Szakdolgozat

**FELADATKIÍRÁS**

**Bíró László**

Mérnökinformatikus hallgató részére

Járműkölcsönző rendszer fejlesztése

A szakdolgozat témája egy járműkölcsönző cég teljes rendszerének a felépítése, amelyhez több modern technológiát is szükséges lesz majd felhasználni.

A rendszer fog állni egy folyamatosan futó szerverből, ami Java Spring keretrendszer felhasználásával fog történni, egy adatbázisból és egy kliensalkalmazásból, amelyhez react/react native vagy kotlin nyelvek lesznek felhasználva. A kommunikáció REST API-k, illetve JSON adatstruktúrák segítségével fog megvalósulni.

A felhasználó könnyen képes lesz a lehetőségek közül a neki legmegfelelőbb jármű kiválasztására, az útvonala nyomon követésére és a járművek egyértelmű azonosítására. A járművek azonosításánál lehetősége lesz majd bárkódok, illetve QR kódok leolvasására a gyorsabb kölcsönzés végett, de a manuális beírás is egy lehetőség lesz. A felhasználók adatainak a biztonságos kezelése érdekében többszintű validáció lesz megvalósítva (adatbázis, szerver, kliensalkalmazás).

A hallgató feladatának a következőkre kell kiterjednie:

* Ismerje meg a felhasználandó technológiák lehetőségeit, amelyek segítségével képes lesz elvégezni megoldani a feladat kihívásait.
* Tervezze meg az adatbázis szerkezetét, illetve az alkalmazáson belül adatküldésre használt JSON adatstruktúrák szerkezetét.
* Készítsen el egy működő szerver alkalmazást, amely REST API-k segítségével képes kommunikálni egy kliens alkalmazással.
* Készítsen el egy működő kliens alkalmazást, amely képes lesz majd kommunikálni a szerver alkalmazással.
* Tesztelje az elkészült alkalmazást.

**Tanszéki konzulens:** Kövesdán Gábor, tanársegéd

Budapest, 2022. október 6.

Dr. Charaf Hassan

egyetemi tanár

tanszékvezető



**Budapesti Műszaki és Gazdaságtudományi Egyetem**

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Bíró László

**Járműkölcsönző rendszer fejlesztése**

Konzulens

Kövesdán Gábor

BUDAPEST, 2022

**Tartalomjegyzék**

[Szakdolgozat 1](#_Toc121344971)

[Összefoglaló 6](#_Toc121344972)

[Abstract 7](#_Toc121344973)

[1 Köszönetnyílvánítás 8](#_Toc121344974)

[2 Bevezetés 10](#_Toc121344975)

[3 Felhasznált technológiák 12](#_Toc121344976)

[**3.1 Backend technológiák** 12](#_Toc121344977)

[3.1.1 Java 12](#_Toc121344978)

[3.1.2 MySQL 12](#_Toc121344979)

[3.1.3 Spring keretrendszer 12](#_Toc121344980)

[3.1.4 Spring Boot 13](#_Toc121344981)

[3.1.5 Spring MVC 14](#_Toc121344982)

[3.1.6 Spring Security 15](#_Toc121344983)

[**3.2 Kliens technológiák** 16](#_Toc121344984)

[3.2.1 Android 16](#_Toc121344985)

[3.2.2 Kotlin 18](#_Toc121344986)

[3.2.3 Retrofit2 19](#_Toc121344987)

[3.2.4 Okhttp3 19](#_Toc121344988)

[3.3 Tooling 20](#_Toc121344989)

[3.3.1 GIT 20](#_Toc121344990)

[3.3.2 Github 20](#_Toc121344991)

[3.3.3 Visual studio code 21](#_Toc121344992)

[3.3.4 Android studio 21](#_Toc121344993)

[3.3.5 IntelliJ IDEA 21](#_Toc121344994)

[4 Tervezés 23](#_Toc121344995)

[4.1 Kommunikáció és a REST 25](#_Toc121344996)

[4.2 CRUD és a REST 26](#_Toc121344997)

[5 Implementáció 30](#_Toc121344998)

[5.1 Szerver alkalmazás 30](#_Toc121344999)

[5.1.1 A domainek 31](#_Toc121345000)

[5.1.2 A Controller osztályok 33](#_Toc121345001)

[5.1.3 A Service osztályok 36](#_Toc121345002)

[5.1.4 A Repository osztályok 37](#_Toc121345003)

[5.1.5 A biztonság 38](#_Toc121345004)

[5.1.6 A validációk 39](#_Toc121345005)

[5.2 Kliens alkalmazás 40](#_Toc121345006)

[5.2.1 A kliens applikáció felépítése 40](#_Toc121345007)

[5.2.2 Felületek 44](#_Toc121345008)

[6 Tesztelés 52](#_Toc121345009)

[7 Összefoglalás 54](#_Toc121345010)

[8 Irodalomjegyzék 55](#_Toc121345011)

[Függelék 57](#_Toc121345012)

**Hallgatói nyilatkozat**

Alulírott **Bíró László**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2022. 10. 24

...…………………………………………….

Bíró László

**Összefoglaló**

A szakdolgozat témája egy járműkölcsönző cég teljes rendszerének a felépítése, amelyhez több modern technológiát is szükséges lesz majd felhasználni.

A rendszer fog állni egy folyamatosan futó szerverből, Java Spring keretrendszer felhasználásával, egy adatbázisból és egy kliensalkalmazásból, amelyhez react/react native vagy kotlin nyelveket szeretnék felhasználni. A kommunikáció REST API-k, illetve JSON adatstruktúrák segítségével fog megvalósulni.

A szakdolgozat elkészítése során a felhasználandó technológiák lehetőségeit kihasználva a kliens többfajta jármű közül biztonságosan választhat. Könnyen képes lesz a lehetőségek közül a neki legmegfelelőbb jármű kiválasztására és a járművek egyértelmű azonosítására. A járművek azonosítására lehetőség lesz majd barkódok, illetve QR kódok leolvasására a gyorsabb kölcsönzés végett, de a manuális beírásra is lehetőség lesz. A felhasználók adatainak a biztonságos kezelése érdekében többszintű validáció lesz megvalósítva (adatbázis, szerver, kliensalkalmazás). A beléptetést pedig a Java Spring lehetőségeit kihasználva szeretném még biztonságosabbá tenni. (Spring Security)

Összefoglalva a szakdolgozatom témája kiterjed egy komplex rendszer minden fontosabb elemére, ezek stabil megalkotására, biztonságossá tételére és a felhasználók szemszögéből a minél egyszerűbb kezelhetőségre a legmodernebb technológiák és módszerek felhasználásával.

**Abstract**

The topic of the thesis is the construction of the entire system of a vehicle rental company, for which it will be necessary to use several modern technologies.

The system will consist of a continuously running server using the Java Spring framework, a database, and a client application for which I would like to use react/react native or kotlin languages. Communication will be implemented using REST APIs and JSON data structures.

The client can safely choose from several types of vehicles. In addition, the client will easily be able to choose the vehicle that suits him best and clearly identify the vehicles. To identify the vehicles, it will be possible to read barcodes and QR codes for faster rental, but it will also be possible to enter them manually. In order to safely manage user data, multi-level validation will be implemented (database, server, client application), and I would like to make access even more secure by taking advantage of Java Spring. (Spring Security)

In summary, the topic of my thesis covers all the most important elements of a complex system, their stable creation, making them safe and making them as simple as possible from the users' point of view using the most modern technologies and methods.

1. **Köszönetnyílvánítás**

Ebben a fejezetben szeretném megköszönni a szakdolgozatom elkészítése során kapott anyagi, illetve erkölcsi segítséget és támogatást. Ebben több személyt és intézményt is megszeretnék említeni, mert nélkülük valószínűleg nem jutottam volna el idáig.

Első sorban meg szeretném köszönni az édesapámnak, aki már nincsen köztünk, de számomra olyan, mintha végig mellettem lett volna, a jó és főleg a rossz időkben. A tőle kapott erkölcsi és anyagi támogatást már sosem tudom meghálálni sajnos, ezért szerettem volna külön megemlíteni.

Ezenkívül természetesen meg szeretném köszönni édesanyámnak, hogy mindig ott volt mellettem, így mindig volt, akivel megbeszéljem a dolgokat, még akkor is, ha 1000 km volt a távolság és természetesen minden rokonomnak, aki valamilyen módon támogatott egyetemi éveim alatt. Tudom néha erejükön felüli mértékben támogattak ezen évek során.

Megköszönném Dr. Pataricza Andrásnak, aki a témalaboratórium, illetve önálló laboratórium tantárgyaknál konzulensem volt a korábbi tanévben, aki értékes és hasznos megjegyzéseivel, észrevételeivel nagy mértékben a segítségemre volt. Elmagyarázta hogyan lehet jól beosztani az időmet, hogyan lehet egy dokumentumot jól megszerkeszteni, egy prezentációt elkészíteni és bemutatni, illetve a szakirodalom kutatást megfelelő precizitással elvégezni. Elmagyarázta, hogy nem az a jó programozó, aki sok nyelvet ismer, hanem aki jól ismeri a módszereket, az elméleteket, és tudja milyen programozóként gondolkodni. Hiszen a nyelvek változnak, illetve mindig jönnek ki újak, de a mögöttük lévő elméleti háttér állandó.

Szeretném még megköszönni jelenlegi konzulensemnek Kövesdán Gábornak, aki bevezetett a Java szerverek és a Spring működésébe. Sok jó tanáccsal és hasznos észrevétellel látott el a projekt készítése közben és mindig kaptam választ a kérdéseimre, ha valahol elakadtam. Az ő szakmai hozzáértése és hasznos megjegyzései nagy mértékben a segítségemre voltak a szakdolgozatom és a hozzá tartozó szerver-kliens alkalmazásom elkészítése során.

Köszönettel tartozok még a csíkszeredai Magic Solutions srl. munkatársainak, akik bevezettek a Java programozás és a Java szerverek világába régi és új Java technológiák segítségével megalkotott projektjeiben. Nagyon pozitívan fogadtak a munkába állásom kezdetén és a felmerülő kérdéseimre készséggel válaszoltak, illetve segítettek a kezdeti nehézségek leküzdésében.

Legvégül, de persze nem utolsó sorban, szeretném megköszönni a Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatika Karának és valamennyi oktatójának, dolgozójának a lelkiismeretes munkáját, erőfeszítését, amellyel a hallgatók képzését, tanulását, a versenyképes tudás megszerzését bármilyen módon is támogatják, hiszen nélkülük egyikünk sem juthatna el idáig, vagy érhetne el komolyabb eredményeket az életben szakmai téren.

1. **Bevezetés**

A szakdolgozatom elkészítéséhez hosszú út vezetett el, először is el kellett készítenem a projektemet. Ez egy szerver-kliens alkalmazás, amely félig Java és félig Kotlin programozási nyelvek felhasználásával készült el. Rengeteg technológia felhasználása volt szükséges, hogy egy könnyedén működő rendszer keletkezzen a végén, ami stabil és egyben biztonságos is.

A szakdolgozatom elkészítése alatt fejlesztettem egy komplex kliens-szerver rendszert, amelynek segítségével egy járműkölcsönző cég feladatait lehet menedzselni. Ehhez olyan technológiák felhasználását terveztem, amelyek modernek, de a céges világban már elég stabilnak és bejáratottnak vannak elismerve. Lényege az, hogy a felhasználó a különböző típusú járművek közül választva egy felhasználóbarát mobilalkalmazás segítségével képes elvégezni a kölcsönzést, bizonyos esetekben csak egy QR kódot beolvasva. Ezzel a módszerrel sok papírmunka spórólható meg, hiszen egy regisztrált felhasználóról már rengeteg adatot tárol az adatbázis, így a szerződéskötés egy bejelentkezett felhasználó esetén már néhány kattintás alatt elvégezhető jóformán emberi beavatkozás nélkül.

A mai világban már rengeteg helyen elterjedt az a módszer, hogy a hosszadalmas papírmunkával járó procedúrákat egy mobilos alkalmazás segítségével valósítjuk meg ezzel könnyítve meg az éles adatok áttekinthetőségét, a folyamat átláthatóságát és logok segítségével az esetleges problémák minél könnyebb és gyorsabb megoldását is. A technológiák közötti kommunikáció mostanság egyre hatékonyabbá válik, így még jobban kilehet használni ennek az előnyeit.

A kliens alkalmazásom egy Android rendszerre írt alkalmazás, mely Rest API-k segítségével kommunikál a szerver alkalmazással. Programozási nyelvnek a Kotlin-t választottam, amely a háttérkutatásaim alapján egy elég modern nyelv így nagyon dinamikusan és hatékonyan lehet benne fejleszteni. A nagy népszerűség és a komoly fejlesztőgárda következtében egy elég kifejlett programozási tekinthető. Fejlesztés során, ha elakadtam valahol általában elég hamar tudtam kapni megoldást az interneten, vagy egy hasonló példát, így sikeresen tudtam venni az elém kerülő akadályokat. Ami még nagyon megtetszett a Kotlin nyelvben, az az erősen típusos tulajdonsága, amely rengeteget a segít számomra a forráskód átláthatóságában és a hibakeresés folyamatában.

Az applikáció segítségével meg lehet tekinteni, hogy milyen járművek állnak a rendelkezésünkre, lehetőségünk van a járművek közötti szűrésre, illetve köthetünk egy szerződést is a céggel a kiválasztott járműre vagy járművekre. Ehhez először regisztrálni kell a felhasználónkat, amely alatt adatokat adunk meg magunkról, amely segítségével megkülönböztethetővé válunk másoktól az applikáció előtt. A regisztráció után az újonnan készült felhasználónkkal rögtön be tudunk lépni az applikációba. Ezután már dinamikusan és egyszerűen lehet úgymond böngészni a bérlendő járművek között és két három kattintás után kibérelni egy vagy akár több járművet is egyszerre. Ezenkívül lehetőségünk van a saját profilunk módosítására és bizonyos grafikonok/táblázatok/információk megtekintésére.

A regisztrációkor az adataink a szerveren tárolódnak el és teljes biztonságban vannak a Spring Security technológiáinak köszönhetően. A belépés után a szerver generál egy tokent, és ezután az adatok további lekérésekor ezt a tokent használja, mert másként nem is lehet a szerveren lévő bizonyos adatokhoz hozzáférni. Erről a szakdolgozat további részében részletesebben lehet olvasni.

Ha valamilyen gond adódik, vagy valamit nem írtunk be helyesen, akkor az applikáció egy felugró ablak segítségével értesít a megfelelő információkról, vagy figyelmeztet, ha gond adódik például a szerver kapcsolattal. Bármi is történik a kliens alkalmazás nem válik instabillá és így az adatvesztésnek, vagy a hibás adat bevitelének is nagyon kevés az esélye. Ez köszönhető a többszintű validációnak is, amelyet szintén kifejtek majd a szakdolgozatom további részében.

Az applikációból való kijelentkezés során a token törlődik a teljes kliens alkalmazásból, nyoma sem marad kliens oldalon a felhasználónak. Ez egy biztonságos működést eredményez, hiszen minden bejelentkezéskor egy teljesen új token jön létre. Így, ha valaki meg is szerezné a tokent valahogy egy újabb bejelentkezés érvénytelenné és használhatatlanná tenné a régit, ezenkívül egy tokennek van egy lejárai ideje, amely a szerverben generáláskor állítódig be.

1. **Felhasznált technológiák**

Itt szeretném összefoglalni a projektem összerakása során felhasznált technológiákat. Amennyire ismereteim engedték megpróbáltam a legmodernebb technológiákat felhasználni, persze a választásaim során a technológiák kiforrottsági szintjét is figyelembe vettem, hiszen egy régebbi technológiához szakirodalmat és bevált módszereket is könnyebben talál az ember.

* 1. **Backend technológiák**

Ebben a fejezetben foglalom össze azokat a technológiákat, amelyek szükségesek voltak a szerver alkalmazás elkészítése során.

### Java

A Java általános célú, objektumorientált programozási nyelv, amelyet a Sun Microsystems fejlesztett. A Java alkalmazásokat jellemzően bájtkód formátumra alakítják, de közvetlenül natív kód is készíthető Java forráskódból. Ebből fejlődött ki később a Kotlin nyelv is.

A programot objektumok alapján csoportosítja, nem az elvégzett műveletek szerint. A nagy projektek könnyebben kezelhetőek, visszaszorítható az elvétett projektek száma. Különböző hardvereken is hasonlóan fut, köszönhető a platformfüggetlenségnek. A platformfüggetlenséget a JVM, azaz a Java Virtual Machine (JVM) biztosítja, amelyet viszont minden egyes platformra külön meg kell írni. A JRE a JVM működése szempontjából szükséges környezetet és egyéb eszközöket biztosít.

### MySQL

Több adatbázist is megvizsgáltam, hogy egy jó technológiát tudjak választani az adatok tárolásához. Utánanéztem többek között az Oracle adatbázisnak és a Microsoft SQL Servernek. Ezeket jól ismertem, mivel elég sokat találkoztam velük iskolai, illetve munkahelyi környezetben, így első gondolatra jó választásnak tűntek. Nagyon stabilok és könnyen kezelhetőek nagy programozói támogatottsággal, csak ezek fizetős alkalmazások. Ezután NoSQL adatbázisokat is megnéztem, mint a MongoDB, de mivel ezek nem tudják garantálni az ACID tulajdonságokat, ezért tovább keresgéltem. Megnéztem melyiknek vannak a legjobb könyvtárcsomagjai, amelyeket kapcsolatba lehet hozni a Springgel és miután több helyen is láttam, hogy a MySQL-t ajánlják, végül ez mellett döntöttem.

A MySQL egy nagyon gyors, többszálú, többfelhasználós és robusztus SQL (Structured Query Language) adatbázis-kiszolgálót biztosít, amely kiforrott technológiai háttérrel rendelkezik. Megtalálható benne a relációs adatmodell egyszerűsége és rugalmassága. Ezenkívül jól skálázható és erős tranzakciós támogatással rendelkezik.

### Spring keretrendszer

A Spring Framework átfogó programozási és konfigurációs modellt biztosít a modern Java-alapú vállalati alkalmazásokhoz – bármilyen telepítési platformon.

A képen asztal látható

Automatikusan generált leírásA fenti táblázatban látható előnyök miatt a Spring közkedveltebbé vált, könnyebben használható és gyorsabban is fejlődik. Az EJB-nek is meg vannak az előnyei, mivel az elosztott tranzakciókat jobban kezeli.

A Spring keretrendszer alapja a függőséginjektálás (Dependency Injection – DI). Függőséginjektáláskor a keretrendszer kezeli a konfigurációban megadott objektumok példányosítását, a keretrendszertől kérjük el a konkrét példányokat, a használt objektum konfigurációs beállítássá válik és nem szükséges újrafordítás sem, csak a konfigurációs fájl szerkesztése. Korábban ezt a koncepciót nevezték Inversion of Control-nak (IoC).

ábra 1 - EJB és Spring

A Spring gerincét adják a Bean-ek, amely a keretrendszer kulcsfogalma. A Bean-ek konfigurációja történhet XML fájllal, újrafordítás sem szükséges. Ezenkívül történhet szórványos annotációkkal és konfigurációs fájlokkal is.

A Spring Annotations lehetővé teszi a függőségek konfigurálását és a függőségek beillesztésének megvalósítását Java programokon keresztül. Pl.: @Bean, @Controller, @RequestBody, @Service, @Repository, @Configuration, @Entity, @Id, @GeneratedValue(strategy = GenerationType.IDENTITY), @ManyToOne, @Email, @PreAuthorize("isAuthenticated()"), @RestController, @Override, @SpringBootApplication. Ez egy nagyon fontos eszköz, hanem a legfontosabb egy Spring alkalmazás elkészítésénél, de már sok programozási nyelvben lehet használni (Pl. Kotlin). Annotációkat meg is lehet írni, akár egy föggvényt, lehet velük velidálni, funkciót lehet adni egy osztálynak, adatbázis szerkezetet megvalósítani stb. Tehát egy nagyon sokoldalú és hasznos eszköz.

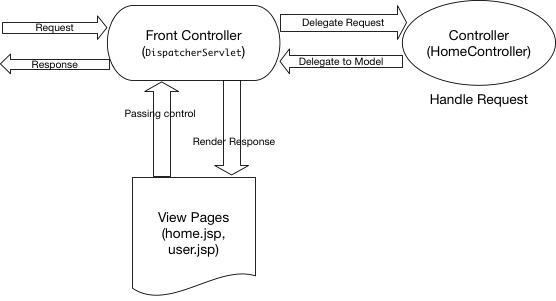
### Spring Boot

A Spring Boot egy háttérrendszer, amely a vállalati Java ökoszisztéma egyik fő szereplőjévé vált. Lehetővé teszi a Java fejlesztők számára, hogy gyorsan, gond nélkül elkezdjenek például webalkalmazásokat készíteni. Igazából ez egy Java-alapú keretrendszer, amelyet egy mikroszolgáltatás létrehozására használnak.

A Spring nagyon sok modult tartalmaz, a valóságban viszont kialakultak ezeknek gyakori részhalmazai: Pl.: webalkalmazások esetén nagyjából ugyanazokat a komponenseket szokás használni. Vannak részek, amelyeket szinte mindegyik Spring alkalmazás használ, Pl.: naplózás.

A Spring Boot a Spring egyfajta tovább gondolása. A Spring Boot lehetővé teszi az olyan szerverek közvetlen beágyazását, mint a Tomcat, a Jetty vagy az Undertow. A Spring Boot leegyszerűsíti az konfigurációkat azáltal, hogy egyedi formálható kezdeti függőségeket biztosít. Egy-egy starter tartalmazza a tipikus, bizonyos tulajdonságú alkalmazás komponenseket. Egy program tetszőleges számú Spring Boot startert használhat. A starterek nevei kezdődhetnek a következőképpen: spring-boot-starter. Pl. a spring-boot-starter-web a webalkalmazások függőségeit tartalmazza, valamint egy beépített webszervert is, ami beállítás nélkül automatikusan elindul. Tehát a Spring Boot egy könnyű keretrendszer, amely leveszi a legtöbb munkát a Spring-alapú alkalmazások konfigurálásából.

### Spring MVC

A Spring MVC a Model-View-Controller architektúrán alapul. Az alábbi kép a Spring MVC architektúráját mutatja magas szinten.

ábra 2 - Spring MVC architektúra

A Front Controller osztály az, amely fogadja az összes kérést és elkezdi feldolgozni azokat. Feladata a kérés átadása a megfelelő Controller osztálynak, és a válasz visszaküldése, amikor a nézetoldalak (View Pages) megjelenítették a válaszoldalt, tehát felelős a Spring MVC alkalmazás áramlásának kezeléséért.

A vezérlő (Controller) egy alkalmazás üzleti logikáját tartalmazza. Itt a @Controller annotációval jelöljük meg az osztályt vezérlőként.

A nézet (View) egy adott formátumban jeleníti meg a megadott információkat.

Egy modell az alkalmazás adatait tartalmazza. Az adat lehet egyetlen objektum vagy objektumok gyűjteménye. Én Json objektumokat használok az adatok továbbítására, de adatokat adok át a fejlécben, vagy úgynevezett Pathvariable-ben is, amikor csak egyszerű változók átadására van szükség.

### Spring Security

A Spring Security egy Java/Java EE keretrendszer, amely hitelesítést, engedélyezést és egyéb biztonsági funkciókat biztosít a vállalati alkalmazások számára. Egy hatékony és nagymértékben testre szabható hitelesítési és hozzáférés-felügyeleti keretrendszer. A projektet 2003 végén „Acegi Security” néven Ben Alex indította el, és 2004 márciusában az Apache License alatt nyilvánosan kiadták. A Spring Security igazi ereje abban rejlik, hogy milyen egyszerűen bővíthető, hogy megfeleljen az egyéni követelményeknek.

Védelmet biztosít az olyan támadások ellen, mint a munkamenetrögzítés, a kattintástörés (clickjacking), a webhelyek közötti kérések hamisítása stb. Servlet API integrációval rendelkezik, és opcionálisan integrálható a Spring Web MVC-vel.

Spring Security segítségével felépíthető egy biztonságos hitelesítési rendszer, amely segítségével a felhasználó akár egy felhasználónév és egy jelszó segítségével biztonságosan lekérhet adatokat a szerverről, anélkül, hogy hitelesítési adatai rossz kezekbe kerülnének. Ehhez nyújt segítséget a JWT.

A JSON Web Token (JWT) egy nyílt iparági szabvány, amelyet két entitás – általában egy kliens (például az alkalmazás frontendje) és egy szerver (az alkalmazás háttérrendszere) – közötti információmegosztásra használnak. JSON-objektumokat tartalmaznak, amelyek rendelkeznek a megosztandó információkkal.

A szerveren a megfelelő beállítások elvégzése után le lehet generálni az adott jelszóból kriptográfia titkosítási módszereket használó tokent, amelyet visszaküldve a kliens alkalmazásnak az már biztonságosan tudja használni akár személyes adatok lekérésére. Egy tokennek be lehet állítani egy időkeretet, amely alatt használható, ezen kívül pedig újbóli bejelentkezés szükséges egy újabb token legeneráláshoz. Én a projektemben Bearer tokent használtam, amelyet a szerver generál egy bejelentkezési kérelemre és a tokent a JSON válasz fejlécében küldi vissza a kliensnek.

* 1. **Kliens technológiák**

Ebben a fejezetben a kliens alkalmazás implementálása során használt technológiákról szeretnék írni.

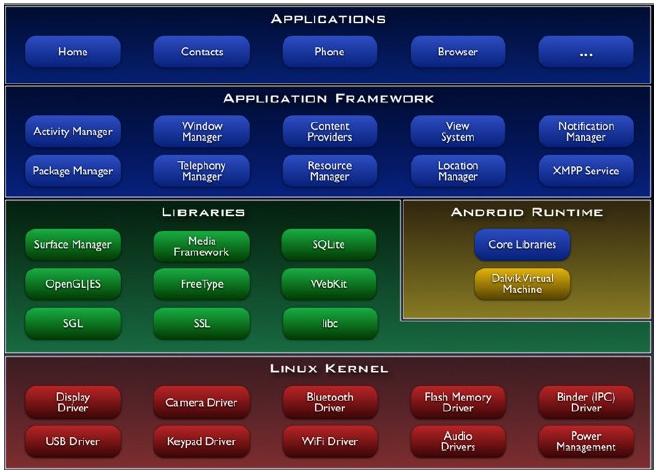
### Android

Az Android egy teljes és modern operációs rendszer. Ez egy linux kernelt használó mobil operációs rendszer, amelyet eredetileg mobil eszközökhöz fejlesztettek ki, de manapság már a legkülönbözőbb helyeken használják fel. Például felhasználják, okostelefenokhoz, táblagépeghez, okos TV – khez, okos órákhoz, háztartási berendezésekhez, autókhoz stb. Az Andoidod kifejezetten alkalmazásokhoz fejlesztették ki.

A Google keze alatt lévő Android op. rendszer egy nyílt forráskódú rendszer, ennek köszönhetem azt, hogy létre tudtam hozni egy kliens-t android telefonra, anélkül, hogy fizetnem kellett volna valamilyen szoftverért. Rengeteg okostelefonon elérhető és jelenleg a piac több mint 72% - át uralja az Android operációs rendszer. Ez egy teljesen nyílt platform fejlesztők, felhasználók és az ipar számára.

Az Android egy többfolyamatos rendszer, amelyben minden alkalmazás (és a rendszer részei) a saját folyamatában fut. Az alkalmazások és a rendszer közötti biztonság nagy része folyamatszinten érvényesül szabványos Linux-szolgáltatásokon keresztül.

Az Android operációs rendszer fontosabb jellemzői a következők:

* Az Android képes több alkalmazást is futtatni egyszerre.
* Támogatja az optimalizált VGA, 2D és 3D grafikát is.
* Lehetővé teszi a komponensek újra felhasználását és cseréjét.
* Az Android támogatja a Java alkalmazásokat.
* Alapértelmezetten mobileszközökre van optimalizálva.

ábra 3 - Android OS architecture

Az Android operációs rendszer szoftverösszetevők halmaza, amely nagyjából öt részre és négy fő rétegre van felosztva, amint a fenti architektúra diagramon látható.

A képen szöveg, képernyőkép, névjegykártya látható

Automatikusan generált leírásA Linux kernel magába foglalja a biztonságot, illetve a memória és folyamatkezelést. Ez egy bevált driver modell. Hatékony számítási kapacitással és erőforrás kezeléssel rendelkezik. Stabil és bevált operációs rendszer mobil platformhoz.

ábra 4 - Linux kernel

A képen szöveg, képernyőkép, névjegykártya látható

Automatikusan generált leírásA Könyvtárak a Linux kernel fölött találhatóak. Ez a réteg különféle C/C++ alapkönyvtárakat és Java alapú könyvtárakat tartalmaz, például Graphics, Media, OpenGL, Surface Manager stb. Az SQLite adatbázis támogatáshoz van. Az SSL (Secure Sockets Layer) egy biztonsági technológia. Az SGL és az OpenGL használható 2D és 3D számítógépes grafikákhoz.

ábra 5 - Könyvtárak és Dalvik VM

A képen szöveg, névjegykártya, képernyőkép látható

Automatikusan generált leírásAz alkalmazás keretrendszere az az eszközkészlet, amelyet minden alkalmazás használ és Android-alkalmazások létrehozására használunk. Ez teljes egészében Java programozási nyelven van megírva. Tartalmaz különböző típusú szolgáltatási tevékenység menedzsereket, értesítéskezelőket, nézetrendszereket, csomagkezelőket stb., amelyek segítik az alkalmazásunk fejlesztését.

ábra 6 - Alkalmazás keretrendszer

Ezeken kívül van az Android Runtime környezet, amely az Android egyik legfontosabb része. Olyan összetevőket tartalmaz, mint a core libraries és a Dalvik virtual machine (DVM). Főleg az alkalmazás keretrendszer alapját adja, és core libraries segítségével működteti az alkalmazásunkat. A Java virtuális géphez (JVM) hasonlóan a Dalvik Virtual Machine (DVM) is egy regiszter alapú virtuális gép, amelyet kifejezetten androidra terveztek és optimalizáltak annak biztosítására, hogy egy eszköz több példányt is hatékonyan tudjon futtatni.

Az Android keretrendszer tetején alkalmazások találhatók. Minden alkalmazás, mint például az otthoni, beállítások, játékok, böngészők, android keretrendszert használ, amely android futási környezetet és könyvtárakat használ. Az Android futtatókörnyezete és a natív könyvtárak Linux kernelt használnak.

### Kotlin

A Kotlin erősen típusos programozási nyelv, amely Java virtuális gépre és JavaScript kódra is lefordítható. Ez egy modern programozási nyelv, amelyet a professzionális Android-fejlesztők több mint 60%-a használja, és amely segít a termelékenység, a fejlesztői elégedettség és a kódbiztonság növelésében.

A Kotlin programnyelvet 2011 júliusában hozták nyilvánoságra, amit akkor már egy éve fejlesztettek. A fordítót és a hozzá tartozó programokat 2012 februárjában adták ki nyílt forráskódú szoftverként Apache 2.0 licenc alatt. A fő fejlesztői a JetBrains szentpétervári csapata, a nyelv a Szentpétervár közelében található Kotlin-szigetről kapta a nevét. A Jetbrains bevallott motivációja az új nyelv fejlesztésében az, hogy az növelje az IDEA fejlesztőeszköz eladásait.

A Kotlin jobban támogatja az Android fejlesztését, mint a Java, ezenkívül az erősen típusos tulajdonságát is nagyon előnyösnek találom, ezáltal a hibakeresés és a kód olvashatósága is javul. Ezért egy Android kliens alkalmazás elkészítéséhez megfelelőnek találtam.

### Retrofit2

A Retrofit egy típusbiztos REST kliens Androidra, Java-ra és Kotlinra, amelyet a Square fejlesztett ki. A könyvtár hatékony keretrendszert biztosít az API-k hitelesítéséhez és interakciójához, valamint hálózati kérések küldéséhez az OkHttp-vel. Amellett, hogy elegáns szintaxist biztosít, könnyen beilleszthető különböző könyvtárakba.

A Retrofit 2.0-ás verziója előtt, ha a lekért választ nem sikerült elemezni a meghatározott objektumban, a rendszer a hibát hívja meg. A Retrofit 2.0-ban, függetlenül attól, hogy a válasz értelmezhető-e vagy sem, az onResponse mindig meghívásra kerül. Abban az esetben viszont, ha az eredmény nem elemezhető az objektumba, akkor a response.body() nullként tér vissza. Bár az el kell mondjam, hogy miközben ezt használtam, volt olyan eset is, hogy egyszerűen kilépett a függvényből. Ezt külön le kellett kezeljem a kliens alkalmazásomnál.

### Okhttp3

A HTTP a modern alkalmazások hálózati kommunikációjának a módja. Így cserélünk adatokat és médiát. A HTTP hatékony végrehajtása gyorsabbá teszi a dolgok betöltését, és sávszélességet takarít meg.

Az OkHttp egy HTTP-kliens, amely alapértelmezés szerint hatékony:

* HTTP/2 támogatás lehetővé teszi, hogy minden kérés ugyanahhoz a gazdagéphez osszon meg egy socketet.
* A kapcsolatkészlet (Connection pooling) csökkenti a kérés késését (ha a HTTP/2 nem érhető el).
* A GZIP csökkenti a letöltési méreteket.
* Végül pedig a válaszok gyorsítótárazása (cache) teljesen elkerüli a hálózatot az ismétlődő kérések esetén.

Ha a hálózat problémás, az OkHttp csendben helyreáll az általános csatlakozási problémákból. Ha a szolgáltatásnak több IP-címe van, az OkHttp megpróbál alternatív címeket adni, ha az első csatlakozás sikertelen.

Használata egyszerű, request/response API-ja gördülékeny építőkkel és stabilitással készült. Támogatja a szinkron hívásokat és az aszinkron hívásokat is a visszahívásokkal (callbacks).

## Tooling

Ebben a fejezetben foglalnám össze azokat az eszközöket, amelyeket a fejlesztés során használtam.

### GIT

A Git egy nyílt forráskódú, elosztott verziókezelő szoftver, vagy másképpen egy szoftverforráskód-kezelő rendszer, amely a sebességre helyezi a hangsúlyt, így a kicsitől a nagyon nagy projektig mindent gyorsan és hatékonyan kezel. A Git könnyen megtanulható, és villámgyors teljesítménnyel rendelkezik. Felülmúlja az olyan SCM-eszközöket, mint a Subversion, CVS, Perforce és ClearCase olyan funkciókkal, mint az olcsó helyi elágazás, kényelmes állomásozási területek (staging area) és többféle munkafolyamat.

### Visual studio code

A Visual Studio Code egy ingyenes, nyílt forráskódú kódszerkesztő, melyet a Microsoft fejleszt Windows, Linux és OS X operációs rendszerekhez. Támogatja a hibakeresőket, valamint beépített Git támogatással rendelkezik, továbbá képes az intelligens kódkezelésre az IntelliSense segítségével.

A Visual Studio Code az Electron nevű keretrendszeren alapszik, amellyel asztali környezetben futtatható Node.js alkalmazások fejleszthetőek. Ugyanakkor a Visual Studio Code nem az Atom forkja, hanem a Visual Studio Online szerkesztőn alapszik (fejlesztési neve: "Monaco").

### Android studio

Az Android Studio Android-alkalmazásokra optimalizált integrált fejlesztői környezetet (IDE) biztosít az alkalmazáskészítőknek az Android platformra való fejlesztéshez. Alkalmazások készíthetőek itt Android-telefonokhoz, -táblagépekhez, Android okosórákhoz, Android TV-hez és Android Auto-hoz. A strukturált kódmodulok lehetővé teszik, hogy a projektet funkcionalitási egységekre lehessen felosztani, amelyeket önállóan lehet építeni, tesztelni és hibakeresést végezni.

Az Android emulátor az Android studio-ban található. Ez egy olyan számítógépes program vagy hardver, ami más programoknak vagy eszközöknek a környezetét (vagy annak részét) „szimulálja”, vagyis lehetővé teszik az adott rendszerrel nem kompatibilis programok (vagy operációs rendszerek) vagy számítógépek futtatását. A fejlesztés során nagyon hasznos volt, hogy egy virtuális telefonon is le tudtam tesztelni a programomat.

1. **Tervezés**

Ez a projekt egy kliens-szerver alkalmazás. Áll egy folyamatosan működő szerver alkalmazásból, egy adatbázisból, és egy kliens alkalmazásból. A szerver kommunikál az adatbázissal és műveletek a Controller osztályokban vannak definiálva.

Ezeken keresztül lehet nagyon szerte ágazóan utasításokat adni a szervernek valamilyen kliens által. Minden felhasználó rendelkezik személyre szabott objektumok fölött (pl. Contract), és vannak közös objektumok is, amelyeket mindenki lát. Ezek fölött nem rendelkeznek teljhatalmú uralommal. Vannak szerepkörök (role-ok) definiálva: admin, user. Bizonyos műveleteket csakis az admin szerepkörrel rendelkező felhasználó képes elvégezn. Ilyen például a POST /createadmin (új admin felhasználó létrehozása) vagy a POST /vehicle (új járművet lehet beszúrni a táblába). Az egyszerű felhasználók csak használni tudják ezeket az adatokat.

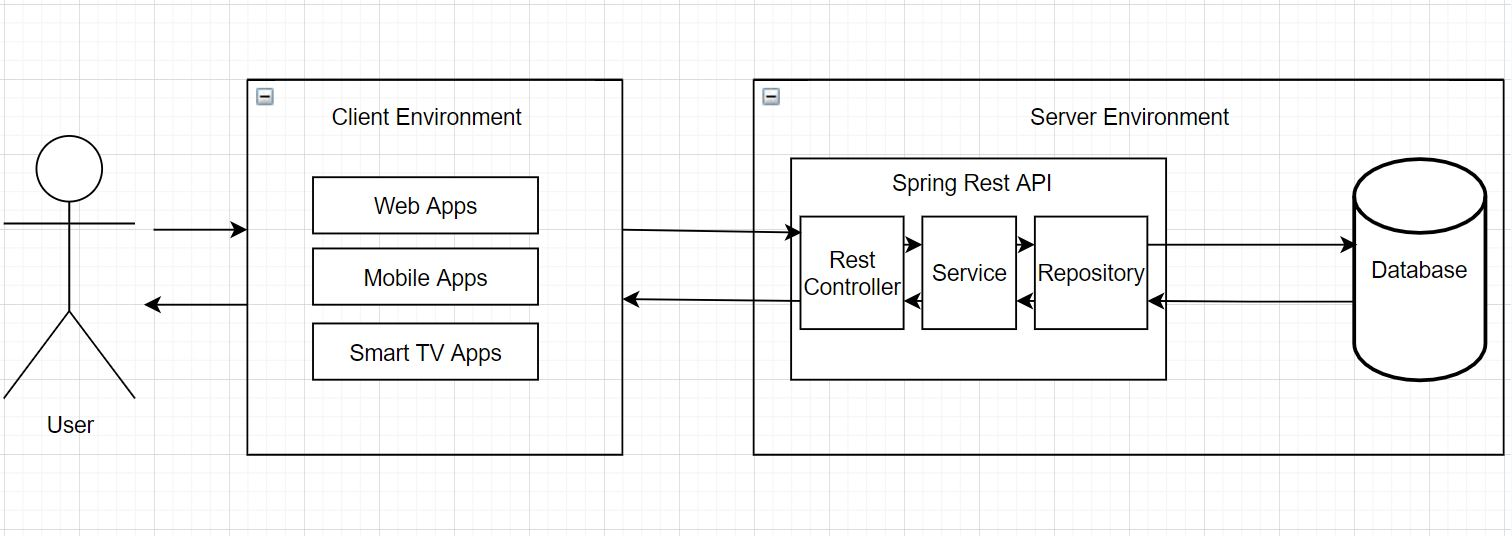
A szerver a következő jellemzőkkel rendelkezik:

* Passzív, a kliensektől várja a kéréseket
* A kéréseket, lekérdezéseket feldolgozza, majd visszaküldi a választ
* Elég nagy számú kliens tud akár egyszerre kiszolgálni a Spring boot tranzakciókezelésének köszönhetően.
* Általában nem áll közvetlen kapcsolatban a felhasználóval, mivel a felhasználó csak a kliens programon keresztül tud kommunikálni a szerverrel és csakis azokat a funkciókat tudja elérni, amely a kliensben le voltak implementálva.

A kliens jelen esetben egy android alkalmazás, amelyben minden fontos funkció elérését implementáltam, így a szerver által nyújtott szolgáltatások nagy hányadát lefedi. A bejelentkezés biztonságos módon van megoldva, a jelszavak már lekódolva vannak elmentve egy Bearer token-ben és bejelentkezés után minden művelet-kor ennek a token-nek a szerverre való felküldése szükséges a műveletek hitelesítéséhez. A regisztrálás (/register), a bejelentkezés (/login) és a szerverrel való kapcsolat ellenőrzése (/welcome) céljából létrehozott rest kéréseken kívül mindig szükséges a fejlécben küldött bearer tokennek a használata. Igazából a kliens a /login kérés sikeres lefutása után kapja meg ezt a tokent visszatérítve szintén a headerben. A token minden sikeres /login esetében újra generálódik szerver szinten.

A felület felhasználóbarát módon van megalkotva, minden fontosabb eseménykor felugró ablak értesíti a felhasználót az esemény sikeres, avagy sikertelen megtörténtéről. A reszponzivitás görgetősávokkal és felülethez alkalmazkodó beimprtált elemekkel van megoldva, így egy kisebb képernyővel rendelkező telefononon, vagy nagyobb képernyővel rendelkező tableten is használhatók az alkalmazás funkciói.

A validációk több szinten is meg vannak valósítva különböző beépített és saját kezűleg megírt függvények segítségével. A kliens szinten vannak előleges validációk, ha ezen túljutunk, akkor a szerver is le ellenőrzi egy függvénnyel, kivételek dobásával vagy annotációkkal. Az adatbázis validációkat és a tranzakciókezelést a spring boot keretrendszer biztosítja.

A java szerver és az adatbázis között automatikus leképezés van Object-Relational Mapping (ORM) segítségével. Ennek az implementálásához hibernate-t használtam. A hibernate fő előnye az Objektum Orientált (OO) szemlélet, amely során a relációs adatbázis elérése lehetségessé válik JDBC vagy SQL kód írása nélkül.

ábra 7 - Szerver-Kliens alkalmazás szerkezete

A kliens-szerver (magyarul: ügyfél-kiszolgáló) a programozásban a főprogram-alprogram viszonynak feleltethető meg. A kommunikációt mindig az ügyfél kezdeményezi, sohasem a kiszolgáló, vagyis a szerver szolgáltatást nyújt a munkaállomások (kliensek) részére. Egy erős hardverrel és jó konfigurációval rendelkező szerver nagyon sok kliens-t ki tud szolgálni egyszerre, amelyek akár többfélék lehetnek: webböngésző, telefon, okos tv, okos óra, hűtő stb.

A kliens-szerver architektúra legalapvetőbb formájában mindössze kétfajta csomópont (node) van, a kliens és a szerver. Ezt az egyszerű architektúrát két szintűnek (angolul two-tier) hívják.

Bonyolultabb architektúrák is léteznek, amelyek 3 különböző típusú csomópontból állnak: kliensből, alkalmazás szerverből (application server) valamint adatbázis szerverből (database server). A háromszintű kiépítésben az alkalmazásszerverek azok, amelyek kiszolgálják a kliensek kéréseit, és az adatbázisszerverek az alkalmazásszervereket szolgálják ki adatokkal. Ennek a rendszernek nagy előnye a bővíthetőség. A rendszer így képes lesz egyensúlyozni és elosztani a feldolgozásra váró adatmennyiséget és munkát a több és gyakran redundáns, specializált csomópont között. Hátránya, hogy nagyobb az adatátviteli forgalom a hálózaton és hogy nehezebben programozható, illetve tesztelhető egy kétszintű architektúránál, mert több eszközt kell összehangolni a kliensek kéréseinek kiszolgálásához.

## Kommunikáció és a REST

A kommunikációhoz Rest API-t használok. Ez egy alkalmazásprogramozási felület (API vagy webes API), amely megfelel a REST architektúra stílusának. A REST a reprezentatív állapottranszfer rövidítése, és Roy Fielding informatikus vezette be és definiálta 2000-ben a doktori disszertációjában. Azokat a rendszereket, amelyek eleget tesznek a REST megszorításainak, "RESTful"-nak nevezik. Tehát ez egy szoftver architektúra típus, amely lehetővé teszi skálázható, nagy teljesítményű elosztott hálózati alkalmazások fejlesztésére.

A REST megszorítások a következők:

* kliens-szerver modell,
* egységes interfész,
* állapotmentes kommunikáció,
* réteges felépítés,
* gyorsítótárazhatóság
* kiegészíthetőség (code on demand)

## CRUD és a REST

A számítógép-programozásban a létrehozás, az olvasás, a frissítés és a törlés az állandó tárolás négy alapvető művelete. REST környezetben a CRUD gyakran a POST, GET, PUT és DELETE HTTP metódusoknak felel meg. Ezek a tartós tárolási rendszer alapvető elemei és ezekről szeretnék kicsit részletesebben beszélni. A példákat a saját projektemből fogom venni.

**Létrehozás (POST)**: REST környezetben leggyakrabban a http POST metódust használjuk erre. Például képzeljük el, hogy egy új felhasználót szeretnénk létrehozni. Ehhez egy <http://localhost:8085/register> POST kérést kell küldeni a szervernek egy JSon objektum kíséretében. A JSon objektum a User modell kikötéseinek kell eleget tegyen, különben nem fogadja el a szerver. Ha elfogadta visszaküldi a kész objektumot és CREATED (201) http válasz érkezik a szervertől. A hibás válaszok minden esetben szerver szinten le vannak kezelve.

Request JSon:

{

"firstName": "Laszlo",

"lastName": "Biro",

"username": "biro",

"email": "birolaszlo@edu.bme.hu",

"password": "12345",

"phoneNumber": "0612345678",

"homeAddress": {

"city": "Budapest",

"country": "Magyarorszag",

"county": "Lagymanyos",

"streetName": "Baross",

"number": 1,

"zipCode": "1117",

"door": "A1"

}

}

Response code: 201

Response JSon:

{

"id": 3,

"username": "biro",

"password": "$2a$10$Szf3K7BeRD9W74G7dbJOBelN5hLfHwiZd9JTDmuflLVf/6wNvJ6IW",

"email": "birolaszlo@edu.bme.hu",

"firstName": "Laszlo",

"lastName": "Biro",

"phoneNumber": "0612345678",

"homeAddress": {

"id": 3,

"country": "Magyarorszag",

"county": "Lagymanyos",

"city": "Budapest",

"zipCode": "1117",

"streetName": "Baross",

"number": "1",

"door": "A1"

},

"billingAddress": null,

"contract": {

"id": 3,

"vehicles": [],

"startDate": null,

"endDate": null

},

"roles": [

{

"id": 2,

"name": "ROLE\_USER"

}

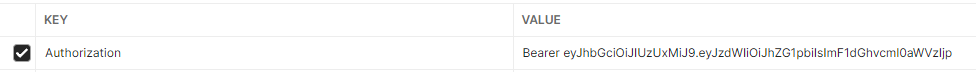
]

}

Látható a válaszból, hogy létrejön egy billingAddress is, amely opcionálisan kitölthető objektum regisztráláskor és még két másik objektum. Az első a Contract, amely minden felhasználónal egyedi, ezzel lehet járművet bérelni és a második, hogy mindenkinek van valamilyen szerepköre, amely a Roles objektumban tárolódik el. Ez utóbbi közös objektum a többi felhasználóval, vagy admin-nal.

**Olvasás (GET)**: Az erőforrások REST környezetben történő olvasásához a GET metódust használjuk. Egy forrás elolvasása soha nem változtathat meg semmilyen információt – csak lekérheti azt. Ha egymás után tízszer hívja a GET-et ugyanazon az információn, akkor az első híváskor ugyanazt a választ kell kapnia, mint az utolsó hívásnál. Például a <http://localhost:8085/welcome> kérésre a “Hello RentApp” karakterlánc érkezik válaszként 200 (OK) állapot kóddal.

**Frissítés (PUT)**: Az update művelettel lehet frissíteni az adatbázis adatait.

Például, ha egy új járművet szeretnék hozzáadni a szerződésünkhöz, akkor egy <http://localhost:8085/contract/1> PUT kérést kell leadni, amely azt jelenti, hogy az 1-es id-val rendelkező járműt szeretnénk hozzáadni a kölcsönzési listánkhoz. Persze ilyenkor soha nem szabad elfeljteni a tokent a fejlécből, amint a lenti ábrán látható.

ábra 8 - Header tokennel a postmanben

Siker esetén válaszként egy 200 as állapotkódot és egy Contract modell szerinti JSon kell kapni:

{

"id": 1,

"vehicles": [

{

"id": 1,

"name": "f40",

"brand": "Ferrari",

"price": 5,

"description": "Ez egy szép auto.",

"category": "CAR"

}

],

"startDate": null,

"endDate": null

}

**Törlés (DELETE)**: A CRUD Delete művelet a DELETE HTTP metódusnak felel meg. Egy adat, adatok eltávolítására szolgál a rendszerből. Példaként vegyük azt, hogy egy járművet már nem áll módomban kikölcsönözni, mert már kicsit elavult. Ebben az esetben egy Ikarus buszról van szó, amely több mint 50 éves és már veszélyessé vált.

Ezért a <http://localhost:8085/vehicle/6> DELETE kérést küldöm a rendszernek. Ennek hatására a 6-os id-val rendelkező jármű törlődni fog a rendszerből. Természetesen most sem szabad elfelejteni a token-t a fejlécből.

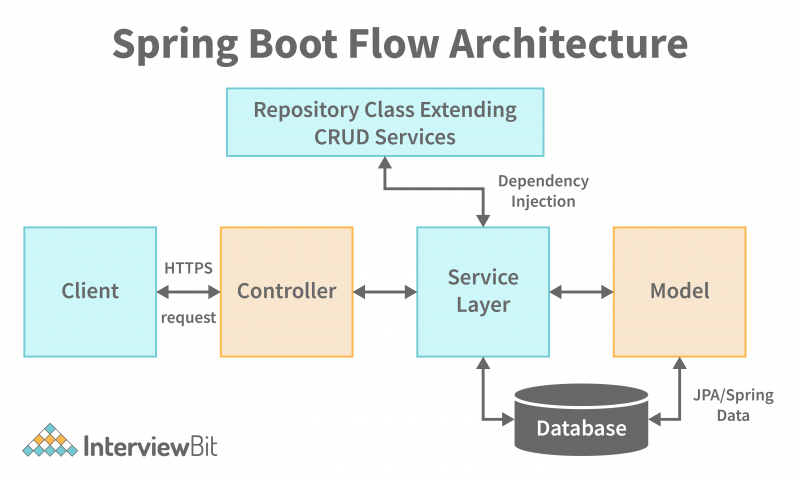
Válaszként egy 204-es állapotkódot fogunk kapni, amelynek jelentése no content, vagyis már nincs megjeleníthető adat. Ezzel jelezve a kliensnek, hogy sikeresen végrehajtódott a kérés és nem létezik jelenleg 6-os id-val rendelkező jármű az adatbázisban.

1. **Implementáció**

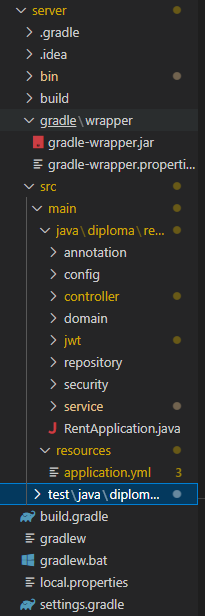
Az implementáció részletezéséhez az implementációt két részben mutatom be, mivel a fejlesztés során is két jól elválasztható alkalmazás implementálása történt.

## Szerver alkalmazás

A szerver fejlesztéséhez alapvetően a Spring boot keretrendszer által adott lehetőségeket használtam fel, amely Tomcat szerveren fut. A Gradle projektépítő eszközt használtam a függőségek behúzására.

Elkészítéséhez a Spring Boot Worklflow Architektúrát használtam fel, amelynek a szerkezete a lenti ábrán látható.

ábra 9 - Spring Boot Workflow Architecture

A szerver főosztálya a RentApplication.java, amelyet @SpringBootApplication annotációval láttam el, ezzel jelölve, hogy egy Spring boot keretrendszert használó alkalmazásról van szó. Elindításhoz a legegyszerűbb mód, ha Visual Studióban jobb klikkel rákattintunk a főosztályra, majd Run Java-ra kattintunk bal egérgombbal. Ha lefutottak a konfigurációs fájlok is, akkor a szerver sikeresen elindult.

ábra 10 - Kliens fájlszerkezete

Az ügyfél HTTP kérést küld (GET, POST, DELETE stb.), amely Rest elvek alapján történik. A http kérés továbbításra kerül a Controller-nek. A controller leképezi a kérést, feldolgozza a leírókat (handlers) és meghívja a szerver logikáját a Service rétegen keresztül. A szerver üzleti logikája a szolgáltatási (service) rétegben található. A Spring boot végrehajtja az összes logikát az adatbázison, amely le van képezve a Java Persistance Library modellekben. A Repository biztosítja az objektumok tárolási, visszakeresési, keresési, frissítési és törlési műveleteinek mechanizmusát, röviden a CRUD mechanizmusokat.

A képen szöveg látható

Automatikusan generált leírásEgyszerübben tehát a Controller fogadja a kérést, továbbítja a Service rétegnek, amely kommunikál a többi réteggel. A Model tárolja az adatbázis táblák szerkezetét és kapcsolatait perzisztens módon, tehát az adatbázis reagál a szerkezeti változásokra. A Repository pedig megmutatja a Service-nek, hogy miként tud kommunikálni az adatbázissal. Az application.yml fájlban konfigurációs adatok megadására van lehetőség hierarchikus formában.

ábra 11 - Modell fájlok

### A domainek

Ezek azok a perzisztens modellek, amelyek az adatok szerkezetét tárolják Java osztályokban. Minden ilyen osztály @Entity annotációval kell ellátni. Összesen 5 modell osztályt hoztam létre, ezenkívül található itt két enum, illetve az Email küldéséhez használt modell objektum.

Példaként a User entitást mutatnám be:

@Entity

@Table( name = "users",

uniqueConstraints = {

@UniqueConstraint(columnNames = "username"),

@UniqueConstraint(columnNames = "email")

})

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@NotBlank

@Size(min= 3, max = 24)

private String username;

@NotBlank

@Password

private String password;

@NotBlank

@Email

private String email;

@NotBlank

private String firstName;

@NotBlank

private String lastName;

@NotBlank

private String phoneNumber;

@NotNull

@OneToOne

private Address homeAddress;

@OneToOne

private Address billingAddress;

@NotNull

@OneToOne

private Contract contract;

@ManyToMany(fetch = FetchType.LAZY)

@JoinTable( name = "user\_roles",

joinColumns = @JoinColumn(name = "user\_id"),

inverseJoinColumns = @JoinColumn(name = "role\_id"))

private Set<Role> roles = new HashSet<>();

Látható, hogy a @Table annotációval megadom, hogy milyen néven szerepeljen a tábla az adatbázisban, illetve egyediség megszorítást is rakok a username és az email oszlopokra, vagyis ezekbe az oszlopokba minden bekerülő érték egyedi kell legyen és ez a vizsgálat adatbázis szinten történik. Az id automatikusan generált, tehát nem kell megadni, amely a @GeneratedValue annotációnak köszönhető. Ezenkívül vannak validációk is: @NotBlank – nem lehet üres, @Size(min,max) – karakterszámra vonatkozik, @Password – ez egy általam írt annotáció a jelszóra, @Email – ez egy beépített annotáció és az email standard formátumát ellenőrzi. A @OneToOne 1 az 1-hez típusú adatbáziskapcsolatot jelöl, illetve a @ManyToMany jelentése több a többhöz, de természetesen létezik még a @ManyToOne és a @OneToMany kapcsolatra is annotáció. A FetchType.LAZY jelöli azt, hogy amikor betölt a rendszer egy adatbázist, akkor a hozzá tartozó adatbázis adatokat is rögtön (serényen) betöltse (EAGER), vagy csak akkor töltse le amikor ez irányú kérés érkezik a szervertől, vagyis lustán (LAZY). Van még egy @JoinTable annotáció is a hozzá tartozó @JoinColumn-okkal, amelyekkel az SQL nyelvek adatbázis kapcsolatait lehet leírni.

### A képen szöveg látható Automatikusan generált leírásA Controller osztályok

ábra 12 - Vezérlő fájlok

A Spring Boot rendszerben a vezérlőosztály feladata a bejövő REST API kérések feldolgozása, a modell elkészítése és a nézet visszaküldése válaszként. A Spring vezérlőosztályait a @Controller vagy a @RestController annotáció jelöli. Ahhoz, hogy kéréseket API kérésekkel lehet elvégezni, amelyhez szükség van egy baseURL-re. Ebben az esetben a baseURL a <http://localhost:8085>. Ehhez adódik a @RequestMapping változójába beírt karakterlánc, illetve a Mapping annotáció (@GetMapping, @PostMapping, @PutMapping és @DeleteMapping) változójába beírt karakterlánc. Az előbbit osztály fölé, az utóbbit pedig függvények fölé lehet elhelyezni. A Controller osztályok a Service osztályokkal kommunikálnak, tehát a Service osztályok függvényeit használják a kérések feldolgozásához. Az adatbázis műveletek elvégzéséhez pedig a Service osztályok használják a Repository osztályok függvényeit.

Példaként a UserController osztályt szeretném bemutatni:

@RestController

@RequestMapping(path = "/user")

@PreAuthorize("isAuthenticated()")

public class UserController {

Logger logger = LoggerFactory.getLogger(UserController.class);

private final UserService userService;

@Autowired

public UserController(UserService userService) {

this.userService = userService;

}

@GetMapping

public ResponseEntity<User> getUser() {

Authentication auth = SecurityContextHolder.getContext().getAuthentication();

String username = (String)auth.getPrincipal();

User user = userService.getUserByUsername(username);

logger.info("Response user");

return new ResponseEntity<User>(user, HttpStatus.OK);

}

@PutMapping

public ResponseEntity<User> updateUser(@RequestBody User user){

Authentication auth = SecurityContextHolder.getContext().getAuthentication();

User currentUser = userService.getUserByUsername((String)auth.getPrincipal());

user = userService.updateUser(currentUser, user);

logger.info("User updated");

return new ResponseEntity<User>(user, HttpStatus.OK);

}

@DeleteMapping

public ResponseEntity<User> deleteUser(){

Authentication auth = SecurityContextHolder.getContext().getAuthentication();

String username = (String)auth.getPrincipal();

userService.deleteUserByUsername(username);

logger.info("User " + username + " deleted");

return ResponseEntity.status(HttpStatus.NO\_CONTENT).build();

}

}

A @RestController annotáció jelzi, hogy egy vezérlő osztály speciális változatáról van szó, amely tartalmazza a @Controller és a @ResponseBody megjegyzéseket, és ennek eredményeként leegyszerűsíti a vezérlő megvalósítását. A @RequestMapping path változója segítségével állítjuk be, hogy csakis a <http://localhost:8085/user> kéréseket fogadja ez az osztály. A @PreAuthorize("isAuthenticated()") azt jelenti, hogy ezeket a kéréseket csak akkor dolgozzuk fel, ha a felhasználó be van jelentkezve. Ami azt jelenti, hogy a kérés indításakor el kell küldeni a fejlécben, azon belül az Authorization mezőben a bejelentkezéskor generált token-t. Ezek után következik a logger változó, mely egy slf4j függőségből ered, és Java naplózási API-t biztosít. Ezután példányosítom a UserService osztályt, amelyben az üzleti logika van implementálva, majd jön a Constructor függvény, amelyhez egy UserService kell. Majd a végén következnek a függvények, amelyek lekezelik a GET, a PUT, illetve DELETE kéréseket a @GetMapping, a @PutMapping és a @DeleteMapping annotációkkal. Értelemszerűen az első lekéri az aktuálisan bejelentkezett felhasználó adatait egy User objektumba, a második frissíti a felhasználóhoz tartozó User objektumot a kéréskor felküldött JSon objektum szerint, a harmadikkal pedig a felhasználó saját magát tudja kitörölni. A POST azért hiányzik, mert az regisztráláskor van:

@PostMapping("/register")

public ResponseEntity register(@RequestBody User user) {

try {

logger.info("/register", user);

return new ResponseEntity<User>(userService.createUser(user), HttpStatus.CREATED);

} catch (ValidationException e) {

logger.warn(e.getMessage());

return new ResponseEntity<String>(e.getMessage(), HttpStatus.BAD\_REQUEST);

}

}

A @RequestBody annotáció jelen esetben azt jelenti, hogy a kérés leadásakor egy JSon objektumot is fel kell küldeni, amely megfelel a User Entity modell objektumnak. A UserService createUser(user) metódusát használja, amely siker esetén visszatéríti az elkészült User-t, különben kivétel keletkezik. A választ a ResponseEntity függvénnyel generáljuk, amely a http státusz kód mellett egy objektumot, vagy tetszés szerint szöveget is vissza tud téríteni. HttpStatus kódok közül az OK a 200-nak, a CREATED a 201-nek, a NO\_CONTENT a 204-nek, a BAD\_REQUEST pedig a 400-nak felel meg.

### A Service osztályok

A képen szöveg látható

Automatikusan generált leírásA Controller osztályok ezeket az osztályokat hívják meg, hogy fel tudják dolgozni a klienstől kapott HTTP kéréseket. Ezek az osztályok @Service annotációval vannak ellátva. Ezekben az osztályokban implementáltam a szerveralkalmazás üzleti logikáját.

ábra 13 - Szolgáltatás fájlok

Példaként a VehicleService osztályból szeretnék bemutatni részleteket:

@Service

public class VehicleService {

private final VehicleRepository vehicleRepository;

private final ContractRepository contractRepository;

@Autowired

public VehicleService(VehicleRepository vehicleRepository, ContractRepository contractRepository) {

this.vehicleRepository = vehicleRepository;

this.contractRepository = contractRepository;

}

public List<Vehicle> getVehicles() {

return vehicleRepository.findAll();

}

public void deleteVehicleById(Long id){

Vehicle vehicle = vehicleRepository.getById(id);

List<Contract> contracts = contractRepository.findAllByVehiclesContains(vehicle);

for (Contract contract : contracts) {

contract.setVehicles(new ArrayList<Vehicle>());

contractRepository.save(contract);

}

contractRepository.flush();

vehicleRepository.deleteById(id);

}

A fenti kódrészletben látható, hogy @Service annotációval jelöljük az osztálynak a funkcióját. Behívjuk a VehicleRepository és ContractRepository osztályokat és a segítségükkel adatbázis műveleteket végzek el. Az első függvény például visszatéríti a az összes járművet, amelyhez a findAll függvényt hívja meg. A második függvény vehicleId alapján töröl elemet a Vehicle táblából, de a biztonság kedvéért a Contract táblát is módosítjuk. Ez úgy történik, hogy megkeressük az összes szerződést, amelynek a járműlistájában szerepel ez a jármű, amit törölni szeretnénk. A biztonság kezdéért ezekből a szerződésekből töröljük a járműveket, hogy később a kölcsönzéskor ne okozhasson problémát. Ehhez a ContractRepository segítségével hozzáférünk az adatbázishoz, lekérünk majd mentünk, és ezek után elvégezzük a tényleges törlést a Vehicle táblában. A kódban látható egy contractRepository.flush() hívás, amelynek azért kell, hogy szerződésmódosítás azonnal megtörténjen, de mindenképp a Vehicle táblából való törlés előtt.

### A Repository osztályok

A képen szöveg látható

Automatikusan generált leírásA Spring boot - ban ez az adatbázis függvények tároló osztálya, amelyben rengeteg előre definiált adatbázis műveletet tudunk használni, de saját magunk is tudunk új függvényeket implementálni. Ezek az osztályok @Repository annotációval vannak ellátva. Összesen 5 Repository osztályt készítettem, minden modell osztálynak egyet.

ábra 14 - Repository fájlok

Példaként a UserRepository osztályt mutatnám be:

@Repository

public interface UserRepository extends JpaRepository<User, Long> {

Optional<User> findByUsername(String username);

@Transactional

Long deleteByUsername(String username);

Boolean existsByUsername(String username);

Boolean existsByEmail(String email);

boolean existsByRolesIn(Set<Role> roles);

}

A fenti kódban látható, hogy a @Repository annotációval kezdünk, amely jelzi a Spring boot-nak, hogy tároló osztályról van szó. Az automatikus függvények generálása a JPARepository kiterjesztésnek köszönhető, amelynek első változója a modell, amire vonatkozik, a második pedig ehhez a modellhez tartozó egyedi azonosítónak (ID) a típusa. Jelen esetben ez User és Long, mivel a User Entity modell hez rendel hozzá meghívható műveleteket, amely osztály id-jának a típusa Long.

Tehát amint említettem korábban vannak automatikusan létrejövő függvények, mint a findAll(), amely az összes tárolt objektumot visszatéríti, de lehet kézzel is megírni függvényeket, amint a fenti kódban látható. Ezek a függvények igazából csak interfészként vannak létrehozva, ezzel tudatva a JPARepository – t, hogy ezekre is szükség lesz. Ezeket is a JPARepository a nevük alapján implementálja, ezzek könnyítve meg a fejlesztő dolgát. Így a legtöbb esetben nem kell jpql kódot sem írni, elég a megfelelő szavakat a megfelelő sorrendben “megfogalmazva” leírni, hogy mit szeretnénk és a JPARepository fogja tudni értelmezni, ha jól írtuk meg. Persze ennek használata korlátozott, de nekem a szerver írása alatt elégnek bizonyult ez az eljárás.

### A biztonság

A képen szöveg látható

Automatikusan generált leírásA képen szöveg látható

Automatikusan generált leírásA biztonságot a JSON Web Token (JWT) internetes szabvány, illetve a Spring Security hitelesítést, engedélyezést és egyéb biztonsági funkciókat biztosító keretrendszerének a segítségével valósítom meg. Itt történik a jelszó kódolása, amelyet a BCryptPasswordEncoder(Int strength) függvénnyel végzek, amely egy kriptográfiai titkosítást végez. A strength változó pedig a titkosítást erősségét jelzi, amelyet lehet állítani 4 és 31 között. Értelemszerűen a magasabb szám biztonságosabb, de több időbe is kerül létrehozni. Én 10-es értéket választottam. Ezzel létrejön a token, amelynek az időtartama mindig bejelentkezéskor adódik meg, amelyet 2 napra állítottam be:

ábra 16 - Biztonsági funkciókat ellátó fájlok

ábra 15 - JWT konfigurációs fájlok

String token = Jwts.builder()

.setSubject(authResult.getName())

.claim("authorities", authResult.getAuthorities())

.setIssuedAt(new Date())

.setExpiration(java.sql.Date.valueOf(LocalDate.now().plusDays(2)))

.signWith(secretKey)

.compact();

### A validációk

Az ellenőrzések szerver szinten sajátkezűleg megírt függvényekkel történnek, de a modell osztályokban annotációkkal is sok ellenőrzés van elvégezve. Ilyenek például: @Email, @Password, @NotBlank, @Size(min,max) stb. Adatbázis szintű megszorítások is itt a Java kódban vannak megírva. Az alábbi példakódban egyediségre vonatkozó megszorítások vannak.

uniqueConstraints = {

@UniqueConstraint(columnNames = "username"),

@UniqueConstraint(columnNames = "email")

}

Gradle segítségével beépítettem a javax validation csomagot is, amely tartalmazza a bean validation API-kat, amelyek leírják a JAVA Bean-ek előre programozott ellenőrzését.

## Kliens alkalmazás

Kliens alkalmazásként egy Android telefonos applikációt készítettem. Először React-native programozási nyelvre gondoltam, mivel abban nagyobb a tapasztalatom. Miközben a háttérkutatást végeztem kicsit jobban elmélyedtem más technológiákban is, így mélyültem el jobban a Java alapú Kotlin programozási nyelvben, annak is a mobil applikációval kapcsolatos vonatkozásában. A nyelv rendelkezik a Java minden előnyével, konkrétan használhatja a Java könyvtárakat, és ezenkívül a nyelvezete nagyon könnyed, könnyen átlátható kód keletkezik. Egy modern programozási nyelv, és akár Java kódra is átalakítható, illetve fordítva. Ezenkívül az egyetemen is tanultam egy rövid ideig. Ez elég is volt nekem, hogy ebben kezdjek el egy projektet, amelyet sikeresen egy használható állapotba hoztam összekötve a Spring boot szerver alkalmazásommal. A Kotlin felhasználói felületéhez XML kódot lehet használni. Mivel az XML bővíthető és nagyon rugalmas, sok különböző dologra használják, beleértve az Android-alkalmazások felhasználói felületének meghatározását.

Ez az alkalmazás arra hivatott, hogy egyszerű és felhasználó barát módon adjon lehetőséget kibérelni egy járművet. Összesen hét kategória közül válogathatunk: autó, roller, kerékpár, teherhordó jármű, kamion, busz és motorbicikli. A lehetőségek között böngészhetünk, megnézhetjük őket kategória szerint, részleteket is megtekinthetünk róla és közben, ha szeretnénk az ADD gomb segítségével felvehetjük a listánkra akár lista nézetben, akár a tulajdonságok oldalon tartózkodunk.

Megtekinthetjük a kész listát, amelyet módosíthatunk, kitörölhetünk belőle, de van lehetőségünk az egészet is kitörölni egyetlen gombnyomással és elölről kezdeni az egészet. Ha megvan a lista, akkor bérlési felületen kiválasztjuk a dátumot amikortól ki szeretnénk venni, illetve amikor visszahozzuk. Ezután egy gombnyomással létrejön a szerződés. A szerződés adatait egy emailben kapja meg a felhasználó, amelyben részletesen ki van listázva, hogy mit vásárolt, melyik járműnek a bérlése mennyibe kerül naponként és összesen. Az utolsó sorban egy összegzés található, hogy mekkora a teljes összeg, amelyet ki kell fizetni. Ezek az összegek függenek az egyes járművek napi bérlési díjától és természetesen attól, hogy hány napra béreltük ki őket.

### A kliens applikáció felépítése

Alapvetően az MVC architektúrát használtam. A funkcionalitást a legjobb tudásom szerint építettem be. Utána olvastam modernebb architektúráknak is, mint az MVP vagy az MVVM, amelyek jobb hatásfokkal rendelkeznek.

A modell-nézet-prezenter szoftvertervezési minta az MVC-mintából alakult ki. Az MVP-modell főleg olyan alkalmazásokban nyújt jelentős előnyöket – hasonlóan az MVC-hez –, ahol komplex adathalmazokon kell műveleteket végezni, és ezek eredményeit a felhasználó elé tárni. A modell-nézet-nézetmodell minta pedig egy architekturális minta. A minta leválasztja a grafikus felhasználói felületet és az üzleti logikát. A nézetmodell értékkonverter, ez a felelős az adatok átalakításáért a könnyű kezelhetőséghez és reprezentálásához.

Tudom, hogy például egy MVVM architektúra segítségével optimálisabb alkalmazást tudtam volna létrehozni, de ezt nem tanultam az iskolában, így inkább egy olyan architektúrát választottam, amelyben otthonosabban mozgok. Végül összeraktam egy működő alkalmazást egy általam jól ismert architektúrában, habár lehet nem ez volt a legoptimálisabb megoldás. Ennek ellenére nagyon stabilnak éreztem, és ahol lehet felhasználóbarát módon lekezeltem a történéseket, beépített felugró ablakokkal láttam el az alkalmazást.

A modelleknek felelnek meg a modell objektumok, amelyek az adatok küldésekor, illetve fogadásakor használatosak a kérésekben. A nézetek (view) kotlinhoz igazított xml struktúrákban vannak megalkotva, amelyek meghatározzák az oldalak kinézetét. A controllert pedig a nézetek-hez megírt Activity osztályok alkotják, amelyek a működésért felelősek, vagyis az alkalmazás backendjét adják.

#### Activity osztályok

A képen szöveg látható

Automatikusan generált leírásEzekben az osztályokban van implementálva az oldalak back-end-je. Itt van megírva, hogy milyen eseményre milyen logika fusson le, melyik Rest API kérés történjen meg. Tehát ezek az osztályok kezelik le a szerverrel való a kapcsolatot és írják le a belső működést. Minden oldalhoz pontosan egy Activity osztály tartozik.

ábra 17 - Felüleletek Controller osztályai

A képen szöveg látható

Automatikusan generált leírás

ábra 18 - Modell osztályok

#### Modell osztályok (models)

Ezek most nem a perzisztens adattárolás miatt vannak, mint ahogy a szerver alkalmazásban látható volt, hanem az adatok fogadásához, illetve küldéséhez. Itt egy JSon konverzió történik és megkülönböztessük az adatok fogadásához és küldéséhez használt objektumokat.

Így néz ki egy adat fogadásához használt adatobjektum:

data class AddressModel (

val id : Long? = null,

val country : String? = null,

val county : String? = null,

val city : String? = null,

val zipCode : String? = null,

val streetName : String? = null,

val number : String? = null,

val door : String? = null,

)

Így néz ki egy adat küldéséhez használt adatobjektum, ahol Objektumról JSon adatstruktúrára történik az átalakítás:

data class AddressModelPost(

@SerializedName("id") val id : Long? = null,

@SerializedName("country") val country : String? = null,

@SerializedName("county") val county : String? = null,

@SerializedName("city") val city : String? = null,

@SerializedName("zipCode") val zipCode : String? = null,

@SerializedName("streetName") val streetName : String? = null,

@SerializedName("number") val number : String? = null,

@SerializedName("door") val door : String? = null,

)

Az átalakításhoz szükség van a @SerializedName annotációra, amely megfeleltetést végez, hogy JSon szerkezetet tudjunk küldeni akár létrehozáshoz vagy frissítéshez. Az eredmény körülbelül így nézhet ki:

{

"id": 7,

"city": "Szeged",

"country": "Magyarorszag",

"county": "Csongrad",

"streetName": "Hajlat utca",

"number": 1,

"zipCode": "6700",

"door": "A1"

}

Ez a szerkezet az adatátviteli objektum (DTO) szerkezethez áll a legközelebb, mivel ez is folyamatok között közvetít adatokat.

#### Szolgáltatások (service)

A képen szöveg látható

Automatikusan generált leírásEbben az alkalmazásban a szolgáltatások biztosítják a szerverrel való kapcsolatot. Ehhez OkHttpClient, Retrofit2 és Gson függőségeket használok. A függőségek beinjektálásához Gradle projektépítő eszközt választottam, amelyet a szerver alkalmazásnál is használtam. A Kotlin a Java nyelvből fejlődött ki, ezért elég sok párthuzamot lehet vonni a kettő között.

ábra 19 - Service fájlok

Itt vannak megírva az API hívások függvényei is. Itt van erre egy példa:

@GET("/vehicle/{vehicleId}")

fun getVehicleById(@Header("Authorization") authorization: String?, @Path("vehicleId") vehicleId: Long?): Call<VehicleModel>

A @GET annotáció jelzi, hogy egy lekérésről van szó, és zárójelben utána a kérés szövege található, amelyet a ServiceGenerator egészít ki. A @Path a Springben használt @PathVariable-hez hasonló. Ezenkívül látható, hogy a küldéskor a fejlécben is elküldünk egy karakterláncot, amely a Bearer Token. Ezt a módszert a legtöbb kéréskor használjuk, ahol azonosítanunk kell magunkat. A Call<VehicleModel> a visszatérési értéket mutatja, amely egy VehicleModel objektum.

#### Ellenőrzések (validations)

Az ellenőrzéseket egy külön fájlba írtam össze, hiszen több helyen is kell használjam ugyanazokat a függvényeket. Ezeknek a lényege annyi, hogy a megkapott értékekről megállapítsa, hogy megfelelnek-e egy bizonyos kritériumnak. Van hogy egyszerre egy listát is leellenőriztetek, de legtöbb esetben egy érték helyességéről van szó. Itt egy példa:

fun validationUsername(editText: EditText): Boolean{

val len: Int = editText.text.toString().length

return len in 3..24

}

Itt látható, hogy egy EditText objektumot kap és egy Boolean értéket térít vissza. Ez a felhasználónév mezőnek az ellenőrzése. Ennek a hossza nagyobb kell legyen mint 3 és kisebb kell legyen mint 24. Látható, hogy a Kotlinaban nagyon röviden és elegánsan lehet ezt megfogalmazni.

#### Erőforrások (res)

A képen szöveg látható

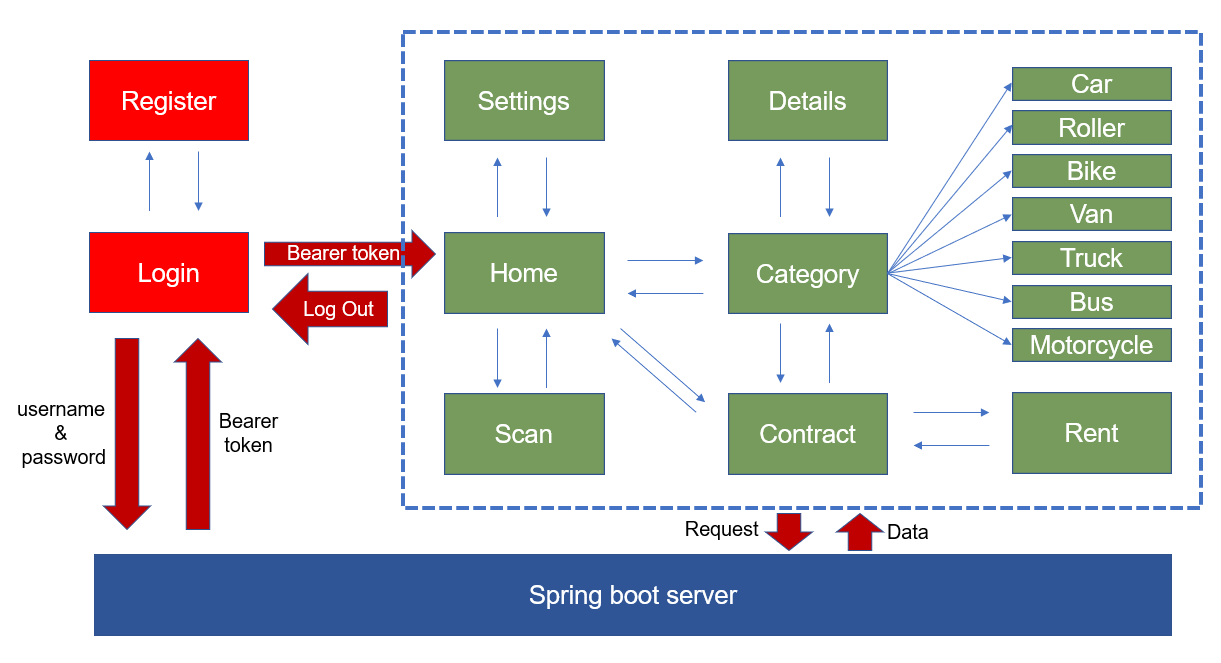
Automatikusan generált leírásEbben a könyvtárban vannak leírva a felületek kinézete XML leírónyelvet használva, itt találhatók különböző konfigurációs fájlok, rendszerváltozók (színek, karakterláncok) és itt vannak eltárolva a képek, ikonok és miden, ami a kinézet alapját adja.

ábra 20 - Erőforrások fájlszerkezete

Ez a réteg tartalmazza az MVC architektúra szerinti nézeteket. Hasonlít a webprogramozásban használt HTML struktúrához, csak mintha egybe lenne olvasztva a CSS-el. Hiszen itt például egyszerre hozunk létre egy gombot, mondjuk meg, hogy hol legyen, illetve milyen színű legyen, és hogy mekkora betűkkel rendelkezzen. Ez mind egy XML objektumon belül. De ehhez képest én elég használhatónak találtam.

### Felületek

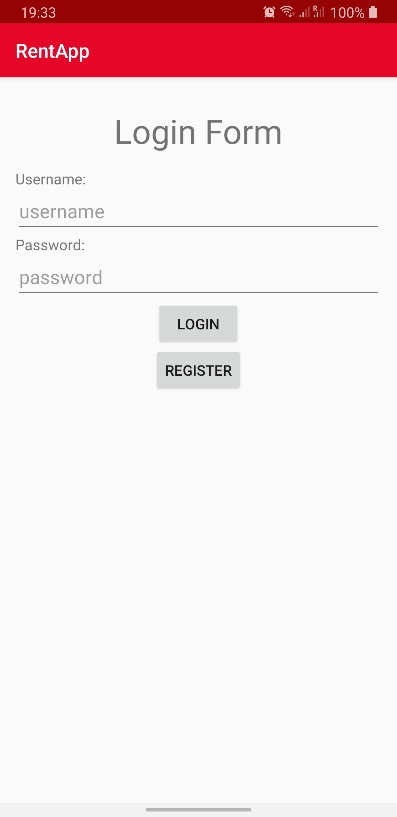
A Kliens alkalmazás számos felülettel rendelkezik. Ezek közül csak a Login és a Register felület érhető el bejelentkezés nélkül, a többi csakis érvényes felhasználónév és jelszó beírása után lehetséges. Ahhoz, hogy a belső oldalak érvényes kéréseket tudjanak küldeni szükség van egy érvényes bearer tokenre, amely bejelentkezéskor jön létre a szerverben és küldődik el a kliensnek. A kliens elmenti és ezt a tokent használja egy újabb bejelentkezésig, vagy amíg az a token érvénytelenné nem válik és ezáltal a szerverrel való kommunikáció lehetetlenné válik. Tehát a szaggatott vonalon kívüli oldalak olyan kéréseket használnak, amelyek személytelenek, így nem kell token, addig a szaggatott vonalon belül lévő oldalak csak úgy tudnak működni, ha egy érvényes bearer tokent tudnak elküldeni a kéréseik fejlécében, amelyhez be kell jelentkezni.

A kliens alkalmazás felületeinek kapcsolati hálója a lenti ábrán látható.

ábra 21 - Kliens architektúrája

A piros nyilak azt jelölik, hogy ott adatáramlás is történik, a vékony kék nyilak pedig az oldalak közötti átjárhatóságot jelölik. Rögtön látható, hogy két részre oszlik a kliens alkalmazás. Egy bejelentkezés előtti és egy bejelentkezés utáni részre A bejelentkezés előtt a Login és a Register felületek érhetők el, a többi csak utána. A bejelentkezés után eléggé szerte ágazóan járkálhatunk az oldalak között, úgymond böngészhetünk az adatok között, használhatjuk az általam megírt funkcionalitásokat. A Kliens alkalmazásban lévő műveletek legnagyobb része Rest API kérések segítségével történik. Az ellenőrzés sok esetben többszinten történik meg.

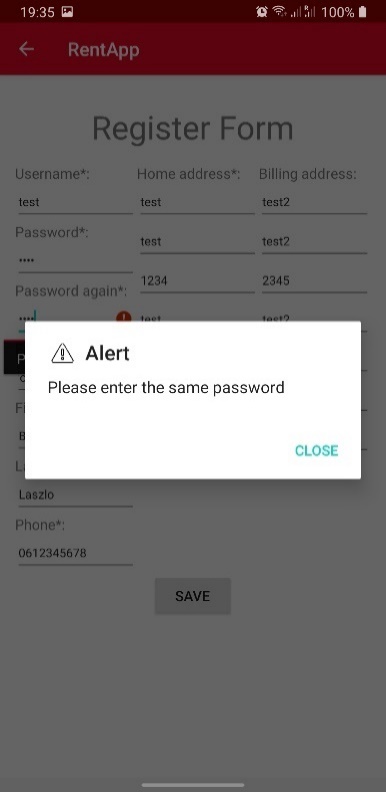
#### Login felület

Elindítás után a Login képernyőre jutunk. Innen be tudunk lépni, ha már rendelkezünk egy érvényes felhasználóval és persze tudjuk a hozzá tartozó jelszót. Ezeket beírva a megfelelő mezőkbe és megnyomva a Login gombot belépünk az alkalmazásban. Ha nincs még felhasználónk, akkor a Regisztrációs oldalra lehet menni. Itt, ha a \*-al jelölt mezőket megfelelően kitöltjük, akkor tudunk regisztrálni.

ábra 22 - Login oldal

#### A képen asztal látható Automatikusan generált leírásRegisztráció felület

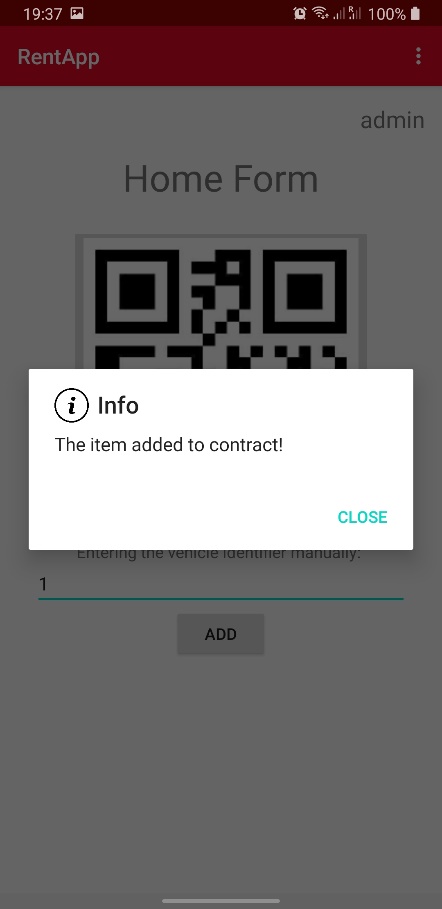
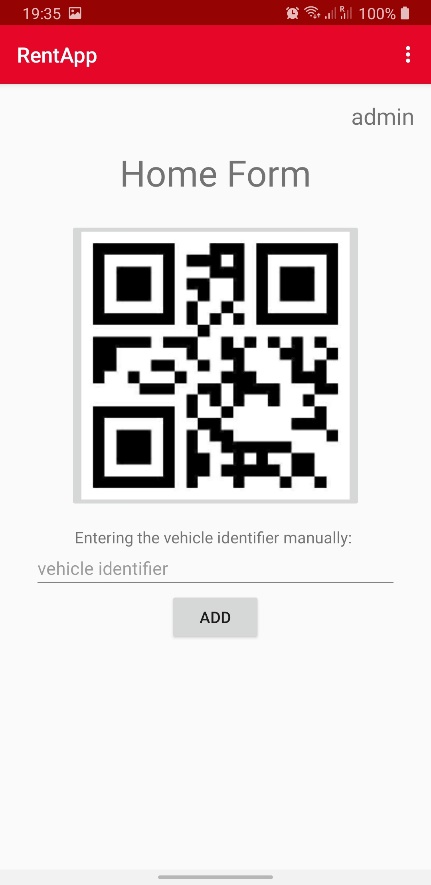
ábra 23 - Regisztrációs oldal

Ahogy már írtam a Regisztrációs felületre csakis a Login felületről tudunk eljutni és a \*-al jelölt mezők kötelezőek. Ezenkívül, ha számlázási címet is szeretnénk kitölteni, akkor ugyanazok a mezők kötelezőek, amelyek a sima címnél, de nem kell aggódni, hiszen a felület mindenre fel van készítve, így jelez, ha üres vagy hibás érték kerülne valamelyik mezőbe. Ekkor egy felugró ablakkal vagy figyelmeztetéssel adja tudatunkra, hogy mivel van probléma és mi a teendő.

ábra 24 - Felugró ablak

#### Home felület

Sikeres bejelentkezéskor minden esetben a Home felületre jutunk, ahol a felső jobb sarokban olvasható a felhasználó nevünk. Ez minden oldalon így van, ahol be vagyunk jelentkezve. Ez az alkalmazás útválasztója. Innen a menü segítségével eljuthatunk a Settings, a Category vagy a Contract oldalakra.

Ezenkívül lehetőségünk van egy barkód, vagy QR kód beolvasására. Ennek a segítéségével tudjuk beolvasni a kibérlendő jármű azonosító kódját, amelynek sikeres beolvasása után be is rakja a felhasználó Contract objektumának a járműveket tartalmazó listájába. Itt mindig van egy ellenőrzés, hogy nehogy ugyanaz a jármű többször is felvételre kerüljön, amely szerver szintről jön, de kliens oldalon is le kellett kezelni. Ez hasznos, ha ott van mellettünk a jármű, például egy elektromos roller, és nehézkes lenne kikeresni a listából, sok adat között szinte lehetetlen, de ezzel egyszerűen egy QR- vagy bárkódot beolvasva a kliens és ezáltal a szerver azonosítja a járművet berakja a listába, pár gomb lenyomása után pedig már gurulhatunk is a rollerrel. Persze valóságban ez bonyolultabb lehet és több folyamatnak is le kell futnia, de hasonló rendszerek már működésben vannak Magyarországon (Mol Bubi).

ábra 25 - Home felület

ábra 26 - Felugró ablak manuális beírás után

Ennek az azonosítónak a beolvasása történhet manuálisan is, hiszen kell gondolni azokra is, akiknek például elromlott a kamerája, vagy lehet az is, hogy elszakadt a járműre ragasztott QR kód, de az azonosítót még ki lehet olvasni.

#### Settings felület

A képen asztal látható

Automatikusan generált leírásIde a Home oldalról lehet eljutni, és a felhasználó profiljának az adatai jelennek meg szerkeszthető formában. Ezek átírása után lehetőségünk van módosítani saját adatainkon az update gomb megnyomásával. Itt is működnek ugyanazok az ellenőrzések a mezőkre, amelyek a Register felületen megvoltak írva, mivel a két felület nagyon hasonló nem volt nehéz dolgom ezek beüzemelésében. Ezenkívül, ha szeretnénk az egész profilunkat tudjuk törölni, amely után az alkalmazás visszalép a Login felületre, hiszen már nem lehetünk ezzel a profillal bejelentkezve.

ábra 27 - Beállítások oldal

#### Category felület

A képen asztal látható

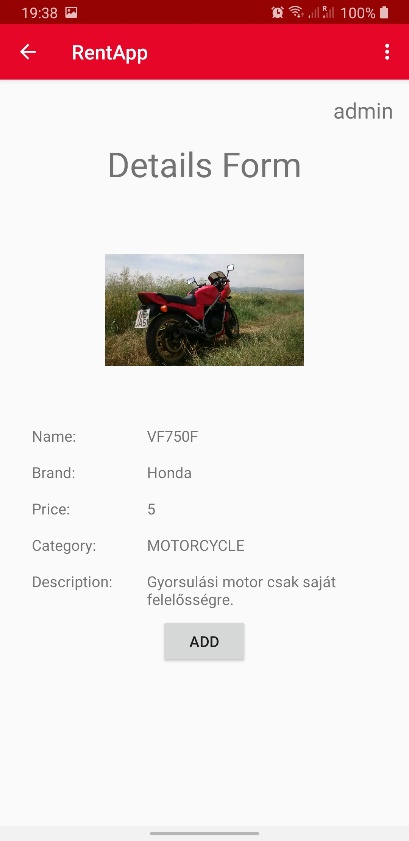
Automatikusan generált leírásA képen asztal látható

Automatikusan generált leírásIde több helyről is eljuthatunk. Itt jelenik meg a járművek listája, ahol a szerveren tárolt összes jármű megjelenik, mellettük egy Details és egy Add gombbal. A Details gombot megnyomva az adott jármű tulajdonságai jelennek meg. Az Add gomb megnyomása után pedig hozzáadjuk az adott járművet a szerződésünkhöz. A menüben lépkedve lehetőségünk van jármű vek kategóriái szerint szűrni az adatokat. Ezek mind külön kérésekkel működnek, ezért mindig a legfrissebb adatokat kapjuk, ami adatbázisban el van tárolva.

ábra 28 - Kategóriák oldal

ábra 29 - Kategóriák

#### Details felület

Ide mindig onnan tudunk eljutni, ahol megjelenik a járművek listája valamilyen formában. Itt mindig lesz egy-egy Details gomb járművenként, amelynek a megnyomása után betöltődnek erre az oldalra az aktuális jármű adatai. Ezek a következők: név, márka, napidíj, kategória, leírás. Ezeken kívül itt található egy Add gomb, amely hozzáadja ezt a járművet a szerződéshez.

ábra 29 - Részletek oldal

#### Contract felület

A képen asztal látható

Automatikusan generált leírásA Contract oldalt a több helyről is el tudjuk érni a menüben a Contract menüpontra lépve érhetjük el. Itt látható a szerződésbe rakott járművek listája. Egy listaelemen a Details gomb hatására a járműhöz Details oldalra jutunk, a Delete gomb megnyomásának hatására pedig kitörli az aktuális járművet a szerződésből. A To Rent gomb megnyomása utána Kölcsönzési oldalra jutunk. A Delete All gomb megnyomása után minden jármű törlődik a Szerződés járműlistájából.

ábra 30 - Szerződés felülete

#### Rent felület

A képen asztal látható

Automatikusan generált leírásIde a Contract oldalról tudunk eljutni. Itt található két DatePicker, a felsővel a beállítható, hogy mikortól szeretnénk kivenni, az alsóval pedig, hogy meddig szeretnénk kivenni a járművet/járműveket. Ezekre vannak különböző megszorítások:

* Az első dátum nem lehet kisebb, mint a mai dátum.
* A második dátum szigorúan nagyobb kell legyen, mint a kezdeti dátum

A Rent gomb megnyomásának hatására létrejön a szerződés, illetve az alkalmazás elküldi e-mailben a szerződés fontosabb jellemzőit, ez egyben egy visszaigazoló e-mail is. Ezért kötelező megadni az e-mail címet, amelynek 3 szintű validációja van. Emailben a felhasználó minden aktuálisan kikölcsönzött járműről kap adatot, ezenkívül azt, hogy mennyit kell fizetnie az egyes tételekért és összesen az egészért. Ezt a szerver számolja ki a napidíjak és a DatePickerekből elküldött adatok alapján. Ezek PathVariable ben küldődnek el Rest API kérésben.

ábra 31 - Foglalás felülete

Itt látható egy rendelés esetén küldött e-mail szövege:

You have successfully rented vehicles!

StartDate: 2022-12-06

EndDate: 2022-12-23

Vehicles:

1. Vehicle:

Name: T680

Brand: Kenworth

Price/day: 20

Total price: 340

Description: Gyorsulás 0-ról 100-ra? Igen.

Category: TRUCK

2. Vehicle:

Name: VF750F

Brand: Honda

Price/day: 5

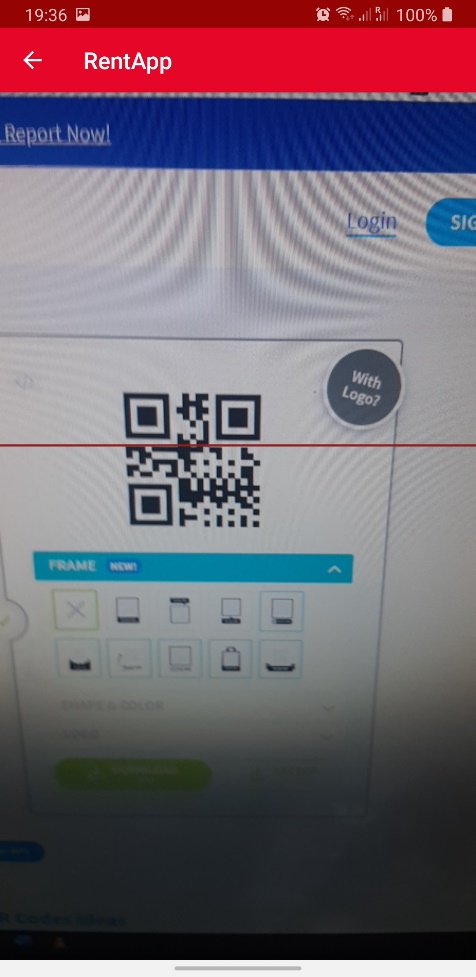
Total price: 85

Description: Gyorsulási motor csak saját felelősségre.

Category: MOTORCYCLE

Sum: 425

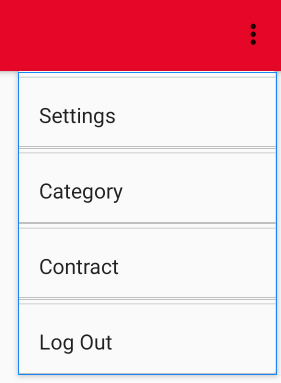
#### Scan felület

Ide a Home oldalról tudunk eljutni a QR kód képére nyomva és az a szerepe, hogy beolvasson egy QR kódot, vagy bárkódot. Ehhez engedélyt kell adnunk az alkalmazásnak, hogy használhassa a kamerát. Ha ez engedélyezve van, akkor nem kéri többé az alkalmazás. Amint erre az oldalra lépünk elkezd keresni egy képet, amit le tud olvasni, ha leolvasta és értelmezte megvizsgálja a beolvasott kódot. Ha felismeri egy jármű azonosítójaként, akkor megnézi, hogy be volt-e rakva már a listába, ha nem volt berakja, különben a kiírja a megfelelő üzenetet. Ha nem ismeri fel természetesen az is le van kezelve.

ábra 32 - Barcode szkenner működés közben

Beolvasás után visszalép a Home oldalra, ha esetleg még egy járművet be szeretnénk olvasni, csak újra meg kell nyomni a képet.

#### Log Out

Bár ennek külön felülete nincs, de ez is nagyon fontos része a felületeknek, hiszen majdnem minden felületen megtalálható. Az oldalakról kilépés akármikor lehetséges a menü használatával minden esetben az utolsó menüpontra lépve. Ilyenkor a bearer token törlődik a rendszerből és visszalépünk egy teljesen új Login oldalra.

ábra 33 - menü (Home)

1. **Tesztelés**

A képen szöveg látható

Automatikusan generált leírásAutomatikus tesztelés csak a szerver környezetben található. A kliens alkalmazás esetén statikus teszteléssel bizonyosodtam meg a funkciók helyességéről, amely alatt manuális tesztet és code review-t értek. A tesztelés során az üzleti logika tesztelését végeztem el, amelyet megpróbáltam kiterjeszteni a Service osztályok függvényeinek egészére. Ezek JUnit tesztek, amelyeket könnyen lehet kezelni egy Java alkalmazásban. Ezzel persze a Repository osztályok tesztelése is történik, mivel a Service-ben lévő függvények ezt használják fel feladataik elvégzése során. Mivel a Repository osztályban adatbázis műveleteket végzünk el, és ezek kigenerálásához a Modell osztályokat használja fel, ezért az egész szerver működésének tesztelése is megtörténik. Elvileg, ha a tesztek lefutnak, akkor a szerver alkalmazás hibamentes, persze feltételezve, hogy a Controller osztályok megfelelően fogadják a kéréseket és jól is válaszolnak. Összesen 35 tesztet írtam meg az 5 Service osztályra, amelyek egyszerre, vagy külön is lefuttathatók és kezelésük nagyon egyszerű a Visual Studio Code tesztelő kiterjesztésének köszönhetően.

ábra 34 - Tesztek

Példaként a RoleServiceTest.java osztályból szeretnék részleteket:

@SpringBootTest

@TestInstance (Lifecycle.PER\_CLASS)

public class RoleServiceTest {

@Autowired

private RoleRepository roleRepository;

@Autowired

private UserRepository userRepository;

@Autowired

private RoleService roleService;

private Role role = new Role(ERole.ROLE\_ADMIN);

private Role role2 = new Role(ERole.ROLE\_USER);

@BeforeAll

public void beforeAll() {

userRepository.deleteAll();

userRepository.flush();

}

@BeforeEach

public void before() {

roleRepository.deleteAll();

roleRepository.flush();

}

@Test

public void createRole(){

Role result = roleService.createRole(role);

assertNotNull(result);

assertNotNull(result.getId());

}

A fenti példakódban látható, hogy @SpringBootTest annotációval kezdünk, amely jelzi a Spring boot keretrendszernek, hogy ez egy Teszt osztály. Ha hozzáadjuk a @TestInstance (TestInstance.Lifecycle.PER\_CLASS) elemet a tesztosztályhoz, elkerülhető, hogy az osztály minden tesztjéhez új példány jöjjön létre. Ez különösen akkor hasznos, ha sok teszt van ugyanabban a tesztosztályban, és ennek az osztálynak a példányosítása költséges. Az @Autowired kulcsszóval egyszerűen adattagként fel vettem a szükséges Service osztályokat, példányosítás nélkül. Ezután a role modelleket példányosítom. Összesen 2 darab van a szerveralkalmazásban, tehát itt most mind a 2 látható (Admin, User). A @BeforeAll annotációval ellátott függvény egyszer fut le a tesztek végrehajtása előtt, a @BeforeEach pedig minden egyes @Test annotációval ellátott függvény előtt lefut, amely ebben az osztályban található. A @Test annotáció magától értetődően egy tesztesetet jelöl. A fenti kódban látható teszt létrehoz egy szerepkört (Admin-t) és leellenőrzi az assertNotNull függvénnyel, hogy az objektum, illetve az id-ja létrejött-e. Többfajta ellenőrzés is létezik: assertFalse, assertEqual, assertTrue stb.

1. **Összefoglalás**

A szerver-kliens alkalmazás elkészítése során úgy érzem elég sok tapasztalatot szereztem a hasonló rendszerek felépítésének megértésében. Nagyrészt sikerült elérnem, amit a feladatkiírásban írt, bár a kliens alkalmazás szerkezetét modernebb tervezési minta szerint is összeállíthattam volna.

Továbbfejlesztési lehetőség bőven akad, ha egy valóban eladható szoftver szintjére szeretném felemelni ezt az alkotást. A kölcsönzési folyamatot ki lehetne egészíteni pdf szerződés generálásával vagy online fizetési lehetőséggel. Lehetne bővíteni a bejelentkezést úgy, hogy létező facebook, vagy google felhasználóval is lehessen használni. Ehhez a szervert kell továbbfejleszteni akár többfajta titkosítási technológiák és biztonsági könyvtárak felhasználása is szükséges lehet. Ezenkívül lehetne egy másik klienssel is bővíteni a portfóliót, hogy ne csak telefonról lehessen használni, hanem egy böngészőből is egy webkliens segítségével.

Összességében el szeretném mondani, hogy nagyon jól éreztem ennek a projektnek az elkészítése során és örülök, hogy idáig eljutottam. Úgy érzem egy stabilan működő rendszert tudtam elkészíteni, ahol a kliensek az internet segítségével bárhonnan képesek elérni a szervert akár egyidőben is.

1. **Irodalomjegyzék**

|  |  |
| --- | --- |
| [1] | Java [Online]. Elérhető: https://www.java.com/en/download/help/whatis\_java.html. [Hozzáférés dátuma: 31 október 2022]. |
| [2] | Spring [Online]. Elérhető: https://spring.io/, [Hozzáférés dátuma: 05 november 2022] |
| [3] | Android [Online]. Elérhető: https://www.javatpoint.com/android-software-stack, [Hozzáférés dátuma: 26 november 2022] |
| [4] | Alkalmazásfejlesztési környezetek (VIAUAC04), 2022 tavaszi előadások, Tárgy holnapja: https://portal.vik.bme.hu/kepzes/targyak/VIAUAC04 |
| [5] | MySQL [Online]. Elérhető: https://dev.mysql.com/doc/, [Hozzáférés dátuma: 27 november 2022] |
| [6] | Spring boot [Online]. Elérhető: https://spring.io/projects/spring-boot, [Hozzáférés dátuma: 05 november 2022] |
| [7] | Spring MVC [Online]. Elérhető: https://www.digitalocean.com/community/tutorials/spring-mvc-example, [Hozzáférés dátuma: 26 november 2022] |
| [8] | CRUD [Online]. Elérhető: https://www.codecademy.com/article/what-is-crud, [Hozzáférés dátuma: 27 november 2022] |
| [9] | Hibernate, ORM, JPA [Online]. Elérhető: https://www.javatpoint.com/hibernate-tutorial, [Hozzáférés dátuma: 26 november 2022] |
| [10] | OkHttp [Online]. Elérhető: https://square.github.io/okhttp/, [Hozzáférés dátuma: 14 november 2022] |
| [11] | Sdk [Online]. Elérhető: https://www.redhat.com/en/topics/cloud-native-apps/what-is-SDK, [Hozzáférés dátuma: 13 november 2022] |
| [12] | Git, [Online]. Elérhető: https://git-scm.com/doc. [Hozzáférés dátuma: 19 október 2022]. |
| [13] | Github [Online]. Elérhető: https://github.com/about, [Hozzáférés dátuma: 13 november 2022] |

**Függelék**

**Build-elés** – az a folyamat, amely során a számítógép a forráskódot valamilyen futtatható állományba tömöríti össze, amelyet később fel tudunk használni akármikor, vagy amikor direkt feltelepíti a programot egy céleszközre.

**Git Branch** – hatékonyan mutatják a változtatások pillanatképét. A fájlok könyvtárból könyvtárba másolása helyett a Git egy branchet tárol a véglegesítésre való hivatkozásként. Ebben az értelemben egy branch a véglegesítések sorozatának csúcsát jelenti – ez nem a véglegesítések tárolója.

**Git Merge** – A mergelés a Git módszere arra, hogy újra összerakja az elágazó brancheket. A git merge parancs lehetővé teszi a git branch által létrehozott független fejlesztési vonalak felvételét és egyetlen ágba való integrálását.

**Fetch** - A git fetch parancs egy távoli tárolóból tölti le a véglegesítéseket, fájlokat és hivatkozásokat a helyi tárhelyre. A lekérés az, amit akkor csinál, ha látni szeretné, min dolgozott már mindenki. Ez teszi a lekérést biztonságossá, mielőtt integrálná a véglegesítéseket a helyi tárolóba.

**Pull** - A git pull parancs a tartalom távoli tárolóból való lekérésére és letöltésére szolgál, valamint a helyi lerakat azonnali frissítésére, hogy megfeleljen a tartalomnak.

**Commit** – A git commit parancs pillanatképet készít a projekt jelenleg végrehajtott módosításairól.

**Push** - A git push parancsot a helyi lerakat tartalmának távoli tárolóba való feltöltésére használják. A leküldés az a mód, ahogyan átviheti a commitokat a helyi lerakatból egy távoli tárolóba.

**Debug folyamat** – hibakeresési folyamat, ami lehet statikus (fejlesztői eszköz általában elvégzi), és dinamikus, amelynek során a program lefuttatásával ellenőrizzük, hogy az megfelelően működik-e. Ez általában magával vonja annak ellenőrzését, hogy megadott bemeneti értékek esetén a program az elvárt kimenetet produkálja-e. Sok esetben megállási pontokat helyezünk el a kód azon részén, ahol szerintünk végig fog haladni, aminek hatására ott kicsit szünetel a futás és meg tudjuk nézni a pillanatnyi változók értékét.

**SDK** – Szoftverfejlesztői csomag (Software development kit). Szoftverfejlesztő eszközök gyűjteménye egyetlen telepíthető csomagban.

**IDE** - Integrált fejlesztői környezet (Integrated development environment). Az integrált fejlesztői környezet egy olyan szoftveralkalmazás, amely átfogó létesítményeket biztosít a számítógépes programozók számára a szoftverfejlesztéshez. Az IDE általában legalább egy forráskódszerkesztőből, automatizálási eszközökből és hibakeresőből áll. Pl: Eclipse, NetBeans, Visual Studio, Android Studio, IntelliJ IDEA...

**CMD** - A Command Prompt, más néven cmd.exe vagy cmd, az OS/2, eComStation, ArcaOS, Microsoft Windows és ReactOS operációs rendszerek alapértelmezett parancssori értelmezője, amely 1987-ben jelent meg először. A cmd.exe parancssori felületen keresztül kommunikál a felhasználóval. Megvalósítása operációs rendszerenként eltérő, de a viselkedés és a parancsok alapkészlete konzisztens.

**Repository (REPO)** - A szoftvertár vagy angolul repository a szoftvercsomagok tárolási helye. Gyakran egy tartalomjegyzéket is tárolnak a metaadatokkal együtt. A szoftvertárakat általában forrásvezérlők vagy lerakatkezelők kezelik. A csomagkezelők lehetővé teszik a tárolók automatikus telepítését és frissítését.

**HTTP (HyperText Transfer Protocol)** - egy információátviteli protokoll elosztott, kollaboratív, hipermédiás, információs rendszerekhez. A HTTP fejlesztését a World Wide Web Consortium és az Internet Engineering Task Force koordinálta RFC-k formájában.

**RFC** – ez egy olyan dokumentum, mely egy új Internet-szabvány beiktatásakor adnak közre. Az új szabvány első tervezete saját számmal kerül a nyilvánosság elé, egy adott időtartamon belül bárki hozzászólhat.

**Connection pool** - A szoftverfejlesztésben a kapcsolatkészlet az adatbázis-kapcsolatok gyorsítótárát jelenti, amelyet úgy karbantartanak, hogy a kapcsolatok újra felhasználhatók legyenek, amikor az adatbázishoz jövőbeli kérésekre van szükség. A kapcsolati készletek a parancsok végrehajtásának teljesítményét javítják az adatbázison.

**Cache (gyorsítótár)** - a számítástechnikában az átmeneti információtároló elemeket jelenti, melyek célja az információ-hozzáférés gyorsítása. A gyorsítás egyszerűen azon alapul, hogy a gyorsítótár gyorsabb tárolóelem, mint a hozzá kapcsolt, gyorsítandó működésű elemek.

**Node (csomópont)** – Ez a hálózatban nem más, mint egy kapcsolódási pont. Lehet egy újraelosztási pont (adattovábbítás) vagy pedig az adatátvitel végpontja. Általában a csomópontok úgy vannak felállítva, hogy képesek felismerni és továbbítani az adatokat más csomópontok felé is.

**Biztonsági token** – Lehet egy fizikai eszköz, amit egy számítógépes szolgáltatás autorizált felhasználója kap, az autentikáció segítése érdekében, de a mi esetünkben egy generált szöveg. Egy felhasználó hozzáférési tokenje tartalmazhatja a felhasználó nevét, azokat a csoportokat, amelyekhez a felhasználó tartozik, a felhasználó biztonsági azonosítóját és az összes olyan csoport biztonsági azonosítóját, amelyekhez a felhasználó tartozik.

**Responzív design** - A reszponzív design egy olyan kialakítása felhasználói felületnek, amely rugalmasan alkalmazkodik a különböző eszközökhöz, böngészőkhöz, képernyőméretekhez annak érdekében, hogy optimális megjelenést biztosítson a felhasználónak minden alkalmas eszközön. Létrejöttének oka az egyre többféle méretű kijelző megjelenése.

**CRUD** - Létrehozás, olvasás, frissítés és törlés (CRUD) az a négy alapvető funkció, amelyet a modelleknek legfeljebb képesnek kell lenniük.

**Front-end** - A programoknak, weboldalaknak az a része, amelyik a felhasználóval közvetlenül kapcsolatban van. Feladata az adatok megjelenése, befogadása a felhasználó (vagy ritkábban egy másik rendszer) felől.

**Back-end** - A backend a többrétegű architektúrának az a rétege, ami a frontend (prezentációs réteg) felől érkező adatok és kérések feldolgozását végzi, illetve ami a szerver oldalon keletkező eredményeket visszaadja a felhasználói felület felé. Backend része az üzleti logika megvalósítása és adattárolási réteg elérése.