

**D O K U M E N T Á C I Ó**

**Vizsgaremek**

**SZOFTVERFEJLESZTŐ ÉS -TESZTELŐ TECHNIKUS SZAKMA**

Tartalomjegyzék

[I. Bevezetés 4](#_Toc208954631)

[II. Fejlesztői dokumentáció 6](#_Toc208954632)

[II.I. Felhasznált technológiák 6](#_Toc208954633)

[II.II. Fejlesztői környezet 7](#_Toc208954634)

[II.II.I. Hardver 7](#_Toc208954635)

[II.II.II. Szoftver 7](#_Toc208954636)

[II.III. Backend 8](#_Toc208954637)

[II.III.I. Adatbázis és adatszerkezet 8](#_Toc208954638)

[II.III.II. Kiemelt kódrészletek 11](#_Toc208954639)

[II.III.III. Végpontok 14](#_Toc208954640)

[II.IV. Frontend 17](#_Toc208954641)

[II.IV.I. Komponensek, szervizek és útvonalak 17](#_Toc208954642)

[II.V. Fejlesztési lehetőségek 20](#_Toc208954643)

[III. Felhasználói dokumentáció 22](#_Toc208954644)

[III.I. Bemutató 22](#_Toc208954645)

[III.II. Rendszerkövetelmények 22](#_Toc208954646)

[III.III. Telepítés és indítás 23](#_Toc208954647)

[III.IV. Egyes oldalak és funkciók 23](#_Toc208954648)

[III.IV.I. Regisztráció 23](#_Toc208954649)

[III.IV.II. Bejelentkezés 24](#_Toc208954650)

[III.IV.III. Funkciók felhasználóknak 24](#_Toc208954651)

[III.IV.IV. Funkciók adminisztrátoroknak 26](#_Toc208954652)

[III.V. Biztonság 26](#_Toc208954653)

[IV. Tesztelés 28](#_Toc208954654)

[IV.I. Dinamikus tesztek - backend és adatbázis 28](#_Toc208954655)

[IV.II. E2E tesztek - Selenium 31](#_Toc208954656)

[IV.III. Manuális tesztek - frontend 34](#_Toc208954657)

[V. Összefoglalás 38](#_Toc208954658)

# I. Bevezetés

A projektünk célja egy webes alkalmazás elkészítése volt, amely egy autókölcsönző rendszer működését szimulálja. Az elkészült program lehetőséget biztosít a felhasználók számára, hogy online felületen keresztül gépjárműveket kölcsönözzenek meghatározott időszakra, míg az adminisztrátorok kezelhetik a flottát, azaz felvihetik, módosíthatják vagy törölhetik a rendelkezésre álló autókat. Az alkalmazás három fő szerepkört különböztet meg: az adminisztrátort és a felhasználót, illetve a bejelentkezés nélküli nézelődőt.

A projektötlet kiválasztásában döntő szerepet játszott a csapatunk mindennapi tapasztalata. Budapesten élünk és dolgozunk, ahol a városi közlekedésben egyre elterjedtebbek a különböző autómegosztó és autókölcsönző szolgáltatások. Szinte mindannyian rendszeres felhasználói vagyunk ezeknek az alkalmazásoknak, legyen szó akár egy rövid városi út megtételéről, akár hosszabb vidéki utazásról. Az, hogy személyes élményeink is kötődnek a témához, különösen motiválóvá tette számunkra a feladatot, hiszen így nemcsak programozási gyakorlatot szereztünk, hanem egy olyan alkalmazást készíthettünk, amelynek működését saját szemszögünkből is jól értettük.

Másik fontos szempont volt, hogy a feladat összetettsége megfelelő kihívást jelentett a tanulmányaink során megszerzett ismeretek gyakorlati alkalmazásához. Az autókölcsönző rendszer modellje magában foglalja mindazokat az alapvető funkciókat, amelyek egy modern webes alkalmazásban tipikusan előfordulnak: adatbázis-kezelés, autentikáció és autorizáció, adminisztrációs felület, valamint a felhasználói interakciók. Így a projekt alkalmas volt arra, hogy átfogó képet adjon egy teljes szoftverrendszer megvalósításáról.

Továbbá úgy gondoltuk, hogy egy ilyen alkalmazás fejlesztése a jövőbeli pályánk szempontjából is hasznos tapasztalatot nyújt. Az informatikai iparban egyre nagyobb az igény a skálázható, biztonságos és felhasználóbarát online szolgáltatásokra. Egy autókölcsönző rendszer kidolgozása során megtanultuk, hogyan kell egy projektet a nulláról felépíteni, hogyan lehet a csapaton belül feladatokat megosztani, valamint hogyan lehet a különböző komponenseket integrálni.

Az elkészült rendszer főbb funkciói:

* **Felhasználói funkciók**: regisztráció, bejelentkezés, autók listázása, autóbérlés adott időszakra
* **Adminisztrátori funkciók**: új autók felvitele, meglévők módosítása vagy törlése
* **Foglalási logika**: a bérlések létrehozása, státuszuk nyomon követése (például függőben lévő, megerősített vagy lemondott állapot)

A fejlesztés során a Java Spring keretrendszert alkalmaztuk, mivel a modern webes szolgáltatások egyik legelterjedtebb és legmegbízhatóbb megoldását nyújtja. A REST architektúrát követve különböző végpontokat (endpointokat) hoztunk létre, amelyek a kliens és a szerver közötti kommunikációt biztosítják. A felhasználók biztonságos belépését és az adatok védelmét a bejelentkezési és jogosultságkezelési funkciók látják el, amelyek JWT token alapú hitelesítéssel valósulnak meg.

A rendszer kialakításánál törekedtünk a valós életből vett működés modellezésére. Ez azt jelenti, hogy a felhasználók nem közvetlenül az adatbázist módosítják, hanem a megfelelő kontrollereken keresztül végzik a műveleteket, míg a háttérben a szolgáltatásréteg (service) gondoskodik az üzleti logikáról, például annak ellenőrzéséről, hogy egy autó adott időszakban valóban szabad-e. A technikai megvalósításban szerepet kaptak olyan alapfogalmak, mint a DTO-k (Data Transfer Object), a repository mintázat, valamint a Spring Security használata a jogosultságok ellenőrzésére.

Összességében tehát a témaválasztásunkat egyszerre indokolta a személyes érdeklődésünk, a gyakorlati hasznosulás, valamint a szakmai fejlődés lehetősége. Az autókölcsönző alkalmazás elkészítése számunkra nemcsak egy vizsga feladat teljesítését jelentette, hanem egy olyan projektet is, amely közel áll a mindennapjainkhoz, és amelynek megvalósítása során értékes tapasztalatokat szereztünk a szoftverfejlesztés gyakorlatáról.

# II. Fejlesztői dokumentáció

## II.I. Felhasznált technológiák

Az alkalmazás adatainak kezelésére **PostgreSQL** adatbázist használtunk, amely megbízhatósága, skálázhatósága és gazdag funkciókészlete miatt ipari szabványnak számít. Előnye, hogy jól kezeli a tranzakciókat, támogatja a relációs integritást, és összetett lekérdezéseket is hatékonyan futtat. A fejlesztés során a PostgreSQL-t **Docker** konténerben futtattuk, amely nagyban megkönnyítette a környezet telepítését és újraindítását. Ezáltal minden fejlesztő ugyanazzal az adatbázis verzióval tudott dolgozni, elkerülve a lokális különbségekből adódó hibákat.

Az adatbázis sémaváltozásait a **Flyway** eszközzel kezeltük. A Flyway lehetővé tette a verziózott migrációkat, így az adatbázis struktúrája következetesen frissült minden környezetben. Ez különösen fontos volt, mivel a csapat több tagja is párhuzamosan dolgozott, és így biztosítani tudtuk, hogy az adatbázis mindig ugyanabban az állapotban legyen mindenkinél.

Az adateléréshez **JPA Hibernate**-et alkalmaztunk, amely lehetővé tette, hogy az adatbázis táblákat Java osztályokként kezeljük, és így a lekérdezések nagy részét egyszerűen, a nyelv szintaxisán belül tudtuk megfogalmazni.

A backend fejlesztéshez **Java 17** nyelvet választottunk, amely hosszú távon támogatott LTS verzió. Ez biztosította, hogy a fejlesztés során stabil, biztonságos és korszerű nyelvi környezetben dolgozhattunk. A keretrendszerként a **Spring Boot** szolgált, aminek nagy előnye, hogy rengeteg előre konfigurált modult tartalmaz, így gyorsan lehet működő alkalmazást építeni, és könnyen integrálható más könyvtárakkal.

A biztonsági funkciókat a **Spring Security** keretrendszer biztosította. Ez tette lehetővé a felhasználók regisztrációját, bejelentkezését és a jogosultságok kezelését. A rendszerben a hitelesítés **JWT tokenekkel** történt, ami biztosította a kliens és a szerver közötti biztonságos kommunikációt, valamint lehetővé tette, hogy a felhasználói munkameneteket könnyen lehessen azonosítani.

A felhasználói felület fejlesztésére a **React** könyvtárat használtuk. A választás fő oka az volt, hogy a React ipari szinten is az egyik legelterjedtebb frontend technológia, amely jól támogatja az újrafelhasználható komponensek fejlesztését. Ezáltal könnyen tudtunk egységes és dinamikus felületet kialakítani, amely reszponzív és gyorsan reagál a felhasználói interakciókra. A stílusozásban a **Tailwind** segített.

Ezzel a tanult háromrétegű architektúrát hoztuk létre: prezentációs / üzleti logikai / perzisztencia réteg elkülönülése.

## II.II. Fejlesztői környezet

### II.II.I. Hardver

Core i5 vagy i7 típusú processzorokkal rendelkeztek, amelyek kellően nagy számítási kapacitást nyújtottak a program fordításához és futtatásához. A fejlesztés során fontos szempont volt a megfelelő memória, ezért minden gép legalább 16 GB, de jellemzően 32 GB RAM-mal volt felszerelve, amely elegendőnek bizonyult a párhuzamosan futó fejlesztőeszközök, adatbázisok és konténerek használatához. A háttértár SSD alapú volt, minimum 512 GB kapacitással, ami gyors fájlhozzáférést és rövid buildidőt biztosított. Ez különösen a Docker konténerek és a Node.js alapú front-end fordításánál jelentett előnyt.

A hardverkörnyezet teljes mértékben megfelelt a fejlesztési igényeinknek, és a projekt során nem merült fel erőforráshiány miatti akadály. A biztosított gépek tehát nemcsak az alapvető fejlesztési feladatokra voltak alkalmasak, hanem a komplexebb tesztelési környezetek futtatását is zökkenőmentesen támogatták.

### II.II.II. Szoftver

A fejlesztői környezet operációs rendszere Windows volt, amely megfelelő támogatást biztosított a szükséges eszközök telepítéséhez és futtatásához. A backend fejlesztéshez az IntelliJ IDEA Ultimate verzióját használtuk, amely fejlett támogatást nyújtott a Spring keretrendszerhez és a Java nyelvhez. A konténerizációhoz a Docker Desktop szoftvert alkalmaztuk, amely lehetővé tette az adatbázis és a kapcsolódó szolgáltatások izolált futtatását. A frontend fejlesztéshez Node.js és npm környezetet telepítettünk, amely biztosította a React és Tailwind alapú alkalmazás buildelését és futtatását. A választott szoftverek mindegyike ipari szabvány, így nemcsak a fejlesztési folyamatot támogatták, hanem gyakorlati tapasztalatot is adtak a modern vállalati eszközök használatában.

## II.III. Backend

### II.III.I. Adatbázis és adatszerkezet

*Adatbázis futtatása*

Az alkalmazás háttéradatbázisa a népszerű PostgreSQL rendszerre épül, amely széles körben elterjedt az üzleti alkalmazások és webfejlesztés területén. Az adatbázis létrehozásához a modern konténerizációs technológia, a Docker volt segítségünkre, amely megkönnyíti a fejlesztési és üzemeltetési folyamatokat. Azért szerencsés a Dockerben futtatni az adatbázist, mert egységes fejlesztői környezetet biztosít mindenkinél (azonos verziók és config-ok), gyors és tiszta telepítés, izoláció a local rendszertől, egyszerű adatmegőrzés (volume), könnyű újraindítás és visszaállítás, hibák esetén törölhető konténer (de nem vész el adat), egyszerű elérés (port-on keresztül).

Az adatbázis létrehozását és konfigurálását az alábbiakban bemutatott postgres-compose.yml fájl segítségével valósítottuk meg, amely tartalmazza a szükséges beállításokat és a szolgáltatások automatizált indítását.

version: '3.9'  
services:  
*# adatbázis és alapvető paraméterei  
# itt érjük el: jdbc:postgresql://localhost:25431/car  
# lent megadott user, password és db-névvel* db:  
 image: postgres:16  
 environment:  
 POSTGRES\_USER: postgres  
 POSTGRES\_PASSWORD: postgres  
 POSTGRES\_DB: postgres  
 ports:  
 - "25431:5432"  
 volumes:  
 - pgdata:/var/lib/postgresql/data  
*# adatbázis kezelő grafikus felület és alapvető paraméterei  
# itt érjük el böngészőből: http://localhost:5060/  
# lent megadott e-mail és jelszóval lehet bejelentkezni  
# pgadmin:  
# image: dpage/pgadmin4  
# environment:  
# PGADMIN\_DEFAULT\_EMAIL: admin@admin.com  
# PGADMIN\_DEFAULT\_PASSWORD: admin  
# ports:  
# - "5049:80"*volumes:  
 pgdata:

A postgres-compose.yml két szolgáltatást indít: a PostgreSQL adatbázist és a pgAdmin felületet. Az adatbázis a postgres:16 image-ből fut, és a host gépről a localhost:26432 címen érhető el. A belépéshez a felhasználónév posgres, a jelszó postgres, az adatbázis neve pedig szintén postgres. A pgdata nevű volume biztosítja, hogy az adatok konténer újraindítás után is megmaradjanak. A pgAdmin egy böngészőből elérhető grafikus kezelőfelületet nyújt az adatbázishoz, amely a http://localhost:5049 címen érhető el. Ide az admin@example.com felhasználónévvel és az admin jelszóval lehet belépni. A két konténer közös hálózatban fut, így a backend közvetlenül a db:5432 címen is elérheti a PostgreSQL-t. Ez a beállítás egyszerűvé és egységessé teszi a fejlesztést, mivel minden csapattag azonos konfigurációval dolgozik.

A Dockeres DB-vel ugyanazt a verziót és configot futtatjátok fejlesztésen, mint tesztben. A Flyway migrációk induláskor automatikusan szinkronizálják a sémát. Az adat lokálisan tartós, de egyszerűen resetelhető, ha tiszta állapotra van szükség. A port-kitettség miatt az IDE-ből és a tesztekből is stabilan elérhető az adatbázis.

*A projekt futtatása*

A projekt build- és függőségkezelését a Maven végezte (3.10.1-es verzió). A Spring Boot beépített szerverrel érkezik (pl. Tomcat, Jetty), így nincs szükség külön telepített alkalmazásszerverre. Az alapértelmezett szerver testre szabható az application.properties fájlban vagy konfigurációs osztályokkal.

Ez alapján a projekt futtatásának a menete:

1. Dockerben elindítjuk az adatbázist
2. Maven clean install parancs futtatása
3. backend indítása: CarRenterApiApplication elindítása (SpringBootApplication)
4. frontend mappából terminal-ból elindítani a frontendet: npm run dev
5. a webalkalmazás elérhető a <http://localhost:5173/> linken

Tekintve, hogy webes alkalmazást hoztunk létre, így biztosítani tudjuk a platfrom függetlenséget és a központi adatkezelést. Mobilos és számítógépes nézetet is kezelünk (reszponzív felületek), valamint a JWT tokenek és a Spring Security segítségével az authentikáció is könnyen és gyorsan kezelhető.

*Adatszerkezet*

A projekt indításakor az adatbázis táblák betöltését és a kezdő adatok betöltését két sql script végzi: az init sql az adatbázis táblákat tartalmazza, míg a seed sql alap rekordokkal tölti fel Természetesen ez csupán a könnyebb tesztelhetőség miatt van – éles környezetben user adatokkal nem töltenénk fel a táblákat. Viszont éles környezetben is előfordulhat induló rekordok felvétele: pl. a webalkalmazás adatbázisának feltöltve az induló bérelhető autó állománnyal vagy admin jogokkal rendelkező felhasználók hozzáadása automatikusan.

Táblák létrehozásáért felelős sql:

CREATE EXTENSION IF NOT EXISTS "uuid-ossp";  
  
  
CREATE TABLE users (  
 id UUID PRIMARY KEY DEFAULT uuid\_generate\_v4(),  
 email VARCHAR(255) UNIQUE NOT NULL,  
 password\_hash VARCHAR(255) NOT NULL,  
 role VARCHAR(32) NOT NULL  
);  
  
CREATE TABLE cars (  
 id UUID PRIMARY KEY DEFAULT uuid\_generate\_v4(),  
 licensePlate VARCHAR(120) NOT NULL,  
 type VARCHAR(60) NOT NULL,  
 numberOfPassangers INT,  
 fuelType VARCHAR(120) not null  
);  
  
CREATE TABLE rentals (  
 id UUID PRIMARY KEY DEFAULT uuid\_generate\_v4(),  
 car\_id UUID NOT NULL REFERENCES cars(id) ON DELETE CASCADE,  
 user\_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,  
 start\_ts TIMESTAMP WITHOUT TIME ZONE NOT NULL,  
 end\_ts TIMESTAMP WITHOUT TIME ZONE NOT NULL,  
 status VARCHAR(32) NOT NULL  
);  
  
CREATE INDEX idx\_rentals\_car\_time ON rentals(car\_id, start\_ts, end\_ts);

Tábla rekordok feltöltéséért felelős sql:

INSERT INTO users(email, password\_hash, role)  
VALUES ('admin@example.com', '$2a$10$Ge8uuN.HE4J4O0hEqkLLgu1x2VwskwjwphRhVTyjcJgbiT95t5I0C', 'ADMIN');   
  
  
INSERT INTO cars(licensePlate, type, numberOfPassangers, fuelType) VALUES  
 ('ABC-123', 'Audi', 5, 'PETROL'),  
 ('DEF-456', 'BMW', 4, 'PETROL'),  
 ('GHI-789', 'Mercedes', 7, 'DIESEL');  
INSERT INTO users(email, password\_hash, role)  
VALUES ('user@example.com', '$2a$10$8NaAL4H4iG3c1F8iDdvuU.h71m7RhjxEiFjKV7AMACIIM8LSnIZ1W', 'USER');  
-- jelszó: User123!

### II.III.II. Kiemelt kódrészletek

Két megoldást szeretnénk kiemelni a kódbázisból.

*Jelszó és biztonság*

A jelszavakat természetesen csak azért kommenteltük be a betöltő init sql-be, mert ez egy vizsga applikáció.

Viszont már jól látszik a kódolt jelszó tárolásán, hogy az adatbiztonságot szem előtt tartottuk. A uuid-ossp kiterjesztés engedélyezése biztosítja a szerveroldali UUID generálást (uuid\_generate\_v4()), így minden elsődleges kulcs ütközésmentesen, adatbázisban keletkezik. Ezen kívül egyediséget kényszerítünk ki adatbázis szinten az e-mailekre, valamint sem a szerepkör, sem az e-mail, sem a jelszó nem lehet üres természetesen.

A rendszer belépési folyamata vezérelt, a POST /api/auth/login végpont kizárólag a felhasználó hitelesítését végzi, majd siker esetén **JWT hozzáférési tokent** ad vissza AuthResponse formában. A token a későbbi kérésekhez a Authorization: Bearer <token> fejlécben kerül felhasználásra, így az üzleti végpontokon nem kell újra elküldeni a felhasználó jelszavát. A regisztráció POST /api/auth/register elkülönítve működik, siker esetén 201 Created és **üres törzs** a válasz, ami minimalizálja a kimenő személyes adatok mennyiségét. Duplikált e-mail esetén a szolgáltatás 409 Conflict státuszkódot ad vissza egy rövid üzenettel (Email already exists). A tesztek azt is ellenőrzik, hogy sikeres regisztrációkor nincs válaszbody, így a rendszer nem ad vissza érzékeny mezőket, például hash-eket vagy szerepköröket.

@RestController  
@RequestMapping("/api/auth")  
public class AuthController {  
 private final AuthService authService;  
 private final JwtService jwtService;  
  
  
 public AuthController(AuthService authService, JwtService jwtService) {  
 this.authService = authService; this.jwtService = jwtService; }  
  
  
 @PostMapping("/register")  
 public ResponseEntity<?> register(@RequestBody RegisterRequest req) {  
 try {  
 User u = authService.register(req.email(), req.password());  
 return ResponseEntity.*status*(HttpStatus.*CREATED*).build();  
 } catch (IllegalArgumentException ex) {  
 return ResponseEntity.*status*(HttpStatus.*CONFLICT*).body(ex.getMessage());  
 }  
 }  
  
  
 @PostMapping("/login")  
 public ResponseEntity<AuthResponse> login(@RequestBody LoginRequest req) {  
 User u = authService.authenticate(req.email(), req.password());  
 return ResponseEntity.*ok*(new AuthResponse(jwtService.generate(u)));  
 }  
}

A fenti bejelentkezést lefedtük tesztekkel is, erről a tesztelés fejezetben írunk bővebben.

Másik kiemelendő kódrészlet ahhoz kapcsolódik, hogy különösen fontos volt a megfelelő időütközés-kezelést, hiszen ez az egyik legfontosabb pontja a bérlésnek. Vegyük végig, hogy hogyan fut végig egy foglalás a backenden és hogyan történik meg az ellenőrzés.

Amikor egy felhasználó szeretne lefoglalni egy járművet egy adott időszakra, a backendben található RentalService felelős azért, hogy ellenőrizze a foglalás érvényességét. Ez több lépésből áll:

Felhasználó azonosítása:

A bejelentkezett felhasználó JWT tokennel hitelesíti magát. A tokenből a rendszer kiolvassa az e-mail címet, majd az adatbázisból lekéri a hozzá tartozó felhasználói azonosítót. Így biztosítható, hogy minden foglalás egy konkrét, regisztrált felhasználóhoz kötődjön.

Adatellenőrzés:

A bérlés során kötelező megadni az autó azonosítóját, valamint a bérlés kezdő- és végidőpontját. A rendszer először validálja, hogy a dátumok helyes sorrendben vannak-e (a kezdőidőpont nem lehet későbbi a végidőpontnál). Ezek a frontenden is le vannak validálva mezőbevitelnél, de biztonság kedvéért itt is elvégezzük az ellenőrzést.

Időütközések kezelése:

A kulcslépés a konfliktus-ellenőrzés. Egy autó egyszerre csak egy felhasználónál lehet, ezért a rendszer minden új bérlés létrehozásakor lekérdezi, hogy az adott járműhöz tartozik-e már meglévő foglalás, amely átfedi a megadott intervallumot.

Az időütközés ellenőrzésének logikája:

* Az új foglalás rendelkezik-e kezdeti dátummal (végdátum nem kötelező).
* Ha az új foglalás kezdete beleesik egy már meglévő foglalás időszakába, ütközés történik.
* Ha az új foglalás vége beleesik egy már meglévő foglalás időszakába, ütközés történik.
* Ha az új foglalás teljesen körbefogja a meglévő foglalást, szintén ütközés van.

Például:

* Meglévő foglalás: 2025.09.02. 10:00 – 14:00
* Új foglalás: 2025.09.02. 13:00 – 15:00 → ütközés (az átfedés 13:00–14:00).
* Új foglalás: 2025.09.02. 14:00 – 16:00 → nem ütközik, mert az egyik pontosan a másik végén kezdődik.

public Rental create(Rental b) {  
 if (b.getEnd() == null) {  
 throw new IllegalArgumentException("Start date is mandatory.");  
 }  
 if (b.getEnd() != null && !b.getStart().isBefore(b.getEnd())) {  
 throw new IllegalArgumentException("End date is before start date.");  
 }  
 if (!rentalRepo.findOverlaps(b.getCarId(), b.getStart(), b.getEnd()).isEmpty()) {  
 throw new IllegalArgumentException("Overlapping renting");  
 }  
 b.setStatus("CONFIRMED");  
 return rentalRepo.save(b);  
}

Ha a rendszer ütközést talál, IllegalArgumentException dobódik, amelyet a controller réteg lekezel, és HTTP 409 Conflict választ küld a kliensnek. Ez a visszajelzés a felhasználó számára egyértelművé teszi, hogy az adott autó a választott időszakban nem érhető el.

Bérlés mentése:

Amennyiben nem találtunk időütközést, a rendszer elmenti a foglalást az adatbázisba PENDING státusszal. Később lehetőség van a foglalás státuszát módosítani (például megerősíteni vagy lemondani), ami szintén az alkalmazás fontos funkciója.

Az elkészült rendszer tehát nem csupán egy egyszerű adatbázis-kezelő alkalmazás, hanem egy valós életből vett folyamatot modellez. A felhasználók biztonságosan tudnak autót foglalni, az adminisztrátorok menedzselhetik a flottát, miközben a háttérben a rendszer gondoskodik az adatbiztonságról és az üzleti logika érvényesítéséről.

A projekt megvalósítása során különösen értékes tapasztalatot szereztünk a konfliktuskezelés és az időalapú foglalások logikájának implementálásában.

### II.III.III. Végpontok

A rendszer a JWT alapú hitelesítésre épít: a bejelentkezés során kapott token minden védett végpontnál szükséges az Authorization fejlécben. A method level security a kritikus műveleteket – mint az autók létrehozása, módosítása, törlése – ADMIN szerepkörhöz köti, így a felhasználói tokenekkel ezek nem hívhatók. A bérlés létrehozása hitelesítést igényel, a felhasználó identitása a Spring Security kontextusából származik, így a foglalás mindig a bejelentkezett személyhez kapcsolódik, ami audit és jogosultsági szempontból is tiszta működést biztosít.

**AuthController – /api/auth**

Regisztráció 🡪 POST /api/auth/register

A regisztrációs végpont új felhasználói fiók létrehozására szolgál. A kliens egy RegisterRequest törzset küld e-mail és jelszó mezőkkel, a háttérben az AuthService.register végzi az üzleti ellenőrzéseket és a perzisztálást. Siker esetén 201 Created státuszkód érkezik, törzs nélkül, ami egyértelmű jelzés a kliensnek, hogy a fiók elkészült és a továbbiakban a bejelentkezés hívható. Ha az e-mail már foglalt vagy az üzleti szabályok megsértése történik, 409 Conflict kerül visszaadásra értelmezhető hibaüzenettel. A végpont jellemzően nyitott (nem igényel előzetes hitelesítést), hiszen a cél épp egy új hitelesítési identitás létrehozása

Bejelentkezés 🡪 POST /api/auth/login

A bejelentkezési végpont hitelesíti a felhasználót az AuthService.authenticate hívással, majd siker esetén a JwtService.generate által előállított JWT tokent ad vissza AuthResponse formában. A kliens ezt a tokent a későbbi hívásoknál a Authorization: Bearer <token> fejléccel küldi, amely alapján a Spring Security azonosítja a felhasználót és a szerepköreit. Sikertelen hitelesítés esetén a szolgáltatás megfelelő hibát ad vissza (tipikusan 401 Unauthorized), siker esetben 200 OK és a token érkezik, amely a védett végpontok elérésének alapja

**RentalController – /api/rentals**

Bérlések listázása 🡪 GET /api/rentals

Ez a végpont a bérlések listázását végzi, opcionális időintervallum szűréssel. A from és to paraméterek ISO időformátumban adhatók meg, és a RentalRepository.findInRange biztosítja, hogy csak az adott ablakba eső rekordok érkezzenek vissza. A kód alapján a végpont nincs külön jogosultság-szűréssel ellátva, így az elérhetősége a globális Spring Security konfigurációtól függ. Tipikus beállítás szerint ez authenticated lehet, azaz érvényes JWT szükséges a híváshoz, ami védi az ügyféladatokat és csökkenti az adatszivárgás kockázatát. Visszatérési érték egy List<Rental>, ami megkönnyíti az admin és felhasználói felületek naptár vagy táblázatos nézetének feltöltését

Új bérlés 🡪 POST /api/rentals

A bérlés létrehozása védett művelet, amely hitelesített hívást feltételez. A Spring Security az Authentication objektumban adja át a bejelentkezett felhasználó identitását, a kód a principal e-mail címe alapján kikeresi a felhasználó azonosítóját és a beérkező Rental entitáshoz rendeli. Az ütközésvizsgálatot és az üzleti validációt a RentalService.create végzi, amely időintervallum-átfedés esetén IllegalArgumentException-t dob. A controller ezt lefordítja 409 Conflict válaszra, így a kliens egyértelmű jelzést kap az elérhetetlenségről. Siker esetén 200 OK és a mentett bérlés adatai érkeznek, amely már tartalmazza a végleges státuszt, azonosítót és az egyeztetett időpontokat

**CarController – /api/cars**

Autók listázása 🡪 GET /api/cars

Az autóflotta listázását végzi és a kód alapján nyitott vagy authenticated lehet a globális biztonsági beállítástól függően. A hívás a CarRepository.findAll metódusra épül, így a felhasználói felület könnyen tud járműkatalógust megjeleníteni kereséssel vagy szűrőkkel. A visszatérési típusnak célszerű List<Car>-nak lennie, a jelenlegi List<Cars> egy apró elírásnak tűnik, amit érdemes egységesíteni a DTO-val és az entitással

Új autó létrehozása 🡪 POST /api/cars

Új autó felvételére szolgál és admin jogosultságot igényel. A @PreAuthorize("hasRole('ADMIN')") kifejezetten a Spring Security method level security képességét használja, így csak olyan JWT-vel hívható, amelynek a szerepkörei között szerepel az ADMIN. Érvényes kérés esetén a repo.save eltárolja az autót, és 200 OK válasz érkezik a létrehozott erőforrás adataival, amellyel a felület azonnal frissítheti a flottalistát

Autó adatmódosítás 🡪 PUT /api/cars/{id}

Meglévő autó módosítását végzi és szintén ADMIN szerepkört igényel. Az útvonalparaméterként kapott azonosítót a kód a törzsben érkező objektumra ráírja, így egyértelmű, melyik rekord frissül. A mentést a repository végzi, tipikusan 200 OK válasszal és a frissített állapottal, ami megkönnyíti a kliens oldali optimista UI frissítéseket

Autó törlés 🡪 DELETE /api/cars/{id}

Az autó törlését valósítja meg, ADMIN szerepkörrel. A végpont először ellenőrzi, létezik-e az adott entitás, nem létező erőforrás esetén 404 Not Found-ot ad vissza, sikeres törlésnél 204 No Content érkezik, ami jelzi, hogy a kérés teljesült, de nincs visszaadandó törzs. Ez a viselkedés REST konzisztens és egyértelműen kezelhető a kliens oldalon.

## II.IV. Frontend

Az alábbi fejezet a megadott React + TypeScript kódbázis komponenseit, segédfüggvényeit és útvonalait ismerteti. Áttekintést adunk röviden a felépítésről, majd mélyebben bemutatjuk az adatszerkezeteket, a tipikus algoritmusokat, a tesztelési nézőpontokat és a továbbfejlesztési lehetőségeket.

Architektúra-áttekintés

* SPA frontenden React + TypeScript + React Router
* Stílus Tailwind CSS osztályokkal
* Auth JWT tokennel, lokális tárolásban (localStorage)
* api modul a REST hívásokhoz
* Oldalak: autólista, belépés, regisztráció, új foglalás, admin autó-nyilvántartó
* Útvonalvédelem: ProtectedRoute és AdminRoute komponensek

### II.IV.I. Komponensek, szervizek és útvonalak

Útvonalak

* / 🡪 autólista mindenki számára
* /login 🡪 belépés
* /registration 🡪 regisztráció
* /new-rental 🡪 védett, bejelentkezett felhasználóknak
* /admin/cars 🡪 védett, csak adminnak
* \* 🡪 fallback kezdőlapra

Útvonalvédelem szerepkör és bejelentkezés alapján. A szerepkör AuthContext-ből származik, a token payload alapján kerül kinyerésre. **Működés:**

* AdminRoute: csak ADMIN esetén enged, különben átirányít kezdőoldalra
* ProtectedRoute: USER vagy ADMIN esetén enged, különben login oldalra irányít

A komponensek minimális, célzott logikát tartalmaznak, jól illeszkednek a routerhez.

App

* **Feladat**
  + Keretoldal navigációval, menüvel, route-okkal
  + Felső navigáció szerepkör-érzékeny linkekkel
* **Megjegyzés**
  + Mobil menü állapotkezelése open state alapján
  + Logout gomb közvetlenül az AuthContext logout-ját hívja

CarList

**A komponens logikája tisztán prezentációs és adatbetöltő jellegű. Nincs saját paginálás vagy szűrés, a backend GET /api/cars teljes bérelhető autólistáját adja vissza.** Read-only nézet, reszponzív kártyás és rácsos megjelenítés.

* **Osztályváltozók / state**
  + data: Car[] useState, a betöltött autólista tárolása
* **Metódusok és hookok**
  + useEffect(() => api.get('/api/cars')...) 🡪 egyszeri betöltés
  + render kártyákba és üres állapot kijelzés

Login

Belépési űrlap validációval. Sikeres bejelentkezéskor token eltárolása, navigáció kezdőoldalra.

* **Osztályváltozók / state**
  + React Hook Form által menedzselt form state Form = { email, password }
  + isSubmitting, errors a validáció és a folyamat jelzésére
* **Metódusok és hookok**
  + handleSubmit(async v => { const res = await api.post('/api/auth/login', v); login(res.data.accessToken); nav('/') })
  + useAuth().login beírja a tokent az AuthContextbe és localStorage-ba

Registration

Új felhasználó regisztrálása, majd visszanavigálás a belépéshez. A backend tervezése szerint regisztrációkor még nincs token.

* **Osztályváltozók / state**
  + React Hook Form kezeli a Form = { email, password } értékeket és validációt
* **Metódusok és hookok**
  + api.post('/api/auth/register', v) majd nav('/login')
  + A kód jelenleg a visszatérő body-ból írná a tokent, de a backend 201-nél üres body-t ad vissza
  + a localStorage.setItem('token', res.data.accessToken) sor egy élesítés előtti tisztításra szorul, mivel nincs token a válaszban

NewRental

Új foglalás felvétele, autó választással és időablak megadásával. Ütközés esetén 409 kezelése felhasználóbarát üzenettel. A dátumkonverzió datetime-local → Date a böngésző zónáját használja.

* **Osztályváltozók / state**
  + cars: Car[] az autóválasztóhoz
  + React Hook Form Form = { carId, start, end }
* **Metódusok és hookok**
  + useEffect betölti a cars listát
  + handleSubmit POST /api/rentals, időpontok Date(...) típusra konvertálása
  + Két hibaág: 409 átfedés üzenet, egyéb hibákra általános jelzés

VehicleRegister

Admin CRUD felület a flottához. Új autó felvétele, meglévő szerkesztése, törlése. Az útvonalat AdminRoute védi szerepkör alapján. A numberOfPassangers helyes típus és optional kezelés szükséges a form inputoknál.

* **Osztályváltozók / state**
  + items: Car[] az aktuális lista
  + form: Car a szerkesztett vagy új autó adatai
* **Metódusok és hookok**
  + load tölti az items listát GET /api/cars hívással
  + save ágválasztás POST vagy PUT között form.id jelenléte alapján
  + edit beemeli a sor adatait a formba
  + del törlés DELETE /api/cars/:id
  + Reszponzív tábla és mobil-kártya nézet

Auth utilok és AuthContext

JWT payload dekódolása Base64URL-ből, szerepkör kinyerése és normalizálása, auth állapot és token életciklus kezelése Contextben. A Context bevezetése megkönnyíti a komponensekben a jogosultsági állapot fogyasztását. A tokenből történő szerepkör-olvasás rugalmas, kezeli az eltérő claim neveket.

* **Kulcs függvények**
  + decodeBase64Url és variánsa: Base64URL → Base64 → atob → JSON parse
  + getRole: role vagy roles[0] olvasása, ROLE\_ prefix levágása, validálás ADMIN|USER
  + AuthProvider: token state, role memo, login, logout akciók
  + useAuth: Context elérés és típusgarancia

## II.V. Fejlesztési lehetőségek

A jelenlegi alkalmazás egy példa működést mutat csak be. Számos olyan piaci / üzleti igény van, amely jogosan merülne fel valós környezetben. A teljesség igénye nélkül összegyűjtöttünk pár olyan fejlesztési irányt, amelyek felmerültek bennünk és amik megfontolandók lennének, amennyiben egy ilyen programot a piacra szeretnénk dobni.

Jelenleg LocalDateTime-ot használunk, amely nem foglalkozik az időzónákkal. Amennyiben ez egy globális piaci termék lenne, abban az esetben az időzónák okozta eltéréseket kezelni kellene.Érdemes lenne a validációs réteget kiemelni (Validator osztályok), valamint egységes hibakezelést alkalmazni saját hibaosztállyal, melyet a frontenden egyedileg (pl animált felugró ablakként) jelenítenénk meg. Ezek kaphatnának súlyozást és típust, hogy a frontenden akár színekkel jelöljük a fontosságát: zöld a visszaigazoló üzenet (minden OK), piros a súlyos hiba, sárga a figyelmeztetés, stb.

Érdemes lehet szigorítani a jelszavakat (pl. tartalmaznia kell különleges karaktert is a min. 6 hosszúság mellett, stb.). Össze lehetne kötni Google vagy Facebook profilokkal is (hagyományos név-jelszó regisztráció helyett).

Átgondolandó lenne további kedvezmények vagy felhasználói szintek bevezetése. Akciók és kedvezmények rögzíthetősége admin oldalon. Mind admin, mind pedig felhasználói oldalon érdemes lenne jól látható és szűrhető naptári foglalás nézetet kialakítani (színkódokkal jelölve). Admin oldalon egyes autókra, míg a user esetén a saját foglalásokra. Összességében a user számára egy Profil oldal létrehozása célszerű lenne (pl. számlázási adatok, korábbi foglalások, visszajelzések, panaszkezelés). Emailes és push üzenetes visszaigazoló üzentek / értesítések főként mobilos nézetben. Később mobil applikációként is érdemes lenne piacra lépni, nem csupán web-es mobil nézettel.

Az adatbázist jobban le kellene védeni nagy adatmennyiség esetén: paginált listázás bevezetése (rentals és cars végpontokra is), illetve például felhasználó törlés esetén létező foglalások töröltre állítása (ne maradjon bent „szemétként”).

Érdemes automatizált telepítéseket (javított CI/CD) kellene bevezetni a folyamatos teljesítésért. Audit loggal is érdemes lehet kiegészíteni a szoftvert, hogy a karbantarthatóságát javítsuk.

# III. Felhasználói dokumentáció

## III.I. Bemutató

Ez a szoftver egy **webalapú autókölcsönző rendszer**, amelynek segítségével felhasználók egyszerűen és biztonságosan foglalhatnak gépjárműveket adott időszakra. Az alkalmazás kétféle szerepkört támogat: **felhasználót** és **rendszergazdát (adminisztrátort)**. A felhasználók foglalni tudnak, az adminisztrátorok pedig kezelhetik az autóflottát.

Fontos tudni, hogy ez egy számítógépes szoftver, amelyet **webböngészőből** lehet elérni, tehát nincs szükség külön telepítésre a felhasználói oldalon. A háttérben egy **adatbázis** működik, amely a foglalások és a felhasználói adatok tárolását végzi, míg a grafikus felület a böngészőben jelenik meg.

## III.II. Rendszerkövetelmények

A szoftver modern webes technológiákra épül, így az alábbi eszközökkel és programokkal használható:

* **Eszközök:** asztali számítógép, laptop, tablet vagy okostelefon
* **Operációs rendszer:** Windows 10/11, Linux, macOS, Android, iOS
* **Böngésző:** Google Chrome, Mozilla Firefox, Microsoft Edge, Safari (legfrissebb verziók ajánlottak)
* **Hálózati kapcsolat:** stabil internetkapcsolat szükséges

Az adminisztrátorok számára további szoftverek szükségesek a szerver oldali futtatáshoz:

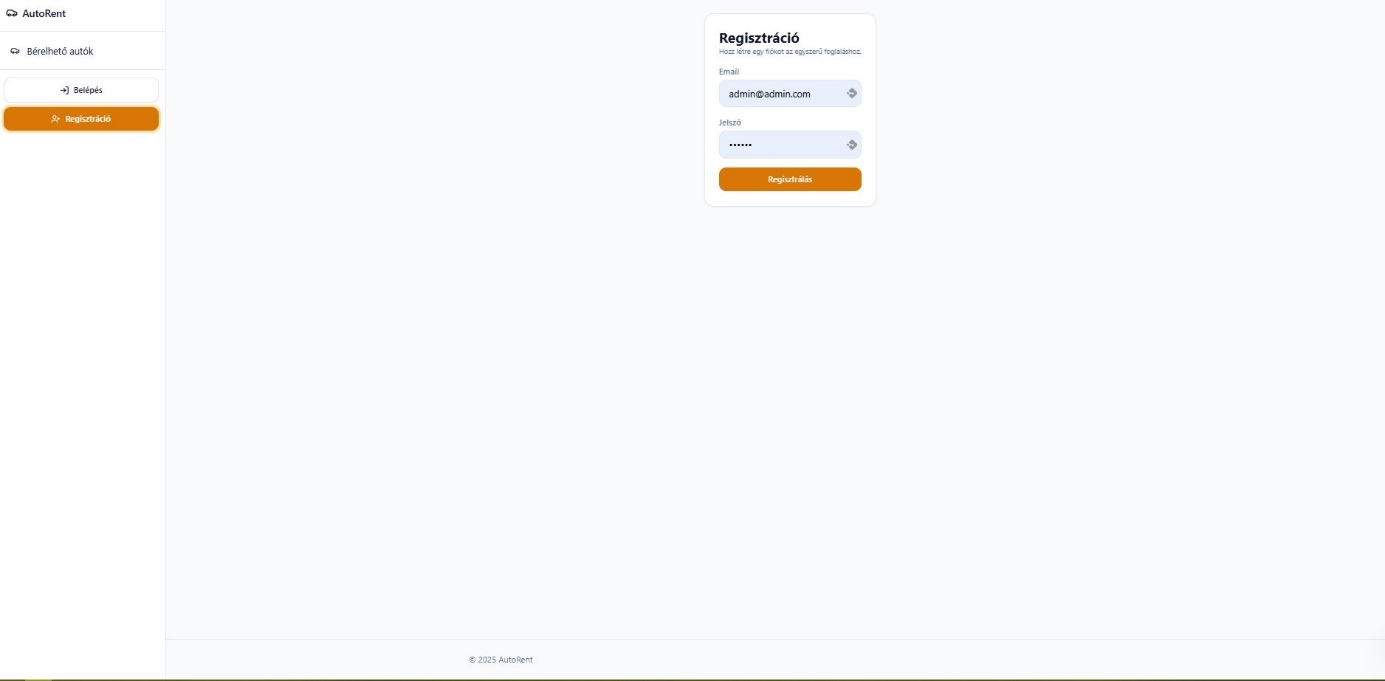
* **Java 17** futtatókörnyezet
* **Maven** a futtatáshoz és buildhez
* **Docker Desktop** a PostgreSQL adatbázis futtatásához

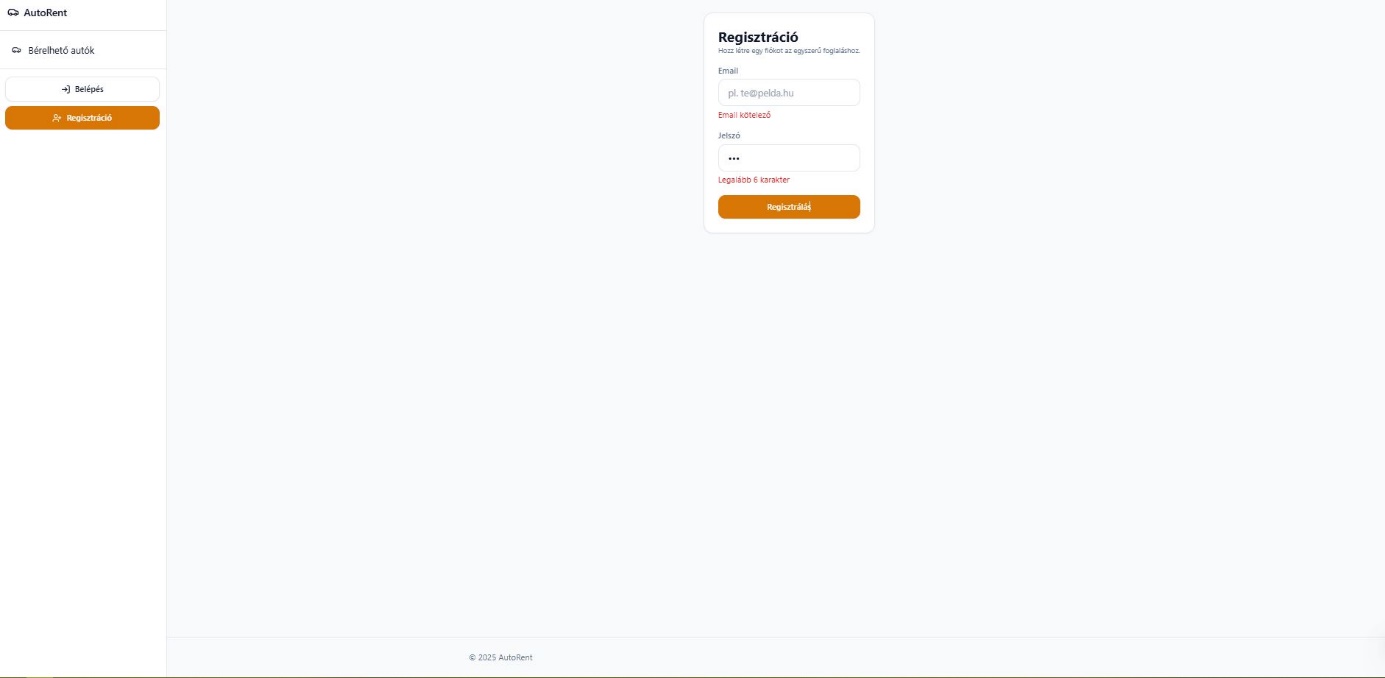
## III.III. Telepítés és indítás

A felhasználónak nincs szüksége telepítésre, a rendszer egy böngészőből érhető el a szerver címének megadásával (pl. http://localhost:5173 vagy éles környezetben egy domain néven).

## III.IV. Egyes oldalak és funkciók

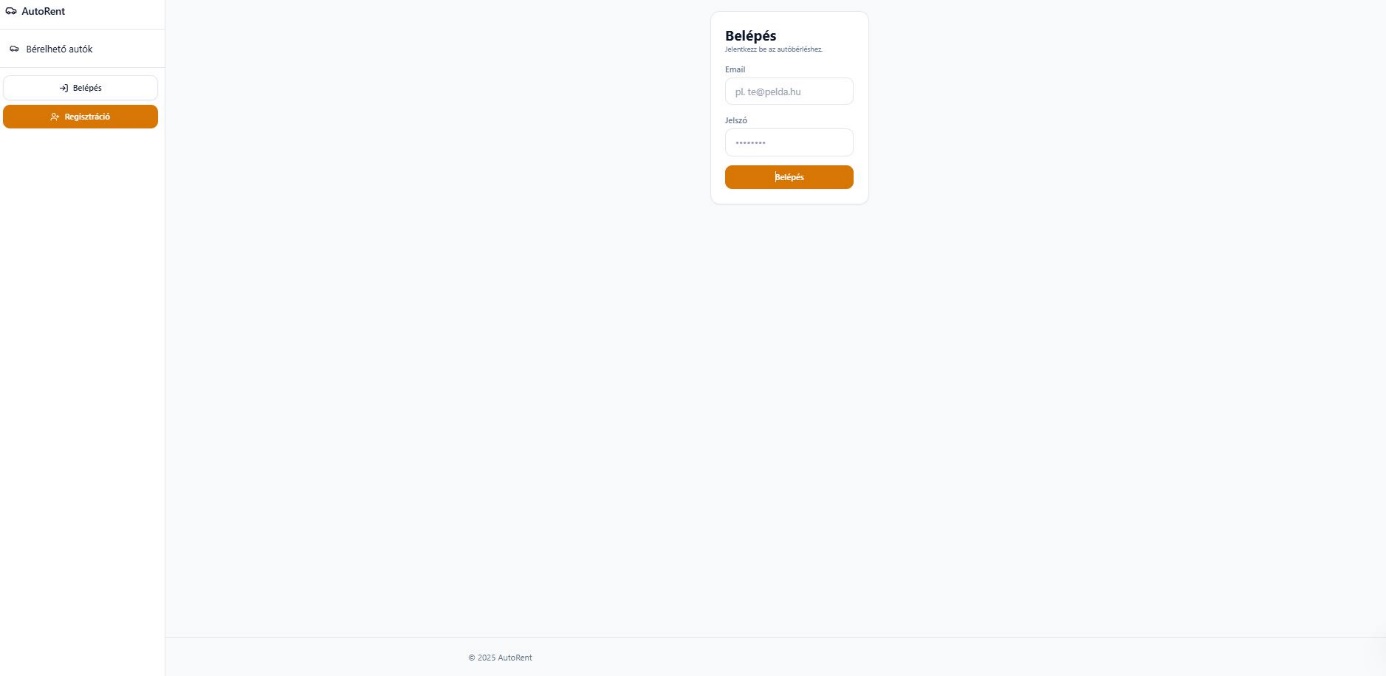
### III.IV.I. Regisztráció





* Nyisd meg a bejelentkező/regisztrációs oldalt
* Add meg az e-mail címed és egy biztonságos jelszót (pl. minimum 8 karakter, kis- és nagybetűk, számok és speciális karakterek használatával)
* Nyomd meg a **Regisztráció** gombot
* Siker esetén üzenetet kapsz, hogy a fiók létrejött, és a rendszer átirányít a bejelentkezéshez

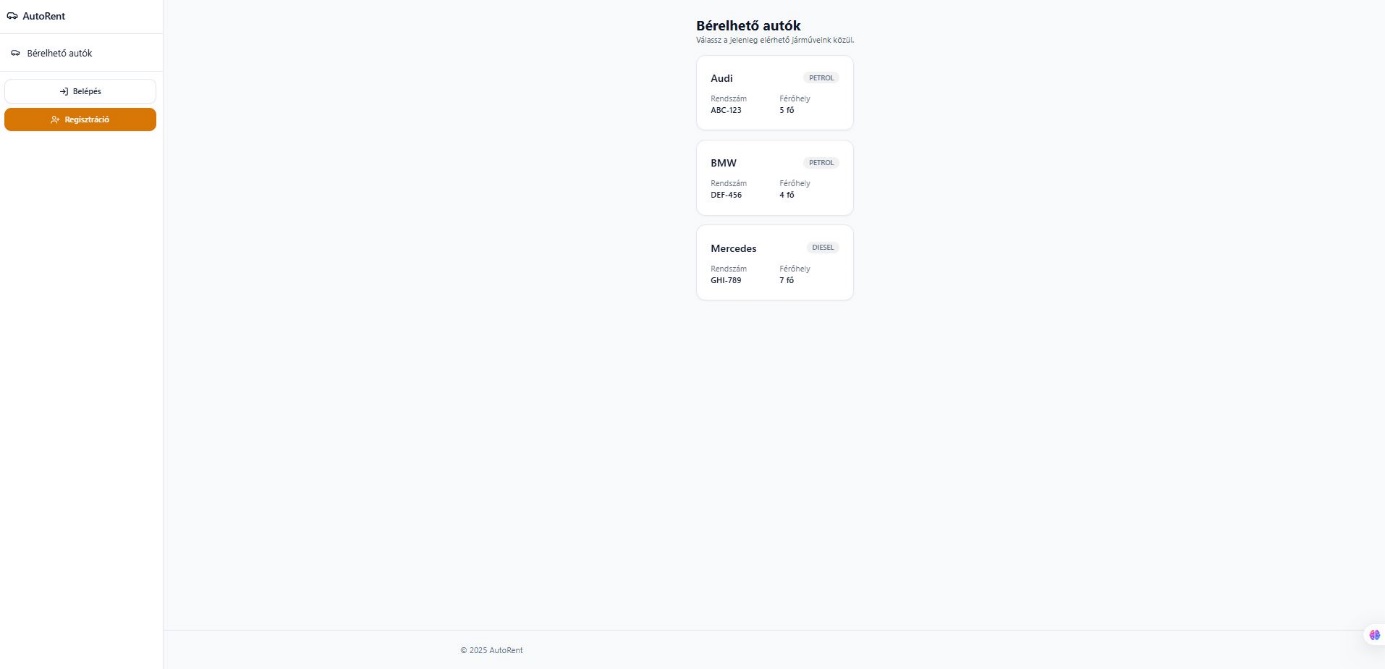
### III.IV.II. Bejelentkezés



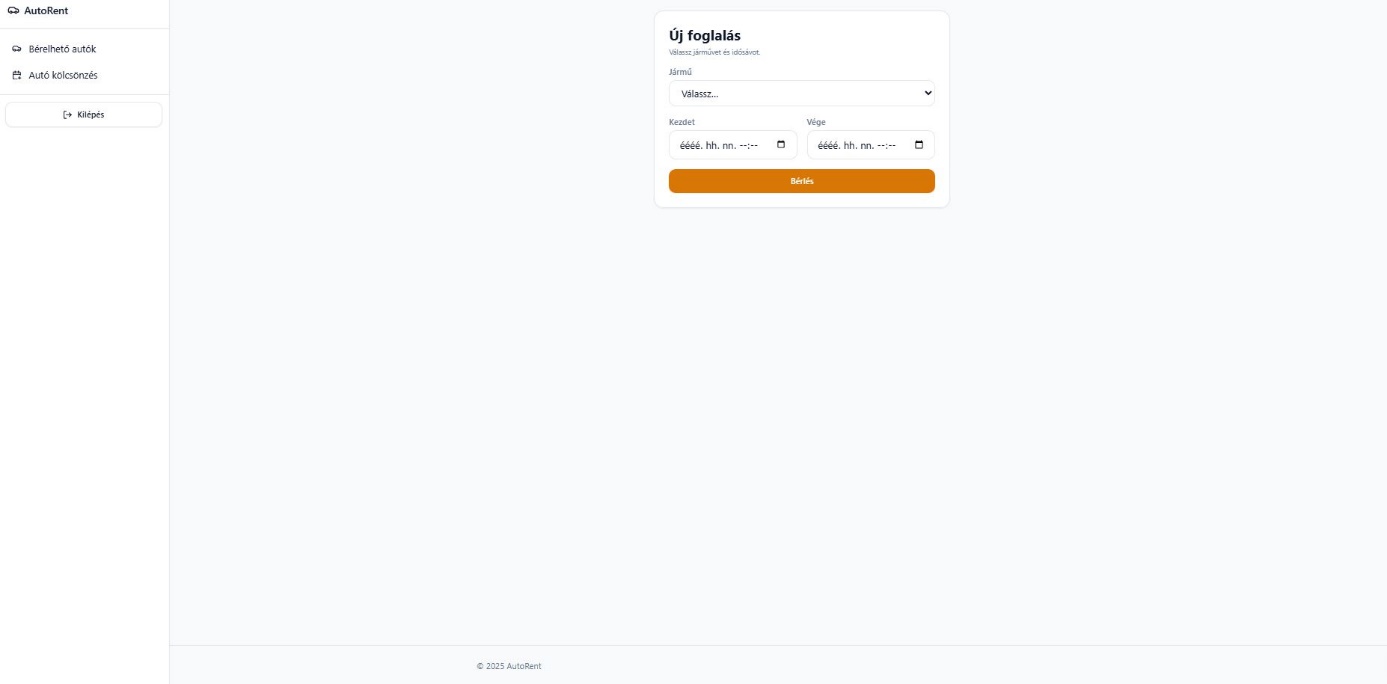
* Add meg a regisztrált e-mail címed és jelszavad
* A rendszer a háttérben ellenőrzi a hitelességet
* Sikeres belépés után kapsz egy **JWT tokent**, amely automatikusan el van tárolva, és minden további kéréshez felhasználásra kerül
* Sikertelen próbálkozásnál hibaüzenetet kapsz (pl. helytelen jelszó vagy nem létező felhasználó)

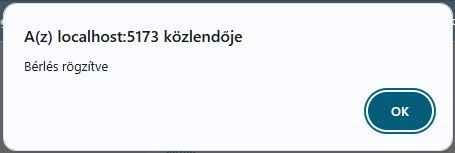
### III.IV.III. Funkciók felhasználóknak

* **Autók listázása:** A főoldalon megjelenik az összes elérhető autó a rendszámmal, típussal, férőhellyel és üzemanyag-típussal



* **Bérlés létrehozása:** A **Bérlés** gombbal a rendszer létrehozza a bérlést, amely automatikusan hozzád lesz rendelve

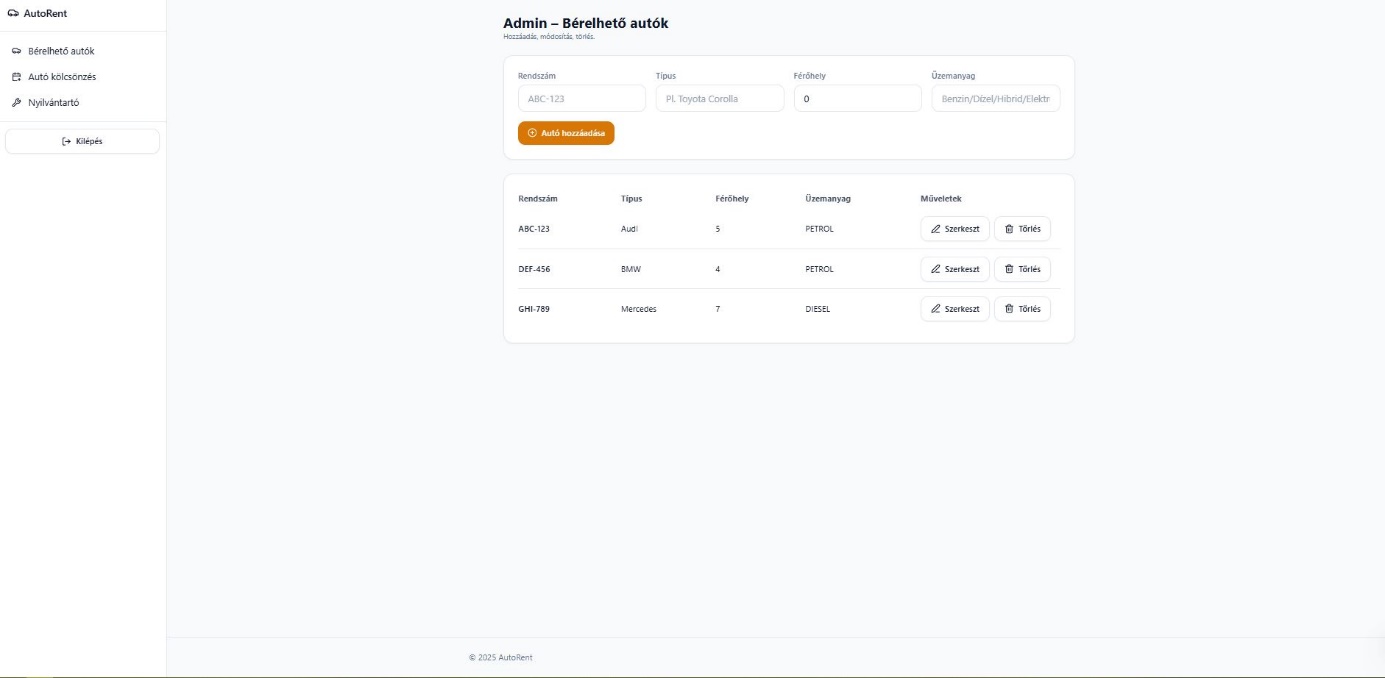




* **Ütközéskezelés:** Ha a megadott időszak ütközik egy korábbi foglalással, a rendszer hibaüzenetet küld (409 Conflict), és új időpontot kell választanod



### III.IV.IV. Funkciók adminisztrátoroknak



* **Autók felvétele:** Új autó hozzáadása rendszám, típus, férőhely és üzemanyag típus megadásával
* **Autók módosítása:** Meglévő autó adatait frissítheted (pl. ha változik a rendszám vagy a férőhely)
* **Autók törlése:** Ha egy autó már nem elérhető, törölheted a rendszerből; ezzel együtt a hozzá tartozó foglalások is törlődnek

## III.V. Biztonság

* **Bejelentkezés ellenőrzése (hitelesítés):** amikor belépsz a rendszerbe, kapsz egy digitális “belépőkártyát” (úgynevezett token). Ezt a kártyát a böngésződ minden további kérésnél automatikusan elküldi, így a rendszer tudja, hogy te vagy az, aki bejelentkezett, és ennek alapján enged hozzáférést a megfelelő funkciókhoz
* **Szerepkör alapú jogosultság:** az admin funkciók csak ADMIN szerepkörrel érhetők el
* **Jelszavak tárolása:** a rendszerben jelszavak csak titkosított (Bcrypt hash) formában szerepelnek
* **Űrlapbiztonság (CSRF védelem):** a rendszer megakadályozza, hogy más weboldalak a tudtod nélkül küldjenek be űrlapokat vagy módosítsák az adataidat. Ez a védelem a háttérben automatikusan működik, amikor például foglalást indítasz vagy törölsz valamit
* **Adatbiztonság:** minden foglalás az adott felhasználóhoz van kötve, így más nem férhet hozzá

# IV. Tesztelés

## IV.I. Dinamikus tesztek - backend és adatbázis

Dinamikus tesztelést végeztünk a controller rétegen a Spring MVC segítségével. Az alkalmazás kódját futtatjuk és a viselkedését ellenőrizzük konkrét bemenetek és elvárt kimenetek alapján. Az MVC teszt izoláltan, a webes réteget indítva ellenőrzi a controllerek működését anélkül, hogy a teljes alkalmazást és az adatbázist fel kellene húzni. A MockMvc használatával HTTP-kéréseket szimulálunk, így ez integrációs jellegű tesztelés a webes interfész szintjén, de nem teljes end-to-end teszt, mert a mögöttes szolgáltatásokat (AuthService, JwtService) mockoljuk.

Ehhez hoztunk létre egy teszt security config-ot. Ez a konfiguráció egy **Spring Security test profile**-hoz tartozó beállítás, amely leegyszerűsíti a tesztelési környezetet. A @TestConfiguration és @Profile("test") annotációk miatt ez csak a test profilban töltődik be, így az éles alkalmazást nem érinti. A @Primary biztosítja, hogy ez a biztonsági konfiguráció felülírja az alapértelmezett filterláncot.

@TestConfiguration  
@Profile("test")  
public class TestSecurityConfig {  
 @Bean  
 @Primary

SecurityFilterChain testFilterChain(HttpSecurity http) throws Exception {  
 http  
 .csrf(csrf -> csrf.disable())  
 .authorizeHttpRequests(auth -> auth.anyRequest().permitAll())  
 .sessionManagement(sm -> sm.disable())  
 .headers(h -> {});  
 return http.build();  
 }  
}

A testFilterChain metódusban a HttpSecurity-t úgy állítjuk be, hogy:

* CSRF védelem kikapcsolásra kerül, így a tesztekben nem kell minden kéréshez CSRF tokent generálni
* Minden kérés engedélyezett (permitAll()), így a hitelesítés és az autorizáció tesztelés közben nem akadályozza a működést
* Session kezelés letiltott, mert a JWT alapú hitelesítéshez a tesztben nincs rá szükség
* HTTP fejlécek konfigurálása üres, nincs extra védelem hozzáadva

Ennek köszönhetően a tesztkörnyezetben az API végpontok biztonsági filterek nélkül, szabadon hívhatók, így az automatizált tesztek kizárólag az üzleti logikát és a válaszok helyességét ellenőrzik, nem pedig a biztonsági réteget.

CarController-en végzett tesztek – autók CRUD műveletéhez tartozó tesztek:

* **Autók listázása**
  + összes autó visszaadása a repositoryból
  + válaszkód: 200 OK
  + JSON tömbben az elemek meglétének ellenőrzése
* **Autó létrehozása (ADMIN)**
  + új autó mentése admin jogosultsággal
  + válaszkód: 200 OK
  + válaszban az új entitás id értékének ellenőrzése
* **Autó frissítése (ADMIN)**
  + meglévő autó módosítása admin jogosultsággal
  + válaszkód: 200 OK
  + válaszban a frissített autó id értékének ellenőrzése
* **Autó törlése – nem létező erőforrás (ADMIN)**
  + autó törlése admin jogosultsággal, ha az id nem található
  + válaszkód: 404 Not Found

RentalController-en végzett tesztek – bérléshez kapcsolódó tesztek:

* **Bérlések listázása időintervallum alapján**
  + bérlések visszaadása a from és to paraméterek szerint szűrve
  + válaszkód: 200 OK
  + JSON tömbben az elemek meglétének ellenőrzése
* **Bérlés létrehozása – felhasználó hozzárendelése**
  + új bérlés létrehozása a bejelentkezett felhasználóhoz kötve
  + userId automatikus beállítása az Authentication alapján
  + válaszkód: 200 OK
  + válaszban az új bérlés id értékének ellenőrzése
* **Bérlés létrehozása – időütközés kezelése**
  + átfedés esetén kivétel dobása a RentalService által
  + controller 409 Conflict státuszt ad vissza
  + rövid, érthető hibaüzenet (Overlaps) a válasz törzsben

Példának hoztuk az új autóbérlés létrehozásához készített tesztet, amely azt célozza, hogy hogyan veszi ki a user adatokat és köti a rendszer automatikusan a foglalás készítéséhez:

@Test  
void create\_assignsUserFromAuthPrincipal() throws Exception {  
 String email = "user@exam.com";  
 UUID uid = UUID.*randomUUID*();  
  
 User user = new User();  
 user.setId(uid);  
 user.setEmail(email);  
 user.setPasswordHash("hash");  
 user.setRole("USER");  
  
 Mockito.*when*(users.findByEmail(email)).thenReturn(Optional.*of*(user));  
  
  
 Rental saved = new Rental(); saved.setId(UUID.*randomUUID*());  
 Mockito.*when*(service.create(*any*(Rental.class))).thenReturn(saved);  
  
  
 Authentication auth = Mockito.*mock*(Authentication.class);  
 Mockito.*when*(auth.getPrincipal()).thenReturn(email);  
  
  
 mvc.perform(*post*("/api/rentals").principal(auth)  
 .contentType(MediaType.*APPLICATION\_JSON*)  
 .content("{\"carId\":\"" + UUID.*randomUUID*() + "\",\"start\":\"2030-01-01T10:00:00\",\"end\":\"2030-01-02T10:00:00\"}")  
 .with(org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.*csrf*()))  
 .andExpect(*status*().isOk())  
 .andExpect(*jsonPath*("$.id").exists());  
}

Az **autentikáció és autorizáció** területén a következőket ellenőrizzük (AuthControllerTest):

* **Sikeres regisztráció**
  + új felhasználó létrehozása
  + válaszkód: 201 Created
  + üres body visszaadása (nem szivárog ki érzékeny adat)
* **Sikertelen regisztráció**
  + már létező e-mail cím esetén ütközés kezelése
  + válaszkód: 409 Conflict
  + rövid, érthető hibaüzenet visszaadása
* **Sikeres bejelentkezés**
  + helyes e-mail és jelszó páros hitelesítése
  + JWT token generálása a JwtService segítségével
  + válaszkód: 200 OK
  + JSON válaszban az accessToken mező megléte és értéke
* **Biztonsági mechanizmusok a tesztekben**
  + CSRF token meglétének ellenőrzése a POST kéréseknél
  + ellenőrzés, hogy érzékeny adatok (pl. jelszó, hash) nem szerepelnek a válaszban

## IV.II. E2E tesztek - Selenium

Az end-to-end tesztek célja, hogy a felhasználó szemszögéből, egy igazi böngészőben lefuttatva ellenőrizzük, hogy az autókölcsönző alkalmazás minden fontos funkciója hibátlanul működik. A tesztekhez a **Selenium WebDriver**-t használtuk, amely automatizáltan vezérli a Chrome böngészőt. A teszteket Java nyelven írtuk, a **Page Object Model** mintát követve, így minden képernyő külön osztályban szerepel (LoginPage, CarListPage, NewRentalPage, AdminCarsPage stb.), a tényleges tesztek pedig jól olvasható, felhasználói lépésekre bontott szcenáriók.

#### Bejelentkezés (Login) tesztelése

* **Elvárás:**
  + A felhasználó helyes adatokkal be tud jelentkezni.
  + Hibás adatokkal nem engedi be a rendszer.
* **Tesztlépések:**
  + Megnyitottuk a „Belépés” oldalt.
  + Helyes e-mail/jelszó párost adtunk meg.
  + Ellenőriztük, hogy sikeres belépés után visszairányít a nyitóoldalra, és a menüben megjelenik a „Kilépés” gomb.
  + Hibás jelszóval is próbálkoztunk, ekkor a rendszer rövid hibaüzenetet adott, és nem léptetett be.
* **Eredmény:**
  + A sikeres belépésnél a JWT token tárolódik, a menüpontok (pl. „Autó kölcsönzés”) megjelennek.
  + Hibás adatok esetén a rendszer nem enged tovább, és jól látható hibaüzenetet ad.

@Test  
void user\_can\_login\_and\_see\_cars() {  
 var nav = new NavBar(driver, wait);  
 var login = new LoginPage(driver, wait);  
 var cars = new CarListPage(driver, wait);  
  
 // kezdőlap  
 cars.open(baseUrl);  
 *assertThat*(cars.isLoaded()).isTrue();  
  
 // belépés  
 nav.goLogin();  
 login.login("user@example.com", "User123!");  
 // redirect után autólista látszik  
 *assertThat*(cars.isLoaded()).isTrue();  
}

#### Autólista (CarList) tesztelése

* **Elvárás:**
  + Az összes bérelhető autó megjelenjen a listában.
  + Ha nincs autó, a rendszer felhasználóbarát üzenetet adjon.
* **Tesztlépések:**
  + Nyitóoldal betöltése után ellenőriztük, hogy a backendből érkező autók helyesen listázódnak.
  + Megnéztük a reszponzív viselkedést: asztali nézetben táblázatosan, mobilon kártyás nézetben jelentek meg az autók.
* **Eredmény:**
  + Az autók minden adatukkal helyesen jelentek meg (típus, rendszám, üzemanyag, férőhely).
  + Üres adatbázis esetén korrekt szöveges üzenet jelent meg.

#### Új foglalás (NewRental) tesztelése

* **Elvárás:**
  + A felhasználó ki tud választani egy autót és időintervallumot.
  + Hibás adatoknál a rendszer figyelmeztessen.
  + Ütközés esetén a foglalást ne engedje.
* **Tesztlépések:**
  + Belépett felhasználóként megnyitottuk a „Új foglalás” oldalt.
  + Kiválasztottunk egy autót és jövőbeli időpontokat.
  + Ellenőriztük, hogy sikeres mentés után megerősítő üzenet jelent meg.
  + Próbálkoztunk hibás adatokkal (pl. a vége korábban volt, mint a kezdete), ilyenkor a rendszer piros hibaüzenetet mutatott.
  + Kipróbáltuk ugyanarra az időszakra a dupla foglalást is, a rendszer 409 hibával jelezte, hogy ütközés van.
* **Eredmény:**
  + A helyes foglalás rögzült, hiba esetén a rendszer megfelelő figyelmeztetést adott.

@Test  
void user\_can\_create\_rental\_non\_overlapping() {  
 var nav = new NavBar(driver, wait);  
 var login = new LoginPage(driver, wait);  
 var rental = new NewRentalPage(driver, wait);  
  
 // login  
 driver.get(baseUrl + "/login");  
 login.login("user@example.com", "User123!");  
 // új foglalás  
 nav.goNewRental();  
  
 // datetime-local mezőknek a billentyűs input formátuma: YYYYMMDDThhmm  
 // Példa: 20300101T1000 → 2030-01-01 10:00  
 rental.createRental(null, "2030.01.01. 10:00", "2030.01.02. 10:00");  
}

#### Admin autókezelő (VehicleRegister) tesztelése

* **Elvárás:**
  + Admin jogosultsággal lehessen autót hozzáadni, szerkeszteni és törölni.
  + Azonnali visszajelzést adjon a változásokról.
* **Tesztlépések:**
  + Admin belépéssel új autót adtunk hozzá.
  + Ellenőriztük, hogy a lista azonnal frissült és megjelent az új autó.
  + Egy meglévő autót szerkesztettünk, és figyeltük, hogy a módosítás a táblázatban is frissült.
  + Végül egy autót töröltünk, ami azonnal eltűnt a listából.
* **Eredmény:**
  + Az admin felület teljes CRUD funkciója hibátlanul működött.
  + Reszponzív nézetben (mobil vs. desktop) is helyesen jelentek meg az adatok.

#### Útvonalvédelem (Route protection) tesztelése

* **Elvárás:**
  + Be nem jelentkezett felhasználó ne férhessen hozzá a védett oldalakhoz.
  + Csak admin láthassa az admin felületet.
* **Tesztlépések:**
  + Belépés nélkül próbáltuk elérni az „Új foglalás” oldalt → a rendszer átirányított a loginhoz.
  + Nem admin felhasználóként próbáltuk megnyitni az admin autókezelőt → a rendszer visszairányított a főoldalra.
* **Eredmény:**
  + A jogosultságok helyesen érvényesültek, a védett oldalakhoz nem lehetett illetéktelenül hozzáférni.

## IV.III. Manuális tesztek - frontend

A rendszer felhasználói felületének manuális tesztelését képernyőnként végeztük el. A cél az volt, hogy a valós használatot szimulálva ellenőrizzük, hogy minden funkció helyesen működik, a hibák jól kezelhetők, és a felület a felhasználó számára logikus és átlátható marad. Az egyes oldalak működését többféle bemenettel, helyes és hibás adatokkal is kipróbáltuk, hogy lássuk, a rendszer hogyan reagál különböző helyzetekben.

**Nyitóoldal és autólista**

A tesztelést a kezdőlapon kezdtük, amely az összes bérelhető autót megjeleníti. Ellenőriztük, hogy a backendből érkező adatok helyesen jelennek meg: minden autó típusának, rendszámának, üzemanyag típusának és férőhelyszámának szerepelnie kell a listában. Külön figyeltünk arra, hogy mi történik, ha az adatbázis üres – ekkor a felület megfelelően egy üzenetet mutatott: „Jelenleg nincs megjeleníthető autó.” Ez a teszt igazolta, hogy a rendszer nem hibázik üres lista esetén sem, hanem felhasználóbarát üzenetet jelenít meg. Megnéztük azt is, hogyan viselkedik a lista különböző képernyőméreteken: asztali nézetben rácsos elrendezésben, mobilon pedig kártyás formában jól áttekinthető módon jelennek meg az adatok.

**Regisztrációs felület**

A következő lépésben a regisztrációs képernyőt teszteltük. Itt manuálisan próbáltunk különböző kombinációkat megadni: helyes e-mail és jelszó párost, túl rövid jelszót, valamint üresen hagyott mezőket is. Ellenőriztük, hogy a validáció azonnal jelzi a hibákat, például ha a jelszó kevesebb, mint hat karakter, akkor a rendszer egyértelmű üzenetet mutat: „Legalább 6 karakter”. A sikeres regisztráció után a felhasználót automatikusan átirányítja a bejelentkezési oldalra, így biztosítva a következő logikus lépést a folyamatban. Ezzel a teszteléssel megerősítettük, hogy az űrlapvalidáció és az adatok backend felé történő továbbítása is hibátlanul működik.

**Bejelentkezés**

A login oldal tesztelése során először egy érvényes, már regisztrált felhasználóval próbáltunk belépni. A sikeres hitelesítés után a rendszer visszairányított a nyitóoldalra, és a felső menüben megjelent a kijelentkezés gomb, ami igazolta, hogy a rendszer észlelte a bejelentkezett állapotot. Ezt követően hibás adatokkal (rossz jelszóval, nem létező e-mail címmel) is próbálkoztunk. A rendszer ilyenkor megfelelő hibaüzenetet adott, és nem engedett tovább, így a biztonsági réteg működése manuálisan is igazolhatóvá vált. Külön figyeltünk arra, hogy a teszt során ne jelenjen meg semmilyen érzékeny információ (például a jelszó hash), csupán rövid, érthető hibaüzenet.

**Új foglalás rögzítése**

A foglalási képernyőn ellenőriztük a legkritikusabb funkciót: az időpont és az autó kiválasztását. Teszteltük, hogy a rendszer megfelelően betölti az elérhető autókat egy legördülő listába. Ezt követően többféle időintervallummal próbálkoztunk. Ha a végdátum korábban volt, mint a kezdődátum, a validáció rögtön figyelmeztetett. Ha helyes időszakot adtunk meg, a foglalás sikeresen rögzült, és a felület megerősítő üzenetet jelenített meg. Végül szimuláltunk egy ütközést: ugyanarra az autóra ugyanarra az időszakra próbáltunk foglalni. Ekkor a backend 409 hibát adott, amit a felület egy felhasználóbarát üzenettel közölt: „Az adott időszakban az autó foglalt! Válassz másik időpontot.” Ez a manuális teszt bizonyította, hogy a kritikus üzleti logika a kliens oldalon is megfelelően van kezelve.

**Admin autókezelő felület**

Az adminisztrátori felület külön tesztelést igényelt, mivel itt a teljes CRUD műveletet el kell érni. Először ellenőriztük az autóhozzáadás funkciót: egy új rendszám, típus, férőhely és üzemanyag típus megadásával mentettük az autót, ami a lista alján megjelent. Ezután szerkesztettünk egy meglévő autót, és figyeltük, hogy a változtatás a táblázatban is azonnal láthatóvá válik. Végül kipróbáltuk a törlést, amely után az autó eltűnt a listából. Teszteltük az üres inputokkal való próbálkozást is, ahol a rendszer megfelelően hibát jelzett. Külön kiemelendő, hogy a felület reszponzívan működött: asztali nézetben táblázatot, mobilon pedig kártyás listát kaptunk.

**Navigáció és útvonalvédelem**

A manuális tesztelés része volt annak ellenőrzése is, hogy a védett oldalakhoz való hozzáférés megfelelően szabályozott. Be nem jelentkezett felhasználóként próbáltuk megnyitni a „Új foglalás” oldalt, de a rendszer visszairányított a bejelentkezéshez. Ugyanígy adminisztrátori jogosultság nélkül az autókezelő felület nem volt elérhető, és a főoldalra irányított a rendszer. Ez megerősítette, hogy a frontend útvonalvédelme (ProtectedRoute és AdminRoute) megfelelően együttműködik a háttérben tárolt JWT tokennel.

**Összegzés**

A manuális tesztelés során minden fő funkciót lefedtünk. Az alkalmazás a vizsgálatok alapján stabil, a validációk és a hibakezelések következetesen működnek. Az autók listázása, a regisztráció és bejelentkezés, az új foglalások rögzítése, valamint az adminisztrátori CRUD műveletek mind jól használhatónak bizonyultak. Kisebb hiányosságokat azonosítottunk, például a regisztráció utáni token tárolás következetlenségét, illetve a részletesebb hibavisszajelzések hiányát, de ezek a projekt további iterációiban javíthatók. Összességében a manuális tesztelés igazolta, hogy a frontend megfelel a szakmai céloknak, és alkalmas az alapvető autókölcsönzési folyamatok támogatására.

# V. Összefoglalás

A projekt elején megfogalmazott célunk egy egyszerűen használható, ugyanakkor szakmailag konzisztens autókölcsönző webalkalmazás elkészítése volt. Ezt a célt a saját értékelésünk szerint teljesítettük: a háromrétegű architektúra tisztán kirajzolódik, a backend és a frontend szerepei jól elválnak, az adatperzisztencia és a biztonság pedig a vizsgafeladat keretei között következetesen megvalósul. A rendszer képes a teljes alapfolyamatra a regisztrációtól és bejelentkezéstől kezdve az autók listázásán és adminisztratív kezelésén át a bérlés felvételéig, beleértve az időütközések kiszűrését is.

Szakmai szempontból a legnagyobb hozzáadott értéket az adatintegritás és a biztonsági réteg megtervezése adta. A bérlések átfedésének kezelése látszólag triviális feladat, a gyakorlatban viszont több apró döntést igényel, például az intervallumok szemantikáját vagy azt, hogy mely státuszok számítsanak ütközőnek. Ezek a döntések végig hatnak a repository lekérdezésektől a szolgáltatásrétegen át a kliensoldali hibaüzenetekig. Ugyanígy a JWT alapú hitelesítés és a szerepkörök alkalmazása egyszerű mintának tűnik, de a token tartalmának normalizálása, a lejárat és a kliensoldali tárolás gyakorlati kérdéseket vet fel, amelyek kezelésével kézzelfogható tapasztalatot szereztünk.

A fejlesztés során a legnagyobb kihívást az egységesség megőrzése jelentette több komponens és csapattag párhuzamos munkája mellett. Itt bizonyult kulcsnak a Docker alapú fejlesztői környezet és a Flyway migrációk és a Git repository. A tesztelés terén a Spring MVC tesztek adták a legtöbb gyors visszajelzést. Ezekkel validáltuk a REST szerződést, a státuszkódokat, a hibaágakat és az autentikációs folyamatot úgy, hogy közben nem kellett teljes alkalmazást és adatbázist indítani. Ez a megközelítés különösen hasznos volt a controllerek finomhangolásánál és az üzenetek egységesítésénél. A jövőben érdemes ezt kiegészíteni szolgáltatás- és repository-szintű tesztekkel Testcontainers Postgres környezetben, illetve néhány end-to-end teszttel, hogy a felhasználói utakat is automatizáltan lefedjük.

A felhasználói felületnél a cél a sallangmentes működés volt. A komponensek egyszerűek és célzottak, ezáltal könnyen bővíthetők. Külön pozitív tapasztalat, hogy a jogosultságkezelés és az útvonalvédelem a Contexten keresztül minimális kóddal fenntartható. Ahol kompromisszumot kötöttünk, az a fejlettebb UX terület: jelenleg a listázás és az űrlapok korrektek, de hiányzik a szerveroldali paginálás, a szűrők, a naptárnézet, illetve a konzisztens hiba- és státuszvisszajelzés komponensek. Ezek ugyan nem gátolják a használhatóságot, de egy érettebb termékszinthez szükségesek.

Mint kész projekt, a rendszer jelen állapotában alkalmas demonstrációra és alapfunkciók ellátására. A backenden a fő domain logikák megvannak autorizációval és autentikációval.

Az utóéletről gondolkodva két pályát látunk. Az egyik a szakmai mélyítés: adatbázis-szinten kizáró ütközéskezelés, egységes hibaséma és nemzetköziesítés. A másik a termékesítés: dinamikus árazás, promóciók, hűségprogram, fizetési integráció és üzemeltetési riportok, bejelentkezési profil bővítés. Ez a kettő nem zárja ki egymást, sőt egymást erősítik, mert az üzleti funkciók megbízhatóan csak stabil technikai alapokra építhetők.

Összességében ez a vizsgaprojekt elérte az eredeti céljait, és tiszta alapot ad a folytatáshoz. Megmutatta, hogy egy egyszerű üzleti problémát hogyan lehet fegyelmezett, ipari mintákat követő módon megoldani, miközben a tanulási görbe is meredek volt és sok, később is hasznosítható tapasztalatot hozott.