

SZAKDOLGOZAT

Aranyi Ádám

Debrecen

2024

Debreceni Egyetem

Informatikai Kar

Hotelszoba foglaló webalkalmazás Spring Boot keretrendszerben

Témavezető:

Dr. Kovács László

Adjunktus

Készítette:

Aranyi Ádám

Programtervező Informatikus

Debrecen

2024

Tartalom

1.	Bevezetés.....	3
2.	Felhasznált eszközök és technológiák	5
2.1.	Apache Maven	5
2.2.	Verziókezelés és Git.....	6
2.3.	MySQL.....	7
2.4.	IntelliJ IDEA	8
2.5.	Spring Core.....	9
2.5.1.	Inversion of Control.....	9
2.5.2.	ApplicationContext.....	9
2.5.3.	Függőség Befecskendezés	10
2.6.	Spring Data JPA	10
2.6.1.	ORM	10
2.6.2.	JPA	11
2.6.3.	Hibernate.....	11
2.6.4.	Entitások.....	11
2.6.5.	Relációk	12
2.6.6.	@OneToOne.....	12
2.6.7.	@ManyToOne és @OneToMany	12
2.6.8.	@ManyToMany	12
2.6.9.	Spring Data	13
2.7.	Spring Web MVC.....	14
2.7.1.	Modell-Nézet-Vezérlő.....	14
2.7.2.	DispatcherServlet	14
2.8.	Spring Security	16
2.9.	Spring Boot	16
2.10.	Thymeleaf	17
2.11.	Bootstrap	17
2.12.	Docker	18
2.13.	Lombok	20
3.	A felület bemutatása	21
3.1.	Felhasználói fiók kezelése	21
3.2.	Szobák keresése	23
3.3.	Szobák lefoglalása	25

3.4.	Foglalások kezelése	26
3.5.	Szállodák értékelése.....	26
3.6.	Az adminisztrátor felület.....	27
3.7.	Navigációs menük	29
4.	Az alkalmazás felépítése	30
4.1.	Monolitikus architektúra.....	30
4.2.	Többrétegű architektúra	30
4.3.	Adatelérés réteg.....	31
4.3.1.	Az adatbázis táblái.....	32
4.3.2.	Repository interfészek.....	35
4.4.	Üzleti logika réteg	38
4.4.1.	Modell entitások és DTO	38
4.4.2.	ModelMapper és Transformer osztályok	39
4.4.3.	RepositoryService osztályok.....	40
4.4.4.	Service osztályok	41
4.4.5.	Képek kezelése	42
4.4.6.	E-mail.....	42
4.5.	Felhasználói felület réteg	42
4.5.1.	@Controller osztályok.....	43
4.5.2.	Spring SecurityConfiguration.....	44
4.5.3.	i18n.....	44
5.	Unit tesztelés.....	46
5.1.	Persistence modul tesztelése.....	46
5.2.	Service modul tesztelése	47
5.3.	Presentation modul tesztelése	48
5.4.	Jacoco.....	49
6.	Konténerizáció.....	50
6.1.	Dockerfile	50
6.2.	Docker Compose	50
7.	Összefoglalás	52
8.	Köszönetnyilvánítás	53
9.	Irodalomjegyzék	54
10.	Függelék	56

1. Bevezetés

Egyetemi tanulmányaim alatt a Java programozási nyelv volt az, amit a legjobban sikerült megismernem. Emiatt tudtam, hogyha egy komolyabb alkalmazás fejlesztésébe szeretnék belekezdeni, akkor ezt érdemes választanom annak alapjául. Ezt a döntésemet csupán megerősítette az, hogy szakmai gyakorlatomon mélyebb betekintést nyertem a Spring keretrendszer világába, melynek fő célja, hogy megkönnyítse a nagyvállalati Java alkalmazások fejlesztését.

A célom nem csupán egy működő alkalmazás elkészítése volt. Eddigi munkatapasztalataimat felhasználva igyekeztem utánozni egy nagyvállalati környezetben történő alkalmazás fejlesztésének folyamatát. Ennek érdekében törekedtem tiszta kód (clean code) írására, hogy azt más fejlesztők is könnyen megérthessék. Alkalmazásom részeit felelősségi körönként külön részekre bontottam, így egy bizonyos mértékben egymástól függetlenül végezhetünk rajtuk fejlesztést, vagy cserélhetjük le őket. A kódbázisom megfelelő működését unit tesztek segítségével ellenőriztem. A különböző változtatásokat a Git verziókezelő segítségével külön ágakon vezettem be. Ezen változtatásokhoz részletes leírásokat adtam majd végül befésültem a fő ágba.

A konténerizáció egy olyan téma, amit már hosszú ideje tudatosan elkerültem, pusztán azért, mert túl idegen volt számomra. Viszont a konténerek használata napról napra egyre elterjedtebbé válik, így előbb vagy utóbb nekem is meg kell tanulnom bánni velük. Ez a projekt egy kezdőlökést ad számomra, hogy megismerkedjek a konténerek kezelésének alapvető módjaival. Célul tűztem ki ugyanis, hogy a fejlesztés végén alkalmazásom egy Docker konténerben futtatható legyen.

Alkalmazásom témájául egy szálláshelyek keresésére és lefoglalására alkalmas webalkalmazás fejlesztését választottam. A cél az, hogy a felhasználók hatékonyan, különböző keresési feltételek alapján tudjanak számos szálloda szállásai között keresgélni. A megfelelő szobákat könnyedén le tudják foglalni, meglévő foglalásaikat szükség esetén törölhetik. Egy e-mailben kapott kód alapján a szállodába zökkenőmentesen be és ki tudnak majd jelentkezni. Továbbá még felhasználói fiókjaik jelszavát szükség esetén módosíthatják. A rendszer

működéséhez az is kell, hogy adminisztrátorok létre tudjanak hozni szállodákat, szobákat, valamint az érkező vendégeket kezelni tudják.

Tervezés közben oda kellett figyelnem arra, hogy az alkalmazást egyszerre több felhasználó is igénybe veszi. Így több különböző kérés kezelésénél is oda kell figyelni arra, hogy más felhasználó nem változtatott-e az adott erőforráson. Ilyen például a szobák keresése. Tegyük fel, hogy egy felhasználónak megtetszik egy szoba, amit le is akar foglalni, de mielőtt véglegesíthetné azt, valaki más már lefoglalta ugyan arra az időpontra. Ezesetben nem hagyhatjuk, hogy két foglalás jöjjön létre egyszerre.

A felhasználói felület előállításához a Thymeleaf szerveroldali sablon motort használtam, mellyel könnyedén illeszthetünk különböző logikát vagy attribútumokat weblapunk kódjába, amiből a felhasználók már csak a kész weboldalt fogják látni. A felület formázására Bootstrap-et és egyszerű CSS utasításokat használtam.

Az adatok tárolására egy MySQL adatbázist használtam, melyet eleinte feltelepítettem saját gépemre és saját kezűleg állítottam be. A fejlesztés végére már a Docker Compose segítségével automatizáltam az adatbázis előkészítését és egy konténerben futtattam

2. Felhasznált eszközök és technológiák

2.1. 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 fájl(ok)ba, 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.

2.2. 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.

Az 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 alapszükségüket sikerült elérnem. Többször előfordult, hogy egy funkció fejlesztése közben vettem észre hibákat a kód különböző részeiben. Ezeknek a kijavítása nem az aktuális funkció fejlesztésének része, így nem is ajánlott ezen az ágon hozzányúlni. Helyette a `dev` ágból létrehoztam egy újabb elágazást, mely csakis ennek a hibának kijavítására szolgál. A kijavított változatot előbb visszafésültem a `dev` ágba, majd ezeket a változtatásokat befésültem az aktív `feature branch`-be.

Az 1. ábrán látható a projektemnek több ága, köztük olyanok is, ahol párhuzamosan végeztem változtatásokat különböző ágakon.

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

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

2.3. 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 betartatja 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.

2.4. 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 megtippegni, 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 kódukot lefordítani, tesztelni és becsomagolni.

Ha a meglévő szolgáltatásoknál többre lenne szükségünk, akkor lehetőséget kínál különböző bővítmények telepítésére.

2.5. 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.

2.5.1. 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 kóduk lazán csatolt lesz, ezáltal sokkal könnyebb karban tartani, függőségeit cserélgetni és a részeit önállóan tesztelni.

2.5.2. 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.

2.5.3. Függőség Befecskendezés

A Függőség Befecskendezés egy tervezési minta, ami megvalósítja az Inversion of Control elvet. Ezt a következőképpen éri el: Ahelyett, hogy minden objektum létrehozna 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 projektben 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 lévő fogja választani. A kódban ezeket az értékeket konstruktor paraméterként adom meg, így a teszt környezetemben könnyen tudom más értékekre cserélni.

2.6. Spring Data JPA

2.6.1. 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.

2.6.2. 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.

2.6.3. 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.

2.6.4. 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.

2.6.5. 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.

2.6.6. @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.

2.6.7. @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.

2.6.8. @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.

2.6.9. 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 példákön 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.

2.7. Spring Web MVC

2.7.1. Modell-Nézet-Vezérlő

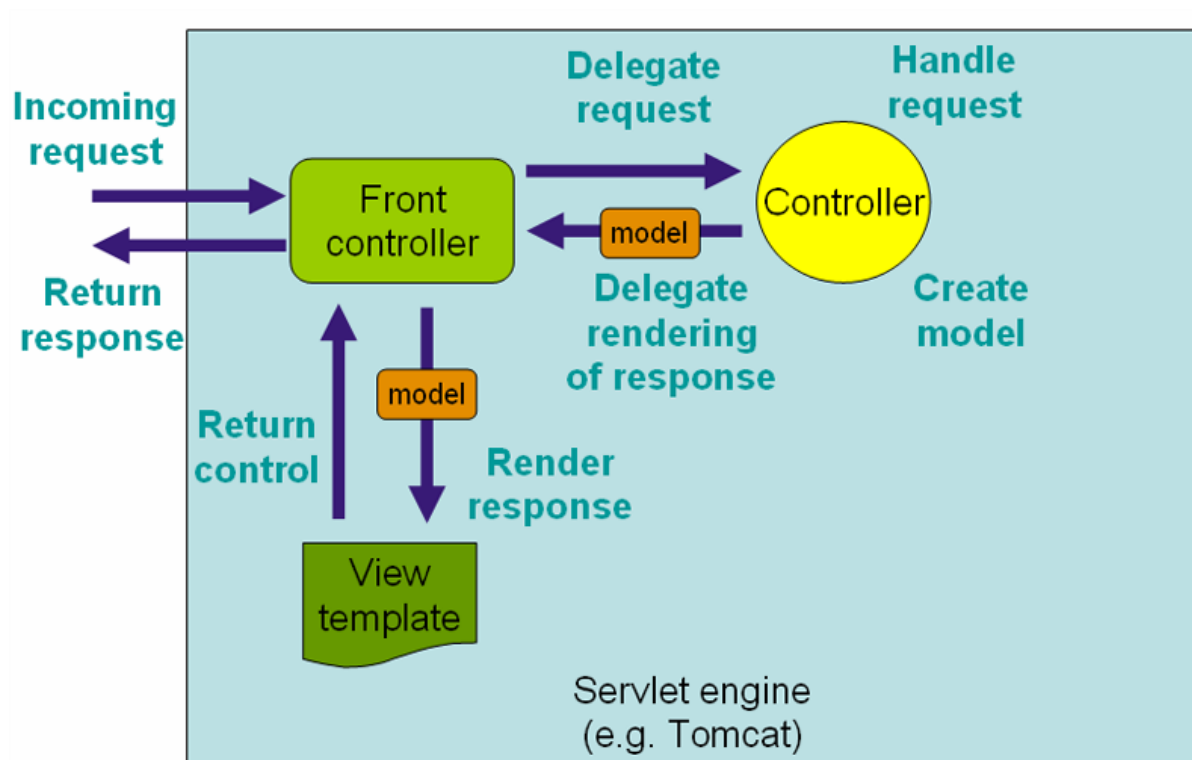
A Modell-Nézet-Vezérlő (Model-View-Controller vagy MVC) egy tervezési minta melyet a kezelői felületek készítése, adatok továbbítása és vezérlési logika megadására használnak. A következő három részből áll:

- Nézet: A felhasználói felület. 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.

2.7.2. 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. A 2. ábrán a kérések feldolgozásának folyamata látható



2. ábra – A Spring Web MVC DispatcherServlet kérés feldolgozási folyamata

(Forrás: <https://docs.spring.io/spring-framework/docs/2.5.x/reference/mvc.html>)

A beérkező kérések először a Front controller-be jutnak. Ez egy tervezési minta megvalósítása, mely kimondja, hogy minden kérést egyetlen központi mechanizmussal kezeljük. 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.

2.8. 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.

Úgy mond *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.

2.9. 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.

2.10.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 szűrhetünk be egy adott elembe, `th:if`-fel valamilyen feltételhez köthetjük egy elem megjelenítését, vagy `th:action`-nal meghívhatjuk egy `@Controller`-ben megadott metódusunkat.

2.11.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.

2.12.Docker

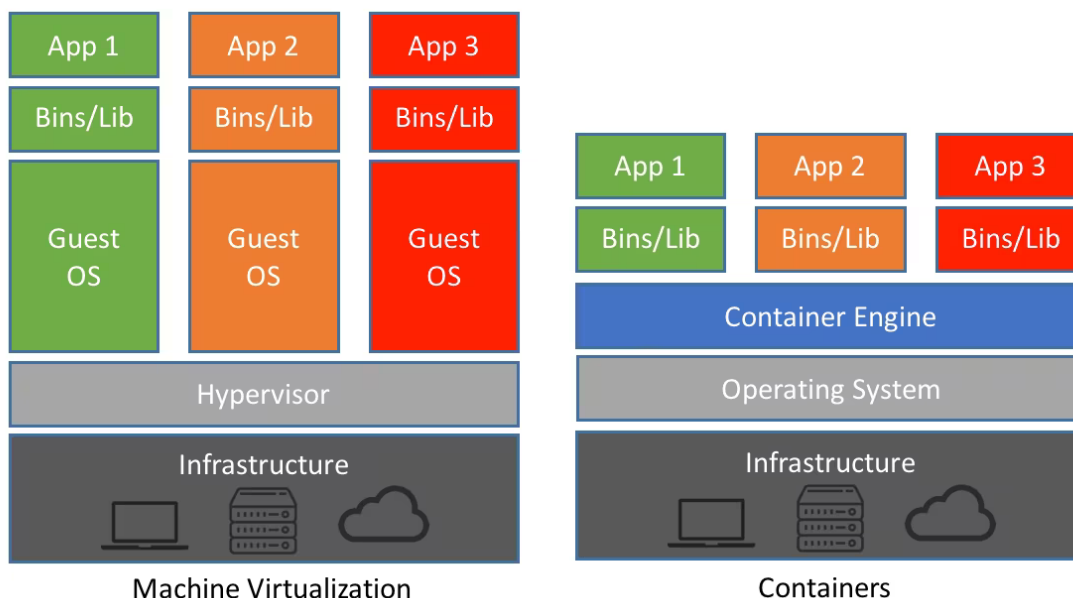
Valószínűleg voltunk már olyan helyzetben, hogy a megírt kódunk tökéletesen működött a saját gépünkön, de egy másik eszközön nem sikerült lefuttatni. Ennek oka leggyakrabban az, hogy eltér a két környezet, amiben az alkalmazás fut. Ilyen problémák lehetnek például eltérő Java verziók, hiányzó függőségek 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, melyeket egyetlen konténer motor kezel. Konténereknek megszabhatjuk, hogy gépünk mely erőforrásaihoz és milyen módon férhetnek hozzá. Egy konténer tartalmaz egy futtatandó alkalmazást, valamint a hozzá szükséges

függőségeket, fájlokat, beállításokat. A virtuális gépek és konténerek felépítésének különbségét a 3. ábra szemlélteti.



3. ábra – Virtuális gépek összevetése a konténerekkel

(Forrás: <https://www.netapp.com/blog/containers-vs-vms/>)

A Docker image-ek utasításokat tartalmaznak egy konténer létrehozásához. Általában egy image felhasznál más image-eket is saját feladatuk ellátásához. Például alkalmazásomban felhasználtam az eclipse-temurin:17-jdk-alpine base image-et 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 image-et, ezután valamilyen felhasznált fájlt, vagy csak a Dockerfile egy sorát megváltoztatjuk, majd újra felépítjük, 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.

2.13.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 az Építő tervezési mintát. Ennek elérésére egy könnyen kezelhető felületet ad változó számú konstruktor paraméterrel rendelkező osztályok példányosítására. A 4. ábra szemlélteti, hogyan példányosíthatunk egy osztályt annak építőjével, ahol szabadon választhatunk, hogy mely tulajdonságait, és azokat milyen sorrendben adjuk meg.

```
17         private static final User USER = User.builder()  
18             .username("TEST_USER_NAME")  
19             .password("TEST_PASSWORD")  
20             .email("TEST_USER_EMAIL")  
21             .enabled(true)  
22             .locked(false)  
23             .build();
```

4. ábra – A Builder tervezési minta használata

(Forrás: saját szerkesztés)

3. A felület bemutatása

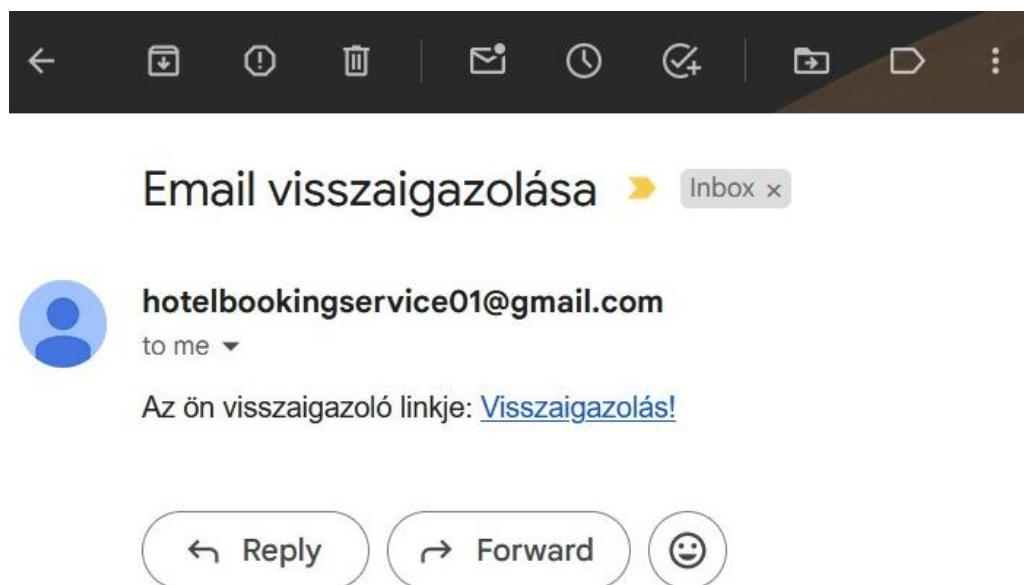
3.1. Felhasználói fiók kezelése

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.

Az így kapott link például így nézhet ki:

<http://localhost:8080/hotelbooking/register/confirm-email/confirm-token?confirmationToken=fafd61b2-7f74-438a-ad8c-adeecaa59969>

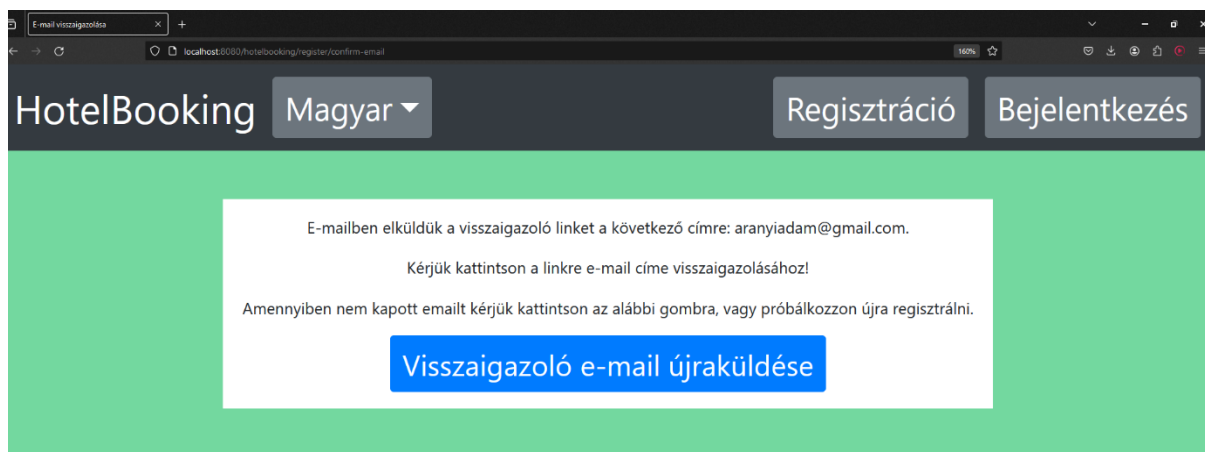
A regisztrációval a felhasználó kap egy az 5. ábrán láthatóhoz hasonló emailt, ami tartalmazza a fiókjához tartozó egyedi aktiváló linket. A link két részből áll. Az alap link, ami az aktiváló metódushoz vezet és az egyedi aktiválási azonosító, amit a metódus paraméterül kap.



5. ábra – Visszaigazoló e-mail

(Forrás: saját szerkesztés)

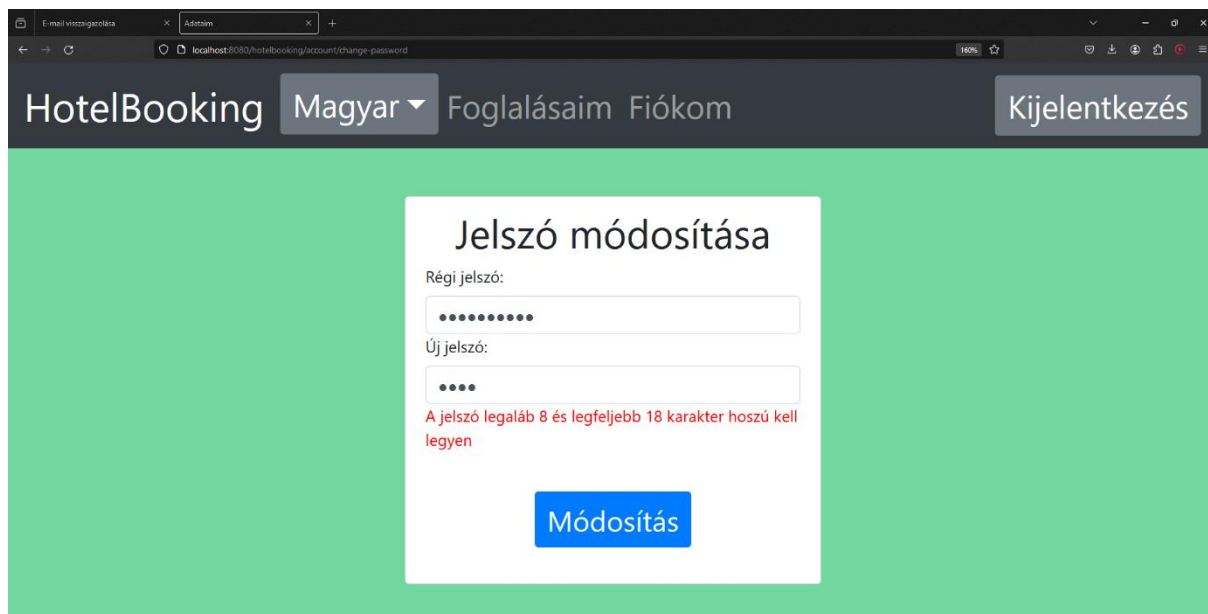
Előfordulhat, hogy valami hiba történik az e-mail küldése során, és a felhasználó nem kapja meg az aktiváló linket. Ebben az esetben kérheti új e-mail küldését a 6. ábrán látható 'Visszaigazoló e-mail újra küldése' gombbal.



6. ábra – Visszaigazoló e-mail újraküldése

(Forrás: saját szerkesztés)

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, különben a 7. ábrán látható hibaüzenetet fogja megkapni.

The image shows a web browser window displaying the 'HotelBooking' website. The page has a dark header with the site name, a language dropdown set to 'Magyar', and links for 'Foglalásaim' and 'Fiókom'. A 'Kijelentkezés' button is on the right. The main content area has a green background and features a white modal box titled 'Jelszó módosítása'. Inside the modal, there are two input fields: 'Régi jelszó:' and 'Új jelszó:'. The 'Új jelszó:' field has a red error message below it: 'A jelszó legalább 8 és legfeljebb 18 karakter hosszú kell legyen'. At the bottom of the modal is a blue button labeled 'Módosítás'.

7. ábra – Jelszó módosítása

(Forrás: saját szerkesztés)

3.2. Szobák keresése

Szobákat a kezdőlapra 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 köztük 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 visszkapunk.

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.

A 8. ábra szemlélteti egy keresés eredményét. Ha a kurzorunkat egy szállodára rávisszük, akkor megjelenik egy lista annak különböző szobáival melyek megfelelnek a keresési feltételeknek.

3.3. Szobák lefoglalása

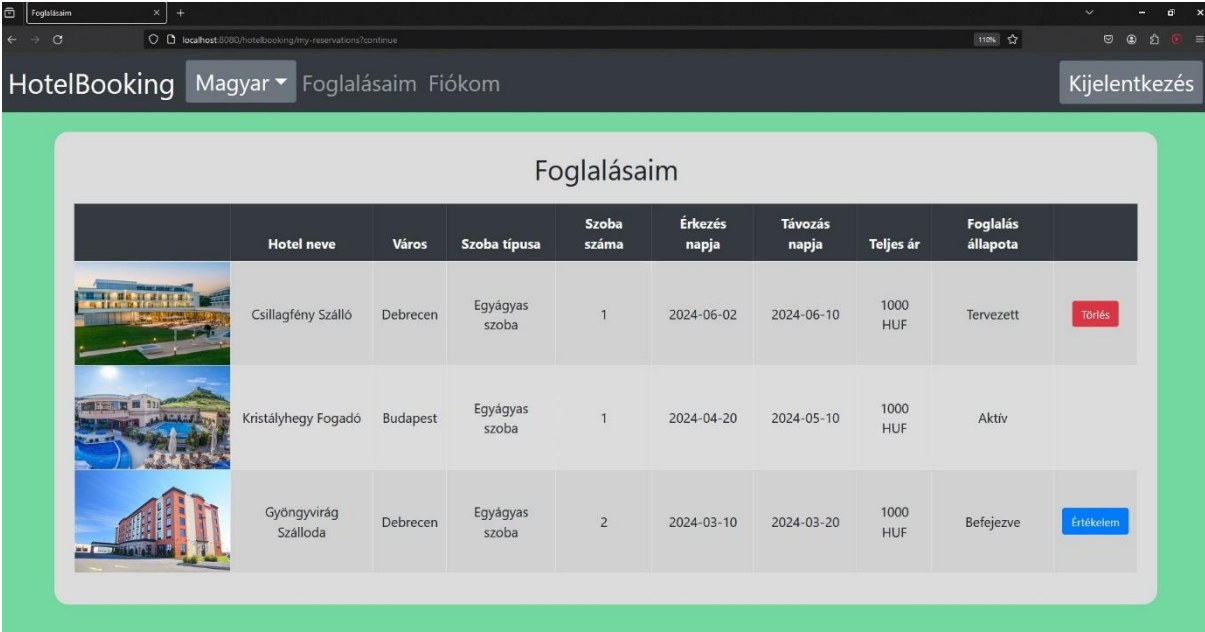
Azzal, hogy a felhasználó kiválasztja a neki tetsző szobát, átkerül a „Szobafoglalás” oldalra. Amennyiben eddig nem volt bejelentkezve, előbb a „Bejelentkezés” oldalra kerül, majd sikeres belépés után lesz ide továbbítva. Itt egy helyen láthatja a tervezett foglalás adatait, mielőtt még azt véglegesítené.




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.

3.4. 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. A 9. ábrán látható, hogyan jelennek meg a felhasználó különböző állapotú foglalásai.



	Hotel neve	Város	Szoba típusa	Szoba száma	Érkezés napja	Távozás napja	Teljes ár	Foglalás állapota	
	Csillagfény Szálló	Debreceen	Egyágyas szoba	1	2024-06-02	2024-06-10	1000 HUF	Tervezett	Törlés
	Kristályhegy Fogadó	Budapest	Egyágyas szoba	1	2024-04-20	2024-05-10	1000 HUF	Aktív	
	Gyöngyvirág Szálloda	Debreceen	Egyágyas szoba	2	2024-03-10	2024-03-20	1000 HUF	Befejezve	Értékelem

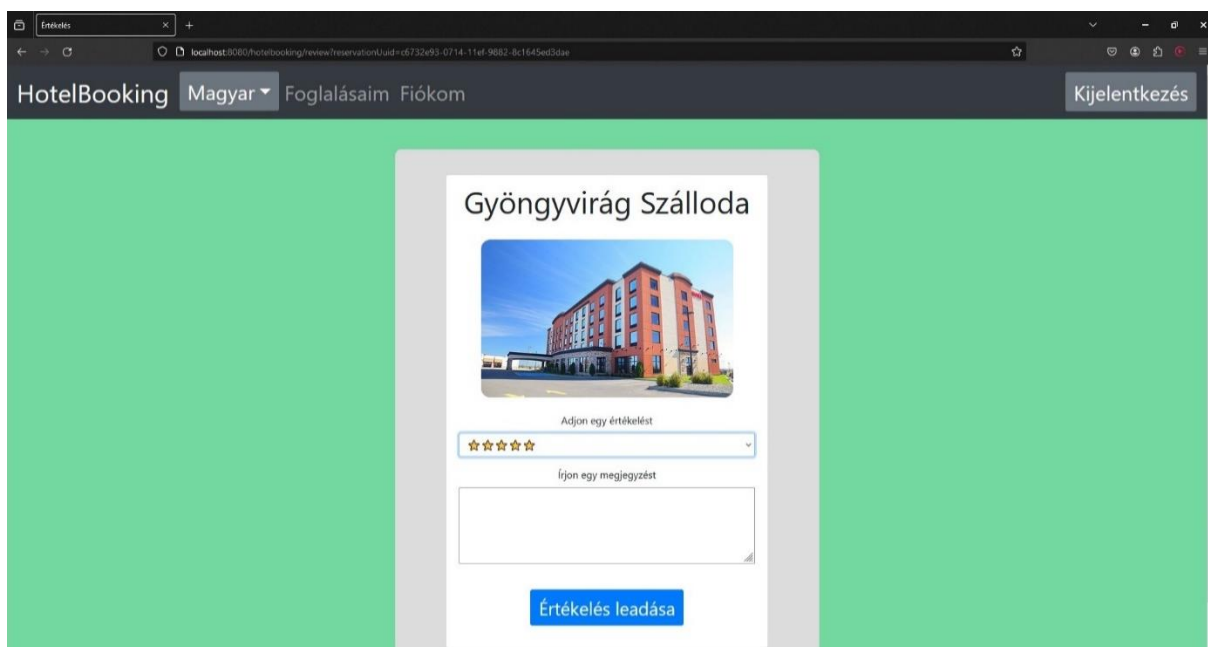
9. ábra – Foglalásaim kezelése

(Forrás: saját szerkesztés)

3.5. 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 a 10. ábrán látható „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.



10. ábra – Szállodák értékelése

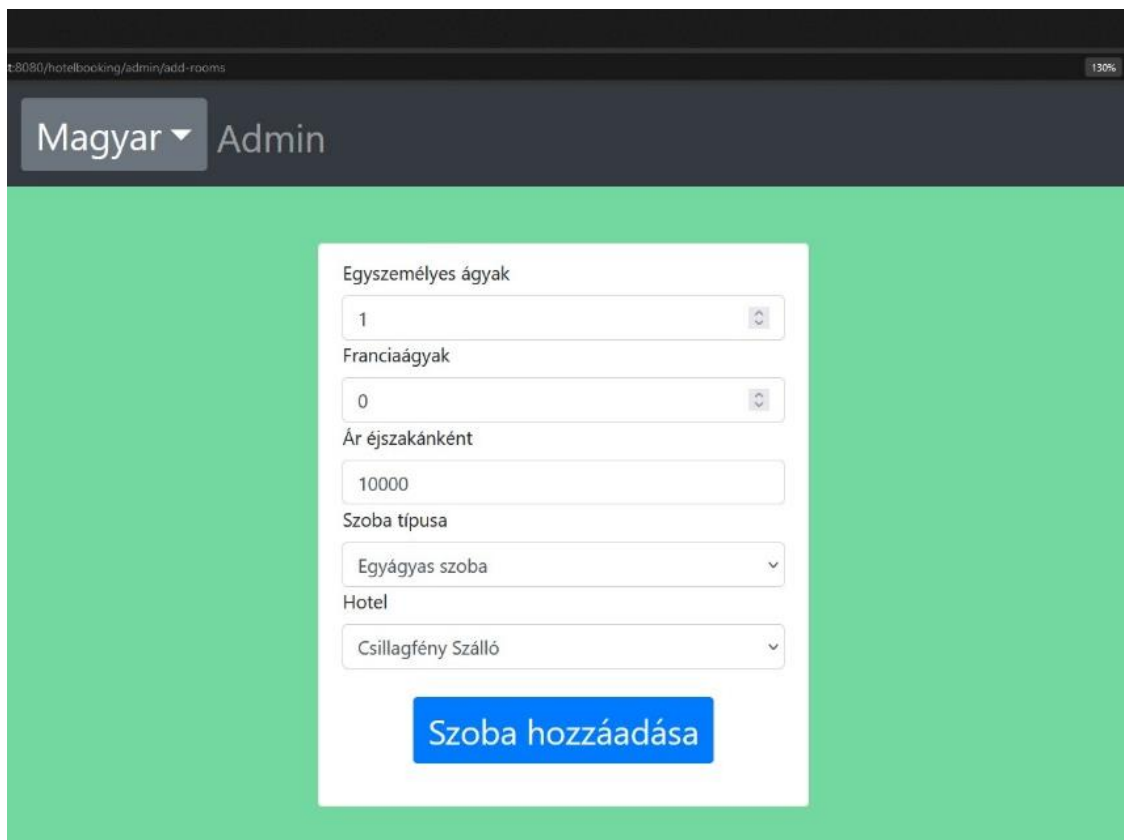
(Forrás: saját szerkesztés)

3.6. 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,

ennyibe kerül egy éjszakára lefoglalni, milyen típusú a szoba, és melyik konkrét szállodához tartozik. A 11. ábrán látható egy szoba létrehozása.

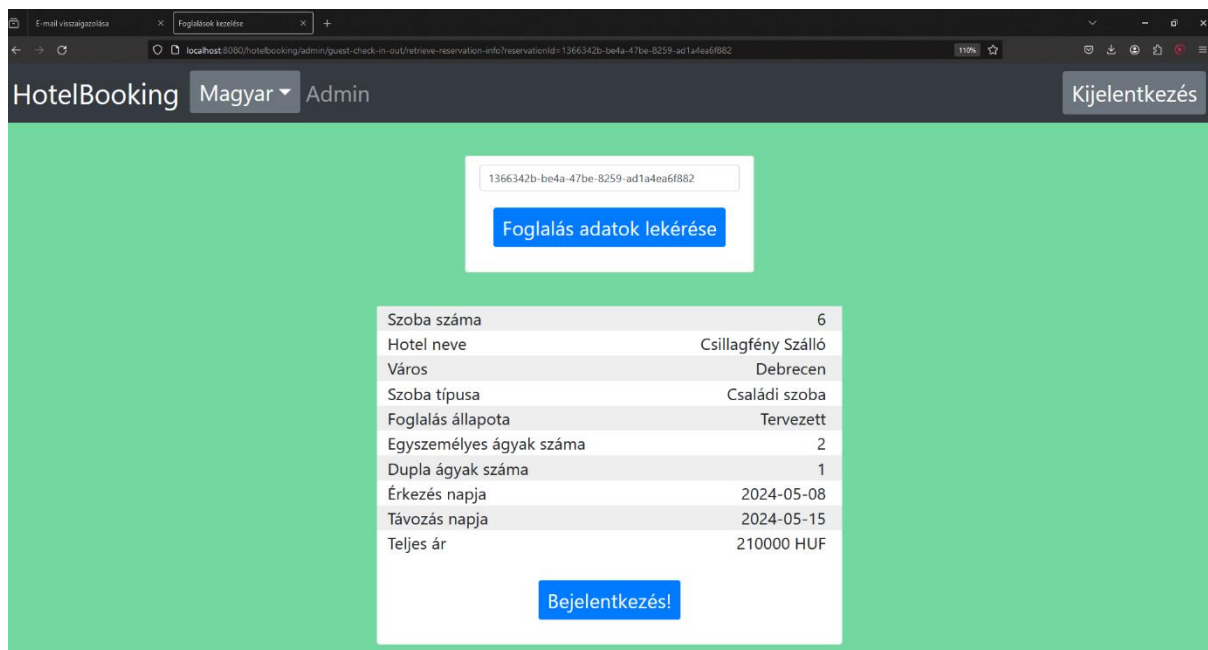


The screenshot shows a web browser window with the URL `localhost:8080/hotelbooking/admin/add-rooms` and a 130% zoom level. The page has a dark header with a language dropdown set to 'Magyar' and the word 'Admin'. The main content area has a green background. In the center is a white form titled 'Egyszemélyes ágyak'. The form contains the following fields: a numeric input for 'Egyszemélyes ágyak' with the value '1'; a numeric input for 'Franciaágyak' with the value '0'; a numeric input for 'Ár éjszakánként' with the value '10000'; a dropdown for 'Szoba típusa' with 'Egyágyas szoba' selected; and a dropdown for 'Hotel' with 'Csillagfény Szálló' selected. At the bottom of the form is a blue button labeled 'Szoba hozzáadása'.

11. ábra – Szobák hozzáadása

(Forrás: saját szerkesztés)

Korábban említettük, hogy egy szoba lefoglalásakor a felhasználók egy egyedi azonosítót kapnak e-mailben. Egy vendég bejelentkezésekor ezt az adminisztrátorok a 12. ábrán látható „Foglalások kezelése” oldalon tudják kezelni. Az azonosító alapján lekérdezik a foglalás adatait és bejelentkeztetik. Amikor a vendég távozna, akkor ugyanúgy az azonosítót felmutatva kijelentkeztetik.



12. ábra – Szobák hozzáadása

(Forrás: saját szerkesztés)

3.7. 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.

4. Az alkalmazás felépítése

4.1. Monolitikus architektúra

Nagyvállalati alkalmazások fejteése 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 kódbasis 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 többretegű architektúra is.

4.2. Többretegű 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 lévőből. 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.

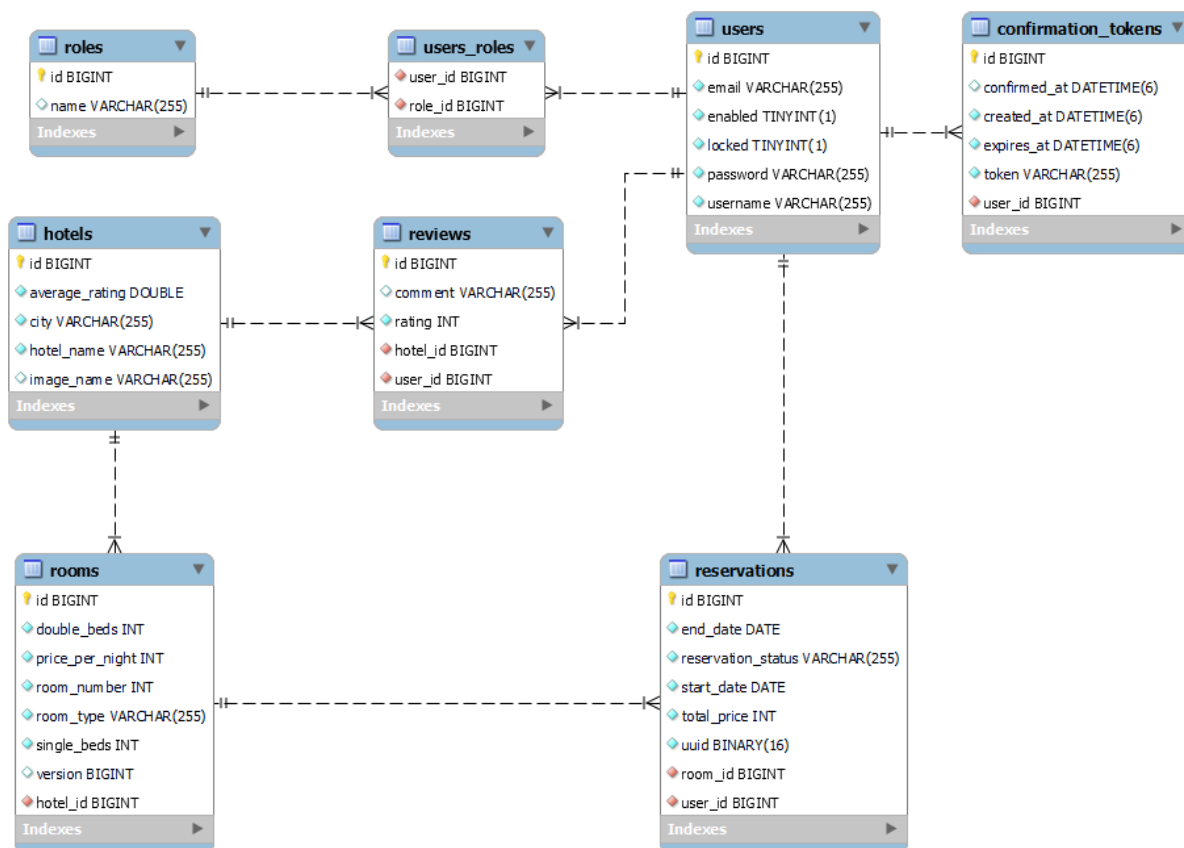
A rétegekre bontás sokat segít, de nem old meg minden problémát. Előfordulhat, hogy a felhasználói felületen szeretnénk, ha a szállodák konkrét címe is megjelenne. Ez egy olyan változtatás, amit nem lehet csak egy rétegen végrehajtani. Ezt a mezőt fel kell venni az adatbázisba, szükség van üzleti logikára, 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 a 13. ábra szemlélteti. Ezen jól látható, hogy az egyes rétegek az alattuk lévőeknek mindig csak kéréseket küldenek, míg a fölöttük lévőnek csakis ezen kérésekre adott választ küldik vissza. Továbbá az is látható, hogy a rétegek csakis a közvetlen szomszédjaikkal kommunikálnak.

4.3. 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-persistence` modul alkotja ezt a réteget,

mely a MySQL adatbázissal való kommunikációért felel. Adatbázisom végső sémája a 14. ábrán látható, melyet a MySQL Workbench segítségével állítottam elő.



14. ábra – Az adatbázis sémája

(Forrás: saját szerkesztés)

4.3.1. Az adatbázis táblái

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. Ezt a táblát a `Users` entitás osztályban definiálom 15. ábrán látható módon.

```
41
42 @ManyToMany(fetch = FetchType.EAGER)
43 @JoinTable(
44     name = "users_roles",
45     joinColumns = @JoinColumn(
46         name = "user_id", referencedColumnName = "id", nullable = false),
47     inverseJoinColumns = @JoinColumn(
48         name = "role_id", referencedColumnName = "id", nullable = false))
49 private Collection<Role> roles;
50
```

15. ábra – Join table kapcsolási mód definiálása

(Forrás: saját szerkesztés)

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 kapcsolat ezen oldalát a 16. ábra szemlélteti. 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.

```
28     @OneToMany(fetch = FetchType.EAGER, mappedBy = "hotel")
29     private List<Room> rooms;
30
31     @OneToMany(fetch = FetchType.EAGER, mappedBy = "hotel")
32     private List<Review> reviews;
```

16. ábra – @ManyToOne kapcsolások inverz oldala

(Forrás: saját szerkesztés)

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 a 17. ábrán látható `uniqueConstraints` megszorítással értem el. Miután a Hibernate előállítja a táblát ebből az osztályból, abba sem JPA-n keresztül, sem külön SQL utasítással nem tudunk olyan sort beszúrni, ami megszegi ezt a megszorítást.

```

15  @Entity
16  @Table(name = "rooms",
17         uniqueConstraints =
18         @UniqueConstraint(columnNames =
19         {"room_number", "hotel_id"}))
20  public class Room {
21

```

17. ábra – Egyedi megszorítás a Room entitás számára

(Forrás: saját szerkesztés)

A `rooms` tábla `version` mezőjét korábban az ütköző foglalások vizsgálatának optimalizálására használtam. Egy kiválasztott szoba foglalásainak újbóli vizsgálata előtt először csak a verziót vettem 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 foglalásait 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.

4.3.2. 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 `JpaRepository` 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 műveletekre 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 a 18. ábrán látható `UserRepository` interfészemben hoztam létre.

```
9      @Repository
10     public interface UserRepository extends JpaRepository<User, Long> {
11
12         7 usages   Admadma
13         Optional<User> findByUsername(String username);
14         7 usages   Admadma
15         Optional<User> findByEmail(String email);
16         7 usages   Admadma
17         boolean existsByUsername(String username);
18         7 usages   Admadma
19         boolean existsByEmail(String email);
20     }
```

18. ábra – UserRepository

(Forrás: saját szerkesztés)

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 cserélni, 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 19. ábrán látható `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.

```

6 usages  👤 Admadma +1
13  @Repository
14  public interface RoomRepository extends JpaRepository<Room, Long> {
15
16      10 usages  👤 Admadma
17      Optional<Room> findRoomByRoomNumberAndHotelHotelName(@Param("roomNumber") int roomNumber,
18                                                              @Param("hotelName") String hotelName);
19
20      10 usages  👤 Admadma
21      @Query("SELECT r.id FROM Room r JOIN r.hotel h WHERE" +
22              "(:singleBeds IS NULL OR r.singleBeds = :singleBeds)" +
23              "AND (:doubleBeds IS NULL OR r.doubleBeds = :doubleBeds)" +
24              "AND (:roomType IS NULL OR r.roomType = :roomType)" +
25              "AND (:hotelName IS NULL OR h.hotelName = :hotelName)" +
26              "AND (:city IS NULL OR h.city = :city)"
27      )
28      List<Long> findRoomsWithConditions(@Param("singleBeds") Integer singleBeds,
29                                       @Param("doubleBeds") Integer doubleBeds,
30                                       @Param("roomType") RoomType roomType,
31                                       @Param("hotelName") String hotelName,
32                                       @Param("city") String city);
33  }

```

19. ábra – RoomRepository

(Forrás: saját szerkesztés)

4.4. 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.

4.4.1. Modell entitások és DTO

A felhasználótól 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 a 20. ábrán látható `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.

4.4.2. `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. A 21. ábrán a `UserTransformer` metódusai láthatóak.

```

12  @Data
13  @Builder
14  @NoArgsConstructor
15  @AllArgsConstructor
16  public class RoomSearchFormServiceDTO {
17      private Integer singleBeds;
18      private Integer doubleBeds;
19      private RoomType roomType;
20      private String hotelName;
21      private String city;
22
23      @DateTimeFormat(pattern = "yyyy-MM-dd")
24      private LocalDate startDate;
25
26      @DateTimeFormat(pattern = "yyyy-MM-dd")
27      private LocalDate endDate;
28  }

```

20. ábra – RoomSearchFormServiceDTO

(Forrás: saját szerkesztés)

```

14  @Autowired
15  private ModelMapper modelMapper;
16
17  4 usages  Admadma
18  public User transformToUser(UserModel userModel) { return modelMapper.map(userModel, User.class); }
19
20  4 usages  Admadma
21  public UserModel transformToUserModel(User user) { return modelMapper.map(user, UserModel.class); }
22
23  12 usages Admadma
24  @
25  public Optional<UserModel> transformToOptionalUserModel(Optional<User> user){
26      if (user.isPresent()){
27          return Optional.of(modelMapper.map(user, UserModel.class));
28      } else {
29          return Optional.empty();
30      }
31  }

```

21. ábra – UserTransformer metódusai

(Forrás: saját szerkesztés)

4.4.3. 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 más osztályok az adatbázissal akarnak kommunikálni, azt ezeken keresztül tehetik 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.

4.4.4. 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 megvalósítja 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 szállások keresése, amit csupán a `searchHotelsWithReservableRooms` 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 ebbe az egy metódusba íránk bele, akkor más fejlesztőknek nehéz lenne megérteni annak működését. A megoldás az, hogy kiszervezzük a logika különböző részeit privát metódusokba vagy akár külön `service` osztályokba. Ekkor a metódusunknak

csupán meg kell hívnia ezeket a megfelelő paraméterekkel. A 22. ábrán látható kódrészlet a leegyszerűsített `searchHotelsWithReservableRooms` metódust ábrázolja.

```
89 public List<HotelWithReservableRoomsServiceDTO> searchHotelsWithReservableRooms(  
90     RoomSearchFormServiceDTO roomSearchFormServiceDTO) {  
91     List<Long> roomIds = roomRepositoryService.getRoomsWithConditions(roomSearchFormServiceDTO);  
92     List<Long> availableRoomsIds = filterAvailableRooms(roomSearchFormServiceDTO, roomIds);  
93  
94     return assignUniqueRoomsToHotels(availableRoomsIds, roomSearchFormServiceDTO);  
95 }
```

22. ábra – RoomSearchFormServiceDTO

(Forrás: saját szerkesztés)

4.4.5. 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 fájl 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.

4.4.6. E-mail

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

4.5. 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 adat érvényesítéséért, valamint itt található az alkalmazás elindításáért felelős main metódus, és több konfigurációért felelős osztály is.

4.5.1. @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. A 23. ábrán az AddRoomsController egy metódusa látható, mely a szobák létrehozására irányuló kéréseket kezeli.

```
43      @PostMapping("/create-new-room")
44      @ public String saveNewRoom(@Valid @ModelAttribute("roomCreationDTO") RoomCreationDTO roomCreationDTO,
45                                BindingResult result,
46                                Model model){
47          if (result.hasErrors()){
48              LOGGER.info("Error while validating");
49              return "addrooms";
50          }
51
52          try {
53              roomService.createRoomFromDTO(
54                  roomCreationDTOTransformer.transformToRoomCreationServiceDTO(roomCreationDTO));
55              model.addAttribute( attributeName: "successMessage", attributeValue: "Success");
56          } catch (Exception e){
57              result.reject( errorCode: "admin.room.validation.global.error");
58              return "addrooms";
59          }
60          return "addrooms";
61      }
```

23. ábra – AddRoomsController saveNewRoom metódusa

(Forrás: saját szerkesztés)

4.5.2. Spring SecurityConfiguration

Szeretnénk korlátozni, hogy a különböző felhasználók milyen kéréseket küldhetnek a rendszernek. Erre a Spring Security által nyújtott `SecurityFilterChain`-t használtam fel. A 24. ábrán látható ennek definiálása. 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.

```
15 @Bean
16 public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
17     http
18         .csrf() CsrfConfigurer<HttpSecurity>
19         .disable() HttpSecurity
20         .authorizeHttpRequests(auth -> {
21             auth.requestMatchers(_patterns: "/hotelbooking/admin/**").hasAnyAuthority(_authorities: "ADMIN");
22             auth.requestMatchers(_patterns: "/hotelbooking/reserve-room/**", "/hotelbooking/my-reservations/**",
23                 "/hotelbooking/review/**").hasAnyAuthority(_authorities: "USER");
24             auth.requestMatchers(_patterns: "/hotelbooking/default/**", "/hotelbooking/account/**").authenticated();
25             auth.requestMatchers(_patterns: "/hotelbooking/login/**", "/error/**", "/images/**",
26                 "/hotelbooking/home/**").permitAll();
27             auth.requestMatchers(_patterns: "/hotelbooking/register/**").anonymous();
28         })
29         .formLogin(form -> form
30             .loginPage("/hotelbooking/login")
31             .defaultSuccessUrl("/hotelbooking/default")
32             .permitAll())
33         .logout() LogoutConfigurer<HttpSecurity>
34             .logoutUrl("/logout");
35     return http.build();
36 }
```

24. ábra – A `SecurityFilterChain` Bean

(Forrás: saját szerkesztés)

A `SecurityConfiguration` osztályban szintén definiáltam egy `PasswordEncoder` Bean-t, mely által az alkalmazás a `BCryptPasswordEncoder`-t fogja használni jelszavak titkosítására.

4.5.3. 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 tesszük.

5. 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 hamarabb kiszűrhetjük. Egy másik előnye a tesztek írásának az, hogy arra készíteti 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 tesztől 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 végéigmeny (Through).

Egy teszt eset 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.

5.1. Persistence modul tesztelése

A `@DataJpaTest` annotáció segítségével írhatunk tesztek 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 `ApplicationContext`-be. A valódi adatbázisunk helyett alapértelmezésben egy memóriában tárolt adatbázist fog használni. Erre az H2

adatbázist használtam fel. Minden teszt eset egy tranzakcióból áll, melyeket a teszt után visszavonunk, így azok egymástól függetlenül manipulálhatják az adatbázist.

A 25. ábrán látható metódusban a `UserRepository` `findByUsername` metódusát tesztelem. A környezet előkészítése abból áll, hogy elmentek egy `User` entitást a teszt adatbázisba. Ezután meghívom a tesztelt metódust és az eredményét eltárolom. Végül ellenőrzöm, hogy az elvárt értékkel tért vissza.

```
59      @Test
60      public void testFindByUsernameReturnsOptionalOfUserWithProvidedUsername(){
61          User user = userRepository.save(User.builder()
62              .username(TEST_USER_NAME)
63              .password(TEST_PASSWORD)
64              .email(TEST_USER_EMAIL)
65              .roles(List.of(SAVED_ROLE))
66              .enabled(true)
67              .locked(false)
68              .build());
69
70          Optional<User> resultUser = userRepository.findByUsername(TEST_USER_NAME);
71
72          Assertions.assertThat(resultUser).isNotNull();
73          Assertions.assertThat(resultUser).isNotEmpty();
74          Assertions.assertThat(resultUser.get().getUsername()).isEqualTo(TEST_USER_NAME);
75          Assertions.assertThat(resultUser.get().getEmail()).isEqualTo(TEST_USER_EMAIL);
76      }
```

25. ábra – A `findByUsername` metódus egy teszt esete

(Forrás: saját szerkesztés)

5.2. 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.

A 26. ábrán a `HotelService` interfész egy implementációjának egy tesztesete látható, melynek mockoljuk a `HotelRepositoryService` függőségét. Ebben a tesztesetben azt vizsgáljuk, hogy a tesztelt metódus az elvártnak megfelelően meghívta a mockolt függősége metódusait.

```
42     @InjectMocks
43     private HotelServiceImpl hotelService;
44
45     18 usages
46     @Mock
47     private HotelRepositoryService hotelRepositoryService;
48
49     Admadma
50     @Test
51     public void testCreateHotelShouldCallSaveWhenHotelNameIsFree(){
52         when(hotelRepositoryService.findHotelByHotelName(HOTEL_NAME)).thenReturn(EMPTY_HOTEL_MODEL);
53         when(hotelRepositoryService.create(HOTEL_CREATION_SERVICE_DTO)).thenReturn(HOTEL_MODEL);
54
55         hotelService.createHotel(HOTEL_CREATION_SERVICE_DTO);
56
57         verify(hotelRepositoryService).findHotelByHotelName(HOTEL_NAME);
58         verify(hotelRepositoryService).create(HOTEL_CREATION_SERVICE_DTO);
59     }
```

26. ábra – A `createHotel` metódus egy tesztesete

(Forrás: saját szerkesztés)

5.3. Presentation modul tesztelése

Mivel a kontroller metódusaink különböző külső kérések kezelésére szolgálnak, így azoknak a tesztekben 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 szerveret futtatnánk a kérések küldésére, a `MockMvc` segítségével mockolt kérések és válaszokkal kommunikálhatunk a különböző metódusainkkal. Egyszerű metódushívással ellentétben így megadhatunk modellt 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. A 27. ábrán a szobák létrehozására irányuló kérések kezelésének egy tesztесе látható. Itt a küldeni kívánt kérés részeként megadom a szükséges paramétereket, majd vizsgálom, hogy a megfelelő választ adja vissza.

```
99      @Test
100      @WithMockUser(authorities = "ADMIN")
101      public void testCreateNewRoomShouldReturnToAddRoomsPageWithErrorIfBindingResultHasErrors() throws Exception {
102          mockMvc.perform(MockMvcRequestBuilders
103              .post(urlTemplate: "/hotelbooking/admin/add-rooms/create-new-room")
104              .flashAttr(name: "roomCreationDTO", ROOM_CREATION_DTO_WITH_FIVE_INVALID_FIELDS))
105              .andExpect(status().isOk())
106              .andExpect(model().attributeDoesNotExist(Names: "successMessage"))
107              .andExpect(view().name(expectedViewName: "addrooms"))
108              .andExpect(model().attribute(name: "roomCreationDTO", ROOM_CREATION_DTO_WITH_FIVE_INVALID_FIELDS))
109              .andExpect(model().attributeErrorCount(name: "roomCreationDTO", expectedCount: 5));
110      }
```

27. ábra – A create-new-room kérés tesztelése

(Forrás: saját szerkesztés)

5.4. 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. A 28. ábra és a 29. ábra szemlélteti a JaCoCo jelentéseit.

6. Konténerizáció

6.1. Dockerfile

Alkalmazásom konténerizációjához egy többlépcsős `Dockerfile`-t használtam (multi-stage build), melyet a 30. ábra szemléltet. Egy `Dockerfile`-on belül megadhatunk több `FROM` utasítást, amelyek által különböző fázisokra bontjuk azt. A különböző fázisokban megadhatjuk, hogy csak azokat a fájlokat másolják át az előttük lévőkből, amikre szükségük van. Ezáltal a kész image csak olyan fájlokat fog tartalmazni, ami elengedhetetlen a futtatásához. Továbbá, ha újraépítés előtt egy fázisban változás történt, akkor csak a módosult fázistól kezdve építjük újra az image-et. A `Dockerfile`-om két fázisból áll. Az első a futtatható `jar` fájl előállításáért felelős `build` fázis. Itt a forráskódból előállítom a futtatható `jar` fájlt a `maven:3.9.6-eclipse-temurin-17` base image-ből származó `maven` segítségével. A `runtime` fázisban átmásolom az alkalmazás futtatásához szükséges fájlokat. A `build` fázisból csupán a futtatható `jar` fájlt másolom át, így az elkészült image nem fogja tartalmazni a forráskódot. Végül az `ENTRYPOINT`-al megadom, hogy a konténer indításával induljon el az alkalmazás.

Létrehoztam egy SQL szkript-et, amellyel feltölthetem az adatbázist előre elkészített Szoba, Felhasználó, Foglalás és egyéb adatokkal. Ez csakis kizárólag a felület szemléltetésének megkönnyítésére szolgál azáltal, hogy már egyből van adat a rendszerben, amit kezelhetünk, nem kell egyesével hozzáadni mindent. Használatát környezeti változókon keresztül szabályozhatjuk. Alapértelmezésben ezeket az adatokat nem töltjük be az adatbázisba.

6.2. Docker Compose

Az alkalmazásunkat már konténerbe helyeztük, így feltételezhetjük, hogy minden gépen az elvártaknak megfelelően fog működni. Viszont azt is szeretnénk garantálni, hogy az elvárt adatbázist használja, ezért ezt is egy konténerben fogjuk futtatni. A 31. ábrán látható `docker-compose.yml` fájl segítségével könnyedén konfigurálhatjuk és futtathatjuk több konténerből álló alkalmazásunkat.

A különböző konténereket a `services` (szolgáltatások) részben adhatjuk meg. Itt a `docker-mysql` szolgáltatást úgy definiáltam, hogy a Docker Hub-ról tölts le a MySQL 8.3-as verziójú `base` `image`-et. Egy `.env` fájl segítségével kulcs-érték párként megadhatunk környezeti változókat, melyekre a `docker-compose.yml` fájlban hivatkozhatunk. Ha azt szeretnénk, hogy ezeket egy adott konténer használja, akkor továbbadhatjuk az `env_file` részlegben. Egy konténer leállításával a benne tárolt összes adat elvész. Az adatbázis és a feltöltött képek megtartása érdekében `volume`-okat használtam, míg a létrejött log fájlokat `bind mount` segítségével tárolom, vagyis az alkalmazást futtató gép fájlrendszerére másolom. A `volume`-oknak többek között előnye, hogy független a konténert futtató operációs rendszertől, és könnyen mozgathatóak más környezetekbe. A Docker Compose kezeli a hálózatot is, amin keresztül a konténerek kommunikálnak.

Az alkalmazás beállításában nem adtam meg konkrét `image` nevet. Helyette ez a `Dockerfile` alapján fog felépülni a konténerek indításakor. Ha a szükséges `image`-ek már létre vannak hozva akkor azok alapértelmezésben nem lesznek újraépítve. A Spring számára megadom az adatbázis elérhetőségét, melyben a konténerben futó adatbázisra hivatkozok. Az alkalmazás futtatásához szükség van az adatbázisra is, így a `depends_on` segítségével jelezzük, hogy erre szüksége van. Azonban ez a beállítás csak azt jelenti, hogy az adott szolgáltatás megvárja míg ez a másik elindul. Indítás után az adatbázisnak időbe telik mire elvégzi saját inicializálási logikáját. Ezalatt az alkalmazás már megpróbált hozzá kapcsolódni és le is áll, mert ez sikertelen volt. A Docker Compose nem nyújt olyan beállítást, amivel közvetlenül megszabhatjuk, hogy egy szolgáltatás várja meg míg egy másik teljesen elkészül. Erre megoldásként a `restart: unless-stopped` beállítás segítségével addig próbáljuk újra és újra elindítani az alkalmazást, míg sikeresen nem csatlakozik vagy nem állítjuk le manuálisan.

7. Összefoglalás

Az alkalmazás funkcióinak szempontjából sikerült elérnem a bevezetésben kitűzött célokat. Ezeken felül még bevezettem egy újabb funkciót is, amely által a felhasználók értékelhetik a szállásokat, ahol tartózkodtak.

A legtöbb időt egyértelműen a unit tesztek írásával töltöttem. Ahhoz, hogy a kódomat megfelelően tudjam tesztelni sok helyen átalakításokat kellett végeznem. Ennek az a pozitív mellékhatása volt, hogy nagy összetett osztályokat több kicsire választottam szét, ezáltal javítva a kód olvashatóságát. Tesztek írása miatt arra kényszerültem, hogy más szemszögből tekintsek a meglévő logikámra, így észrevettem korábban figyelmen kívül hagyott lehetőségeket a kód minőségének javítására.

A funkcionális követelményeken kívül megszabtam több technikai követelményt is. Úgy érzem ezeket összességében sikerült elég jól teljesítenem. Amin a jövőben még javítanom kell az például tesztekben a konstans értékek használata. Olvashatóság és kódismétlés csökkentésének céljából ajánlott egy osztályban megadott értékeket konstans értékekbe kiszervezni. Ez eleinte segített átláthatóbbá tenni a tesztjeimet, viszont olyan teszt osztályok esetén, amik már összetettebb metódusokat futtattak már inkább megnehezítették az eligazodást. Szintén nem vagyok megelégedve a DTO osztályaimmal. Ezeket a fejlesztés során mindig csak az aktuális probléma megoldására hoztam létre, nem terveztem előre, hogy mi lesz mindegyikük szerepe a végleges alkalmazásban. Feladatukat így is megfelelően ellátják, de amikor a jövőben ismét DTO osztályokat kell használnom, akkor már tudni fogom, hogy alaposabb tervezést igényelnek.

Amivel viszont túlteljesítettem saját elvárásaimon az egyértelműen a konténerizáció volt. Eleinte megelégedtem volna csupán azzal, hogy egy kész `jar` fájlt konténerbe tudok csomagolni és futtatni tudom. De ahogy ezen dolgoztam lépésről lépésre próbáltam egyre bővíteni a folyamatot. A végeredmény egy több konténerből álló konténerizált alkalmazás lett, melyet bármilyen rendszeren futtathatunk, amely rendelkezik Docker-el

8. Köszönetnyilvánítás

Szeretném megköszönni családomnak, hogy végig motiváltak engem és hogy támogattak dolgozatom írásának nehezebb időszakában is.

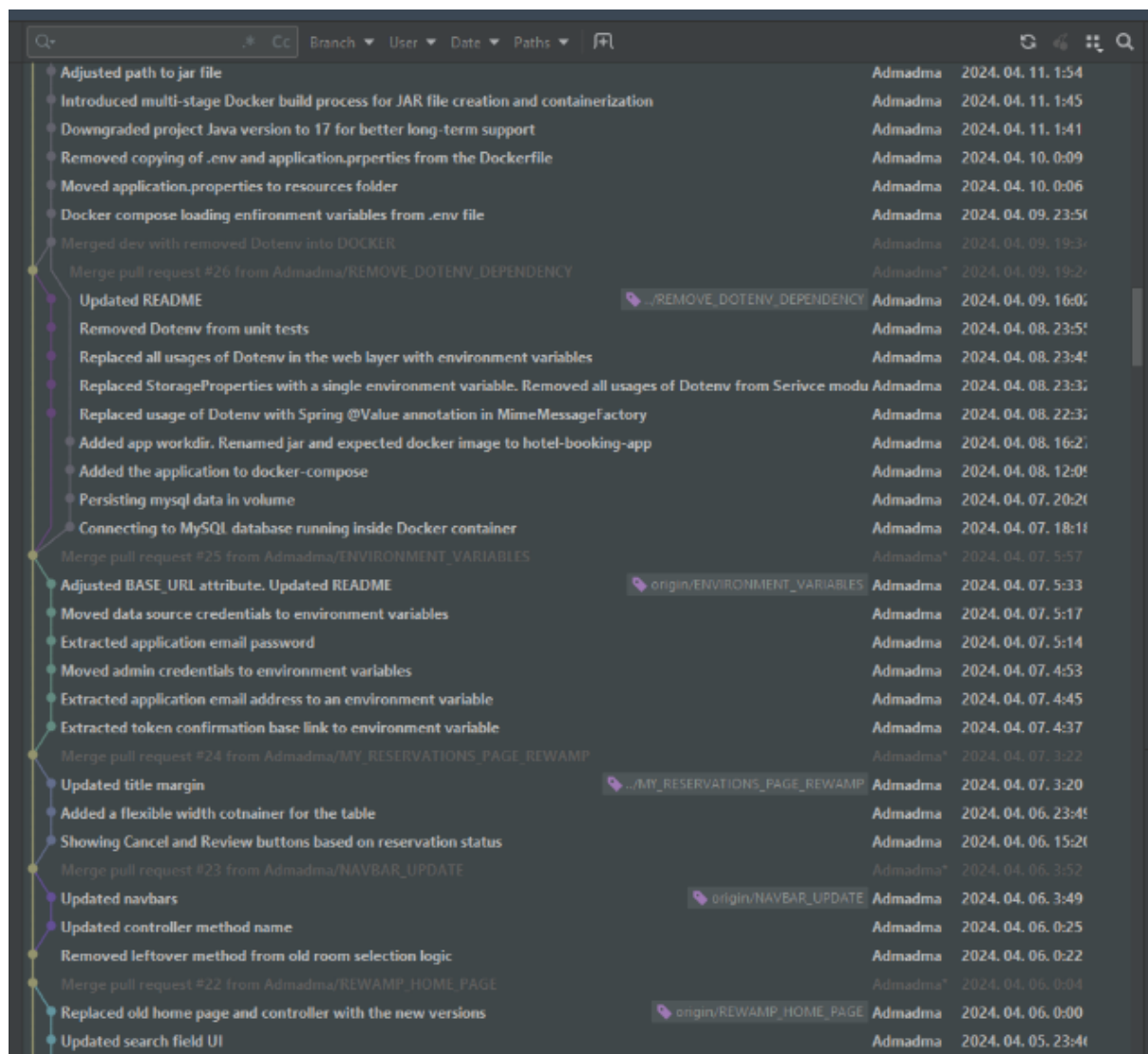
Továbbá szeretném megköszönni témavezetőmnek Dr Kovács Lászlónak, hogy támogatott dolgozatom témájának módosításában és hogy türelmes volt velem a vártnál sokkal tovább tartó munkám ellenére is.

9. Irodalomjegyzék

- Christian Bauer, Gavin King, Gary Gregory: Java Persistence with Hibernate, Second Edition
- Erich Gamma, John Vlissides, Richard Helm, Ralph Johnson: Design Patterns: Elements of Reusable Object-Oriented Software
- Martin Fowler: Patterns of Enterprise Application Architecture
- Maven dokumentáció: <https://maven.apache.org/what-is-maven.html> (elérés dátuma 2024. 04. 30.)
- IntelliJ IDEA szolgáltatásai: <https://www.jetbrains.com/idea/features/> (elérés dátuma 2024. 04. 30.)
- Parametrizált Fragment-ek: <https://www.thymeleaf.org/doc/articles/layouts.html> (elérés dátuma 2024. 04. 30.)
- Spring keretrendszer dokumentáció: <https://docs.spring.io/spring-framework/reference/> (elérés dátuma 2024. 04. 30.)
- Spring Data JPA dokumentáció: <https://docs.spring.io/spring-data/jpa/reference/jpa.html> (elérés dátuma 2024. 04. 30.)
- Spring Security konfigurálása WebSecurityConfigurerAdapter kiterjesztése nélkül: <https://spring.io/blog/2022/02/21/spring-security-without-the-websecurityconfigureradapter> (elérés dátuma 2024. 04. 30.)
- Spring Web MVC dokumentáció: <https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html> (elérés dátuma 2024. 04. 30.)
- ViewResolver interfész dokumentáció: <https://docs.spring.io/spring-framework/docs/3.0.0.RELEASE/reference/html/mvc.html#mvc-viewresolver-resolver> (elérés dátuma 2024. 04. 30.)
- Spring Boot dokumentáció: <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/> (elérés dátuma 2024. 04. 30.)
- Spring Boot tesztelés: <https://docs.spring.io/spring-boot/docs/1.4.2.RELEASE/reference/html/boot-features-testing.html> (elérés dátuma 2024. 04. 30.)

- Gmail SMTP szerver használata: <https://support.google.com/a/answer/176600?hl=en> (elérés dátuma 2024. 04. 30.)
- Docker áttekintés: <https://docs.docker.com/get-started/overview/> (elérés dátuma 2024. 04. 30.)
- Többlépcsős Docker build: <https://docs.docker.com/build/building/multi-stage/> (elérés dátuma 2024. 04. 30.)
- MySQL összefoglaló: <https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html> (elérés dátuma 2024. 04. 30.)
- ModelMapper: <https://modelmapper.org/> (elérés dátuma 2024. 04. 30.)

10. Függelék



1. ábra – Git ágak

(Forrás: saját szerkesztés)

HotelBooking

Magyar

Regisztráció

Bejelentkezés

Egyszemélyes ágyak száma

2

Franciaágyak száma

Szoba típusa

-- Szoba típusa --

Város

-- Város --

Hotel

-- Hotel --


Érkezés napja

05 / 05 / 2024

Távozás napja

05 / 12 / 2024

Szobák keresése



Csillagfény Szálló

Debrecen

Átlagos értékelés: 4.5

Családi szoba

Egyszemélyes ágyak száma: 2

Franciaágyak száma: 1


Teljes ár: 210000 HUF

lkerszoba

Egyszemélyes ágyak száma: 2

Franciaágyak száma: 2

Teljes ár: 280000 HUF



Kristályhegy Fogadó

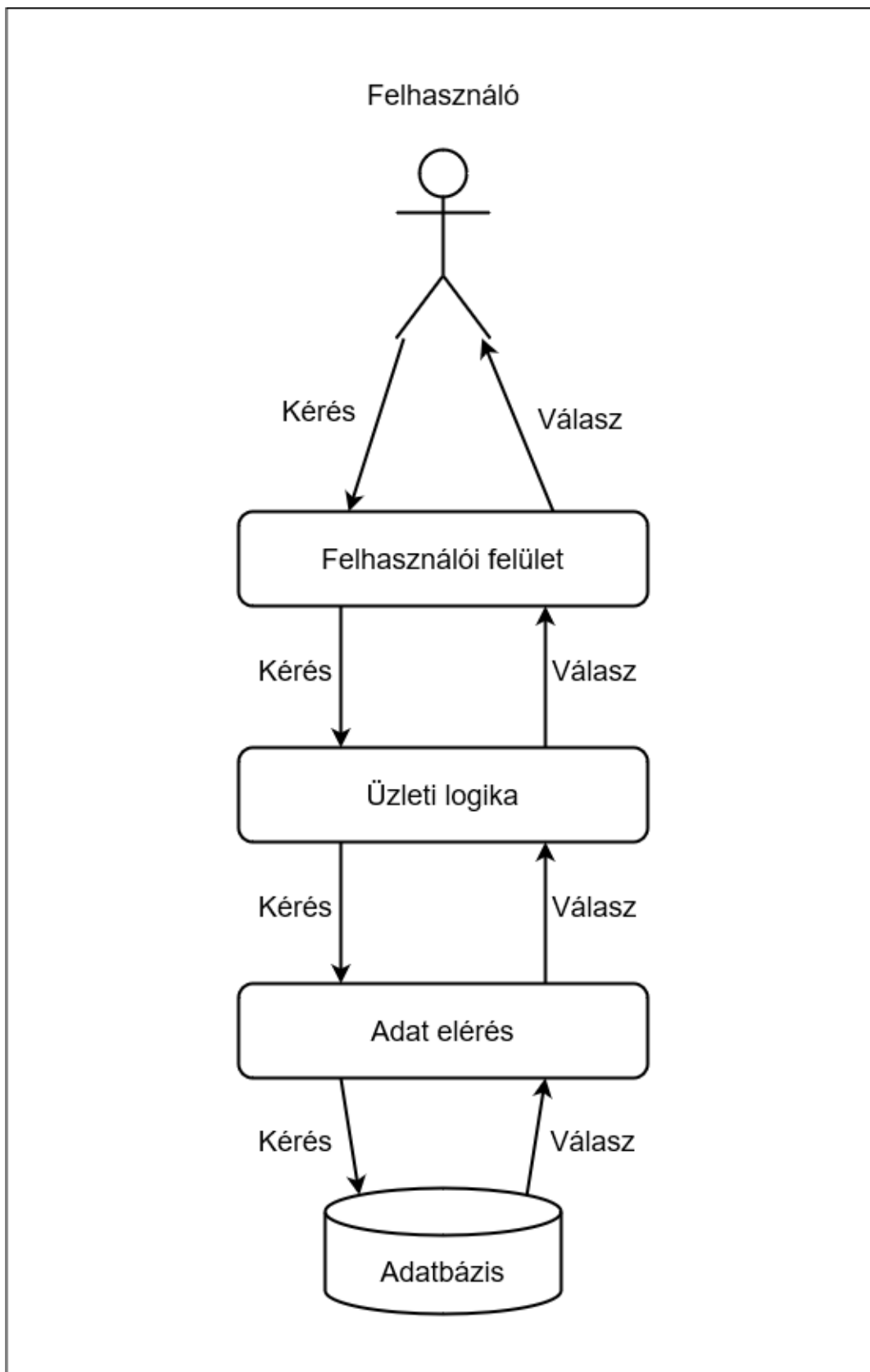
Budapest

Átlagos értékelés: 4.1

8. ábra – Szobák keresése

(Forrás: saját szerkesztés)

57



13. ábra – Háromrétegű alkalmazás szerkezete

(Forrás: saját szerkesztés)

hotel-booking-app-service

hotel-booking-app-service

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.application.hotelbooking.dto		46%		55%	11	13	2	6	2	3	1	2
com.application.hotelbooking.factories		0%	n/a	n/a	2	2	10	10	2	2	1	1
com.application.hotelbooking.wrappers.implementations		0%	n/a	n/a	5	5	8	8	5	5	2	2
com.application.hotelbooking.models		85%	n/a	n/a	1	3	2	12	1	3	1	3
com.application.hotelbooking.services.implementations		99%		100%	1	92	0	275	1	64	0	14
com.application.hotelbooking.transformers		98%		100%	1	45	0	55	1	38	0	8
com.application.hotelbooking.services.repositoryservices.implementations		100%	n/a	n/a	0	35	0	44	0	35	0	7
com.application.hotelbooking.services.imagehandling		100%		100%	0	7	0	23	0	4	0	2
com.application.hotelbooking.exceptions		100%	n/a	n/a	0	10	0	20	0	10	0	10
Total	143 of 2,108	93%	9 of 96	90%	21	212	22	453	12	164	5	49

Created with JaCoCo 0.8.8.2020

Created with JaCoCo 0.8.8.20221

28. ábra – JaCoCo jelentés a service modul lefedettségéről

(Forrás: saját szerkesztés)

```

53.
54. private void addToHotelsListIfNotPresentYet(RoomModel roomModel, List<HotelWithReservableRoomsServiceDTO> hotelsWithReservableRoomsServiceDTO, UniqueReservab
55. Optional<HotelWithReservableRoomsServiceDTO> optionalHotelWithReservableRoomsServiceDTO = hotelsWithReservableRoomsServiceDTO.stream().filter(hotel -> ho
56.
57. if (optionalHotelWithReservableRoomsServiceDTO.isPresent()){
58. HotelWithReservableRoomsServiceDTO hotelWithReservableRoomsServiceDTO = optionalHotelWithReservableRoomsServiceDTO.get();
59. if (hotelWithReservableRoomsServiceDTO.getUniqueReservableRoomOfHotelServiceDTOList().stream().filter(room -> room.equals(uniqueReservableRoomOfHotel
60. hotelWithReservableRoomsServiceDTO.getUniqueReservableRoomOfHotelServiceDTOList().add(uniqueReservableRoomOfHotelServiceDTO);
61. }
62. } else {
63. List<UniqueReservableRoomOfHotelServiceDTO> uniqueReservableRoomOfHotelServiceDTOList = new LinkedList<>();
64. uniqueReservableRoomOfHotelServiceDTOList.add(uniqueReservableRoomOfHotelServiceDTO);
65.
66. hotelsWithReservableRoomsServiceDTO.add(
67. new HotelWithReservableRoomsServiceDTO(
68. roomModel.getHotel().getHotelName(),
69. roomModel.getHotel().getCity(),
70. roomModel.getHotel().getImageName(),
71. roomModel.getHotel().getAverageRating(),
72. uniqueReservableRoomOfHotelServiceDTOList
73. ));
74. }
75. }
76.
77. private List<HotelWithReservableRoomsServiceDTO> assignUniqueRoomsToHotels(List<Long> availableRoomsIds, RoomSearchFormServiceDTO roomSearchFormServiceDTO) {
78. List<HotelWithReservableRoomsServiceDTO> hotelsWithReservableRoomsServiceDTO = new LinkedList<>();
79. for (Long availableRoomsId : availableRoomsIds){
80. RoomModel roomModel = roomRepositoryService.getRoomById(availableRoomsId).get();
81. addToHotelsListIfNotPresentYet(
82. roomModel,
83. hotelsWithReservableRoomsServiceDTO,
84. buildUniqueReservableRoomOfHotelServiceDTO(roomModel, roomSearchFormServiceDTO));
85. }
86. return hotelsWithReservableRoomsServiceDTO;
87. }
88.
89. public List<HotelWithReservableRoomsServiceDTO> searchHotelsWithReservableRooms(RoomSearchFormServiceDTO roomSearchFormServiceDTO) {
90. List<Long> roomIds = roomRepositoryService.getRoomsWithConditions(roomSearchFormServiceDTO);
91. List<Long> availableRoomsIds = filterAvailableRooms(roomSearchFormServiceDTO, roomIds);
92.
93. return assignUniqueRoomsToHotels(availableRoomsIds, roomSearchFormServiceDTO);
94. }
95.
96. public boolean isEndDateAfterStartDate(LocalDate startDate, LocalDate endDate){
97. return endDate.isAfter(startDate);
98. }
99. }

```

29. ábra – JaCoCo jelentés RoomServiceImpl osztály sorainak lefedettségéről

(Forrás: saját szerkesztés)

```

1  FROM maven:3.9.6-eclipse-temurin-17 AS build
2
3  WORKDIR /app
4
5  COPY . .
6
7  RUN mvn clean install -DskipTests
8
9  FROM eclipse-temurin:17-jdk-alpine AS runtime
10
11  WORKDIR /app
12
13  RUN mkdir images
14
15  RUN mkdir demo_images
16
17  COPY images ./demo_images
18
19  COPY copy_files.sh .
20
21  RUN chmod +x copy_files.sh
22
23  COPY data.sql .
24
25  COPY --from=build \
26      /app/hotel-booking-app-web/target/hotel-booking-app-web-0.0.1-SNAPSHOT.jar \
27      hotel-booking-app.jar
28
29  ENTRYPOINT ["java", "-jar", "hotel-booking-app.jar"]

```

30. ábra – A Dockerfile

(Forrás: saját szerkesztés)

```

1  version: '3'
2
3  volumes:
4    my-datavolume:
5    files-volume:
6
7  services:
8    docker-mysql:
9      image: mysql:8.3
10     env_file:
11       - .env
12     environment:
13       - MYSQL_ROOT_PASSWORD=${MYSQL_ROOT_PASSWORD}
14       - MYSQL_DATABASE=hotel_db
15       - MYSQL_USER=${DATASOURCE_USERNAME}
16       - MYSQL_PASSWORD=${DATASOURCE_PASSWORD}
17     volumes:
18       - my-datavolume:/var/lib/mysql
19     ports:
20       - 3307:3306
21
22   app:
23     build: .
24     restart: unless-stopped
25     volumes:
26       - ./logs:/app/logs
27       - files-volume:/app/images
28     ports:
29       - 8080:8080
30     env_file:
31       - .env
32     environment:
33       SPRING_DATASOURCE_URL: jdbc:mysql://docker-mysql:3306/hotel_db?allowPublicKeyRe
34     depends_on:
35       - docker-mysql
36

```

31. ábra – A docker-compose.yml fájl

(Forrás: saját szerkesztés)