjünk meg róla, hogy a Node.js telepítve van a gépen (ajánlott a legfrissebb LTS verzió, pl. 16.x vagy 18.x). Az Angular CLI globális telepítése is hasznos (npm install -g @angular/cli), bár a projekt közvetlenül npm start vagy ng serve parancsokkal is futtatható, ha a node\_modules telepítve van.
2. **Java fejlesztőkörnyezet:** Telepítve kell legyen egy JDK (Java Development Kit), minimum Java 17 (mivel a modern Spring Boot verziók ezt igénylik, ha a projekt Spring Boot 3.x-et használ). Győződjünk meg arról is, hogy a JAVA\_HOME megfelelően be van állítva.
3. **Maven vagy Gradle:** A projekt a Spring Boot alapértelmezett eszközével (általában Maven) épül. Ha Maven-t használunk, telepítve kell legyen (vagy használhatjuk az IDE beépített Mavenjét). Alternatív: ha a projekt build.gradle-t tartalmazna, akkor Gradle szükséges – de a library-management valószínűleg Maven projekt (pom.xml alapján).
4. **PostgreSQL adatbázis:** Telepítsük a PostgreSQL szervert (ha még nincs). Hozzunk létre egy új adatbázist a projekt számára, például bibliosys néven. Hozzunk létre egy felhasználót is (pl. bibliosys\_user) egy erős jelszóval, és adjunk neki megfelelő jogosultságot erre az adatbázisra (CREATE, UPDATE, DELETE stb.). Jegyezzük fel az adatbázis elérhetőségét (host, port – alapértelmezett a localhost:5432 –, adatbázisnév, felhasználó, jelszó).

**Backend (library-management) futtatása:**

1. Nyissuk meg a projektet egy IDE-ben (IntelliJ IDEA, Eclipse vagy VSCode megfelelő bővítménnyel). Töltsük le a függőségeket Maven segítségével (az IDE ezt automatikusan megteszi a pom.xml alapján, vagy futtassuk a mvn install parancsot a projekt gyökérben).
2. Állítsuk be az adatbázis kapcsolatot a **src/main/resources/application.yml** fájlban. Ebben a fájlban konfiguráljuk a spring.datasource beállításokat:

spring:

datasource:

url: jdbc:postgresql://localhost:5432/bibliosys

username: bibliosys\_user

password: <a-jelszó>

jpa:

hibernate.ddl-auto: update

generate-ddl: true

show-sql: true

Az update ddl-auto érték azt eredményezi, hogy a hibernate az alkalmazás indulásakor frissíti a sémát a modellek alapján (ha még nincsenek táblák, létrehozza őket). Fejlesztési környezetben ez kényelmes. (Éles környezetben inkább flyway/liquibase használata javasolt migrációkra.)

1. Ellenőrizzük a application.yml-ben a JWT beállításait is (ha vannak). Gyakran be van állítva egy titkos kulcs, pl. jwt.secret, és token lejárati idő. Ezeknek a fejlesztéshez már be kell legyenek keménykódolva vagy konfigurálva, de ha hiányoznak, pótolni kell. (Például: jwt.secret: MyJwtSecretKey – ezt élesben nyilván környezeti változóként kellene kezelni.)
2. Indítsuk el a Spring Boot alkalmazást. Ezt megtehetjük az IDE "Run" funkciójával a main fv. futtatásával (LibraryManagementApplication), vagy parancssorból a projekt gyökérkönyvtárában: mvn spring-boot:run. Ha buildelt JAR fájlt szeretnénk: mvn package után a target/library-management.jar futtatható java -jar paranccsal.
3. Ha minden jól megy, a konzolon látjuk a Spring Boot indulási logokat. Az alkalmazás alapértelmezetten a **8080-as porton** indul (ezt az application.yml server.port beállításával lehet módosítani, de jelenleg maradhat 8080). Mikor megjelenik a „Started LibraryManagementApplication ...” üzenet, a backend készen áll.

**Frontend (library-frontend) futtatása:**

1. Parancssorban navigáljunk a library-frontend mappába.
2. Adjuk ki az npm install parancsot. Ez letölti az Angular projekthez szükséges összes függőséget a package.json alapján a node\_modules könyvtárba. (Ügyeljünk rá, hogy internetkapcsolat szükséges ehhez a lépéshez.)
3. Ha a telepítés kész, indítsuk el a fejlesztői szervert: ng serve --open (vagy npm start, ha be van állítva). Az --open flag hatására a parancs automatikusan megnyit egy böngészőablakot is a megfelelő URL-en.
4. Az Angular fejlesztői szerver alapértelmezetten a **http://localhost:4200/** címen fut. Ezen az URL-en fog betöltődni a BiblioSys webes felülete. Mivel a backend 8080-on van, a böngésző és a szerver közötti kommunikáció cross-origin lesz, de ezt már a CorsConfig engedélyezte.
5. A böngészőben meg kell jelennie a bejelentkező oldalnak. Ha látjuk a bejelentkezési felületet, akkor a frontend sikeresen fut. (Ha nem nyílt meg automatikusan, manuálisan navigáljunk a http://localhost:4200 címre.)
6. Jelentkezzünk be egy létező felhasználóval. Ha ez egy tiszta adatbázis, akkor előbb létre kell hozni egy felhasználót:
   * Ezt megtehetjük pl. a psql parancssorból egy SQL beszúrással a user táblába (figyelve a jelszó hash-re), vagy egyszerűen meghívhatjuk a regisztrációs végpontot. Használhatunk cURL-t vagy Postman-t: pl.

bash

curl -X POST http://localhost:8080/api/auth/register \

-H "Content-Type: application/json" \

-d '{ "username": "admin", "password": "adminpass" }'

(Ha van ilyen végpont és nincs tiltva.)

* + A lényeg, hogy legyen egy **adminisztrátor** felhasználónk. Például hozzunk létre egy admin felhasználót jelszóval adminpass.

1. A bejelentkezési felületen adjuk meg a felhasználónevet és jelszót, nyomjunk **Bejelentkezés** gombot. Sikeres autentikáció után átirányít a rendszer a főoldalra (Dashboard).
2. Most mind a backend, mind a frontend fut, és használhatjuk az alkalmazást.

**Problémamegoldás futtatáskor:**

* Ha a backend indításakor hibaüzenetet kapunk, ellenőrizzük az adatbázis beállításokat. Gyakori hiba, hogy rossz jelszó vagy nincs létrehozva a megadott adatbázis.
* Ha a frontend indul, de nem éri el a szervert (pl. a login gombra nem történik semmi, a konzolon CORS hibát látunk), akkor valószínűleg a backend CorsConfig nincs jól beállítva. Fejlesztéshez általában @CrossOrigin(origins="http://localhost:4200") annotációt is tehetünk a controller osztályokra vagy engedélyezhetjük globálisan. Jelen projektben egy külön CorsConfig enged mindenkit, de ha nem működik, gyors tesztként be lehet kapcsolni a Spring Boot-ban a http.cors().and().csrf().disable() sort a SecurityConfig-ban.
* Ha az Angular nem indul (pl. verzió-kompatibilitási hibák), ellenőrizzük a Node.js verziót és az Angular CLI verzióját. Esetleg futtassuk a ng update parancsot vagy hozzuk a projektet az aktuális Angular verzió alá.
* Ha az Angular indul, de a login után nem navigál tovább, nézzük meg a Developer Console-t a böngészőben. Lehet, hogy a login kérést elutasította a szerver (401) – ebben az esetben ellenőrizzük a megadott jelszót és a felhasználó létezését. Lehet logolni is a AuthService-ben a válaszokat segítségképp.

A futtatási lépések végén egy működő alkalmazást kell kapnunk, ahol a felhasználó bejelentkezhet, láthatja a könyveket, felvehet kölcsönzést és visszavételezhet. A fejlesztői szerverek futása közben az Angular automatikusan újratölti a változtatásokat (hot reload), a Spring Boot pedig újraindítható könnyen. Így a fejlesztés során gyors iterációk lehetségesek.

**Fejlesztési útmutató – új funkciók hozzáadása**

A BiblioSys kódjai könnyen bővíthetők új funkciókkal, köszönhetően a moduláris felépítésnek. Ebben a fejezetben áttekintjük, hogyan érdemes új funkciókat implementálni a projekthez, és mire kell figyelni a fejlesztés során, hogy a rendszer konzisztens és hibamentes maradjon.

**Általános fejlesztési elvek:**

* **Rétegek betartása:** Kövessük a meglévő architektúrát. Amikor új funkciót (pl. új entitást vagy use-case-t) vezetünk be, próbáljuk illeszteni a meglévő rétegstruktúrába:
  + Hozzunk létre új entitást (ha szükséges) az entity csomagban.
  + Készítsünk hozzá repository interfészt a repository csomagban.
  + Az üzleti logikát tegyük külön service osztályba a service csomagban. Itt implementáljuk a funkció lépéseit (tranzakciósan, ha adatbázis módosítást végez).
  + Tegyünk ki szükség szerint DTO-kat a dto csomagban, ha az entitást nem akarjuk közvetlenül kitenni vagy össze kell kombinálni adatokkal.
  + Készítsünk a frontend felé REST végpontokat a controller csomagban. Használjuk a service-t, validáljuk a bemenetet (@Valid és Bean Validation annotációk a DTO-kon), és biztosítsuk a megfelelő HTTP státuszkódokat.
  + Frontend oldalon hozzunk létre új komponenst és/vagy szolgáltatást. Az Angular modul felépítésben, ha a funkció nagyobb moduláris egység (pl. “könyv keresése” oldal), lehet külön Angular modult is létrehozni lazy loadinggal. Ha kisebb kiegészítés (pl. egy új mező kezelése), elég a meglévő komponens módosítása.
* **Konzisztens kód stílus:** Tartsuk be a projektben eddig is alkalmazott kódolási stílust. Például:
  + Java oldalon használjunk beszédes elnevezéseket, CamelCase-t osztály- és metódusnevekhez, konvenció szerint a Spring bean-ek nevei kis kezdőbetűsek.
  + TypeScript oldalon kövessük az Angular style guide-ot (pl. komponens osztálynevek nagy kezdőbetű, változók camelCase).
  + Formázás: Az IDE-kben beállítható auto-format (pl. Prettier Angularhoz, és az IDE formázója Java-hoz). Egységesítsük a behúzásokat, sortöréseket a könnyebb olvashatóságért.
* **Validáció és hiba kezelés:** Új funkciók esetén gondoljunk a bemenő adatok validációjára:
  + Backend: ha új DTO-t vezetünk be, annotáljuk a mezőit pl. @NotNull, @Size stb. ahol kell, és a controller paraméterhez tegyük oda a @Valid-et, hogy a Spring automatikusan validáljon. A ValidationExceptionHandler már globálisan kezeli a hibákat és 400 Bad Request választ ad JSON hibatesttel, így ezt a mechanizmust használjuk.
  + Frontend: érdemes a felhasználói bevitelt HTML5 szinten vagy Angular form validációval ellenőrizni (pl. required, pattern), hogy még azelőtt jelezzük a hibát, mielőtt a szerverig eljutna. Természetesen a szerver oldali validációt ez nem váltja ki, de javítja a felhasználói élményt.
  + Hibaüzenetek: Bővíthetjük a felületet, hogy az API hibákra (pl. 400 vagy 500) kulturált üzenetet adjon. Például, ha a szerver azt válaszolja, hogy “Könyv már ki van kölcsönözve” (409 Conflict), azt megjeleníthetjük egy felugróban vagy piros szövegként az érintett komponensben.
* **Biztonság és jogosultságok:** Új végpont bevezetésekor dönteni kell, védett legyen-e. Alapértelmezésben a SecurityConfig a /api/auth/\*\* kivételével mindent véd. Tehát ha létrehozunk pl. egy /api/report végpontot jelentések lekérdezésére, az automatikusan védett lesz (token kell hozzá). Ha valamit nyilvánossá akarunk tenni, azt a SecurityConfig-ban antMatchers().permitAll() (Spring Security 5.x) vagy újabb Spring Security esetén a HttpSecurity.authorizeHttpRequests() láncolásban kell beállítani. Figyeljünk arra is, hogy a tokennel hívott végpontok esetén a felhasználó adatai lekérdezhetők a SecurityContext-ből, ha szükséges (pl. @AuthenticationPrincipal UserDetails user paraméterrel egy controller metódusban).
* **Adatbázis módosítások:** Ha az új funkció adatbázis változást igényel (új tábla, mező hozzáadása), a fejlesztői fázisban ezt a hibernate.ddl-auto: update megoldja automatikusan. Viszont ügyeljünk arra, hogy a változás kompatibilis legyen a meglévő adatokkal. Prod környezetben érdemes verziózott migrációkat használni (Flyway), de dev-ben az update elég. Ha pl. mezőt adunk hozzá egy entitáshoz, tegyük lehetőleg nem kötelezővé az elején (nullable = true a JPA annotációnál), különben a futó adatbázis update-je nem fog sikerülni, ha vannak már rekordok (persze developmentnél akár drop schema is megengedhető).
* **Teljeskörű funkcionalitás:** Gondoljuk végig az új feature “lifecycle”-jét:
  + Ha például “új könyv kategorizálása” funkciót adunk hozzá, akkor:
    - Adjuk hozzá a kategória mezőt a Book entitáshoz és adatbázishoz.
    - Ezt jelenítsük meg a könyv felvételi űrlapon (ha van külön UI könyv hozzáadásra, vagy implementáljunk ilyet).
    - Biztosítsuk, hogy listázásnál is megjeleníthető (UI változtatás).
    - Adjuk hozzá a filterezési lehetőséget (pl. kategória szerinti szűrés), ha értelmezett.
    - Ehhez kellhet új végpont (pl. GET /api/books?category=X).
    - Ne feledjük a jogosultságokat: valószínűleg csak admin adhat hozzá könyvet, de a mi rendszerünkben minden authentikált felhasználót adminnak tekintünk, szerepkörök nélkül. Ha bevezetnénk külön szerepeket (pl. “admin” aki könyvet is felvihet, “user” aki csak kölcsönözhet), akkor a SecurityConfig-ot és a UI elemeket is ennek megfelelően alakítsuk (ehhez Role alapú engedélyezést állíthatunk be Spring Security-ben, és a JWT tokenbe is bele kell tenni a szerepkört).
* **Tesztelés és hibakeresés:** Új fejlesztés esetén mindig teszteljük le a fő use-case-eket:
  + Egységtesztek írása a service réteghez (ha van rá idő) nagyon ajánlott – a projekt bővülésével a regressziós hibák elkerülésére.
  + Manuálisan is teszteljük a frontendet: próbáljuk ki a tipikus forgatókönyveket (pl. új funkciónál a legjobb, legrosszabb és átlagos eseteket is). Nyissuk meg a böngésző fejlesztői eszközöket és figyeljük a konzolt ill. hálózati kéréseket, hogy minden kérés a várt választ adja.
  + Hibakereséshez backendnél hasznos a logolás. Szükség esetén állítsuk be a logolási szintet (pl. application.yml-ben logging.level.hu.bibliosys=DEBUG), vagy tegyünk ideiglenes logokat a kódba. Frontendnél a console.log hívások segíthetnek.
  + Külön figyeljünk arra, hogy az új funkció ne rontsa el a meglévőket. Például ha a bejelentkezési folyamatot módosítjuk (pl. jelszó reset funkciót adunk hozzá), véletlenül se tegyük tönkre a normál login működését.

**Konkrét példák a bővítésre:**

* *Új entitás hozzáadása:* Tegyük fel, szeretnénk nyilvántartani a könyveknél a kiadót is külön entitásként (Publisher), és minden könyvhöz hozzárendelni. Lépések:
  1. Backend: hozzuk létre a Publisher entitást, benne pl. név, cím mezőkkel. Legyen kapcsolat Book entitással (@OneToMany Publisher -> Book, Book oldalán @ManyToOne Publisher).
  2. Hozzunk létre PublisherRepository, PublisherService, és adott esetben PublisherController-t, ha külön akarjuk kezelni (pl. listázni kiadókat, hozzáadni új kiadót).
  3. Frontend: készítsünk egy felületet a kiadók kezelésére (ha kell), vagy integráljuk a könyvfelviteli űrlapba a kiadó kiválasztását (pl. egy dropdown mező).
  4. Figyeljük meg, hogy a Book entitás új mezője (publisher\_id) miatt a könyv felvitelekor a backend várni fogja ezt is. Tehát a BookDTO-t is frissíteni kell, és a frontenden is küldeni kell ezt az infót.
  5. Teszteljük, hogy könyv hozzáadás működik-e, kiadók listázása stb.
* *Új végpont a meglévő entitáshoz:* Mondjuk bevezetünk egy keresőfunkciót, hogy cím alapján lehessen keresni a könyvek között.
  1. Backend: BookRepository-ba írhatunk egy metódust: List<Book> findByTitleContainingIgnoreCase(String title). BookService-ben egy metódus, ami meghívja ezt.
  2. BookController-ben új végpont: pl. GET /api/books/search?query=valami. Ebből kivesszük a query paramétert, meghívjuk a service-t, visszaadjuk a találatokat.
  3. Frontend: új komponenst nem feltétlen kell, a Dashboard-on lehet egy keresőmező. Az Angular komponensben implementáljuk, hogy a beírt szövegre hívja a BookService.searchBooks(query) metódust, ami meghívja a GET /api/books/search?query=...-t.
  4. Megjelenítjük a találatokat, esetleg ha üres a kereső input, visszaváltunk az összes könyv listájára.
  5. Teszt: keressünk létező és nem létező címrészletekre, nézzük meg az eredményt.

**Mire figyeljünk még:**

* **Kompatibilitás:** Ha a frontend és backend egyszerre fejlesztjük, ügyeljünk rá, hogy a interfészek (API-k) illeszkedjenek. Egy apró eltérés (pl. más JSON mezőnév) bughoz vezethet. Használjunk konzisztens elnevezéseket. Például a backend DTO-ban startDate a mező neve, akkor a frontend modellben is startDate legyen, különben nem fog bindelődni automatikusan Angularban.
* **Teljesítmény:** Az alkalmazás jelenlegi méreténél nem kritikus, de ha bővítjük, figyeljünk a potenciális teljesítmény problémákra:
  + Ne töltsünk le egyszerre túl sok adatot feleslegesen. Pl. ha van 10.000 könyv, nem biztos, hogy egyszerre mindet listázni kell – érdemes lehet lapozást (pagination) bevezetni a Book listázásnál. A Spring Data JPA tud Page objektumot visszaadni, a Spring MVC pedig kezel page, size paramétereket. Frontenden pedig implementálhatjuk a lapozást egy táblázathoz.
  + Hasonlóképp, a kölcsönzések listája is lapozható, bár valószínűleg egyszerre nem lesznek ezres nagyságrendben aktív kölcsönzések.
  + Az Angular alkalmazásnál figyeljünk arra, hogy ne tartsunk feleslegesen sok *two-way binding*-ot nagy listákra, mert az ronthatja a teljesítményt. Egyszerű *one-way binding* (pl. \*ngFor to list items) a megfelelő.
* **Dokumentáció és karbantartás:** Minden új funkcióról érdemes röviden dokumentációt írni (hasonlóan ehhez a kézikönyvhöz). Tartsuk naprakészen a README-t vagy a fejlesztői doksit, hogy mi hogyan működik. Ez segít a csapat új tagjainak, vagy később magunknak is, ha visszatérünk a projekthez.

Összefoglalva, a BiblioSys bővítése során a kulcsszó a **következetesség**. A meglévő kód mintáját követve az új fejlesztések is illeszkedni fognak a rendszerbe, és a fejlesztési, tesztelési folyamat zökkenőmentes lesz. Mindig teszteljünk és gondoljuk át a felhasználói szemszögből is az új funkció működését.

**API végpontok listája és működése**

Az alábbiakban felsoroljuk az alkalmazás által nyújtott fontosabb REST API végpontokat, rövid leírással. Minden végpont JSON formátumú kéréseket vár és JSON válaszokat ad vissza. A védett végpontok megkövetelik a JWT token meglétét az **Authorization** headerben (Bearer <token> formában).

**Autentikációs és felhasználó-kezelő végpontok:**

* **POST /api/auth/register** – *Új felhasználó regisztrációja.* A kérés body-ban egy JSON tartalmazza a regisztrációs adatokat (pl. "username", "password"). A szerver létrehozza az új felhasználót (a jelszót titkosítva tárolja). Válasz: 201 Created, siker esetén tipikusan visszaadja az új felhasználót (vagy üres törzzsel).
* **POST /api/auth/login** – *Bejelentkezés.* A kérés body-ban "username" és "password". Ha sikeres azonosítás, a válasz 200 OK, tartalmazza a JWT tokent és esetleg a felhasználó adatait. Példa válasz:

json

{ "token": "<JWT\_TOKEN\_STRING>", "user": { "id": 1, "username": "admin" } }

Sikertelen bejelentkezés esetén 401 Unauthorized státusz.

* *(Opcionális)* **GET /api/auth/me** – Ha van ilyen, visszaadhatja a bejelentkezett felhasználó profilját a token alapján. (Projekt specifikációjában nem említették, de gyakran van ilyen végpont.)
* **POST /api/auth/logout** – Mivel a JWT stateless, nincs igazi logout végpont a szerveren; a kiléptetés annyi, hogy a kliens törli a tokenjét. Ezt a végpontot akár meg sem kell valósítani. Ha biztonsági okból mégis kezelni akarjuk (pl. token blacklisting egy adatbázisban), implementálható.

**Könyv (Book) erőforrás végpontok:**

* **GET /api/books** – *Könyvek listázása.* Visszaadja az összes könyvet a rendszerben. Védett végpont (csak bejelentkezve). Válasz példa (200 OK):

json

[

{ "id": 10, "title": "A Gyűrűk Ura", "author": "J. R. R. Tolkien" },

{ "id": 11, "title": "Harry Potter és a Bölcsek Köve", "author": "J. K. Rowling" },

...

]

A válaszban nincsenek kölcsönzési információk, csak a könyv adatai. A kliens a Borrower végpontból tudja kideríteni, melyik van kikölcsönözve.

* **GET /api/books/{id}** – *Egy konkrét könyv lekérdezése.* Visszaadja a megadott azonosítójú könyv részletes adatait. Ha nincs ilyen id, 404-et dob.
* **POST /api/books** – *Új könyv hozzáadása.* A kérés body tartalmazza a könyv adatait (cím, szerző, stb.). Siker esetén 201 Created, a törzsben az új könyv adataival (beleértve az adatbázis által generált id-t). Ezt a végpontot csak adminisztrátor használja jellemzően.
* **PUT/PATCH /api/books/{id}** – *Könyv adatainak módosítása.* A kérés body-ban a módosítandó mezők (vagy teljes könyv objektum). Siker esetén 200 OK (esetleg visszaadja a frissített adatokat), 404 ha nem található.
* **DELETE /api/books/{id}** – *Könyv törlése.* Kitörli a könyvet az adatbázisból. 204 No Content válasz siker esetén. Fontos: ha a könyv ki van kölcsönözve, a backend dönthet úgy, hogy tiltja a törlést (409 Conflict ad vissza pl.). Jelen implementációban feltételezzük, hogy nincs ilyen logika beépítve, de érdemes ezt figyelembe venni.

**Kölcsönzési (Borrower) erőforrás végpontok:**

* **GET /api/borrowers** – *Aktív kölcsönzések listája.* Védett végpont, visszaadja az összes jelenleg aktív kölcsönzést (az egyszerűség kedvéért a rendszerben lévő összes Borrower rekordot, mivel visszavételkor töröljük a rekordot). A válasz egy lista BorrowerDTO objektumokkal. Példa 200 OK válasz:

json

[

{

"id": 7,

"name": "Kiss Péter",

"phone": "06201234567",

"email": "kiss.peter@example.com",

"startDate": "2025-07-01",

"endDate": "2025-07-15",

"bookTitle": "A Gyűrűk Ura",

"bookAuthor": "J. R. R. Tolkien"

}

]

Itt egy elem jelzi, hogy “A Gyűrűk Ura” c. könyv ki van kölcsönözve Kiss Péternek július 15-ig.

* **POST /api/borrowers** – *Új kölcsönzési rekord létrehozása (egy könyv kikölcsönzése).* A kérés JSON body tartalmazza a kölcsönvevő adatait és a könyvazonosítást (nálunk bookTitle+bookAuthor formában, de más rendszerben inkább bookId-t küldenének). Siker esetén 201 Created, válaszban az új rekord (vagy 200 OK üzenettel). Hibák:
  + 400 Bad Request, ha valami hiányzik vagy invalid (pl. dátum formátum rossz).
  + 404 Not Found, ha a megadott könyv nem létezik.
  + 409 Conflict, ha a könyv már ki van kölcsönözve (ezt implementálhatjuk úgy, hogy ha már van Borrower arra a könyvre, nem enged újabbat).
* **DELETE /api/borrowers/{id}** – *Könyv visszavétele (kölcsönzés törlése).* A megadott azonosítójú kölcsönzési rekordot töröljük. Ezt tekintjük a visszavétel regisztrálásának. Siker: 204 No Content (a rekord törölve). Lehetséges hibák:
  + 404 Not Found, ha nem található ilyen rekord (pl. már törölték, vagy rossz id).
  + (Elvileg lehetne 400, ha mondjuk egy már visszavett dolgot próbál törölni – de nálunk törlés után nincs is meg, így 404 a helyzet.)
* *(Opcionális)* **PUT /api/borrowers/{id}** – Ezt nem terveztük, de lehetne pl. kölcsönzési határidő módosítására (hosszabbítás). Ha lenne ilyen, a body-ban új endDate, és a szerver frissítené a rekordot. Jelenleg ilyen funkció nincs implementálva, de a rendszer könnyen bővíthető lenne rá.

**Egyéb esetleges végpontok:**

* **GET /api/books/available** – Nem szerepel explicit a rendszerben, de logikailag a /api/books + /api/borrowers kombinációjából a frontenden meg lehet állapítani az elérhető könyveket. Ha a backend nyújtaná, akkor például szűrhetné azokat a könyveket, amikhez nincs aktív kölcsönzés. Implementálása: pl. bookRepository.findAllByIdNotIn(select book\_id from borrower).
* **GET /api/books/{id}/borrowers** – Hasonlóan, nem külön kezelt, de lehetne végpont lekérdezni egy könyv összes (aktív vagy korábbi) kölcsönzését. Mivel töröljük a rekordokat, a történeti adatokhoz nem jutunk hozzá jelenleg.
* **Health check / egyéb technikai végpontok:** Spring Boot Actuator nincs bekapcsolva, de fejlesztéskor hasznos lehet a /actuator/health ellenőrzésre. Ezt be lehet kapcsolni, és akkor nyomon követhető a rendszer állapota.

**Megjegyzések a használathoz:**

* Minden védett végpont JWT tokennel hívandó. A token megszerzéséhez előbb a /auth/login-t kell hívni. A token lejárati ideje alatt (pl. 1 óra) használható a többi művelet.
* A kliens alkalmazás (Angular) ezeket a végpontokat hívja a megfelelő helyeken. Pl. bejelentkezésnél a /auth/login, könyvlista frissítésnél a /books, kölcsönzés felvitelekor a /borrowers (POST), visszavételnél a /borrowers (DELETE).
* Az API tervezés RESTful elveket követ: főneves erőforrás URL-ek (books, borrowers), HTTP metódusokkal jelezve a műveletet (GET=lekérdezés, POST=létrehozás, PUT=teljes módosítás, PATCH=részleges módosítás, DELETE=törlés).

Az alábbi táblázat összefoglalja a legfontosabb végpontokat:

| **Végpont** | **Metódus** | **Leírás** | **Védettség (Auth)** |
| --- | --- | --- | --- |
| /api/auth/register | POST | Új felhasználó regisztráció | Nem (nyilvános) |
| /api/auth/login | POST | Bejelentkezés, JWT token szerzés | Nem (nyilvános) |
| /api/books | GET | Könyvek listázása | Igen (JWT) |
| /api/books | POST | Új könyv felvétele | Igen (JWT) – admin jellegű |
| /api/books/{id} | GET | Könyv részleteinek lekérése | Igen (JWT) |
| /api/books/{id} | DELETE | Könyv törlése | Igen (JWT) – admin jellegű |
| /api/borrowers | GET | Aktív kölcsönzések listázása | Igen (JWT) |
| /api/borrowers | POST | Új kölcsönzés rögzítése (kikölcsönzés) | Igen (JWT) |
| /api/borrowers/{id} | DELETE | Kölcsönzés törlése (visszavétel) | Igen (JWT) |

Ez a lista segít a fejlesztőknek és a klienseknek megérteni, hogyan kommunikálhatnak a szerverrel, és milyen műveleteket lehet végrehajtani a rendszerben.

**Példák – API válaszok és szolgáltatáshívások**

Ebben a fejezetben néhány konkrét példát mutatunk be a rendszer használatára, beleértve a tipikus JSON válaszokat és a front-end oldali szolgáltatáshívásokat.

**Példa JSON válaszok az API-tól**

* **Könyvek listájának lekérdezése (GET /api/books)** – Tegyük fel, hogy a rendszerben jelenleg 2 könyv van nyilvántartva, egyik sincs kikölcsönözve. A szerver válasza:

json

[

{

"id": 101,

"title": "A kis herceg",

"author": "Antoine de Saint-Exupéry"

},

{

"id": 102,

"title": "Pál utcai fiúk",

"author": "Molnár Ferenc"

}

]

Ez egy tömb, benne könyv objektumokkal. Jelenleg egyik könyv sincs jelölve kölcsönzöttnek (ez itt nem is látszik).

* **Aktív kölcsönzések lekérdezése (GET /api/borrowers)** – Tegyük fel, hogy az előző két könyvből az elsőt kikölcsönözték. A válasz:

json

[

{

"id": 15,

"name": "Nagy Anna",

"phone": "0612345678",

"email": "anna.nagy@example.com",

"startDate": "2025-07-01",

"endDate": "2025-07-10",

"bookTitle": "A kis herceg",

"bookAuthor": "Antoine de Saint-Exupéry"

}

]

Itt egyetlen aktív kölcsönzés van (id=15), Nagy Anna kikölcsönözte *A kis herceg* című könyvet 2025. július 1. és 2025. július 10. között. (Ebből következően a könyvlista lekérdezés után a kliens tudhatja, hogy a *A kis herceg* könyv épp kölcsönözve van, mert szerepel a borrowers listában.)

* **Bejelentkezés (POST /api/auth/login)** – Példa kérés:

json

{ "username": "admin", "password": "adminpass" }

Erre a szerver példa válasza:

json

{

"token": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJhZG1pbiIsImlhdCI6MTY4ODM1OTIyMCwiZXhwIjoxNjg4MzYyODIwfQ.LmP9v-F5H...",

"user": {

"id": 1,

"username": "admin"

}

}

A token egy hosszú base64 string (itt lerövidítve) és esetleg visszakapjuk a felhasználónév/id párost. Ezt a front-end eltárolja (pl. localStorage) és a tokennel dolgozik tovább.

* **Kölcsönzés létrehozása (POST /api/borrowers)** – Példa kérés:

json

{

"name": "Nagy Anna",

"phone": "0612345678",

"email": "anna.nagy@example.com",

"startDate": "2025-07-01",

"endDate": "2025-07-10",

"bookTitle": "A kis herceg",

"bookAuthor": "Antoine de Saint-Exupéry"

}

Erre a szerver válasza siker esetén lehet:

json

{

"id": 15,

"name": "Nagy Anna",

"phone": "0612345678",

"email": "anna.nagy@example.com",

"startDate": "2025-07-01",

"endDate": "2025-07-10",

"bookTitle": "A kis herceg",

"bookAuthor": "Antoine de Saint-Exupéry"

}

Vagy akár üres törzs, csak 201 státusz. Ebben a példában visszakaptuk a létrehozott kölcsönzés teljes adatait, immár egy generált ID-vel (15).

* **Könyv törlése (DELETE /api/books/102)** – Ha töröljük a 102-es ID-jű könyvet (Pál utcai fiúk), a szerver válasza: státusz 204 No Content, törzs nincs. Következmény: a GET /api/books hívásból az a könyv eltűnik.
* **Hiba példa – Könyv kétszeri kölcsönzés**: Ha megpróbáljuk kikölcsönözni *A kis herceg* könyvet, miközben Nagy Anna már kivitte és nem hozta vissza, akkor a POST /api/borrowers kérésre a szerver visszaadhat egy hibát, pl. 409 Conflict:

json

{ "error": "Book already borrowed by someone else" }

(A pontos hibaüzenet a szerver implementációjától függ. A lényeg, hogy a kliens ezt le tudja kezelni, és pl. figyelmezteti a felhasználót, hogy nem sikerült a kölcsönzés.)

**Példa front-end szolgáltatáshívások és használat**

A front-end Angular alkalmazás a HttpClient segítségével hívja az API-t. A kód áttekintéséhez íme néhány példa arról, hogyan néz ki a front-end oldalon a service hívás és a komponens reagálása:

* **AuthService – bejelentkezés hívása:**

typescript

// auth.service.ts

login(username: string, password: string): Observable<any> {

return this.http.post<any>('http://localhost:8080/api/auth/login', { username, password });

}

Ezt a LoginComponent használja:

typescript

this.authService.login(this.form.username, this.form.password).subscribe({

next: (response) => {

localStorage.setItem('token', response.token);

// esetleg user infók mentése

this.router.navigate(['/dashboard']);

},

error: (err) => {

this.errorMessage = 'Bejelentkezés sikertelen: hibás adatok vagy szerver hiba.';

}

});

Látható, hogy siker esetén elnavigál Dashboardra, hiba esetén kiírja az üzenetet. A token tárolása localStorage-ban történik.

* **BookService – könyvlista lekérése:**

typescript

// book.service.ts

getBooks(): Observable<Book[]> {

return this.http.get<Book[]>('http://localhost:8080/api/books');

}

Használat a Dashboard komponensben:

typescript

this.bookService.getBooks().subscribe({

next: (data) => { this.books = data; },

error: (err) => { console.error('Nem sikerült betölteni a könyvlistát', err); }

});

Itt this.books egy Book tömb, amit a HTML sablonban \*ngFor-ral listázunk. Ha error van (pl. token hibából), konzolra logoljuk.

* **BorrowerService – kölcsönzések és visszavétel:**  
  (Ha nincs külön BorrowerService, akkor is bemutatjuk egyet.)

typescript

// borrower.service.ts

getActiveLoans(): Observable<Borrower[]> {

return this.http.get<Borrower[]>('http://localhost:8080/api/borrowers');

}

returnBook(loanId: number): Observable<void> {

return this.http.delete<void>(`http://localhost:8080/api/borrowers/${loanId}`);

}

Használat a BorrowedBooks komponensben:

typescript

this.borrowerService.getActiveLoans().subscribe(loans => this.loans = loans);

onReturnBook(loan: Borrower) {

if(confirm(`${loan.bookTitle} visszavétele megtörtént?`)) {

this.borrowerService.returnBook(loan.id).subscribe({

next: () => {

// Távolítsuk el a visszavett kölcsönzést a listából

this.loans = this.loans.filter(l => l.id !== loan.id);

// Esetleg frissítsük a books listát is a Dashboardon keresztül vagy emitterrel

},

error: err => {

console.error('Hiba a visszavétel során', err);

alert('Hiba történt a visszavétel regisztrációja közben.');

}

});

}

}

Itt az onReturnBook egy gomb eseményhez van kötve. Megerősítés (confirm) után hívja a service-t. Siker esetén kitörli a lokális listából a kérdéses elemet, így az UI-ról eltűnik. (Robusztusabb megoldás: újra lekérdezni a friss listát a szervertől). Hiba esetén logolja és jelez a felhasználónak.

* **BorrowDialog – kölcsönzés mentése:**  
  Már korábban bemutattuk részleteiben, de összerakva:

typescript

onSubmit(): void {

if (this.borrower.name && this.borrower.startDate && this.borrower.endDate &&

this.borrower.phone && this.borrower.email) {

this.http.post('http://localhost:8080/api/borrowers', this.borrower).subscribe({

next: () => {

this.dialogRef.close(); // siker, bezárjuk a dialógust

},

error: err => {

console.error('Hiba történt a mentés során:', err);

alert('Nem sikerült menteni a kölcsönzést. Próbáld újra.');

}

});

}

}

Itt HttpClient közvetlen használatát látjuk. Alternatív megoldás, hogy a BorrowerService-be integráljuk a post hívást és itt azt hívjuk meg. A fenti közvetlen megoldás is működőképes, csak kevésbé szép szerkezetileg.

**Összefoglaló a példákból:** A front-end és back-end kommunikáció JSON formátumban zajlik. Az Angular HttpClient Observable-ket ad vissza, melyekre a komponensek feliratkoznak (subscribe). A JWT tokent a HttpInterceptor automatikusan hozzáadja a fejlécekhez, így a fejlesztőnek erre nem kell külön figyelnie minden hívásnál – elég egy helyen (AuthService login után) eltárolni a tokent. A példák rávilágítanak arra is, hogy a hibaágakat mindenhol érdemes kezelni (pl. console.error vagy felhasználói értesítés formájában), különben csendben elnyelődhetnek a problémák.

**Felhasználói felület bemutatása**

Végül tekintsük át a BiblioSys alkalmazás felhasználói felületét és annak elemeit. A UI egyszerű és funkcionális kialakítású, a hangsúlyt a könyvtári műveletek hatékonyságára helyezi. Az Angular és Angular Material kombinációja modern megjelenést biztosít, követve a Material Design irányelveket, ami egységes és letisztult kinézetet eredményez.

**Bejelentkező oldal (Login UI)**

A bejelentkező oldal egy középre igazított **login form**. A form két mezőt tartalmaz:

* **Felhasználónév** – egyszerű szövegmező.
* **Jelszó** – jelszómező (a karakterek beírás közben nem látszanak, type="password").

Valamint egy **"Bejelentkezés"** gomb. A form implementálja a szükséges validációt: mindkét mező kitöltése kötelező. Ha a felhasználó üresen próbálja meg elküldeni, piros keret vagy hibaszöveg jelzi a mezők alatt, hogy meg kell adni az adatokat. A “Bejelentkezés” gomb csak akkor aktív, ha mindkét mező ki van töltve (ezt Angular form bindinggal könnyű elérni, pl. [disabled]="!loginForm.form.valid").

Stílus szempontjából az oldal minimalista: a háttér semleges (fehér vagy világos), a bejelentkező doboz a képernyő közepén egy kártyán vagy panelen jelenik meg. A projekt név (BiblioSys) vagy egy logó szerepelhet a form felett, jelezve a rendszer nevét. Esetleg egy rövid üdvözlő üzenet: *"Lépjen be a könyvtári rendszerbe"*.

Sikertelen belépésnél a form alatt egy piros figyelmeztetés jelenik meg: pl. *"Hibás felhasználónév vagy jelszó."*. Sikeres belépés esetén az oldal átirányít a Dashboardra.

**Dashboard – főoldal könyvlista és kölcsönzések**

A Dashboard a bejelentkezés után a **fő kezelőfelület**. Az oldal tetején található egy navigációs sáv (navbar) vagy fejlécterület:

* Bal oldalon a rendszer neve (pl. *BiblioSys Library System*).
* Jobb oldalon a bejelentkezett felhasználó neve (felhasználónév), és egy **Kijelentkezés** gomb. A kijelentkezés gombra kattintva a felhasználó azonnal kiléptetésre kerül (token törlése) és visszakerül a bejelentkező oldalra.

A fejléc alatt két fő panel látható (vagy egymás alatt, ha egyszerűbben egymás utáni blokkban jelenítjük meg):

1. **Könyvek listája:** Táblázatos vagy kártyás formában jelennek meg a könyvek. Minden könyvnél a következő információk láthatók:
   * Cím (kiemelve félkövéren)
   * Szerző (dőlten vagy kisebb betűvel a cím alatt)
   * Opcionálisan további infó (pl. év, ha lenne)
   * Állapot / művelet: Ha a könyv **elérhető**, egy zöld vagy szürke “Elérhető” címke, valamint egy **"Kölcsönzés"** gomb jelenik meg. Ha a könyv **kikölcsönözve van**, akkor egy piros “Kölcsönözve” felirat, és mellette tipikusan jelöljük, hogy ki és meddig (pl. *"Kölcsönözte: Nagy Anna (2025-07-10-ig)"*). Ilyenkor a "Kölcsönzés" gomb helyett lehet **"Visszavétel"** gomb vagy link.

A megjelenítést implementálhatjuk táblázatként: oszlopok pl. [Cím – Szerző – Állapot – Művelet]. Vagy kártyákként: minden könyv egy card komponens, benne a szövegek és a megfelelő gomb.

A lista tetején lehet egy **“Új könyv hozzáadása”** gomb is (csak akkor, ha tervezünk UI-t erre). Ennek megnyomására egy form jönne elő (modal vagy külön oldal) a könyv adataival. Alapesetben, ha nincs rá UI, ez a gomb hiányozhat.

Ha sok könyv van, érdemes lapozó komponenst betenni alulra, vagy legalább görgethetővé tenni a listát.

1. **Kölcsönzött könyvek (aktív kölcsönzések) listája:** Ez lehet a könyvlista alatt, egy külön szekció címmel: *"Jelenleg kikölcsönzött könyvek"*. Itt minden sor egy aktív kölcsönzés, tartalmazva:
   * Könyv címe (esetleg szerző is)
   * Kölcsönvevő neve
   * Határidő dátuma (pirossal kiemelve, ha közel vagy túl van a határidőn a mai dátum szerint)
   * Művelet: **"Visszavétel"** gomb

A táblázat kinézete pl. [Könyv – Kölcsönvevő – Határidő – Művelet]. Ha nincs aktív kölcsönzés, egy üres állapot üzenetet jelenítünk meg itt (pl. *"Nincs kint lévő könyv."*).

Amikor a "Visszavétel" gombra kattint a könyvtáros, a sorból eltűnik az adott bejegyzés (és a könyvlista fentebb automatikusan frissül, a könyv státusza "Elérhető"-re vált). Ezt azonnali UI visszajelzésként is megoldhatjuk (optimista frissítés), vagy újratölthetjük a listákat a szerverről.

**Kölcsönzés indítása – BorrowDialog UI:** Amikor a könyvtáros egy könyv mellett a "Kölcsönzés" gombra kattint, egy modális dialógus jelenik meg a képernyő közepén. A háttér tartalom elhalványul (overlay). A dialógus tartalma:

* Cím: *"Kölcsönzés"* vagy *"Könyv kölcsönzése"*. Esetleg a könyv címét is belefoglalhatjuk: *"Kölcsönzés – A kis herceg"*.
* Űrlap mezők (mindegyik alatt kisebb betűvel a hibaüzenet, ha invalid):
  + **Név:** beviteli mező (pl. <input type="text" placeholder="Név" ...>). Kötelező.
  + **Kezdés dátuma:** dátum kiválasztó (<input type="date" ...>). Kötelező. Alapértelmezett értéke a mai nap.
  + **Vége dátuma:** dátum kiválasztó a határidőnek. Kötelező. Lehet akár alapból +2 hétre állítva.
  + **Könyv címe:** *nem szerkeszthető* mező (disabled input), ami a kiválasztott könyv címét mutatja.
  + **Könyv szerzője:** hasonlóan egy disabled input a szerzővel.
  + **Telefonszám:** szövegmező a telefonszámnak (patternnel ellenőrizhető, hogy számokat tartalmazzon). Kötelező.
  + **Email:** email típusú mező az email címnek. Kötelező, és valós email formátumot igényel.
* Gombok a dialógus alján:
  + **OK / Mentés:** (Angular Material esetén pl. egy primer színű mat-button). Erre kattintva indul a mentés folyamata. Ha valamelyik mező hiányzik, a gomb tiltva van, vagy a kattintás nem zárja be a dialógust, hanem a mezőknél megjelennek a hibajelzések.
  + **Mégsem:** (mat-button, piros vagy default). Erre kattintva a dialógus bezárul anélkül, hogy mentettünk volna (a form adatait elveti).

A dialógus stílusa a Material Design keretein belül egyszerű: fehér hátterű kártya, lekerekített sarkokkal, max szélessége ~400px, hogy jól kitöltse a képernyő közepét anélkül, hogy túl nagy lenne. Mobilon is használható legyen (reszponzív), a mezők összecsúsznak több sorba.

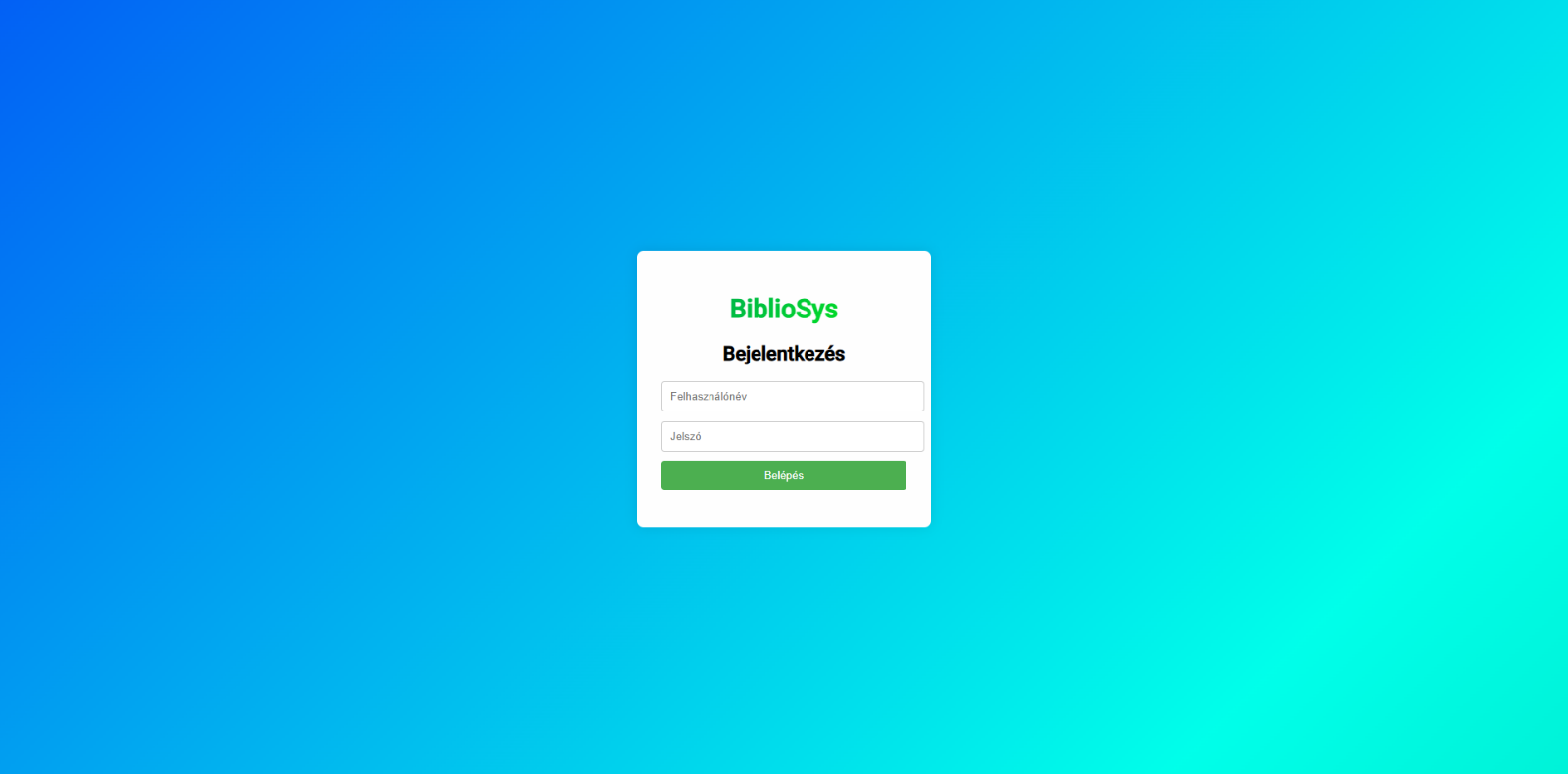
**Felület interakció összefoglalva:**

* A könyvtáros belép az oldalon, majd a Dashboardon látja a könyveket és kölcsönzéseket.
* Ha ki akar adni egy könyvet, megnyomja a könyv sorában a *Kölcsönzés* gombot, kitölti az adatokat a felugró dialógusban, és lementi. Ezután látja, hogy a könyv sorában megjelent a “kikölcsönözve” státusz, és a kölcsönzések listájában is megjelenik egy új bejegyzés.
* Ha visszahoztak egy könyvet, a kölcsönzések listájában a megfelelő sorban rányom a *Visszavétel* gombra. A sor eltűnik a listából, és a fenti könyvlistában a könyv újra kölcsönözhető státuszú lesz.
* Ha új könyvet akar felvinni (ha ez támogatott UI-n), akkor azt megteheti egy “+ Könyv hozzáadása” funkcióval, ami egy hasonló dialógust vagy külön oldalt nyit, ahol megadja a könyv címét, szerzőjét stb., majd mentés után az új könyv megjelenik a listában.

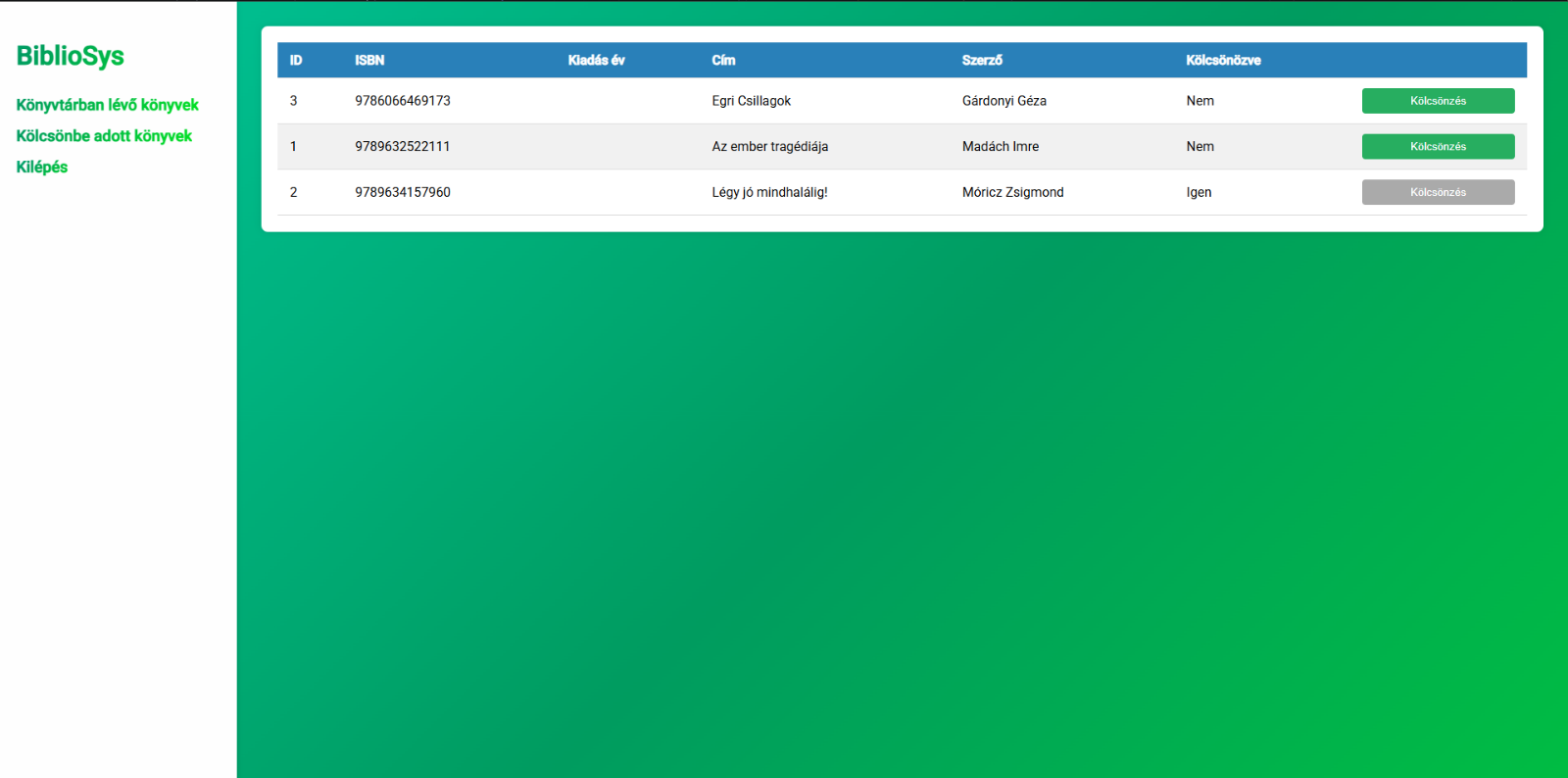
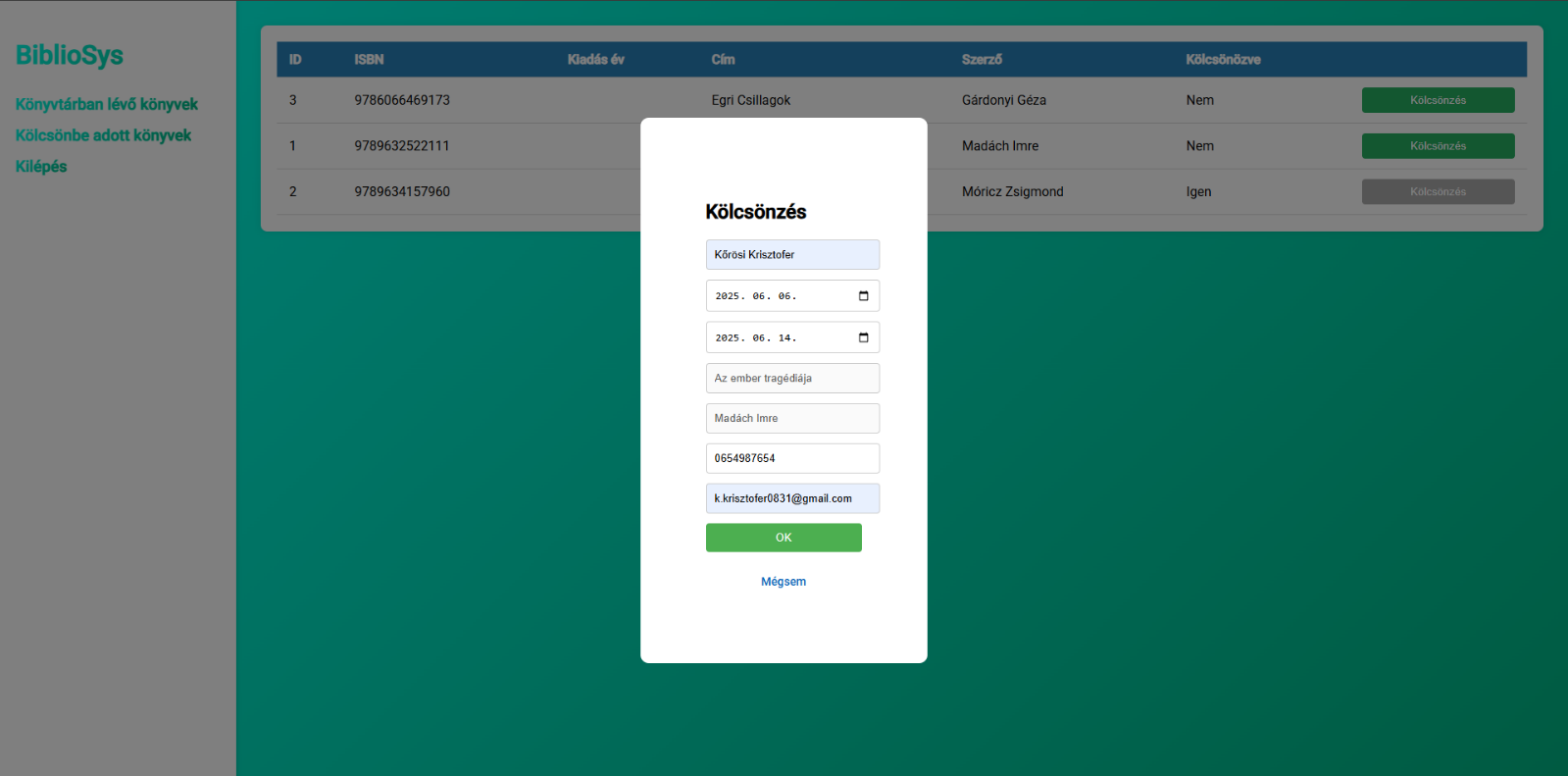
**Reszponzív design:** Az Angular Material grid és flex layout segít abban, hogy az alkalmazás kisebb képernyőkön (pl. tableten) is használható legyen. A táblázatok sorbarendeződhetnek egymás alá, a dialógus kitölti a képernyő nagy részét mobilon stb. A betűméretek és gombméretek elég nagyok ahhoz, hogy érintőképernyőn is könnyen használhatók legyenek.

**Színséma és branding:** A projekt nem feltétlenül rendelkezik egyedi arculattal, de használhatja a Material alapértelmezett színeit: pl. kék árnyalatok kiemelőszínnek. A fontos műveletek (mentés, kölcsönzés) kiemelt (primer) színnel jelennek meg, a veszélyes műveletek (törlés, visszavétel – bár az nem “veszélyes” értelemben, de kritikus művelet) figyelmeztető színnel (pl. piros) jelennek meg gombként.

**Felhasználói élmény:** A felület igyekszik egyszerű maradni. Az információk jól strukturáltak, a könyvtáros számára ismerős kifejezések vannak használva (“Kölcsönzés”, “Visszavétel”, “Határidő” stb.). A rendszer portfólió jellegéből adódóan a cél nem egy extravagáns design, hanem az, hogy bemutassa a fejlesztő képességét egy teljes stack alkalmazás létrehozására. Ennek megfelelően a UI letisztultsága és a funkcionalitás teljessége a legfontosabb.

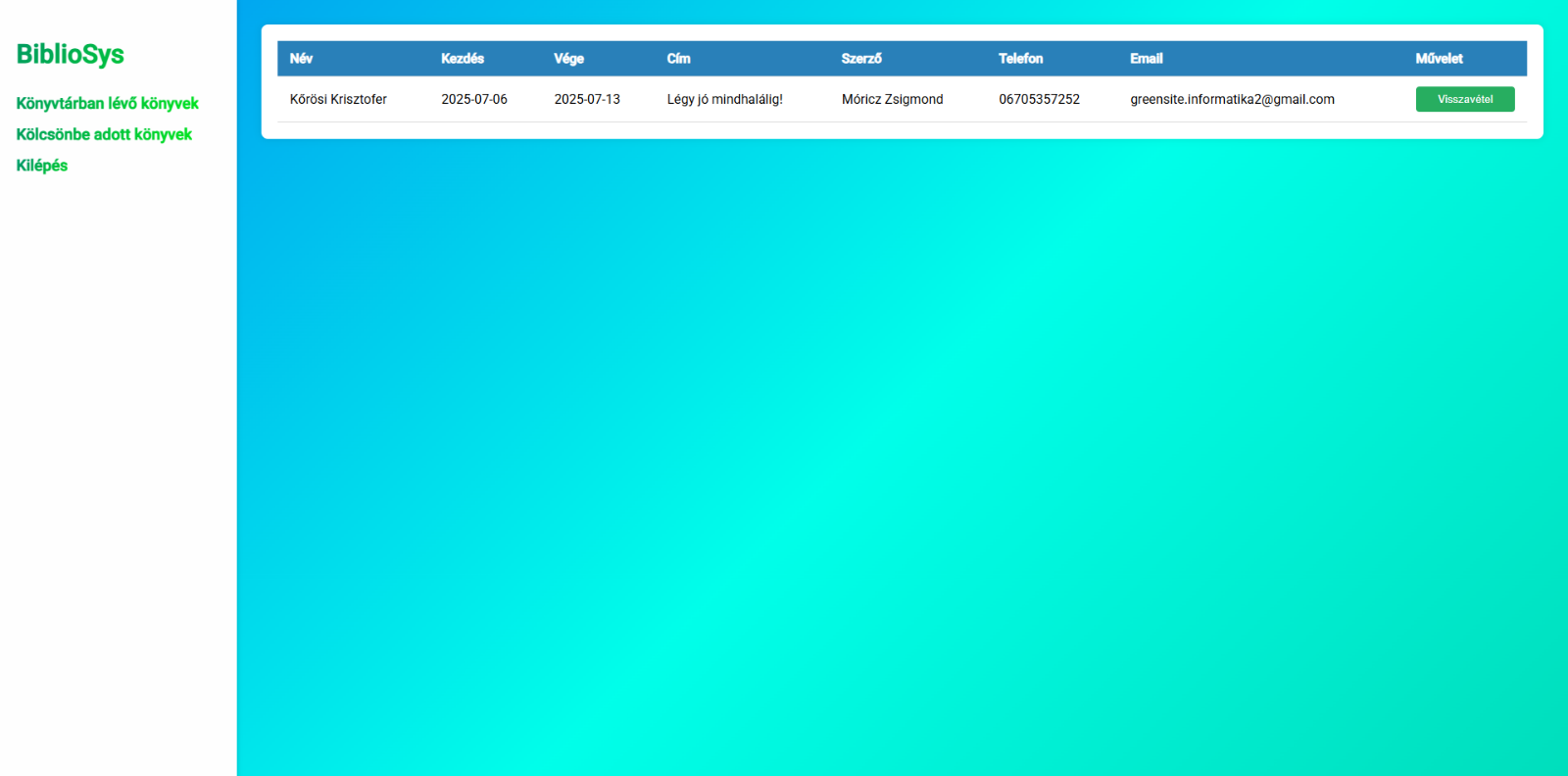
****Képernyőképek:**  
*(Itt a dokumentáció Word változatában érdemes valódi képernyőképeket elhelyezni az alkalmazásból. Például:*

Login képernyő

.

Kölcsönzés dialógus

Dashboard képernyő



Kölcsönzöttek listázása

**Zárszó:** A BiblioSys fejlesztői dokumentációja remélhetőleg átfogó képet adott a rendszer felépítéséről és működéséről. A kézikönyv célja, hogy segítse a jövőbeni fejlesztőket a projekt megértésében és továbbfejlesztésében, valamint bemutassa a projektet szakmai portfólió részeként az érdeklődőknek. A rendszer további bővítési lehetőségei (pl. értesítések küldése, késedelmi díjak számítása, könyvkereső funkció stb.) nyitva állnak, és a most lefektetett alapokra építve viszonylag könnyen megvalósíthatók a jövőben. Köszönjük a figyelmet, és sikeres fejlesztést kívánunk a BiblioSys rendszerrel!