Hotelszoba foglaló webalkalmazás Spring Boot keretrendszerben

Tartalomjegyzék

1. Bevezetés 1
2. Felhasznált eszközök technolódiák 2
   1. Apache Maven
   2. Verziókezelés és Git
   3. Intellij IDEA
   4. MySQL
   5. Spring framework
      1. Spring Boot
      2. Spring Data JPA
      3. Spring Web MVC
      4. Spring Security
      5. Spring Initializr
   6. Lombok
   7. Tesztelés
      1. JUnit
      2. Mockito
   8. Thymeleaf
      1. Fragmentek
   9. Bootstrap
   10. I18n
3. Egyéb
   1. UML diagramm
   2. Adatbázis séma
4. Az alkalmazás felépítése
   1. Domain réteg
5. A felültet bemutatása
6. Rétegelt Architektúra
   1. Domain modul bemutatása
   2. Service modul bemutatása
   3. View modul bemutatása
   4. Domain modul bemutatása
   5. Transformerek, ModelMapper

Bevezetés

Apache Maven

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

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

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

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

Verziókezelés és Git

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

Szintén problémát jelenthet az, amikor több fejlesztő dolgozik ugyan azon 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 ugyan azt a kódbázist szerkeszti. Ezt elkerülendő, lehetőségünk van az eredeti kódbázisból elágazásokat létrehozni. Ezek az elágazások, vagy másnéven branch-ek ugyan azt 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 alap működésüket sikerült elérnem.A képen szöveg, emléktábla, képernyőkép látható

Automatikusan generált leírás

1. ábra – Git ágak

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

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

IntelliJ IDEA

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

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

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

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

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

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

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

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

Spring keretrendszer

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.

Spring Core

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 felelőséget ruházza ét egy keretrendszerre vagy konténerre. Az alkalmazás megmondja milyen objektumai vannak és ezek között mik az összefüggések, de a konténer fogja létrehozni és kezelni őket. Ennek eredményeként kódunk lazán csatolt lesz, ezáltal sokkal könnyebb karban tartani, függőségeit cserélgetni és a részeit önállóan tesztelni.

ApplicationContext

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

Függőség Befecskendezés

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

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

Spring Data JPA

ORM

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

JPA

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

Hibernate

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

Entitások

A JPA Entitások olyan Java osztályok, melyek leírják egy adatbázis tábláját. Az osztályban szereplő attribútumok a tábla egy-egy oszlopának felelnek meg. Az osztály példányai pedig a tábla egy sorát fogják képezni.

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

Relációk

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

@OneToOne

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

@ManyToOne és @OneToMany

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

@ManyToMany

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

Spring Data

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

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

Viszont ezeknél a műveleteknél valószínűleg kicsit bonyolultabb lekérdezéseket szeretnénk végezni. A létrehozott repository-ban megadhatunk saját metódusokat is, melyekhez a @Query annotációval valamilyen saját lekérdezést rendelhetünk. Ám a Spring Data JPA képes arra is, hogy bizonyos metódusneveket értelmezzen és ezekhez futásidőben előállítsa hozzájuk a kívánt lekérdezést. Ezeknek a metódusneveknek viszont meg kell felelnie egy előre meghatározott konvenciónak, hogy értelmezni tudja.

A @Repository annotáció 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 itnerfészeket.

Spring Security

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

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

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

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

Thymeleaf

A thymeleaf egy modern szerver-oldali Java template

Az alkalmazás rétegei

Domain réteg

Tartalomjegyzék

1. Bevezetés 1
2. Felhasznált eszközök technolódiák 2
   1. Apache Maven
   2. Verziókezelés és Git
   3. Intellij IDEA
   4. MySQL
   5. Spring framework
      1. Spring Boot
      2. Spring Data JPA
      3. Spring Web MVC
      4. Spring Security
      5. Spring Initializr
   6. Lombok
   7. Tesztelés
      1. JUnit
      2. Mockito
   8. Thymeleaf
      1. Fragmentek
   9. Bootstrap
   10. I18n
3. Egyéb
   1. UML diagramm
   2. Adatbázis séma
4. Az alkalmazás felépítése
   1. Domain réteg
5. A felültet bemutatása
6. Rétegelt Architektúra
   1. Domain modul bemutatása
   2. Service modul bemutatása
   3. View modul bemutatása
   4. Domain modul bemutatása
   5. Transformerek, ModelMapper

security technikai leírás

Korábban erre úgy volt lehetőség, hogy megadtunk egy konfigurációs osztályt, mely kiterjesztette a WebSecurityConfigurerAdapter osztályt, és felülírtuk annak configure metódusát. Viszont a Spring Security 5.7.0-M2 verzió óta ez elavultnak minősül (deprecated) és helyette a komponens alapú konfiguráció a javasolt. Ehhez csupán létre kell hoznunk egy metódust mely egy HttpSecurity példányt kap paraméterül és visszaad egy SecurityFilterChain-t. A metódust a @Bean annotációval kell ellátni, hogy a komponens szkennelés folyamán a keretrendszer megtalálja, és beregisztrálja a komponensek közé.

A már létező felhasználókat az adatbázisunkban tároljuk. Ahhoz, hogy innen megkapjunk egy adott felhasználót implementálnunk kell a UserDetailService interfészt és meg kell adni a loadUserByUsername()működését. Ez a metódus paraméterül kapja a bejelentkezni kívánt felhasználó nevét, és lekérdezi a hozzá tartozó User entitást az adatbázisból. Ennek adataival létrehoz egy UserDetails objektumot, majd megvizsgálja, hogy a megadott jelszó egyezik-e a tárolt felhasználó jelszavával. Ha igen, akkor úgy tekinti, hogy a felhasználó be van lépve, és hozzáadja adatait a SecurityContextHolder-hez.

A SecurityContextHolder a jelenleg bejelentkezett felhasználó tárolásáért felelős. Ebből egy Authentication objektumon keresztül érhetjük el azt. Lekérdezhetjük ebből a felhasználó nevét, azt, hogy milyen jogosultságai vannak, vagy akár getPrincipal() metódus segítségével megkaphatjuk az egész UserDetails objektumot ami tartalmazza a felhasználó minden tárolt adatát.

A felhasználók jogosultságainak kezelésére szerepköröket hozzáférését szeretnénk valamilyen szerepkör szerint korlátozni.