Hotelszoba foglaló webalkalmazás Spring Boot keretrendszerben

Tartalom

[1. Bevezetés 3](#_Toc165246397)

[2. Felhasznált eszközök és technolódiák 4](#_Toc165246398)

[2.1. Apache Maven 4](#_Toc165246399)

[2.2. Verziókezelés és Git 5](#_Toc165246400)

[2.3. MySQL 7](#_Toc165246401)

[2.4. IntelliJ IDEA 7](#_Toc165246402)

[2.5. Spring Core 8](#_Toc165246403)

[2.5.1. Inversion of Control 8](#_Toc165246404)

[2.5.2. ApplicationContext 8](#_Toc165246405)

[2.5.3. Függőség Befecskendezés 9](#_Toc165246406)

[2.6. Spring Data JPA 9](#_Toc165246407)

[2.6.1. ORM 9](#_Toc165246408)

[2.6.2. JPA 10](#_Toc165246409)

[2.6.3. Hibernate 10](#_Toc165246410)

[2.6.4. Entitások 10](#_Toc165246411)

[2.6.5. Relációk 11](#_Toc165246412)

[2.6.6. @OneToOne 11](#_Toc165246413)

[2.6.7. @ManyToOne és @OneToMany 11](#_Toc165246414)

[2.6.8. @ManyToMany 11](#_Toc165246415)

[2.6.9. Spring Data 12](#_Toc165246416)

[2.7. Spring Web MVC 12](#_Toc165246417)

[2.7.1. Modell-Nézet-Vezérlő 12](#_Toc165246418)

[2.7.2. DispatcherServlet 13](#_Toc165246419)

[2.8. Spring Security 15](#_Toc165246420)

[2.9. Spring Boot 15](#_Toc165246421)

[2.10. Thymeleaf 16](#_Toc165246422)

[2.11. Bootstrap 16](#_Toc165246423)

[2.12. Docker 17](#_Toc165246424)

[2.13. Lombok 17](#_Toc165246425)

[3. A felület bemutatása 17](#_Toc165246426)

[3.1. Felhasználói fiók kezelése 17](#_Toc165246427)

[3.2. Szobák keresése 19](#_Toc165246428)

[3.3. Szobák lefoglalása 20](#_Toc165246429)

[3.4. Foglalások kezelése 21](#_Toc165246430)

[3.5. Szállodák értékelése 21](#_Toc165246431)

[3.6. Az adminisztrátor felület 21](#_Toc165246432)

[3.7. Navigációs menük 22](#_Toc165246433)

[4. Az alkalmazás felépítése 23](#_Toc165246434)

[4.1. Monolitikus architektúra 23](#_Toc165246435)

[4.2. Többrétegű architektúra 23](#_Toc165246436)

[4.3. Adatelérés réteg 24](#_Toc165246437)

[4.3.1. Az adatbázis táblái 25](#_Toc165246438)

[4.3.2. Repository interfészek 28](#_Toc165246439)

[4.4. Üzleti logika réteg 30](#_Toc165246440)

[4.4.1. Modell entitások és DTO 30](#_Toc165246441)

[4.4.2. ModelMapper és Transformer osztályok 31](#_Toc165246442)

[4.4.3. RepositoryService osztályok 31](#_Toc165246443)

[4.4.4. Service osztályok 32](#_Toc165246444)

[4.4.5. Képek kezelése 32](#_Toc165246445)

[4.4.6. E-mail 33](#_Toc165246446)

[4.5. Felhasználói felület réteg 33](#_Toc165246447)

[4.5.1. @Controller osztályok 33](#_Toc165246448)

[4.5.2. Spring SecurityConfiguration 34](#_Toc165246449)

[4.5.3. i18n 34](#_Toc165246450)

[5. Unit tesztelés 35](#_Toc165246451)

[5.1. Persistence modul tesztelése 36](#_Toc165246452)

[5.2. Service modul tesztelése 36](#_Toc165246453)

[5.3. Presentation modul tesztelése 37](#_Toc165246454)

[5.4. Jacoco 37](#_Toc165246455)

[6. Konténerizáció 38](#_Toc165246456)

[6.1. Dockerfile 38](#_Toc165246457)

[6.2. Docker compose 38](#_Toc165246458)

[7. Összefoglalás 38](#_Toc165246459)

[8. Köszönetnyilvánítás 38](#_Toc165246460)

[9. Irodalomjegyzék 38](#_Toc165246461)

# Bevezetés

Bevezetés

# Felhasznált eszközök és technolódiák

## Apache Maven

A Maven egy *build tool*, amely szoftverprojektek kezelésében nyújt segítséget, valamint jelentősen megkönnyíti a build folyamatot a felhasználó részére. A *build tool*-ok olyan programok, melyek automatizálják az alkalmazás *felépítését*. Feladatuk közé tartozik a forráskód lefordítása, automatizált tesztek futtatása, a lefordított kód futtatható fájlokba, például JAR-okba csomagolása, majd ennek elhelyezése egy helyi, távoli vagy központi tárhelyen, másnéven repository-ban.

A build folyamat a Project Object Model (POM) leírása alapján történik. A pom.xml fájl segítségével szabhatjuk testre Maven projektünket. Megadható többek között a projekt neve, verziója, függőségei, több modulból álló projekt esetén a felhasznált modulok és a build életciklus.

Fontos funkciója a Maven-nek a függőségek kezelése. Alkalmazásunkban legtöbbször szeretnénk valamilyen külső, már előre legyártott technológiát felhasználni, mint például a Project Lombok vagy a Spring. Ezelőtt a függőségek beszerzése és karbantartása rendkívül problémás folyamat volt. A beszerzéshez egyesével fel kellett keresni az egyes függőségek weboldalát. Oda kellett figyelni, hogy a különböző függőségek egymással kompatibilisek legyenek, és nyomon kellett követni, mikor jelenik meg valamiből egy újabb verzió.

A Maven a következő megoldást kínálja függőségek kezelésére: POM.xml-ben megadhatjuk a függőségeinket. A build folyamán először megnézi, hogy a felsorolt függőségek már le vannak-e töltve a gépünkre. Ami nincs meg, azt interneten keresztül letölti a központi tárhelyről, majd hozzáadja a helyi tárhelyhez, hogy később ne kelljen még egyszer letölteni. Az mvnrepository.com weboldalon könnyen rákereshetünk a szükséges függőségekre, hogy hozzáadjuk őket projektünkhöz.

## Verziókezelés és Git

Verziókezelésnek nevezzük azt, amikor nyomon követjük és kezeljük egy kódbázison történt változtatásokat. Hagyományos fejlesztés során, ha egy fájlt megváltoztatunk, akkor ezzel elveszítjük annak eredeti változatát. Ha később vissza kell vonni egy változtatást, mert valamilyen új hibát okoz, akkor ez rengeteg problémával járhat. A verziókezelést végző programok lehetőséget nyújtanak arra, hogy eltároljuk fájlok korábbi verzióit, majd szükség szerint böngésszünk közöttük.

Szintén problémát jelenthet az, amikor több fejlesztő dolgozik ugyanazon a kódbázison. A verziókezelés lehetővé teszi, az egymástól való független munkát, a változtatások nyomon követését, és a hibát okozó változtatások visszakeresését.

A Git egy ingyenes és nyílt forráskódú verziókezelő rendszer. Más, központosított verziókezelő rendszerekkel (CVCS) szemben Git egy osztott verziókezelő rendszer (DVCS). Ez azt jelenti, hogy minden egyes fejlesztő gépén le van töltve a teljes kódbázis és annak összes korábbi verziója. Ennek előnye, hogy hálózati kapcsolat nélkül is, a többi változtatástól függetlenül dolgozhatunk. Az egész kódbázis tárolásának ma már, az olcsó tárhely és tömörítésnek köszönhetően, elenyésző a hátránya.

Nagyobb alkalmazások fejlesztésénél szinte garantált, hogy problémát fog okozni az, ha mindenki ugyanazt a kódbázist szerkeszti. Ezt elkerülendő, lehetőségünk van az eredeti kódbázisból elágazásokat létrehozni. Ezek az elágazások, vagy más néven branch-ek, ugyanazt a kódot tartalmazzák, mint ahonnan elágaztunk, de ha itt valamilyen változtatást végzünk, annak nem lesz hatása a főágon lévő kódra. Ezt gyakran akkor használjuk, amikor egy alkalmazáshoz valamilyen új funkciót szeretnénk fejleszteni. Lehetőség van ebből még további ágakat is létrehozni, így a funkció fejlesztését fel tudjuk osztani különböző részeire, melyen így egyszerre több ember is könnyedén dolgozhat. Ha elvégeztük a dolgunkat a mellékágon, azt visszafésülhetjük a főágba.

A képen szöveg, emléktábla, képernyőkép látható

Automatikusan generált leírásAz alkalmazásom fejlesztése során az elágazások használatának egyik oka egy dev ág létrehozása volt. Ennek célja csupán az, hogy a főágon nem ajánlott félkész kódot tárolni, habár ez nem egy konvenció és eléggé véleményfüggő. A másik ok az a különböző funkcióknak szóló ágak, úgynevezett feature branch-ek létrehozása volt. Itt dolgoztam olyan dolgokon, mint a Spring Security bevezetése, vagy a lokalizáció beállítása. Ezek olyan funkciók, amik az alkalmazás fejlesztése során folyamatosan bővültek, így mindet visszafésültem a dev ágba amint az alapműködésüket sikerült elérnem.

1. ábra – Git ágak

Fejlesztés során forráskódomat a saját gépem mellett online is tároltam. Erre a GitHub-ot használtam fel, amely egy ingyenes internetes tárhely elsősorban Git által verziókezelt projektek tárolására. A projektem a következő linken érhető el:

<https://github.com/Admadma/hotel-booking-app>

## MySQL

Adatok rendszerezett gyűjteménye alkot egy adatbázist. Ez lehet bármi egy egyszerű bevásárlólistától kezdve egy hotelszoba-foglaló alkalmazásban kezelt rengeteg információig. Ezen adatok létrehozása, elérése és feldolgozása érdekében szükség van valamilyen adatbázis kezelő eszközre. Erre a célra tökéletesen megfelel a MySQL.

A MySQL egy relációs adatbázis kezelő rendszer. Az adatokat külön táblákban tárolja. Megszabhatjuk, hogy a különböző táblák milyen kapcsolatban álljanak egymással. Különböző megszorításokat helyezhetünk egy tábla oszlopaira, mint például azt, hogy nem lehet üres, vagy egy oszlopban adott érték csak egyszer szerepelhet. Az adatbázis betartattja ezeket a szabályokat, és garantálja, hogy nem lesz benne olyan adat, ami megszegné. Az adatbázis eléréséhez strukturált lekérdezőnyelvet (Structured Query Language vagy SQL) használhatunk, mely a legelterjedtebb nyelv ilyen célokra.

Azért választottam, mert sebessége, megbízhatósága, skálázhatósága és könnyen kezelhetősége miatt számos feladatra használható kisebb alkalmazásoktól kezdve összetett nagyvállalati rendszerekig. Továbbá már közel 30 éve jelen van, így rengeteg segédanyag található hozzá.

## IntelliJ IDEA

Az IntelliJ IDEA egy integrált fejlesztői környezet elsősorban Java és Kotlin programok fejlesztéséhez. Néhány főbb szolgáltatása a következő:

Statikus kód elemzés és intelligens kiegészítés. Folyamatosan értelmezi a leírt kódunkat. Ha valami fordítási hibát okozna, mert például kifelejtettünk a sor végéről egy pontosvesszőt, vagy elírtuk egy változó nevét, akkor egyből jelez a felhasználónak. Elég elkezdeni gépelni, és már írás közben képes megtippelni, hogy mit is szeretnénk leírni, és felkínálja a lehetőségeket.

Egyszerűbbé teszi a kód refaktorálását. Ennek egyik eszköze például a metódus kiemelés, amit már korábbi projektjeimben is rengetegszer használtam. Akár már három kattintással is kijelölhetünk egy kódrészt, és létrehozhatunk neki egy külön metódust, amit az eredeti helyén csak meg kell hívni.

A kódunkat egyetlen kattintással lefuttathatjuk. Ehhez megadhatunk különböző paramétereket is, mint például környezeti változók listáját. Ha futtatás közben előjön valamilyen hiba, de nem tudjuk ennek a konkrét forrását, akkor úgynevezett breakpoint-okat helyezhetünk el a kódban. Debug módban futtatva elemezhetjük az aktuális állapotot az egyes breakpoint-oknál.

Egyszerűen tudunk vele automatizált teszteket írni és futtatni. Tesztjeinket írhatjuk a legelterjedtebb keretrendszerekben, mint például a JUnit, TestNG, Cucumber, és az eredményüket valós időben vizsgálhatjuk.

Támogatja a legelterjedtebb verziókezelő rendszereket, mint például a Git és a Subversion. Megkönnyíti a verziók összehasonlítását, különböző branch-ek létrehozását és kezelését, változtatások commit és push-olását, valamint a merge conflict-ok egyszerű megoldását.

Támogatja a már korábban említett build tool-okat, mint például a Maven vagy Gradle. Az IDE-ben egyszerűen tudjuk velük a kódunkat lefordítani, tesztelni és becsomagolni.

Ha a meglévő szolgáltatásoknál többre lenne szükségünk, akkor lehetőséget kínál különböző plug-in-ok telepítésére.

## Spring Core

A Spring egy nyílt forráskódú keretrendszer, aminek legfőbb célja, hogy megkönnyítse a nagyvállalati Java alkalmazások fejlesztését. Az idő folyamán több modul is létrejött a keretrendszer funkcióinak bővítésére, mint például a Spring Security, vagy Spring Data JPA. Viszont felmerülhet a kérdés, hogy mi van akkor, ha az alkalmazásunkban nem akarjuk ezeket felhasználni. Szerencsére a Spring fő irányelvei közé tartozik a különböző igények kielégítése és a személyre szabás lehetősége minden szinten. Eszerint alkalmazásunk igényeitől függően szabhatjuk meg, hogy mely modulokra van szükségünk.

### Inversion of Control

A keretrendszer alapkövét képezi a Spring Core. Ez a modul valósítja meg az Inversion of Control (IoC) elvet. Ez az elv azt mondja ki, hogy az alkalmazásban lévő objektumok és függőségek kezelését ne maga az alkalmazás végezze. Helyette ezt a felelősséget ruházza át egy keretrendszerre, vagy konténerre. Az alkalmazás megmondja, milyen objektumai vannak, és ezek között mik az összefüggések, de a konténer fogja létrehozni és kezelni őket. Ennek eredményeként kódunk lazán csatolt lesz, ezáltal sokkal könnyebb karban tartani, függőségeit cserélgetni és a részeit önállóan tesztelni.

### ApplicationContext

A Spring keretrendszerben az ApplicationContext lesz az a konténer, ami átveszi a fent említett felelősségeket. Feladata az alkalmazásban jelen lévő objektumok példányosítása, konfigurálása és függőségeik kielégítése. Ezeket a konténerben tárolt objektumokat Bean-eknek nevezzük. Az alkalmazás elindításakor megtörténik a komponens szkennelés, mely minden Spring által kezelt Bean-t, legyen az egy Service, Component, vagy Controller, felkeres és regisztrál. A Bean-eket a konténer kezeli teljes életciklusukon át. A Bean-ek felépítéséhez a konténer XML konfigurációkból vagy annotációkból gyűjti be az objektum adatait, mint például annak nevét.

### Függőség Befecskendezés

A Függőség Befecskendezés egy tervezési minta, ami megvalósítja az Inversion of Control elvet. Ezt a következőképpen éri el: Ahelyett, hogy minden objektum létrehozná magának saját függőségét, egyszerűen csak megmondják, hogy mire van szükségük és megkapják azt az ApplicationContext-ből (feltéve, hogy definiálva van az igényelt Bean).

A függőségek befecskendezésére többféle lehetőség is van. Az @Autowired annotáció segítségével megadhatjuk őket a konstruktorban, vagy setter metódusokban. Én a projektemben a mező-alapú befecskendezést választottam, aminek fő oka az egyszerűség és olvashatóság volt.

Ha a különböző Bean-ek helyett csak egyszerű mezők, például String vagy egy lista értékét szeretnénk megadni, arra a @Value annotációt használhatjuk. Az értékeket leggyakrabban egy properties fájlból olvassuk ki, viszont alkalmazásomban környezeti változók értékének olvasására használom. Fontos megjegyezni, hogy ha a keresett változó mindkét forrásban definiálva van, akkor a környezeti változóban lévőt fogja választani. A kódomban ezeket az értékeket konstruktor paraméterként adom meg, így a teszt környezetemben könnyen tudom más értékekre cserélni.

## Spring Data JPA

### ORM

Az Objektum-relációs leképezés (Object Relational Mapping) az egyszerű objektumok mezői és egy SQL adatbázis tábla oszlopai közötti átalakításnak, majd eltárolásnak az automatizált folyamata. Ennek célja, hogy akár egyetlen SQL parancs írása nélkül is kommunikálni tudjunk relációs adatbázisokkal Java kódunkból.

### JPA

A Java Persistence API egy specifikáció, mely interfészeket ad az ORM megvalósításához és perzisztens objektumok kezeléséhez. Az interfészeihez nem ad megvalósítást, így önmagában nem tudjuk felhasználni.

### Hibernate

A Hibernate egy programkönyvtár, mely megvalósítja a JPA interfészeit. Használatának számos előnye van. Csökkenti az adatbázis kezeléséhez szükséges kódot, így egyből az üzleti logikára tudunk koncentrálni. Könnyebb karbantartani, mivel a kevesebb kódnak köszönhetően sokkal átláthatóbb. Számos lehetőséget kínál a teljesítmény optimalizálására, mint például a gyorsítótárazás, vagy tömeges feldolgozás. Ha szükséges, a fejlesztés során lehetőségünk van más adatbázishoz csatlakozni, vagy lecserélni más implementációra anélkül, hogy programkódunkat változtatni kellene.

### Entitások

A JPA Entitások olyan Java osztályok, melyek leírják egy adatbázis tábláit. Az osztályban szereplő attribútumok a tábla egy-egy oszlopának felelnek meg. Az osztály példányai pedig a tábla egy sorát fogják képezni. A Hibernate képes arra, hogy ezen osztályokból saját maga hozza létre az adatbázis tábláit.

Ezeket az osztályokat az @Entity annotációval kell ellátnunk. Az entitásunk tulajdonságait további annotációk segítségével adhatjuk meg. Például a @Table annotációban megadhatjuk a tábla nevét és sémáját. Az @Id annotációval jelölhetjük az osztály azon mezőjét, ami az entitás elsődleges kulcsa lesz. Az egyes mezőkre a @Column annotációval megszorításokat szabhatunk ki, mint például azt, hogy nem lehet null, vagy csak különböző adatok kerülhetnek abba az oszlopba. A @Transient annotációval jelölhetjük azokat a mezőket, amiket nem akarunk, hogy az adatbázis részei legyenek.

### Relációk

Egy relációs adatbázis legfontosabb tulajdonsága nem más, mint maga a relációk. A JPA az entitások közötti relációk leírására több annotációt is kínál. Két egymással kapcsolatban álló entitás leírására bevezette a reláció birtokosa fogalmat, mely azt a táblát jelenti, ahol a másik táblára mutató idegen kulcs található. Ez a reláció lehet egy- és kétirányú is. Egyirányú esetén csak a tulajdonos táblából érhetjük el a másik táblát. Kétirányú esetén továbbra is csak a tulajdonos tábla tartalmazza az idegen kulcsot, de már a másik táblából, más néven a reláció inverz oldalából is elérhetjük a tulajdonos tábla hozzá tartozó egy vagy több elemét.

### @OneToOne

Olyan kapcsolatot jelöl, ahol egy entitásnak egy példánya a megjelölt mező entitásának csakis egy darab másik példányával áll kapcsolatban. Az alkalmazásomban ezt nem használtam fel, így abból nem tudok példát mondani, de ilyen lehet például egy munkahely alkalmazottai és a parkolóhelyeik kapcsolata. Minden alkalmazott legfeljebb egy parkolóval rendelkezhet, és minden parkolóhelyhez legfeljebb egy alkalmazott tartozhat.

### @ManyToOne és @OneToMany

Olyan relációnak a két oldalát jelölik, ahol az egyik entitásból több példány is tartozhat egy másik entitás egy példányához. Ilyen például az alkalmazásomban a szobákat leíró Room és a foglalásokat leíró Reservation entitások kapcsolata. Feltéve, hogy az időintervallumok nem ütköznek, egy szoba tartozhat egyszerre több foglaláshoz is az adatbázisban, de egy foglalás csak egy darab szobára szólhat.

### @ManyToMany

Olyan relációt jelöl, ahol egy entitásnak több példánya állhat kapcsolatban egy másik entitás több példányával is. Alkalmazásomban ezt a kapcsolatot a felhasználókat leíró User, és a szerepköröket leíró Role entitások között használtam fel. Azért van itt szükség erre a kapcsolási módra, mert egy szerepkör tartozhat több felhasználóhoz is, és egy felhasználó egyszerre több szerepkörrel is rendelkezhet.

### Spring Data

Egy Spring modul, aminek célja, hogy egy absztrakciós réteget képezzen az üzleti logikánk és a JPA között. Ennek eléréséhez szüksége van valamilyen JPA implementációra. Ez alapértelmezésben a Hibernate lesz, de tetszés szerint megadhatunk mást is.

Főbb előnyei közé tartozik a kód nélküli repository-k bevezetése. Előre definiál több repository interfészt, mint például CrudRepository, vagy JpaRepository. Ezek az interfészek egy adott entitás eléréséhez szükséges metódusokat adják meg. A saját repository interfészünknek csupán ki kell valamelyiket terjesztenie, és meg kell adnia neki az adott repository által kezelt entitás osztályát és az azonosítójának típusát. Ezután már egyből használhatjuk a különböző interfészek által kínált metódusokat, mint például save, findById, count.

Viszont ezeknél a műveleteknél valószínűleg kicsit bonyolultabb lekérdezéseket szeretnénk végezni. A létrehozott repository-ban megadhatunk saját metódusokat is, melyekhez a @Query annotációval valamilyen saját lekérdezést rendelhetünk. Ám a Spring Data JPA képes arra is, hogy bizonyos metódusneveket értelmezzen, és ezekhez futásidőben előállítsa hozzájuk a kívánt lekérdezést. Ezeknek a metódusneveknek viszont meg kell felelnie egy előre meghatározott konvenciónak, hogy értelmezni tudja. Az ilyen metódusokat elnevezett lekérdezéseknek (named queries) nevezzük. Használatukat az adatelérés réteg ismertetése során példákon keresztül mutatom majd be.

A repository interfészünket megjelölhetjük a @Repository annotációval, melynek célja, hogy jelezze a Spring felé, hogy az alábbi interfész egy repository. Ez segít a komponensek szkenneléskor való keresésben, a kiadott SQL utasításokból kapott kivételek lefordításában és a tranzakciók kezelésében. Viszont használata csupán ajánlott, nem kötelező. A Spring az annotáció nélkül is képes megtalálni, feltéve, hogy a neve megfelel a konvencióknak, és kiterjesztik a megfelelő repository interfészeket.

## Spring Web MVC

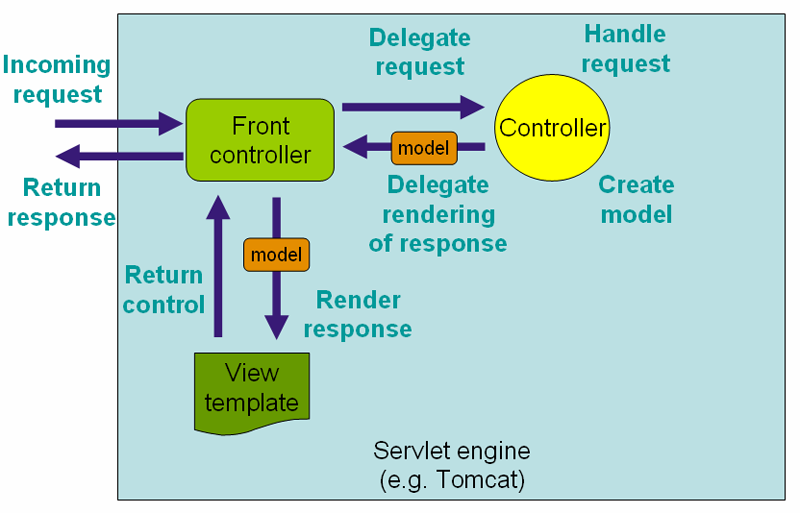
### Modell-Nézet-Vezérlő

A Modell-Nézet-Vezérlő (Model-View-Controller vagy MVC) egy tervezési minta melyet a kezelői felületek készítése, adatok továbbítása és vezérlési logika megadására használnak. A következő három részből áll:

* Nézet: A prezentációs réteg. Itt tud a felhasználó kéréseket küldeni, és itt látja majd azoknak az eredményét.
* Vezérlő: Ez a réteg fogadja a kéréseket. Meghatározza, hogy a nézet hogyan reagál a különböző kérésekre.
* Modell: Ez az objektum felel a rétegek közötti információ továbbításáért. A felhasználó a kéréseivel ezt módosítja, és a vezérlő ebben helyezi el a kérések eredményét.

Az MVC minta bevezetése előtt ezek a szerepkörök összefolytak, aminek eredménye egy nehezen karbantartható kódbázis volt. A felhasználói felület változtatása például okozhatta az adatkezelés logikájának változását is. Szétválasztásuk megkönnyíti a különböző rétegek egymástól független fejlesztését, és azok újra felhasználhatóságát.

### DispatcherServlet

Más MVC keretrendszerekhez hasonlóan a Spring Web MVC központjában is a kérések állnak. Vagyis az alkalmazás funkcióinak célja mindig valamilyen beérkező kérés kielégítése. A DispatcherServlet egy olyan servlet, amely elvégzi ezen kérések feldolgozását és továbbítását a különböző vezérlőkhöz. Teljes mértékben integrálva van a Spring IoC konténerrel, így támogatja a Spring minden más szolgáltatását is, mint például Bean-ek definiálását. Tekintsük meg az következő ábrát, melyen a kérések feldolgozásának folyamata látható (ld. 2. ábra).

2. ábra A Spring Web MVC DispatcherServlet kérés feldolgozási folyamata

A beérkező kérések először a Front controller-be jutnak. Ez egy tervezési minta megvalósítása, mely kimondja, hogy minden kérést egyetlen központi mechanizmussal kezeljünk. Ez a mechanizmus a HandlerMapping, mely elemzi a beérkező kéréseket, és az adatokat tartalmazó modell objektummal továbbítja azokat a megfelelő vezérlő metódusoknak.

A vezérlő elvégzi a szükséges műveleteket, és szükség esetén visszaad egy modellt, ami tartalmazza annak eredményét. Szintén visszaadhatja még annak a nézetnek a nevét is, amire a kérés teljesítése után navigálni kell.

Az MVC tervezési minta nem különíti el a bejövő kérések kezelését, az üzleti logika alkalmazását, és az adatbázis elérést. Ez mind a vezérlő feladatai közé tartozik. A Spring MVC-ben viszont lehetőségünk van erre külön komponenseket létrehozni.

A vezérlőből kapott eredmény és a nézet neve a ViewResolver-hez kerül, mely továbbítja azt a megfelelő nézetet előállító technológiának. Ez lehet például a Thymeleaf, vagy a JSP template engine, mely átalakítja a kapott modellt egy számára érthető formátumba, és előállítja az új nézetet, amit a felhasználó megkap.

## Spring Security

Alkalmazásunkhoz nem szeretnénk, hogy egy egyszerű felhasználó teljes hozzáféréssel rendelkezzen, mivel ezzel komoly károkat okozhat. Szeretnénk bizonyos funkciók, vagy oldalak elérését úgy korlátozni, hogy azt csak a megfelelő jogosultsággal rendelkező felhasználó érhesse el.

A Spring Security egy olyan keretrendszer, mely lehetővé teszi különböző felhasználók azonosítását (authentication), vagyis valamilyen adat, például egy név és jelszó páros alapján a beazonosítását. Valamint a bejelentkezett felhasználók engedélyezését (authorization), ami az a folyamat, amikor megmondjuk, hogy egy adott felhasználó milyen jogosultságokkal rendelkezik.

Úgymond *out of the box* működik, tehát már a projekt függőségeihez hozzáadva nyújt egy alapértelmezett bejelentkezési oldalt, és korlátozza a hozzáférést minden erőforrásunkhoz. Ezt az alapbeállítást természetesen felülírhatjuk saját konfigurációnkban.

Szintén lehetőségünk van különböző szerepkörök megadására. Ezeket a szerepköröket a felhasználókhoz rendelve egyszerűen megadhatjuk, hogy milyen jogosultságokkal rendelkeznek.

## Spring Boot

Az imént felsorolt három keretrendszeren kívül még rengeteg más Spring projekt áll rendelkezésünkre, mellyel tovább bővíthetjük az alapul szolgáló Spring Core funkcionalitásait. Viszont ennek megvan a maga hátulütője. Mint már korábban is említettem, a személyre szabhatóság és a különböző igények kielégítése fontos szerepet játszik a Spring-ben. Ebből adódóan egy alkalmazást sokféleképpen lehet felépíteni, aminek beállításaival rengeteg idő elmehet, amit funkciók fejlesztésével tölthettünk volna. Ráadásul ezek sokszor nem is térnek el két projekt között, így ugyanannak a gyakori beállításnak a megadásával töltjük az időt.

A hivatalos dokumentáció szerint: „A Spring Boot segít olyan önálló, nagyvállalati minőségű Spring-alapú alkalmazások létrehozásában, amelyeket minimális erőfeszítéssel futtathatunk is.”. Ennek elérése érdekében egy alapértelmezett beállítást nyújt bizonyos modulokhoz.

Hagyományos Spring alkalmazások kihelyezéséhez a projektet futtatható war állományba kell csomagolni, majd ezt egy servlet-nek átadni. Spring Boot esetében lehetőségünk van az egész projektet egy beágyazott servlet-tel együtt egy jar fájlba csomagolni, és az egészet csupán a java -jar paranccsal futtatni.

## Thymeleaf

A Thymeleaf egy modern szerver oldali Java sablon motor, mely dinamikus web alkalmazások fejlesztésére lett kifejlesztve. Használatával a fejlesztők könnyen tervezhetnek különböző HTML oldalakat. A weboldal kódjában megadhatjuk, mely különböző modell attribútumainkat szeretnénk felhasználni, és ebből előállítja a végleges oldalt, amit a felhasználó látni fog majd. Lehetőségünk van különböző kódrészletek beszúrására is fragment-ek segítségével. Ezeket arra használtam fel, hogy ugyanazt a navigációs menüt több oldal felső részére is beszúrjam. Különböző kifejezésekkel más logikát is adhatunk az oldalunkhoz. Például th:text-tel szöveget szúrhatunk be egy adott elembe, th:if-fel valamilyen feltételhez köthetjük egy elem megjelenítését, vagy th:action-nal meghívhatjuk egy @Controller-ben megadott metódusunkat.

## Bootstrap

A Bootstrap egy CSS keretrendszer, mely előre létrehozott CSS elemeket ad, amit felhasználhatunk kódunkban. Ezek lehetnek például gombok, form-mezők, táblák és még sok más. Alkalmazásomban többnyire gombok formázására és konténerek elrendezésére használtam, de sok esetben saját CSS kódot írtam, amely megfelelt a konkrét igényeimnek.

## Docker

Valószínűleg voltunk már olyan helyzetben, hogy a megírt kódunk tökéletesen működött a saját gépünkön, de egy másik eszközön nem sikerült lefuttatni. Ennek oka leggyakrabban az, hogy eltér a két környezet, amiben az alkalmazás fut. Ilyen problémák lehetnek például eltérő Java verziók, hiányzó függőségek vagy hibás beállítások.

A probléma áthidalására számos megoldás született. Az első megközelítés olyan konfiguráció kezelő eszközök voltak, mint például az Ansible vagy Puppet. Ilyen eszközökkel fejlesztők az alkalmazáshoz különböző szkripteket írtak, amelyek feladata többek között különböző függőségek telepítése, fájlok kezelése és hálózati kapcsolatok kezelése. Viszont megírásuk rengeteg időt vesz el a fejlesztőktől.

Egy másik megközelítés a virtuális gépek használata. Egy virtuális gép egy olyan számítógépes program, amely egy valódi számítógép működését szimulálja. Egy számítógépen több virtuális gépet is futtathatunk, melyek különböző erőforrásait egy Hypervisor kezeli. Egy virtuális gép magába foglal mindent, ami az alkalmazások futtatásához szükséges, beleértve az operációs rendszert, a különböző függőségeket, beállításokat és magát a futtatandó alkalmazást. Egy virtuális gép képes egyszerre több alkalmazást is futtatni az adott környezetben. Az így létrehozott virtuális gép image-t futtathatjuk más gépeken, vagy kihelyezhetjük egy szerverre az alkalmazás egyszerű futtatása érdekében. Viszont a fő előnye egyben a legnagyobb hátránya is. Az ilyen image-ek kihelyezésével teljes operációs rendszereket is kihelyezünk szerverünkre, amely sok alkalmazás esetén már jelentős mennyiségű tárhelyet is elfoglalhat. Ráadásul a fejlesztőknek a virtuális gép elkészítésével is sok időt kell eltölteniük.

Végül elérkeztünk a konténerekhez. Ma már számos program közül választhatunk konténerek kezelésére. Ezek közül a legelterjedtebb a Docker. Míg a virtuális gépek egy teljesen önálló gépet szimulálnak, amik nem is tudnak az őket futtató rendszerről, addig a konténerekre tekinthetünk úgy, mint egy gépen futó alkalmazásokra. Konténereknek megszabhatjuk, hogy gépünk mely erőforrásaihoz és milyen módon férhetnek hozzá. Egy konténer tartalmaz egy futtatandó alkalmazást, valamint a hozzá szükséges függőségeket, fájlokat, beállításokat.

\*kép virtual vs container\*

A Docker image-ek utasításokat tartalmaznak egy konténer létrehozásához. Általában egy image felhasznál más iamge-eket is saját feladatuk ellátásához. Például alkalmazásomban felhasználtam az eclipse-temurin:17-jdk-alpine image-t az alkalmazás futtatásához. A különböző image-ek előállításához és futtatásához szükséges lépéseket a Dockerfile-ban egyszerű utasításokkal adhatjuk meg, így nem kell a fejlesztőknek bonyolult konfigurációs fájlokat kezelnie. Egy image futtatásával egy konténert hozunk létre. A Dockerfile minden utasítása az elkészült image egy rétege lesz. Amennyiben a docker build paranccsal felépítünk egy iamge-t, ezután valamilyen felhasznált fájlt, vagy csak a Dockerfile egy sorát megváltoztatjuk, majd újra felépíjük, akkor csak a megváltozott rétegtől kezdődően hajtja végre az utasításokat. A futtatással megadhatunk további beállításokat a konténerünk számára. Fontos megjegyezni, hogy amikor leállítunk egy konténert, azzal annak minden tárolt adata elveszlik hacsak nem tároljuk el külön volume-okon vagy a saját fájlrendszerünkben. A Docker Compose eszköz segítségével több konténer együttműködéséből álló alkalmazásokat hozhatunk létre.

## Lombok

A Java nyelv hírhedt arról, hogy rengeteg boilerplate kódot tartalmaz. Ezek olyan egyszerű kódrészek melyeket több helyen is meg kell ismételnünk minimális változtatásokkal. Ilyenek például a privát mezőkhöz létrehozott getter és setter metódusok vagy egy osztály különböző konstruktorai.

A Lombok célja, hogy csökkentse az ilyen jellegű kódismétlést azáltal, hogy különböző annotációkat nyújt, melyekkel automatikusan legenerálhatjuk azokat. Például a @Data annotáció magába foglalja a @Getter, @Setter, @ToString, @EqualsAndHashCode és a @RequiredArgsConstructor annotációkat, melyekkel helyettesíteni tudjuk egy osztály ismétlődő kódjának nagy részét. Továbbá a @Builder annotációval megvalósítja a Builder tervezési mintát. Ennek elérésére egy könnyen kezelhető felületet ad változó számú konstruktor paraméterrel rendelkező osztályok példányosítására.

\*ábra user builder a teszt-ből\*

# A felület bemutatása

## Felhasználói fiók kezelése

Az alkalmazás legtöbb funkciójának, mint például a szobák lefoglalásának és értékelés írásának eléréséhez bejelentkezés szükséges. Ez azt jelenti, hogy aki használni akarja, annak először regisztrálnia kell. Ezt a <http://localhost:8080/hotelbooking/register> címen, vagy az oldal jobb felső sarkában lévő regisztráció gombbal teheti meg. A regisztráció oldalon meg kell adnia a kívánt felhasználónevét, a jelszavát és a saját email címét. Feltétel, hogy a felhasználónév egyedi és legalább 4, legfeljebb 18 karakter hosszú legyen. A jelszó legalább 8, legfeljebb 18 karakter hosszú legyen. Az email feltétele, hogy a saját valós email címét adja meg a felhasználó, amihez ő hozzá is fér. Amennyiben ezen feltételek valamelyike nem teljesül, a felhasználó megfelelő hibaüzenetet fog kapni az adott hibás mezőre. Ha minden mező megfelel a feltételeknek, akkor az alkalmazás létrehoz egy új inaktív felhasználót a megadott adatokkal és átirányít az email visszaigazolás oldalra.

A regisztrációval a felhasználó kap egy emailt, ami tartalmazza a fiókjához tartozó egyedi aktiváló linket. A link két részből áll. Az alap link, ami az aktiváló metódushoz vezet és az egyedi aktiválási azonosító, amit a metódus paraméterül kap. Példa: <http://localhost:8080/hotelbooking/register/confirm-email/confirm-token?confirmationToken=fafd61b2-7f74-438a-ad8c-adeecaa59969>

\*kép az e-mailről\*

Előfordulhat, hogy valami hiba történik az e-mail küldése során, és a felhasználó nem kapja meg az aktiváló linket. Ebben az esetben kérheti új e-mail küldését a ’Visszaigazoló e-mail újra küldése’ gombbal.

\*kép a confirm-e-mail oldalról\*

Bejelentkezett felhasználónak szintén lehetősége van a saját jelszavának megváltoztatására. Ezt a „Fiókom” menüpontban teheti meg. Itt meg kell adnia a régi jelszavát, valamint az új jelszót kétszer. Az új jelszónak ugyanúgy meg kell felelnie az eredeti jelszóra vonatkozó karakterlimitnek.

\*kép a form-ról\*

## Szobák keresése

Szobákat a kezdőlapon tudunk keresni, amit minden felhasználó elér a <http://localhost:8080/hotelbooking/home> címen. Itt található a szobakereső, ami által különböző feltételeknek megfelelő szobákat kereshetünk. Megadhatjuk, hogy hány darab egyfős ágy, franciaágy legyen. Kiválaszthatjuk a szoba típusát, ami lehet például családi szoba, egyágyas szoba és még sok más. Választhatunk a rendszerben lévő konkrét szállodák közül, vagy csupán megadhatjuk, hogy mely városban lévő szállodák között keressen. A megadott paramétereket először itt is megvizsgáljuk, nincs-e közte érvénytelen érték, mint például negatív érték az ágyak számánál. Ezen mezők kitöltése mind opcionális. Egy üresen hagyott mező azt jelenti, hogy arra a tulajdonságra nem végzünk szűrést, így minden értéket visszakapunk.

Amit viszont muszáj kitölteni az az érkezés és távozás napja. Itt először megvizsgáljuk, hogy egyik sem múltbeli érték, és a tervezett távozás napja az érkezés után van. Minden szobához tartozik egy lista, ami tartalmazza a hozzá tartozó különböző foglalásokat. Amikor elérhető szobákat keres az alkalmazás, akkor végigmegy ezen a listán és megvizsgálja, hogy egy sem ütközik ezzel az időintervallummal. Bevett szokás az, hogy egy szállásról délelőtt kell távozni, és délután már új vendégnek adják ki a szobát. Ez egy adott szoba esetében azt jelenti, hogy ha a tervezett érkezésünk napja megegyezik egy már meglévő foglalás távozás napjával akkor az még nem kizáró ok. Szintén megengedett az, ha a tervezett távozásunk napja megegyezik egy meglévő foglalás kezdetének napjával. Az időintervallumok ütközésének vizsgálatát bővebben ki fogom fejteni a Service réteg bemutatása során.

Tegyük fel, hogy helyesen adtuk meg a keresési feltételeket, és a rendszer talált is ezeknek megfelelő szobákat. Ez esetben a különböző szobákat szállodáik szerint csoportosítva listázzuk ki. Minden szállodának láthatjuk a képét, a nevét, a várost, ahol található és az átlagos értékelését. Ha az egeret rávisszük egy kiválasztott szállodára, akkor egy újabb listát láthatunk annak szobáiról.

Egy szálloda rendelkezhet több megegyező szobával is. A felhasználónak nem szükséges ezeket mind látnia. Szimpla kilistázásuk helyett kereséskor egy szálloda szobáit különböző tulajdonságaik alapján csoportosítjuk. Ebből az egyszerűsített listából sokkal könnyebben lehet válogatni. Tételezzük fel, hogy a szállodán kívül nem adtunk meg más opcionális feltételt. Ennek a szállodának a kiválasztott időintervallumban legyen három elérhető szobája. Az első és a második megegyezik, mert azonos az egy- és kétszemélyes ágyak száma, egy éjszaka és a teljes foglalás ára, valamint a szobák típusa. A harmadik szobánál is ez mind megegyezik, leszámítva azt, hogy eggyel több az egyszemélyes ágyak száma. Mivel az első két szoba ugyanolyan, így ebből a típusból elég csak egyet listázni. A harmadik szoba különböző, így ez egy külön elem lesz a listában. A keresés végeredménye egy darab szálloda lesz, aminél kétféle szobát tudunk lefoglalni.

\*kép a szoba keresésről\*

## Szobák lefoglalása

Azzal, hogy a felhasználó kiválasztja a neki tetsző szobát, átkerül a Szobafoglalás oldalra. Amennyiben eddig nem volt bejelentkezve, előbb a „Bejelentkezés” oldalra kerül, majd sikeres belépés után lesz ide továbbítva. Itt egy helyen láthatja a tervezett foglalás adatait, mielőtt még azt véglegesítené.

Előfordulhat, hogy amíg a felhasználó a kezdőlapon tartózkodott, vagy a foglalása tervét ellenőrizte, addig más felhasználók befoglalták az összes szobát az adott időpontban. Amikor a felhasználó a foglalását véglegesíti, az alkalmazás újra átvizsgálja a szobákat és keres egy olyat, ami megegyezik a kiválasztott összes releváns tulajdonságával. Ez a keresés más, mint amikor a kezdőoldalon opcionális mezőket üresen hagyunk, mivel itt már a kiválasztott kategória minden tulajdonságára szűrünk. Ezáltal csak azokat kapjuk vissza, amelyek megegyeznek a kiválasztott tulajdonságaival. Ha tényleg megtörténne az, hogy időközben befoglalták az összes ilyen szobát, akkor a felhasználót átirányítjuk a kezdőlapra, és hibaüzenetet kap a sikertelen foglalásról. Ekkor, ha megpróbál újra rákeresni az adott szobára, akkor már nem fog ilyet találni, mivel mind foglalt.

Amennyiben mégis találunk megfelelő szabad szobát, akkor lefoglaljuk azt. A sikeres műveletről a felhasználó e-mailt kap, ami tartalmazza a foglalás adatait, beleértve a lefoglalt szoba számát is. A foglalás közben generálunk egy egyedi azonosítót, amit szintén elküldünk az e-mailben. A felhasználók szállodába érkezéskor a recepción ezt a kódot felmutatva tudnak be- és kijelentkezni. Ha a művelet sikeresen megtörtént, átirányítjuk a felhasználót a „Foglalásaim” oldalra.

## Foglalások kezelése

A „Foglalásaim” oldalon tekintheti meg a felhasználó az általa lefoglalt szobákat. Itt többek között megtekinthető a szoba száma és a foglalás állapota is. A tervezett állapot olyan foglalásokat jelöl, ahova a felhasználó még nem jelentkezett be. Az ilyen állapotú foglalásokat lehetősége van törölni. Ám mielőtt ez megtörténne, fontos ellenőrizni a kérést. Habár a felületen a „Törlés” gomb csak tervezett foglalásoknál jelenik meg, rossz szándékú felhasználók megpróbálhatnak olyan kérést küldeni a rendszernek, ami egy aktív, vagy már befejezett foglalást próbál törölni. Az is lehet, hogy egy olyan kérést küld, ami egy másik felhasználó foglalását törölné. Ezen problémák elkerülése érdekében törlés előtt vizsgáljuk a foglalás állapotát és azt is, hogy a valódi tulajdonosa küldte -e a kérést. Ha ezek valamelyike nem teljesülne, akkor a felhasználó hibaüzenetet kap.

\*kép a foglalásaim oldal-ról\*

## Szállodák értékelése

Ha egy vendég meg volt elégedve a szállásával, vagy netalán valamilyen rossz élménye volt, arról egy értékelés írásával adhat visszajelzést. A „Foglalásaim” oldalon minden befejezett foglalás sorában megjelenik egy „Értékelés” gomb. Ez átnavigál az „Értékelés” oldalra. Itt egy 1-től 5-ig terjedő skálán értékelheti a szállodát. Ha valamilyen konkrét véleményt szeretne megfogalmazni, azt a megjegyzés rovatban teheti meg. Értékelése írásával frissül a szálloda átlagos értékelése is. Így, ha a kezdőlapon újra rákeresünk akkor már az új átlagos értékelés lesz látható.

\*kép a review page-ről\*

## Az adminisztrátor felület

Közönséges felhasználókon kívül az alkalmazást adminisztrátorok is használni fogják. Jelenleg az alkalmazásban egy adminisztrátor van, amit az első indításkor automatikusan hozunk létre. Így nem szükséges külön e-mail-címmel regisztrálnia. Bejelentkezéshez ugyanazt az oldalt használja, mint a közönséges felhasználók. Míg a felhasználók alapértelmezésben a kezdőlapra, addig az adminisztrátorok az „Adminisztráció" oldalra lesznek átirányítva sikeres belépést követően.

Adminisztrátorok képesek szállodákat felvinni a rendszerbe. Ezt a „Hotelek hozzáadása” oldalon tehetik meg. Itt meg kell adni a kívánt szálloda nevét, a várost, ahol található. és fel kell tölteni egy képet róla. A „Szobák hozzáadása” oldalon adhatnak hozzá szobákat a már létező szállodákhoz. Itt adják meg, hogy hány darab egy- és kétszemélyes ágy van a szobában, mennyibe kerül egy éjszakára lefoglalni, milyen típusú a szoba, és melyik konkrét szállodához tartozik.

Korábban említettük, hogy egy szoba lefoglalásakor a felhasználók egy egyedi azonosítót kapnak e-mailben. Egy vendég bejelentkezésekor ezt az adminisztrátorok a „Foglalások kezelése” oldalon tudják kezelni. Az azonosító alapján lekérdezik a foglalás adatait és bejelentkeztetik. Amikor a vendég távozna, akkor ugyanúgy az azonosítót felmutatva kijelentkeztetik.

\*kép a check-in/out oldalról\*

## Navigációs menük

Minden oldal tetején található egy navigációs menü, ami az adott felhasználóhoz tartozó különböző oldalak gyors elérését biztosítja. Egy be nem jelentkezett felhasználó számára csupán a nyelv kiválasztására szolgáló legördülő menü, valamint a bejelentkezés és regisztráció gombok lesznek elérhetőek. Bejelentkezett felhasználók és adminisztrátorok ehelyett egy kijelentkezés gombot fognak látni. Ezen kívül a felhasználók még láthatják a „Foglalásaim” és a „Fiókom” oldalra vezető gombokat, míg az adminisztrátorok az „Admin” felületre tudnak egyszerűen navigálni. A bal felső sarokban lévő logóra kattintva mindenki a kezdőlapra tud navigálni.

# Az alkalmazás felépítése

## Monolitikus architektúra

Nagyvállalati alkalmazások fejtése során tapasztalhatjuk, hogy az idő elteltével a kód egyre összetettebb és nehezen átláthatóbb lesz. Mindig lesz igény valamilyen új üzleti logika bevezetésére, vagy esetleg a felhasználói felület változtatására. Ezek sokszor csupán apró változtatások, amik egymástól teljesen függetlenek, és önmagukban nem járnának sok munkával. A problémát az okozza, hogy megfelelő tervezés hiányában ezek a feladatkörök nem lesznek megfelelően elkülönítve. Ennek az eredménye egyetlen nagy részből álló monolitikus kódbázis lesz. Ilyen esetben egy feladatkört ellátó kódrészlet szigorúan függ attól, hogy egy másik, tőle viszonylag független hogyan dolgozik. A fejlesztés kezdetekor talán még könnyebbnek tűnhet ez a megközelítés, mert még átlátjuk azt a kevés és viszonylag egyszerű kódunkat. Ez viszont hamar változni fog. A folyamatos fejlesztés során egyre nehezebb lesz egy új funkciót implementálni, egy már meglévő függőséget nagyon nehéz lesz lecserélni, vagy valahol egy apró változtatás a kód sok más részét tönkre is teheti. Erre példa az adatbázis lecserélése. Egy monolitikus alkalmazás valószínűleg szorosan függeni fog attól, hogy milyen konkrét adatbázist használunk. Lehetséges, hogy a felhasználói felület közvetlen SQL kéréseket küld az adatbázisnak. Ez megnehezíti az átállást, mert előfordulhat, hogy az új adatbázis más SQL szintaxist használ, és emiatt módosítani kell a felhasználói felületet is. Sokkal hatékonyabb lenne, ha az adatbázissal való kommunikációért felelős kódrészletet el tudnánk különíteni az alkalmazás többi részétől, így az ilyen változtatások nem hatnának ki az egész alkalmazásra. Ezen az elképzelésen alapul a többrétegű architektúra is.

## Többrétegű architektúra

Ahelyett, hogy az alkalmazásunkat egy nagy tömbként kezelnénk, sokkal célszerűbb azt kisebb részekre bontani. Erre egy megközelítés az, hogy az összefüggő feladatköröket önálló rétegekbe rendezzük, melyek egy meghatározott szabály alapján épülnek egymásra. A felső rétegek használják az alattuk lévőket, de az alsó rétegek nem látják mi van felettük. Továbbá a rétegek általában elrejtik az alattuk lévőket, így egy felső csakis a közvetlen alatta lévőket látja. Alkalmazásomban a felhasználói felületet, az üzleti logikát és az adat elérést rendeztem három külön rétegbe, melyeket a következő részekben részletesen be is mutatok.

Egy ilyen felépítésnek számos előnye van. Sokkal átláthatóbbá teszi a különböző rétegek feladatkörét, így könnyebbé teszi azok megértését. Csökkenti a rétegek közötti függőséget. Egy rétegnek csak azt kell megmondania, hogy milyen funkciókra van szüksége az alatta lévőtől. Az, hogy ezek hogyan valósulnak meg, már nem tartozik rá. Például az üzleti logika számára lényegtelen, hogy milyen adatbázist használunk. Ha az adat elérésért felelős réteg az elvárt módon viselkedik, akkor gondtalanul működik tovább.

Egy réteget tovább bonthatunk kisebb modulokra is, melyek az adott réteg felelősségi körén belül különböző feladatokat látnak el. Tegyük fel, hogy a grafikus felhasználói felületen kívül szeretnénk, ha az alkalmazásunk REST API-on keresztül is elérhető lenne. Ehhez csupán annyit kell tennünk, hogy létrehozunk egy új modult, ami a REST kéréseket kezeli. A rétegekre bontás megkönnyíti az egyes részek újrafelhasználhatóságát is. Ha ennek a két felületnek ugyanazokkal a funkciókkal kell rendelkeznie, akkor felhasználhatjuk a már meglévő üzleti logikát kezelő moduljainkat.

A rétegekre bontás sokat segít, de nem old meg minden problémát. Előfordulhat, hogy a felhasználói felületen szeretnénk, ha a szállodák konkrét címe is megjelenne. Ez egy olyan változtatás, amit nem lehet csak egy rétegen végrehajtani. Ezt a mezőt fel kell venni az adatbázisba, szükség van üzleti logikára, ami kezeli, és csak utána tudjuk megjeleníteni. Szintén fontos megemlíteni a teljesítmény csökkenését. Még ha minimálisan is, de csökkenti a kommunikáció sebességét az, hogy egy kérésnek akár több rétegen is át kell jutnia.

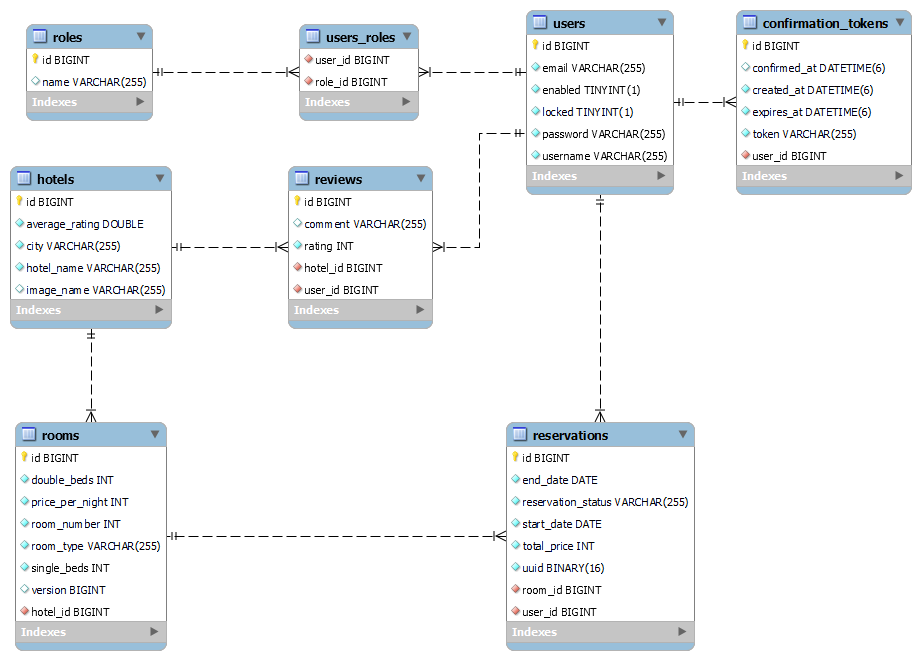
Az alkalmazásom rétegeinek felépítését az alábbi ábra szemlélteti. Itt jól látható, hogy az egyes rétegek az alattuk lévőknek mindig csak kéréseket küldenek, míg a fölöttük lévőnek csak is ezen kérésekre adott választ küldik vissza. Továbbá az is látható, hogy a rétegek csakis a közvetlen szomszédjaikkal kommunikálnak.

\*rétegeket bemutató ábra\*

## Adatelérés réteg

Az adatelérési réteg feladata különböző adatforrások kezelése. Egyik leggyakoribb feladata az adatbázisokkal való kommunikáció CRUD műveletek végrehajtása érdekében. Az ilyen műveletek az adatok létrehozását, olvasását, frissítését és törlését jelentik. Alkalmazásomban a hotel-booking-app-presistence modul alkotja ezt a réteget, mely a MySQL adatbázissal való kommunikációért felel.

### Az adatbázis táblái

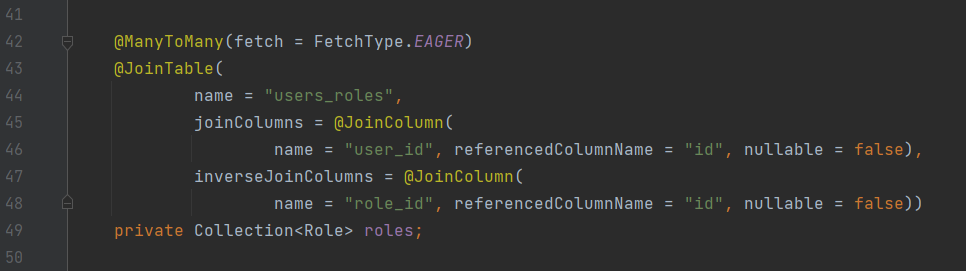


3. ábra Az adatbázis ER diagramja

Alkalmazásunk különböző felhasználóit a users táblában tároljuk. Itt található az egyedi felhasználónév és egyedi e-mail-cím. A jelszavakat egyszerű szöveg helyett titkosítva tároljuk. Az aktivált (enabled) és zárolt (locked) boolean mezők különböző okokból korlátozhatják a felhasználói fiók elérhetőségét. A zárolt tulajdonságot gyakran arra használhatjuk, hogy egy adott sikertelen bejelentkezési próbálkozás után rövid időre zároljuk a fiókot. Ilyen funkciót nem vezettem be, így ez a mező mindig hamis lesz. Az aktivált mezőt arra használjuk, hogy a fiók elérhetőségét valamilyen feltételhez kössük, mint például az e-mail-cím visszaigazolása. Egy újonnan regisztrált felhasználó esetében ez a mező hamis lesz, és csak akkor vált igaz-ra, ha visszaigazolja az e-mail-címét.

Regisztrációval minden felhasználónak küldünk egy visszaigazoló e-mailt, ami tartalmaz egy egyedi azonosítót. Ez egy Univerzálisan Egyedi Azonosító (UUID), amit a confirmation\_token táblában tárolunk. Ehhez többek között tartozik egy mező, ami megmondja, mikor lett aktiválva, vagy mikor jár le. Lejárt kulcs esetén a felhasználó kérheti új e-mail küldését, amivel egy új kulcsot is generálunk.

A felhasználókhoz tartozó különböző szerepköröket a roles tábla tárolja. Alkalmazásomban két szerepkört hoztam létre: USER, amit az általános felhasználókhoz és ADMIN, amit az adminisztrátorokhoz rendelek. Ezek határozzák meg, hogy egy felhasználó mely felületekhez és funkciókhoz fér hozzá.

Mint korábban említettem, a roles és a users tábla között @ManyToMany kapcsolat áll fenn. Ezt a kapcsolatot a users\_roles @JoinTable segítségével valósítom meg. Ez a tábla tartalmazza az összes felhasználó és szerepkör párost. Ha például a rendszerben három felhasználóm van, melyből az első ADMIN, a második USER és a harmadik rendelkezik mindkét szerepkörrel, akkor a users\_roles tábla négy sort tartalmazna. Ebben az esetben a harmadik felhasználónk elér mindent, amihez egy ADMIN –nak, vagy egy USER-nek hozzáférése van.

ábra 3 join table kapcsolálsi mód definiálása a User entitás osztályban

A hotels táblában tároljuk többek között a szállodák nevét és a várost, ahol található. Azokat a képeket, amiket az adminisztrátorok töltenek fel egy-egy szállodához, egy másik helyen tároljuk, és itt csupán a fájl nevére hivatkozunk az image name oszlopban. Az average\_rating oszlopban eltároljuk egy szálloda értékeléseinek átlagát, így azt nem kell minden alkalommal újra kiszámolni, ha meg akarjuk jeleníteni. Az ábrán látható, hogy a hotels táblában nincs hivatkozás a rooms vagy reviews táblára, ám ennek ellenére kódunkban mégis képesek vagyunk egy szálloda entitástól lekérdezni annak szobáit, vagy értékeléseit. Ez azért van, mert kétirányú kapcsolatban áll ezekkel a táblákkal, melynek mindkét esetben a hotels az inverz oldala. A Room vagy Review entitás osztály példányaiból továbbra is hivatkozhatunk a hozzájuk tartozó szállodákra, de ha az alábbi kódrészlet nem szerepelne a Hotel entitás osztályban, akkor ez a kapcsolat visszafelé már nem működne.

\*kódrészlet hotels rooms reviews OneToMany\*

A rooms tábla jelképezi a szállodák szobáit. Itt tároljuk például a szobaszámot, azt, hogy egy szobához hány ágy tartozik, mennyibe kerül egy éjszakára, vagy éppen melyik szállodához tartozik. Erre a táblára egy külön megszorítást szabtam ki, mely szerint a szobaszám és a szálloda azonosító párosának egyedinek kell lennie. Erre azért van szükség, hogy egy konkrét szobát be tudjunk azonosítani a rendszerben. Ezt az alábbi uniqueConstraints megszorítással értem el. Miután a Hibernate előállítja a táblát ebből az osztályból, abba külön SQL utasítással sem tudunk olyan sort beszúrni, ami megszegi ezt a megszorítást.

\*kép rooms megszorítás\*

A rooms tábla version mezőjét korábban az ütköző foglalások vizsgálatának optimalizálására használtam. Egy kiválasztott szoba foglalásainak újbóli vizsgálata előtt először csak a verziót vetettem össze a korábban vizsgált szobával. A változatlan verzió azt jelentette, hogy a szoba nem módosult, továbbra sincsenek ütköző foglalásai. Ha pedig változott, akkor újra vizsgálom a foglalásokat. Erre a logikára viszont már nincsen szükség. Az alkalmazás felhasználói jelenleg már nem konkrét szobákat próbálnak lefoglalni, hanem csak egy megegyező szobákat tartalmazó halmazból egyet.

A szobák lefoglalásait a reservations táblában tartjuk számon. Többek között tartalmazza, hogy egy felhasználó melyik szobát foglalta le és mely időközre. Az itt tárolt UUID segítségével tudják az adminisztrátorok kezelni a vendégek foglalásait.

Végül a reviews táblában tároljuk a különböző szállodák értékelését a felhasználók megjegyzéseivel és minősítéseivel.

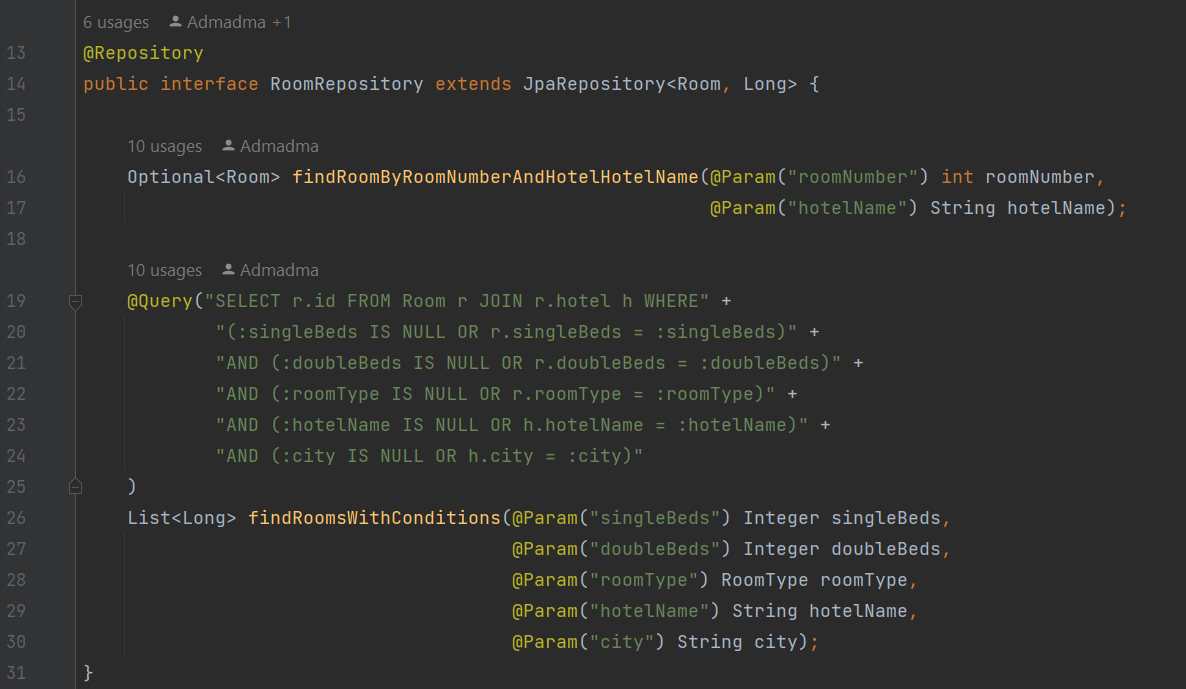
### Repository interfészek

Mint korábban említettem, a repository interfészek a különböző entitásaink eléréséhez nyújtanak különböző metódusokat. A felső rétegek ezek segítségével küldhetnek kéréseket anélkül, hogy közvetlen kapcsolatban állnának az adatbázisunkkal. A JPARepositorty interfész több hasznos metódust is magába foglal. Ilyen például a save, melynek paraméterül adva az entitás osztály egy példányát elmenthetjük azt az adatbázisban, vagy a findAll, mely visszaad egy tábla összes elemét tartalmazó listát.

Az alapértelmezett metódusokon kívül más műveleteke is szükségem volt, ezért létrehoztam több elnevezett lekérdezést is. Például szerettem volna felhasználónév vagy e-mail alapján rákeresni felhasználókra. Ezt a findBy kifejezés és a vizsgált mező nevének kombinációjával tehetjük meg. Itt a metódusban a mező nevének nagy betűvel kell kezdődnie, és a többi részének meg kell egyeznie az entitásban lévő mező nevével. Ha például az entitásomban a felhasználónevet username-ként adtam meg, akkor a metódusom nevének findByUsername kell lennie. Ha az első nagybetűn kívül más eltérés is van, mint például findByUserName akkor a JPA nem fogja megtalálni a kívánt mezőt. Amennyiben jól adtam meg a metódus nevét, az egy Optional<User> példányt fog visszaadni. Az Optional osztály a Java 8-ban lett bevezetve a null visszatérési értékek biztonságos kezelésére. Azt jelzi, hogy a visszaadott érték lehet üres is, és ezért azt ellenőrizni kell mielőtt használnánk. Amennyiben a keresési feltételnek egy elem sem felelt meg, egy üres Optional példány kapunk vissza. Vannak helyzetek, amikor csak azt szeretnénk vizsgálni, hogy létezik-e felhasználó az adott felhasználónévvel vagy e-mail-címmel. Erre az existsBy metódust használhatjuk. Ezeket a metódusokat az alább látható UserRepository interfészemben hoztam létre.

\*UserRepository kép\*

A metódus nevében megadhatunk több keresési feltételt. Ezeket az And szóval kell elválasztani. Ilyen például a RoomRepository interfészben megadott findByRoomNumberAndHotelHotelName metódus, ami az adott szálloda adott szobaszámmal rendelkező szobáját adja vissza. A Room entitások viszont közvetlenül nem rendelkeznek hotelName tulajdonsággal. Ennek eléréséhez először hivatkoznunk kell a hozzájuk rendelt Hotel entitásokra, majd ennek a hotelName tulajdonságára. Ezért szerepel a metódus nevében kétszer a Hotel szó.

Az elnevezett lekérdezések hasznosak, de van, amire nem adnak megoldást, vagy már túl összetettek lennének. A @Query annotációval lehetőségünk van egyedi lekérdezéseket rendelni a metódushoz. Erre használhatunk natív SQL lekérdezéseket. Ilyen esetben felhasználhatunk olyan különleges SQL utasításokat is, amelyek csak az adott adatbázisunkban elérhetőek. Ennek az a hátránya, hogy a kódunk szorosan függeni fog attól, hogy milyen adatbázist használunk. Ha azt le akarjuk cseréln, akkor valószínűleg módosítani kell az SQL utasításokat is. Egy másik opció a JPQL (Jakarta Persistence Query Language) lekérdezések használata, mely az entitás osztályaink alapján kommunikál az adatbázissal. A findRoomsWithConditions metódusom számára JPQL nyelven adtam meg a lekérdezést. Ez a lekérdezés először egy JOIN-nal összeköti a Room és a hozzá tartozó Hotel entitásokat, majd WHERE segítségével vizsgálja, hogy a paraméterül kapott értékek szerepelnek-e az adott szoba tulajdonságai között. Ha valamelyik paramétert nem adtuk meg akkor abból minden értéket elfogad a keresésnél. A metódus paramétereire helyezett @Param annotáció segít abban, hogy azonos típusú paramétereket a lekérdezés megfelelő mezőjéhez rendeljük, ám nincsen rá feltétlenül szükség. Ha a paraméter neve megegyezik a megfelelő mezővel, akkor gond nélkül hozzárendeli.

ábra 3 RoomRepository interfész

## Üzleti logika réteg

Az üzleti logika magába foglalja a különböző műveleteket, amit az alkalmazás végrehajt a felhasználók számára. Ide tartozik a különböző számítások elvégzése a felhasználótól kapott és az adatbázisból kinyert adatok alapján, a felhasználói felület rétegből kapott kérések érvényesítése vagy továbbítása az adat elérési réteg felé. A hotel-booking-app-service modul felelős az alkalmazásom üzleti logikájáért.

### Modell entitások és DTO

A felhasználóktól kapott legtöbb kérésnek a célja, valamilyen adatbázissal kapcsolatos művelet. Szobák hozzáadásához el kell menteni azokat az adatbázisba. Bejelentkezéshez le kell kérdezni az adott felhasználó nevét és jelszavát. Ezek a műveletek mind entitás osztályokkal dolgoznak. A probléma az, hogyha az üzleti logikánkban közvetlen ezekre az entitásokra támaszkodunk, akkor túlságosan függeni fogunk tőlük, ami szembe megy a rétegekre bontás céljával. Továbbá lehet, hogy egy entitásnak vannak olyan tulajdonságai, ami csak az adatbázisra tartozik, és nem szeretnénk, hogy a többi réteg hozzáférjen. Ilyen például az Id mező.

Valahogy mégis el kell végezni a szükséges műveleteket egy adott entitáson. Ennek érdekében létrehoztam a modell entitásokat, amik a hotel-booking-app-presentation modulban lévő entitásaimnak majdnem teljesen megegyező másai. A hotel-booking-app-service modul a saját modell entitás osztályaival dolgozik. Az adatbázisból kapott entitásokat ezekre alakítja át, és mielőtt elküldene valamit, azt ebből alakítja vissza. Ennek köszönhetően, ha valamilyen változás történne az adatelérés működésében, az kisebb eséllyel okozná az üzleti logika változását és elég lenne csak az átalakításért felelős logikát átírni.

Vannak helyzetek, amikor információt szeretnénk átadni a rétegek között, de ezek nem köthetők egy adott entitáshoz, vagy ha mégis, az túl zavaros lenne. Ebben az esetben adatátviteli objektumokat (Data Transfer Object vagy DTO) használhatunk. Ilyen például a felhasználói felület réteg RoomSearchFormDTO osztálya. A keresési feltételek, amiket a felhasználó a kezdőlapon ad meg, egy ilyen objektumba kerülnek. Ezt előbb átalakítjuk RoomSearchFormServiceDTO-objektummá és továbbítjuk a keresésért felelős üzleti logika felé. A modell entitásokra is tekinthetünk DTO-ként, hisz a feladatuk szintén objektumok továbbítása a kód különböző részei között. Csupán azért különítettem el, hogy egyértelművé tegyem, ezek konkrét entitásokat jelképeznek.

\*kép RoomSearchFormServiceDTO\*

### ModelMapper és Transformer osztályok

A Transformer osztályok azért felelősek, hogy az alsó rétegekkel való kommunikáció során átalakítsák az entitás vagy DTO objektumokat. Erre a ModelMapper programkönyvtárat használtam fel. A ModelMapper képes arra, hogy konvenciók alapján átalakítsa egy osztály példányát egy másik típusú, ám hozzá nagyban hasonló osztály példányává. Azokra a különleges esetekre, ahol a meglévő mezőkből nem tudja megfelelően elvégezni az átalakítást megadhatunk saját szabályokat.

\*kép UserTransformer\*

### RepositoryService osztályok

A service metódusokat két külön csoportba rendeztem. Ennek egyike a RepositoryService osztályok. Feladatuk a repository metódusok meghívása, és az elküldött vagy visszakapott objektumok átalakítása transformer-ek segítségével. Ha a többi service osztály az adatbázissal akar kommunikálni, azt ezeken keresztül teheti meg. Ennek egyik előnye, hogy az átalakítással csak itt kell foglalkozni. A többi service osztálynak nem kell transformer-eket használnia. A másik előnye az újra felhasználhatóság. A különböző service metódusok gyakran több különböző repository metódust is meghívnak. Ha a service osztályokban adnánk meg őket, akkor könnyen körkörös függőség alakulhat ki.

### Service osztályok

Itt találhatóak a különböző üzleti logikát megvalósító metódusok. Eredetileg egyszerű logikából állt, minden service a saját repository-jával kommunikált. Például minden szobákkal kapcsolatos logika a RoomService-be került. Ahogy a kód bővült, ennek az osztálynak is nőtt a felelőssége, és túl összetett lett. Ennek megoldása érdekében szétválasztottam két külön osztályra, melyek a szobák létrehozásáért és a szabad szobák kereséséért felelősek.

A különböző service osztályokra nem hivatkozunk konkrétan. Helyette megadunk egy interfészt, ami tartalmazza azokat a metódusokat, amikkel rendelkeznie kell egy ilyen service-nek, és egy konkrét osztály megvalósítja azokat. Ezáltal a kód lazán csatolt lesz. A különböző függőségeket ezekkel az interfészekkel adjuk meg, és a Spring az ApplicationContext-ből kiválasztja hozzá a konkrét osztályt, ami implementálja azt.

Az osztályok átláthatósága mellett fontos az is, hogy az egyes metódusok jól olvashatóak legyenek. Egy metódus, ami túl sok dolgot csinál egyszerre nehezen lesz átlátható. De van, ahol mégis szükséges egyetlen metódus hívással elindítani egy összetettebb folyamatot. Ilyen például a szobák keresése, amit csupán a searchRooms metódus végez el. Ez először lekérdezi az adatbázisból a feltételeknek megfelelő szobákat, kiszűri azokat, amiknek van ütköző foglalása, majd a megfelelő szobákat szállodáik szerint egy listába rendezi. Ha ezt mind ebben az egy metódusban, akkor más fejlesztőknek nehéz lenne megérteni a működését. A megoldás az, hogy kiszervezzük a logika különböző részeit privát metódusokba vagy akár külön service osztályokba. Ekkor a searchRooms metódusnak csupán meg kell hívnia ezeket a megfelelő paraméterekkel.

### Képek kezelése

Az adminisztrátorok által feltöltött képeket a saját fájlrendszerünkben tároljuk. A feltöltésért a FileSystemStorageService osztály felelős. A mappát, ahova a képek kerülnek az IMAGE\_FOLDER\_PATH környezeti változó segítségével adhatjuk meg. A feltöltendő fájlon elvégezzük a szükséges ellenőrzéseket. Megvizsgáljuk, hogy nem üres és hogy megfelelő-e a formátuma. A méretére vonatkozó megszorításokat már a felhasználói felületen ellenőriztük. Az elmentendő fájl nevét felülírjuk egy általunk generált UUID-val. Sikeres mentés után ezt az új nevet adjuk vissza, és ez lesz elmentve a szállodával együtt.

### E-mail

A Spring által nyújtott JavaMailSender interfész egy egyszerű felületet ad e-mailek küldésére, melyeket a Google SMTP szerverén keresztül küldjük el felhasználóinknak. Ehhez szükséges egy Google -fiók, aminek a beállításaiban létre kell hozni egy külön jelszót. Az alkalmazásunk ezt fogja használni a bejelentkezéshez.

## Felhasználói felület réteg

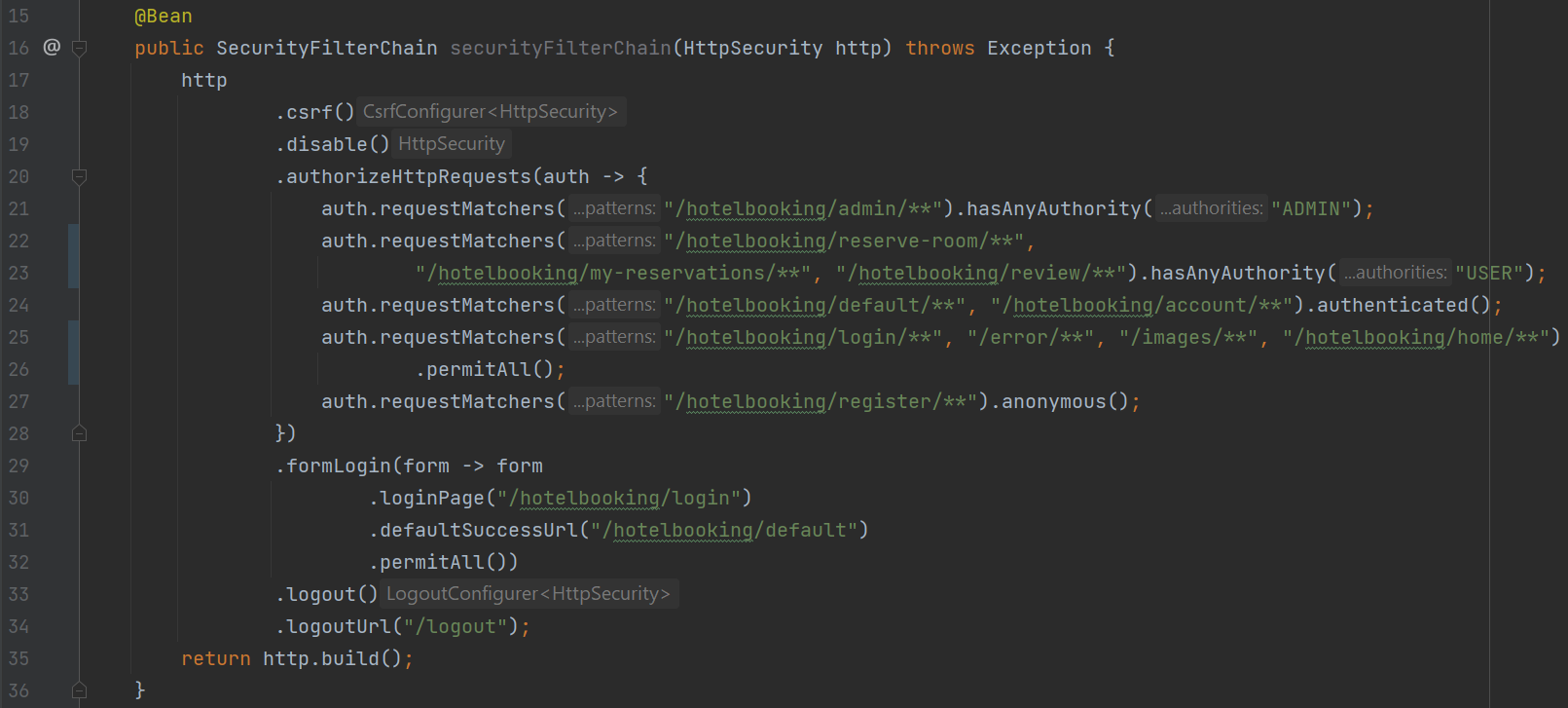
A felhasználók és az alkalmazás közötti kommunikáció kezelésére szolgál a felhasználói felület. Alkalmazásom egy webes felületen érhető el, melyet a hotel-booking-app-web modulban definiálok. Ez a modul felel a különböző weboldalak megjelenítéséért, a felhasználók által megadott valamely értékek érvényesítéséért, valamint itt található az alkalmazás elindításáért felelős main metódus, és több konfigurációért felelős osztály is.

### @Controller osztályok

A @Controller-ek olyan specifikus @Component-ek, melyekben a webről érkező kéréseket különböző metódusokhoz köthetjük. A @RequestMapping annotációval megadhatunk egy alap URL-t, ami hozzá lesz csatolva minden metódus elérési útjához az osztályon belül. Minden kérést kezelő metódus egy String értéket ad vissza. Ez annak az HTML oldalnak a neve, ahova a felhasználót átnavigáljuk a kérés teljesítése után (ez lehet akár a jelenlegi oldal is). A service metódusokból kapott objektumokat transformer-ek segítségével itt is átalakítjuk a modul saját típusaira.

\*kép AdminPageController\*

### Spring SecurityConfiguration

Szeretnénk korlátozni, hogy a különböző felhasználók milyen kéréseket küldhetnek a rendszernek. Erre a Spring Security által nyújtott SecurityFitlerChain-t használtam fel. Itt megadhatjuk például, hogy a /hotelbooking/admin URL alatt található összes erőforráshoz csak adminisztrátor joggal rendelkező felhasználók, vagy éppen a regisztráció oldalhoz csak be nem jelentkezett felhasználók férjenek hozzá. Itt adtam meg, hogy sikeres bejelentkezés után minden felhasználó a /hotelbooking/default oldalra legyen átirányítva, ahonnan a szerepkörének megfelelő kezdőlapra lesz átirányítva. Szintén ebben az osztályban definiáltam egy PasswordEncoder Bean-t, mely által az alkalmazás a BCryptPasswordEncoder-t fogja használni jelszavak titkosítására.

ábra 3 SecurityFilterChain Bean

### i18n

Általában szeretnénk, hogy weboldalunkat más nyelven beszélő felhasználók is használni tudják. Az internacionalizálás vagy más néven i18n az a folyamat, amikor a kódunkat úgy tervezzük, hogy az több különböző nyelvet is képes legyen támogatni. Ehhez szükség van egy konfigurációs osztályra, amely megvalósítja a WebMvcConfigurer interfészt, és definiálja a szükséges Bean-eket. Ezután hozzunk létre minden támogatni kívánt nyelv számára egy properties fájlt. Ennek minden sora egy üzenet kódjából és az adott nyelvű üzenetből fog állni. Innentől, ha a kódban valamilyen szöveges üzenetet szeretnénk megjeleníteni a felhasználó számára, legyen az csupán a bejelentkezés gomb neve, vagy egy hibaüzenet, azt az adott üzenet kódjára való hivatkozással tegyük.

# Unit tesztelés

Tesztek írása segít megbizonyosodni arról, hogy a kódunk adott részei megfelelően működnek. Fejlesztés során a különböző változtatások több olyan kódrészre is kihatással lehetnek, amire nem is számítunk. Ha ezek valamelyike nem működne a változtatás miatt, azt a tesztekkel egyből kiszűrhetjük. Egy másik előnye a tesztek írásának az, hogy arra készteti a fejlesztőt, hogy kisebb önálló részekben tekintsen a kódjára. Fejlesztés alatt számos osztályt bontottam több részre annak érdekében, hogy a külön részeit tesztelni tudjam. Habár a tesztek hiánya gyakran egy rossz minőségű kód előjele, azok megléte még nem garantálja, hogy hibátlan a kódunk. Vannak helyzetek, ahol egyszerűen nem tudunk minden lehetséges esetre tesztet írni, a magas teszt lefedettség nem garantálja, hogy a metódusok minden helyzetben az elvárt módon viselkednek.

A teszteknek meg kell felelniük a F.I.R.S.T elveknek. Azaz, a lehető leggyorsabban fusson le (Fast), a többi teszttől függetlenül működjön (Isolated), újra futtatás esetén ugyanaz legyen az eredménye (Repeatable), egyértelműen igazolja, hogy egy kódrész átment a teszten, és ne kelljen a fejlesztőnek kézzel ellenőrizni azt (Self-validating) és a lehető legtöbb ágat fedjék le, amin különböző adatok alapján a kód végégimeny (Through).

Egy teszteset általában három részből áll. Első a környezet előkészítése (gyakran Given vagy Arrange néven hivatkozunk rá). Egy adatbázis tesztelése során ez lehet például valamilyen adat elmentése a teszt adatbázisba. Második a tesztelni kívánt metódus meghívása (When vagy Act). Például az imént elmentett adat lekérdezése. Harmadik az eredmény összevetése az elvárttal (Then vagy Assert). Itt ellenőrizzük, hogy a metódus tényleg az általunk elvárt adattal tért vissza.

## Persistence modul tesztelése

A @DataJpaTest annotáció segítségével írhatunk teszteket különböző JPA repository metódusainkhoz. Ez először is felkeresi a különböző entitásokat és beállítja a különböző repository-kat. Tesztek futásakor a @Service, @Controller, és egyéb komponensek nem lesznek betöltve az ApplicationContex-be. A valódi adatbázisunk helyett alapértelmezésben egy memóriában tárolt adatbázist fog használni. Erre az H2 adatbázist használtam fel. Minden teszteset egy tranzakcióból áll, melyeket a teszt után visszavonunk, így azok egymástól függetlenül manipulálhatják az adatbázist.

\*kép testFindByUserName\*

## Service modul tesztelése

A különböző service metódusok sokszor több más osztály metódusait is felhasználják saját feladatuk elvégzésére. Például a HotelService osztály szállodák létrehozására szolgáló createHotel metódusa felhasználja HotelRepositoryService több metódusát a név ellenőrzésére, majd az adatbázisba elmentésre, viszont unit teszteléskor csak arra vagyunk kíváncsiak, hogy a createHotel metódus megfelelően végzi-e a dolgát. Vagyis meghívja a kellő metódusokat a kellő adatokkal, és elvégzi a többi szükséges műveletet.

A mockolás segít áthidalni ezt a problémát. A tesztelni kívánt osztályunk függőségeit a Mockito által nyújtott @Mock annotációval mock objektumokra cserélhetjük. Ezek rendelkezni fognak annak a metódusaival, de nem adnak hozzá megvalósítást. Helyette megadhatjuk, hogy adott tesztesetekben hogyan viselkedjenek. Bizonyos paraméterekre milyen adatot adjanak vissza, esetleg dobjanak hibát, vagy pusztán csak azt is ellenőrizhetjük, hogy meg lettek-e hívva a tesztelt metódus által.

\*kép test\_create\_hotel\*

## Presentation modul tesztelése

Mivel a kontroller metódusaink különböző külső kérések kezelésére szolgálnak, így azoknak a tesztekből való közvetlen meghívásával nem sok mindent tudnánk vizsgálni. A @DataJpaTest-hez hasonlóan itt a @WebMvcTest annotációt használva olyan tesztkörnyezetet állíthatunk elő, ami csak a Spring MVC-hez szükséges bean-eket használja fel. Ahelyett, hogy egy valódi szervert futtatnánk a kérések küldésére, a MockMvc segítségével mockolt kérések és válaszokkal kommunikálhatunk a különböző metódusainkkal. Egyszerű metódushívással ellentétben így megadhatunk modell és session attribútumokat, a kérést indító felhasználó azonosságát és még sok más paramétert, ami egy valódi kérés része lehet. A kérés elküldése után ellenőrizhetjük, hogy a válasz megfelel-e minden elvárásnak. A service metódusokhoz hasonlóan a kontrollerek is használnak más osztályokból származó metódusokat. Ezeket a @MockBean annotációval tudjuk mockolni. A SecurityConfiguration osztályban megadott szabályok nem lesznek alkalmazva, így alapértelmezésben minden kérés le van tiltva minden felhasználó számára. Ezt az @Import(SecurityConfiguration.class) annotációval hozzáadtam a teszt osztályokhoz, így azt is tudom tesztelni, hogy mely felhasználók milyen kéréseket tudnak küldeni.

\*kép test\_create\_new\_room\*

## Jacoco

A JaCoCo egy széles körben elterjedt Java programkönyvtár tesztlefedettség kezelésére. Segítségével meghatározhatjuk, hogy a kódnak legalább mekkora része legyen letesztelve. Ha ez nem teljesül, akkor megszakítja a build folyamatot. Részletes jelentést ad a kód lefedettségéről, melyben láthatjuk, hogy egy csomag hány százalékát fedjük le, mely osztály mely sorait futtatta le vagy akár azt is, hogy egy elágazás hány ágát hagytuk ki.

\*kép jacoco report\*

# Konténerizáció

## Dockerfile

Dockerfile bemutatása, többlépcsős build, fájlok másolása

## Docker compose

Docker compose fájl bemutatása, .env fájl kezelése, adatbázis elindítása és csatlakozás, volume-ok

# Összefoglalás

# Köszönetnyilvánítás

# Irodalomjegyzék