SZAKDOLGOZAT

Jurás Bence

Debrecen

2019

Debreceni Egyetem

Informatikai Kar

Java webalkalmazás fejlesztése

Témavezető

Dr. Jeszenszky Péter
adjunktus

Készítette

Jurás Bence
programtervező informatikus BSc

Debrecen

2019

Tartalomjegyzék

1. Bevezetés

A dolgozatom témájának kiválasztásnál azt tartottam legfontosabbnak, hogy egy olyan tudást szerezzek, amivel később munkába állhatok.

A Java nyelvet választottam, amit az egyetemi oktatás során nagyon megkedveltem.

A dolgozatom egy online vásárlással foglalkozó oldal elkészítésére irányult. Elkészítésére célirányosan a modern, valamint az iparban is használt technológiákat választottam. A programmal sikerült minden olyan célt elérni, amit még az elején kitűztem magamnak.

Főbb célom e technológiák megértése, és felhasználása volt közel valós környezetben. Az objektum-relációs leképezés, a web API-k megértése, a hatékony tiszta kód megvalósítása mind céljaim voltak.

A program maga nem teljesít minden célt, amit még az elején kitűztem magamnak az idő hiánya miatt, ám a főbb funkciók elkészültek.

Az oldalra regisztrálást követően tudunk termékeket megtekinteni, termékekre keresni. A keresést tudjuk szűkíteni, valamint a kereséshez egy olyan szűrő megoldást valósítottam meg, amiben attribútumokat, kategóriákat adhatunk át.

A dolgozat első fejezete a felhasznált technológiák rövid bemutatásával kezdődik. A második fejezet a szoftvert ismerteti. A szoftver ismertetését először az entitások bemutatásával, majd a kontrollerek, végül pedig a szolgáltatások bemutatásával zárom.

A szoftverben a keresést megvalósító szűrő lesz az, ami a legtöbb funkcionalitást igénybe veszi.

A szűrőnél egy sajátos terminológiát is megfogalmaztam, ám erről a szoftver ismertetésénél még terjedelmesen írni fogok.

2. A felhasznált technológiák

Ebben a fejezetben szeretnék kitérni a fejlesztéshez felhasznált technológiákra.

2.1. Spring Framework

A Spring Framework egy nyílt forráskódú Java alapú keretrendszer. A függőség befecskendezésre (DI) és kontroll megfordításra (IoC) alapelvre épül.

Mára már nagyon sokrétű a keretrendszer. Magába foglal aspektus-orientált programozást, tranzakció kezelést, MVC-t (modell-nézet-vezérlő) szerkezeti mintát, hitelesítést, autorizációt, tesztelést stb.

A Spring az Enterprise Java Beans (EJB) méltó kihívója lett, később pedig egy sokkalta népszerűbb alternatívává nőtte ki magát.

2.1.1. Történelme

A történelme e nagyobb eseményekben foglalható össze **[**2**]**:

- Az első verzió elkészítésére 2002-ben került sor Rod Johnson által.
- Ezt követte 2004 márciusában az első kiadása a Spring Framework 1.0-nak.
- A Spring 2.0 megjelenése 2006-ban. Ez a verzió az XML alapú konfigurációt könnyíti meg.
- 2007-ben a 2.5-ös verzió az annotációs konfigurációt emelte be.
- 2012-ben a 3.2-es verzió a Java konfigurációt, Hibernate 4-et és Servlet 3.0-át hozta magával. Teljes java 7-es támogatást kapott.
- A Spring 4.0 2014-ben érkezett. Ő már a Java 8-as verzió támogatását biztosította.
- 2012-ben igény merül fel egy sokkalta kompaktabb és egyszerűbb Spring keretrendszerre.

2.2. Spring Boot

A Spring Boot egy Java nyelven írt nyílt forráskódú keretrendszer. Főként mikroszerviz architektúrára épülő alkalmazásokat fejlesztenek vele (azaz modulok fejlesztése, ahol az egyes modulok egymástól teljesen függetlenek, bármikor lecserélhetőek), ám monolitikus (egy nagy komplex alkalmazás) fejlesztésének is van támogatottsága.

Célja a Spring Framework-nek egy egyszerűbb megvalósítása. Ezen céllal a betanulási, fejlesztési idő lecsökkentése is megvalósításra került. A Spring Framework-beli konfigurációk

elérhetőek sokkalta kevesebb kód megírásával, így az egyes konfigurációs annotációk több konfigurációt is magukba foglalnak.

Egy Spring Boot alapú REST API alapjának elkészítéséhez csupán néhány Maven függőságre van szükség, úgymint [2]:

Spring Boot Starter web	Web alkalmazások készítéséhez (RESTful + Spring MVC).
Spring Boot Data JPA Starter	Tartalmazza a Spring Data JPA-át és a Hibernate-et.
Spring Boot Security Starter	Ezen függőség segítségével gondoskodhatunk a biztonságról alkalmazásunkban.
Spring Boot Test Starter	Tesztelő könyvtárakat tartalmaz, úgymint: Mockito, JUnit és Hamcrest.

2.3. Spring Data

Az egyik legelőrehaladottabb megoldást az adatfeldolgozásra a Spring Data Project biztosítja. Képes automatikusan generálni tárolókat az általunk átadott funkcionális interfészből. A generált tároló biztosítja számunkra az adatelérés réteget. Bővebben a [3] cikkben olvashatunk róla.

Az absztrakció alapja a `Repository` interfész. Ebből származnak le olyan interfészek, mint például: `JpaRepository, CrudRepository, MongoRepository`.

A bemutatásában sokszor fogok a dokumentációhoz [4] visszatérni.

2.3.1. Annotáló konfiguráció

Használni úgy tudjuk, hogy a repository-nkat injektáljuk. Az injektálás történhet például mezőn annotálva vagy konstruktoron annotálva.

Például:

```
@Autowired
private UserRepository userRepository;

//Illetve:
@Autowired
public UserService(UserRepository userRepository){
```

```
        this.userRepository=userRepository;
}
```

2.3.2. Lekérdező metódusok

Standard, illetve összetett lekérdező műveletek végrehajtására tárolókat használunk. Egy interfészre van csak szükség, ami fontos, hogy kiterjesszen egy al interfészét a `Repository` interfész gyermek interfészei közül, vagy magát a `Repository` interfészt.

Egy példa egy termék interfész létrehozására:

```
interface ProductRepository extends Repository<Product,Long>{…}
```

Generikus paramétere egy termék entitás, míg a másik a termék entitás egyedi azonosítójának a típusa.

Ezt követően egy lekérdező műveletet a következőképpen készíthetünk el:

```
interface ProductRepository extends Repository<Product,Long>{
        List<Product>findByCategory(String category);
}
```

Magát a lekérdezést a Spring Data absztrakciói mögött elrejtve oldja meg. A függvényünk egy termékből álló listát fog visszaadni, leszűrve őket a kategóriájukra. Ehhez hasonló `findyBy` műveleteket könnyen létre tudunk hozni. Pusztán figyelni kell, hogy mi az a paraméter, amire szűrni szeretnénk.

Például életkorra a következőképpen nézne ki:

```
List<User>findByAge(int age);
```

A leszűrt elemekre lehet használni a distinct műveletet is:

```
List<Product>findDistinctProductByColor(String color);
```

Rendezni is lehet a következőképpen:

```
List<Product>findByColorOrderByAgeAsc(String color);
```

A névben szerepelni kell ilyenkor az OrderBy-nak, aztán a rendező tulajdonság neve következik majd az, hogy növekvő, vagy csökkenő sorrendben szeretnénk látni a leszűrt elemeket.

Előfordulhat, hogy 1-nél több tulajdonságra szeretnénk szűrni, ilyenkor egy összekapcsoló logikai műveletre van szükség (`Or/And`).

Például:

```
List<Product>findByAgeOrHeight(int age,int height);
```

2.3.3. Lekérdezés készítés

Korábban már írtam a lekérdezések használatáról, illetve megadásáról, ám most jobban belemegyek.

A felépítése egy ilyen lekérdezésnek a következő módon alakul:

`{típus} findyBy {tulajdonság_név}[and | or [{tulajdonság_név}[and | or]..] ]([ [típus tulajdonság_név],[típus tulajdonság_név]..]);`

Fontos, hogy a paraméterlista elemszáma meg kell, egyezzen a lekérdezésben átadott tulajdonságnevek számával és a típusainak is meg kell egyeznie.

Felhasználható kulcsszavak:

`And, Or, Is, Equals, Between, LessThan, LessThanEqual, GreaterThan, GreaterThanEqual, After, Before, IsNull, IsNotNull, NotNull, Like, NotLike, StartingWith, EndingWith, Containing, OrderBy, Not, In, NotIn, True, False, IgnoreCase.`

Ezekre néhány példa:

GreaterThan	findByAgeGreaterThan
Before	findByStartDateBefore
Containing	findByFirstnameContaining
Not	findByLastnameNot
NotIn	findByAgeNotIn (Collection <Age> ages)
True	findByActiveTrue ()

2.3.4. @Query használata

Amennyiben saját lekérdezéseket szeretnénk írni azt, megtehetjük a `@Query` annotáció használatával. Itt a függvényünk felett egy JPQL lekérdezést adhatunk meg.

Példa:

```
@Query("select p from Product p where p.color = ?1")
Product findByColor(String color);
```

Ebben a példában a sztringben megjelenő „p” egy a `Product` entitás egy példányát képviseli. A „p” segítségével tudunk hivatkozni a benne található tulajdonságok egyikére.

Alkalmazható a `Like` kifejezés is a következő módon:

```
@Query("select p from Product p where p.name like %?1")
List<Product>findByNameEndsWith(String name);
```

A % karaktert kiszedve a Spring Data egy érvényes JPQL lekérdezést hoz létre.

A lekérdezésben szerepeltetett paramétereket eddig csak pozíció szerint adtuk át, ám lehetséges ezt név szerint kötni is. Példa:

```
@Query("select p from Product p where p.color = :color or p.age = :age")
Product findByColorOrAge(@Param("color")String
color,@Param("age")Integer age);
```

2.4. Hibernate

A Hibernate [6] a JPA-nak az egyik legnépszerűbb megvalósítása. A Hibernate egy objektum-relációs le kepézést megvalósító programkönyvtár.

2.4.1. ORM

Hibernate segítséget nyújt az alkalmazásunknak abban, hogy az adatokat tartósan tároljuk. Maga a perzisztencia nem más, mint az a vágy, hogy az alkalmazás által kezelt adatok túléljék az alkalmazást. Tehát azt szeretnénk, hogy az objektumaink állapotai éljenek a JVM hatáskörén kívül is, így ez az állapot később még visszakapható. ORM segítségével az entitásainkra képezhetjük le adatbázisunk tábláink sorait.

A táblák leképezése POJO osztályokra XML vagy annotációs konfigurációval implementálható.

A Hibernate képes kezelni az egy az egyhez, több a többhöz kapcsolat az osztályok között. Objektum-orientált programozásból átemel többeket között például öröklést, polimorfizmust, kompozíciót és asszociációt. Támogatja az egyedi érték típusok le kepézését is [6]. Saját natív API-ja lehetővé tesz natív DML utasítások és natív tárolt függvények végrehajtását.

2.4.2. Annotációk

A `javax.persistance` csomagot használva érhetjük el őket. Az annotációk [6] segítségével tudjuk konfigurálni az entitásokat, és a kapcsolatot közöttük. Helyette használható XML konfiguráció is.

Néhány annotáció:

`@Entity`	Az entitás babjainkat ezzel az annotációval kell, hogy elássuk.
`@Table (name=táblanév)`	Az átadott táblanév alapján köti az adatbázisban megtalálható táblát ezzel az entitással.
`@Id`	Jelezzük, hogy a tulajdonság egy egyedi azonosító.
`@Column (name=oszlopnév)`	Az átadott oszlopnév alapján köti az adatbázisban található táblaoszlopot ezen entitás tulajdonságához.
`@Lob`	Nagyméretű objektumok használata esetén kell megadni.
`@OneToOne`	Egy az egyhez kapcsolathoz használandó.
`@OneToMany`	Egy többhöz kapcsolathoz használandó.
`@ManyToOne`	Több az egyhez kapcsolathoz használandó.
`@ManyToMany`	Több többhöz kapcsolathoz használandó.

3. A szoftver bemutatása

Ebben a fejezetben szeretnék kitérni az elkészített projekt működésére, történetére bővebben.

3.1. A program főbb funkciói

A cél egy webáruház megvalósítása. Egy webáruház legalapvetőbb funkciói közé tartoznak a termékek feltöltései, illetve ezek megvásárlása, vagy az ezekre való licitálhatóság. Ezt ki kell egészíteni természetesen plusz funkciókkal, hogy egy használható, felhasználóbarát oldalt kapjunk. Ezen plusz funkciók (illetve adalékok):

- felhasználó kezelés
- termékek kategóriái
- termékeken megjelenő attribútumok (tulajdonságok)
- termékekre való kereshetőség
- termékekre való licitálhatóság
- fix áras megvásárolhatóság.

A felhasználói élmény megteremtésé érdekében szükséges lehetőséget teremteni egy olyan kapcsolati formára a potenciális vevő és a kínáló között, mint a kommentelés lehetősége. Továbbá szükséges, hogy a feltölteni kívánt terméket a vevő fontos információkkal láthassa el. Ez nálam egy hosszú leírás megadásában teljesedik ki, illetőleg képeket tölthet fel a termékeihez a felhasználó.

A keresés működéséhez elengedhetetlenül szükségesek a már említett kategóriák, illetve attribútumok. A kategóriák jelentik a termék besorolhatóságát, úgymint például Telefon, Bútor, Számítástechnikai eszköz stb. Az attribútumok megadása a termék egy olyan komolyabb leírása, ami nem csak azt teszi lehetővé, hogy azt jobban megismerjük, de a kereshetőségét is ez teremti meg, természetesen a kategóriákkal kiegészülve.

Attribútum alatt általában egy név-érték párt gondolunk. Itt is valamennyire hasonló a programban az attribútumként hívott megoldás a jólismert név-érték pár attribútumokhoz, de nem ugyanaz. Ezt a későbbiekben még sokkal részletesebben kifejtem.

Egy egyszerű példa lehetne egy valamilyen Körte márkájú telefon, amiről tudjuk, hogy zöld színű, Android operációs rendszer fut rajta, 3GB rendszermemóriával rendelkezik és így tovább. Ez mind olyan leírása a terméknek, ami alapján tudunk szűrni ilyen tulajdonságokra, például ha a 3GB vagy annál nagyobb rendszermemóriával rendelkező, zöld színű telefonokat akarunk megkapni, akkor azt az implementált szűrőmmel megtehetjük.

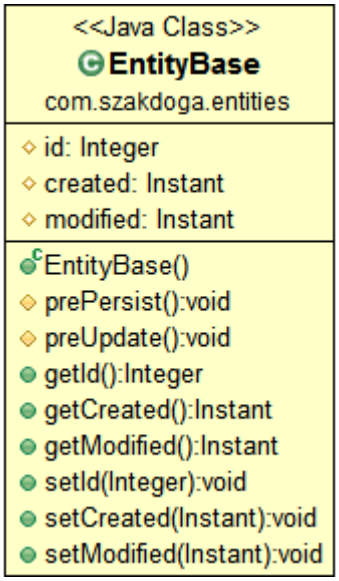
Az alapvető funkciók között megemlíteném a regisztrációt és a belépés lehetőségét. A regisztrációt követően kapunk a megadott e-mail címünkre egy aktiválásra felszólító e-mail-t, amiben ha az aktiváló URL-t nem látogatjuk meg, akkor nem lesz a regisztrált fiókunk aktiválva, ami elvesz tőlünk sok főbb funkciót, úgymint a termékek feltöltése, licitálás, valamint a fix áras megvásárlás. Amennyiben egy regisztrált felhasználó elfelejtette a jelszavát, könnyen kaphat újat.

A bejelentkezett felhasználó kap egy access token-t, amivel őt egyértelműen azonosítjuk. Bármilyen olyan funkció, ami belépést igényel, elérhető a token alapján megszerzett felhasználó nevében.

A szoftver csupán kettő nagyobb szerepkört különböztet meg, az `USER`-t illetőleg, az `ADMIN`-t. Funkciós korlátozások ezek szerint természetesen szükségesek, például a titkos felhasználói információk elrejtése, a kategóriák és az attribútumok csupán az `ADMIN` számára lehetségesek.

3.2. Entitások

3.2.1. Bázis entitás



1. ábra: A Bázis entitás UML osztály diagramja

Az entitások nagy része ezen entitásból származik, hiszen ők is rendelkeznének e mezőkkel, így adja magát az öröklés.

Tartalmazza az `Id`-t, amit elláttam a már korábban bemutatott `Id` annotációval, illetve a generálási stratégiájának az alapértelmezett `auto`-t állítottam be. Ez azt jelenti, hogy a perzisztencia szolgáltató fogja majd kiválasztani a neki megfelelőt.

Az adatbázisnak a MySQL-t választottam, amiben van „auto increment”, így nagyon kényelmessé teszi az egyedi azonosítók automatikus generálását.

Felvettem két további fontos mezőt, a létrehozás dátumát illetve a módosítás dátumát. Ezen kettőt elláttam olyan attribútummal, hogy a `JSON` szerializációban ne szerepeljenek. Pusztán azért, mert ha `Repository`-n keresztül kérnék le entitásokat, akkor a létrehozási és módosítási dátumok ne legyenek elérhetőek, minthogy ezek a felhasználó számára nem szükséges információk.

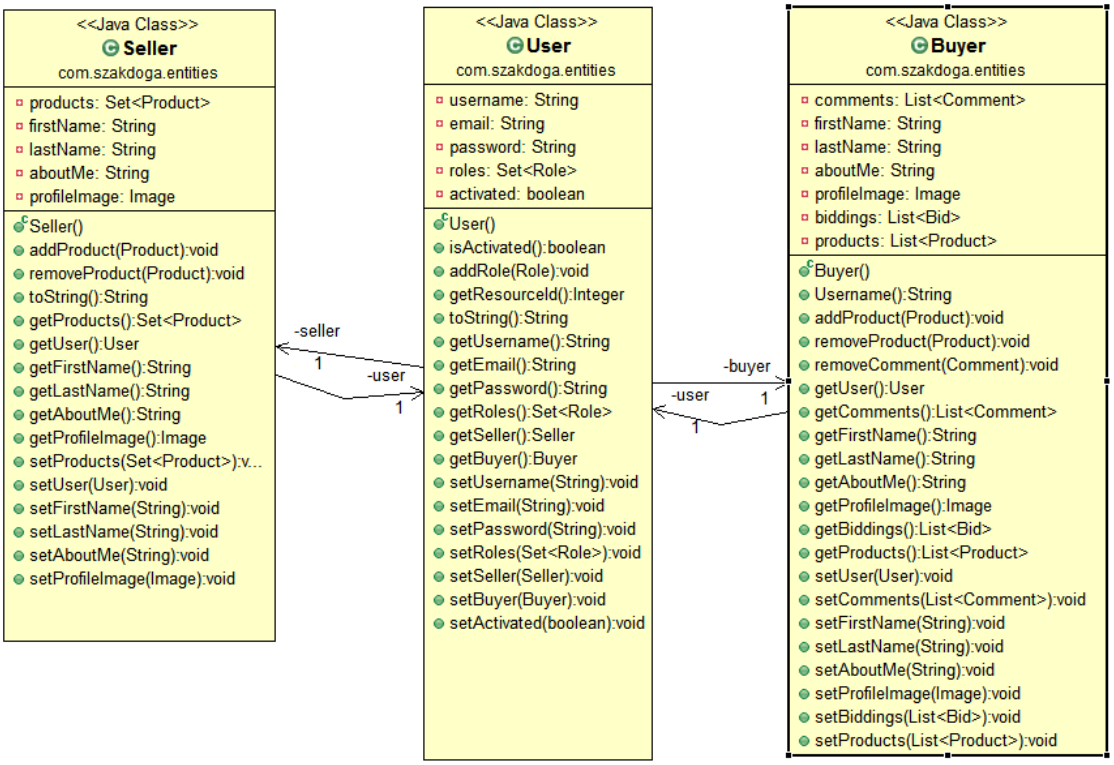
Végül, hogy megmondjam, hogy az entitásom az egy bázis entitás, a `javax.persistance` csomag által szolgáltatott `@MappedSuperClass` annotációt vettem igénybe.

Beemeletem a projektembe a *lombok* [1] nevű függőséget, ami nagyban megkönnyítette a munkámat, hiszen nem szükséges általa a getterek-setterek állandó legeneráltatása, így ezen boilerplate kódtól tisztábbak lettek az entitásaim, illetve a DTO-im.

[1] https://projectlombok.org/

3.2.2. User, Seller, Buyer entitások

A bázis entitás után ezen három osztály az, ami a legtöbb helyen megjelenik. A legtöbb entitás közvetlen vagy közvetett módon, de kapcsolatban áll ezen entitásokkal, főleg, hogy a `Seller` és a `Buyer` szülője a `User`.



2. ábra: A User, Seller, Buyer entitások

3.2.2.1. User entitás

A `User` entitás segítségével menthetünk le illetve érhetünk el felhasználókat. Egy felhasználót az e-mail címe, felhasználói neve, jelszava illetve szerepe alapján jellemezzük.

Előforduló asszociációs kapcsolatai:

`@ManyToMany`	A `Role` entitással
`@OneToOne`	A `Seller` és `Buyer` entitásokkal

Minthogy ez már olyan entitás, amint perzisztálni szeretnénk az adatbázisban, meg kell őt jelölni az `@Entity` annotációval. A jelszót, mint olyan információt, amit nem szabad továbbítani a repozitoris lekérésekhez, elláttam a `@JsonIgnore` annotációval.

Az asszociációra vonatkozó annotáción elforduló `mappedBy` azt a célt szolgálja, hogy megmondjuk, hogy melyik mező birtokolja a relációt. Ezt csak a birtokolt szerepelteti. A reláció másik oldalán egy `@JoinColumn` annotáció szerepel. A benne átadható `name` értéknek meg kell adni az adatbázis táblán található összekapcsoló oszlopnevet (külső kulcs nevét). Tehát azért fontos, hogy a birtokló oldalán legyen ez megadva, mert a birtoklónak az adatbázis táblájában szerepel az összekötő oszlopnév.

Az entitás rendelkezik továbbá egy aktiválságot jelző logikai értékkel is, ami a regisztráció után még hamis, ám a regisztrációs e-mail-ben megadott linkben lévő URL-en tovább haladva ez átíródik igazra.

Itt az aktivációnál megemlíteném, hogy az aktivációs karakterláncot az a `UserActivation` entitás tartalmazza. Rendelkezik továbbá még a lejárati dátummal. Magára az aktivációra a következő fejezetekben fogok kitérni, amikor majd a serviceket és kontrollereket mutatom be.

3.2.2.2. Seller entitás

A vevő entitás, ahogy említettem korábban, egy az egy asszociációban áll a `User` entitással. Létrehozatalára a felhasználó létrehozásánál kerül sor. Előforduló asszociációs kapcsolatai:

`@OneToMany`	A `Buyer` entitással
`@OneToOne`	A `User` és `Image` entitásokkal

Termékek hozzáadásra, illetőleg elvételére hoztam itt létre az `add/removeProduct` metódusokat.

Az `aboutMe` mezőn egy `@Lob` annotáció szerepel. Itt azért választottam a `Lob`-ot, mert ezt nagyobb szöveges mezőnek gondoltam ki, ahol esetleg valamilyen formázott HTML-ben megírt személyes leírást adhatunk át, ami esetleg akár nagyobb méretű is lehet.

3.2.2.3. Buyer entitás

A másik profil (a `Seller` mellett) ami létrejön a `User` regisztrációja során, az a `Buyer` entitás. Mezőket tekintve hasonlít a másik profilhoz, ám vannak kiegészítések.

Előforduló asszociációs kapcsolatai:

`@OneToMany`	A `Product, Bid` és `Comment` entitásokkal
`@OneToOne`	A `User` és `Image` entitásokkal

Az idő hiánya végett a buyer profiljára való kommentelés nem készült el.

3.2.2.4. Attribute entitás

Ezen entitás hivatott a speciális tulajdonságok megfogalmazásra. A kétoldalú kapcsolat végett elérhetőek azon termékek, amelyeken az aktuális attribútum meg lett adva.

Itt szerepel az az érték, amelyet majd egy típushoz kötünk. A típust az `AttributeCore` entitásban helyeztem el. Azért volt fontos nem itt szerepeltetni a típust, mert akkor nem tudnék rákeresni, ha nem volna egy pár egyedi attribútum típus.

A külön szedés előnye, hogy tudok olyan kapcsolatot felállítani, hogy ha például egy `AttributeCore` entitás típusa karakterlánc, neve pedig szín, akkor ehhez állhatnak kapcsolatban olyan attribútumok, melynek értékei a CSS3 színnevei közül valók. Tehát megtudok ehhez adni attribútumokat különböző értékekkel, amelyek mindegyike véges típusokból válogathat.

Típusnak három érték képzelhető el: egész, lebegőpontos szám, karakterlánc. Előforduló asszociációs kapcsolatai:

`@ManyToOne`	A `Product` és `AttributeCore` entitásokkal

3.2.2.5. AttributeCore entitás

Ahogy említettem, ő volna az attribútum típusait összefogó entitás. A típust enum-ban adtam meg. Ahhoz, hogy ennek a perzisztenciája működőképes lehessen, szükséges volt egy `@Enumerated` annotáció megadása, paraméterként pedig a típusát, aminek én ordinálist választottam, ami az enum-ot egész számként tárolja.

Előforduló asszociációs kapcsolatok:

`@OneToMany`	`Az Attribute` entitással

3.2.2.6. Bid entitás

Ezen entitással a licitálást valósítjuk meg. Vásárlóból azért egy listát kapcsolok hozzá, mert így tudom visszakeresni azt, hogy kik is voltak, akik licitáltak.

A licitálás szabályának értelmében szükséges mindig szigorúan nagyobb licitet feladni az adott termékre, így amikor lejár a dátuma a licitálásnak, akkor elegendő csak az utolsó licitálót kivenni a listából. Értéknek egész értéket használ.

Előforduló asszociációs kapcsolatai:

`@ManyToOne`	`A Product` és `Buyer` entitásokkal

3.2.2.7. Product entitás

Az egyik legfontosabb entitás. Hasonlóan a profilokhoz itt is `Lob`-ként adtam meg a leírását.

Képek feltöltésére az `Image` entitás alkalmazható. Az előbb felsorolt entitások nagy része szerepel itt.

Termék feltöltésénél meg kell adni, hogy az fix áras, vagy licitálós. Ezt a termék típust enum-ban kell megadnunk.

Természetesen a licit entitáshoz hasonlóan itt is egészként adhatjuk meg a fix árat, amelyet kötelező megadni a termék feltöltésénél, amennyiben az fix árasnak indul. Ami viszont közös a kettőben, hogy egy bizonyos időintervallumot kell megadni.

A keresés egyszerűsítése végett felvettem egy aktivitást jelző logikai értéket, ami akkor igaz, amikor még nem járt le a termék, vagy ha fix áras, de még nem lett megvásárolva.

Így a keresésnél nem kell a dátumot hasonlítani mindegyiknél, hogy az még érvényes-e, pusztán elég e logikai értéket venni alapul.

A terméket a már megadott attribútumokon kívül jellemezi még a kategória, ami a kereshetőség és specifikálhatóság szempontjából elengedhetetlenül fontos.

A kommentek is nagyon fontosak, hiszen adni kell egy fórumot a potenciális vevőknek a termékre vonatkozó kérdéseik feltételére.

A munka elején úgy képzeletem, hogy egy belső levelezéssel oldom meg a kommunikáció lehetőségét a vevők és az eladók között, ám rájöttem, hogy ezzel azt a potenciált veszítem el, hogy a vásárlók egy még átfogóbb képet kapjanak az adott termékről.

Ezenfelül, ami még nagyon fontos, hogy az eladót is lehessen kommentálni, hiszen az ő munkássága az oldal presztízsét javíthatja, de ugyanilyen arányban rombolhatja is. Sajnálatos módon ez már nem készült el, de amennyiben ez a projekt valós környezetbe kerülne ki, egészen biztos, hogy megvalósításra kerülne.

Előforduló asszociációs kapcsolatai:

`@OneToMany`	A `Comment, Attribute` és `Image` entitásokkal
`@ManyToOne`	A `Buyer`, entitással
`@ManyToMany`	A `Category` entitással

3.2.2.8. Category entitás

Az alapgondolatot már megfogalmaztam a kategóriákkal kapcsolatban, ám azt hozzátenném, hogy az alapelgondolás részét képezi az is, hogy a kategóriák faszerűen képzelhetőtek el. Ezt a faszerű kialakítást úgy gondoltam, hogy kiindulásként van egy null értékű szülő kategória. Belőle származnak le az alkategóriák, melynek mindegyike egy újabb szülő kategória lehet, de nyilván már nullértékű nem.

Előforduló asszociációs kapcsolatai:

`@ManyToMany`	A `Product` entitással

3.2.2.9. Image entitás

A profilokon megjelenő képek eltárolásáért felel, illetőleg a termékeken megjelenő képekért. Lob-ként kezelem és bájt tömbben tárolom.

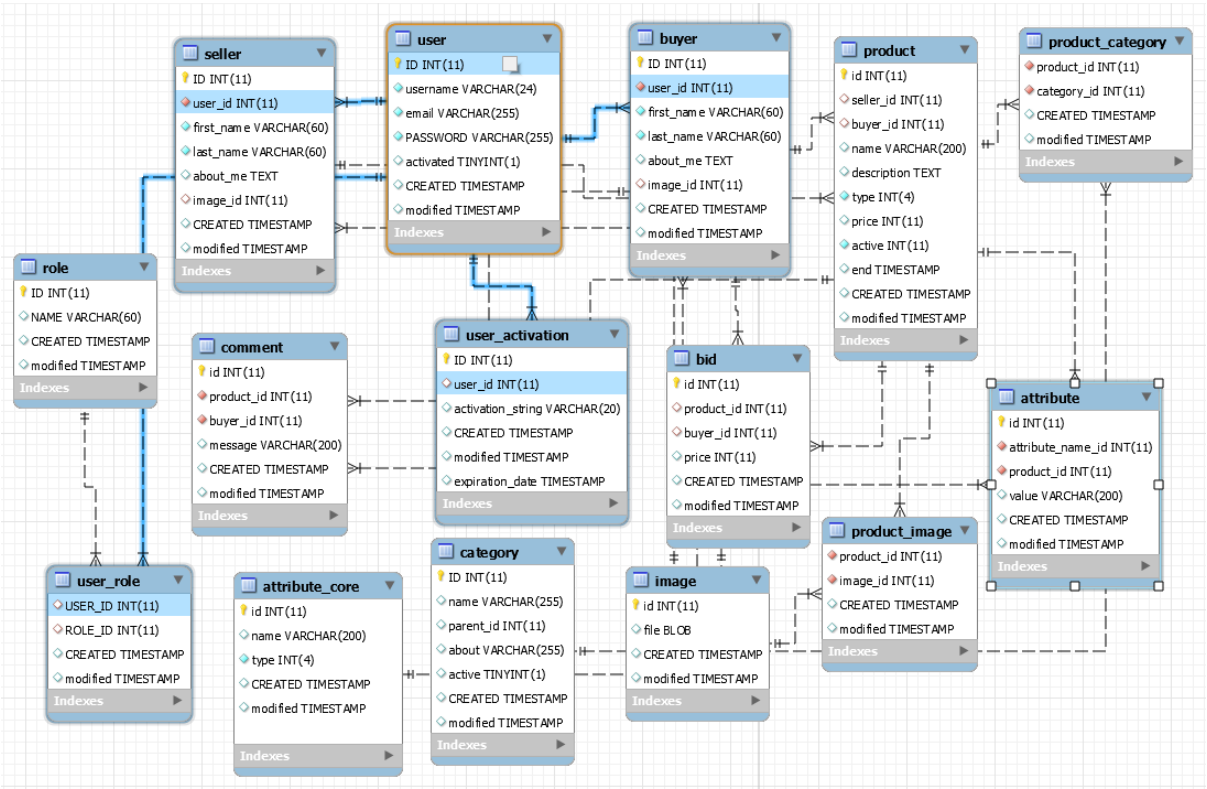
3.2.2.10. AccessTokenEntity, RefreshTokenEntity entitások

Az access token JDBC-vel való tárolására használandók, illetőleg a token megszerzésére, elpusztítására.

3.3. Adatbázis

Ezen rövid fejezettel a már bemutatott entitásokat szeretném bemutatni, hogy az adatbázisban hogyan állnak össze.

3.3.1. Diagram



3. ábra: Az adatbázis séma diagram

3.3.2. Leírás

Magát az adatbázist MySQL-ben valósítottam meg, minthogy ez egy nagyon jól dokumentált, és egy ilyen típusú webalkalmazás felépítésre rendkívül ajánlott a fundamentumok megismerésére.

Összesen 15 táblát hoztam létre. Kevesebbel nehéz volna kiszolgálni az alkalmazást, többel pedig redundancia léphet fel. Kiindulásnál ugyanis úgy alkottam meg a képek perzisztálását, hogy a profil táblákon volt egy lob tárolásra alkalmas oszlop, míg a termékek pedig már a kép entitással álltak kapcsolatban.

Ám a kis átalakítás hatására mindenki a kép táblát használta, ez pedig azért is jó, mert így sokkal kevesebb mezőből álló táblákat hozhattam létre. Maga a kép tábla egy irányú kapcsolatban lett felvéve, így aztán kell egy összekapcsoló tábla, amiben a kapcsolatot szerepeltetem. A képek tábla esetében ez a `product_image` tábla. Kategóriák esetében, minthogy az egy több a többhöz kapcsolatban lett felvéve, ő is kapott egy összekapcsoló táblát, a `product_category`-t.

3.4. Kontrollerek

Ezen fejezetben a kontrollerek bemutatására kerítek sort. Többnyire azonos kialakítást képviselnek, a felesleges komplexitás elkerülése érdekben.

A kontrollerek implementálják a REST API-t, és többnyire JSON-t adnak vissza. Azon egyszerűsítés érdekében, hogy a kontroller metódusait ne kelljen mindig megjelölni a `@RequestMapping` és `@ResponseBody` annotációkkal, helyette a kontrollert látom el a `@RestController` annotációval. A `@RestController` sztereotip annotáció kombinálja a `@ResponseBody`-t és a `@Controller` annotációkat.

3.4.1. Csontváz

A kontrollerjeim nagyrésze ugyanazon függvényeket/metódusokat tartalmazza így törekedtem arra, hogy azok ne térjenek el egymástól, így az egységességnek köszönhetően esetleges javításuk, továbbfejlesztése sokkalta egyszerűbb.

Majdnem minden kontroller rendelkezik egy saját service-szel is, ami a kontroller mögötti logikát valósítja meg, továbbá a perzisztáló repozitorikkal foglalkozik.

Ezen csontváza a következőképpen fest:

`@PostMapping public DTO create (@RequestBody DTO, HttpServletResponse)`	Új entitások felvételére szolgál. Az első paramétert leképezve entitássá mentjük le az érvényesítést követően. A `HttpServletResponse`-t pedig arra használom, hogy ha esetleg valamilyen hiba adódik, akkor egy HTTP hibakód formájában azt itt megadhatom.
`@PutMapping („/{id}”) public DTO update (@PathVariable („id”) id, @RequestBody DTO, HttpServletResponse)`	Már létező entitások frissítésre alkalmazandó. Szükséges továbbá egy egyedi azonosító, ami alapján a már létezőt lekérdezhetjük.
`@GetMapping („/{id}”) public List <DTO>get (@PathVariable („id”) id, HttpServletResponse)`	Egyedi azonosító alapján való lekérés.

`@GetMapping („/all”) public List <DTO> get ()`	Az összes létező entitás (DTO-vá képezve) lekérése.
`@GetMapping („/size”) public List <DTO>size ()`	Az összes létező entitás darabszáma. Szükség lehet rá, ha lapozva szeretnénk lekérni a már létező entitásokat.
`@GetMapping („/all/{page}/{size}”) public List <DTO> getAll (@PathVariable („id”) page, @PathVariable („id”) size)`	Lapozható lekéréshez. Szükséges átadni egy lap és egy darabszámot, hogy melyik lapról szeretnénk hányat.

3.4.2. User kontroller

A csontvázban bemutatott metódusokon, függvényeken kívül szerepel még néhány, ami a felhasználó műveleteihez elengedhetetlen.

Található benne egy egyszeri meghívást biztosító pollozást megvalósító metódus. A benne található funkciót a későbbiekben még bemutatom, ám lényegében csak annyit csinál, hogy a termékeket deaktiválja, ha azok elérték a végdátumot.

Található egy felhasználó aktiválásra alkalmas végpont, a `UserActivation`, amely egy aktivációs kódot vár, majd átirányít vagy egy oldalra, ami tájékoztat a hibás aktivációs kódról, vagy ha sikeres volt az aktiváció, akkor egy bejelentkezési oldalra.

3.4.3. Buyer, Seller kontroller

Az egységes kialakítás végett mindkettő pusztán egy plusz metódust valósít meg, a profilkép feltöltésére vonatkozót. A feltöltött profilkép elérésére az `Image` kontrollert használom, a képek egységes elérése végett. A feltöltést azért volt szükséges ezen a két kontrollerre elkülöníteni, mert nekem így tisztább, mintha az Image kontroller DTO-jában lenne felsorolva az összes entitásra vonatkozó egyedi azonosító, ahova kép kellhet, esetleg több, ha több helyre szeretnénk azt a képet feltölteni. Ám a képek felöltését sose képzeltem el úgy, hogy egy kép több helyen is szerepeljen, pl. a két profilon ugyanaz a profilkép. Mert bár valós igény lehet, hogy valaki azonos képet szeretne itt és ott is, bár én arra bátorítanám a felhasználókat, hogy azt a két profilt kezeljék két különböző profilként.

A képfeltöltő metódusnál szükség volt a `GetMapping`-ben felvenni a `consumes` paramétert, hogy megmondjam, hogy milyen MIME média típusokat fogadok el a klienstől. Jelen esetben a `multipart/form`-data-át engedek meg. Ekkor a kép feltöltésekor meg kell adni a fájlnak egy nevet is, amit én a feltöltő metódusomban „`image`”-nek várok. A feltöltött képet pedig `MultipartFile`-ként kezelem a továbbiakban, amiből könnyen kinyerhető azon bájt tömb, amit tudok majd tárolni a feltöltéskor.

3.4.4. Image kontroller

Az előző pontban kifejtett okok miatt itt nem szerepel, csupán az elérés. Továbbá a képek törlése is hasonlóan oda korlátozódik, ahol azokat feltöltjük azonos analógia végett.

A képek lekérése csupán egy kép id-re van szükség. Lett készítve egy függvény, ami profil képet ad vissza, nem pedig képet, így megkérdőjelezhető, hogy akkor miért is nem, a profil kontrollereken lett elhelyezve, ám ha jobban megnézzük, akkor látjuk, hogy bár az elérése valóban kibővül a „profile”-al, ám ugyanazt a funkcionálisát valósítja meg, mint a sima kép elérés. Amennyiben nincsen profilkép feltöltve, akkor egy alapértelmezett kép kerül átadásra.

A függvények bájt tömbként adják vissza a képeket, ám a kliens oldalon JPEG-ként kezelhetőek, minthogy ezt a média típust adtam át.

3.4.5. Product kontroller

A csontvázon kívül szerepel még a termékekhez képfeltöltés, illetve az eltávolítás lehetősége.

A létező kategóriák elérése is itt található. Bár tisztább lett volna egy kategória kontroller, ahova lehet posztolni kategóriákat, illetőleg akkor onnan elérni, ám az idő hiánya sajnos ezt nem tette lehetővé.

A csontváz részét nem képezi a törlés lehetősége, hiszen az `User, seller`, és `buyer` kontrollerek nem valósítják meg. A `User` kontrollernél bár elképzelhető volna egy logikai törlés, de ez nem lett implementálva.

Ám itt már szükséges lehet a termékek törlése, bár ez is kérdés tárgya lehet, hiszen ha esetleg a felhasználó esetleg obszcén viselkedést tanúsít, akkor kérdéses lehet, hogy esetleg azt a termékét távolítsuk el, ami a gondokat keltette, vagy magát a felhasználót helyezzük olyan pozícióba, hogy már ne legyen képes belépni az oldalra fiókjával.

A kitiltás ideje megint fontos kérdést vet fel, ám ha a felhasználó viselkedése minden határon túlmegy, akkor valóban törlése van szükség, amely megvalósítása szerintem mindenképpen csak logikai lehet. Hiszen annyi helyen szerepelhet, hogy törlése az asszociációs kapcsolat végett nagy munka volna mind eltávolítani. Ha felveszünk hozzá egy flag-et amivel jelezzük, hogy ő ki-tiltott felhasználó, akkor minden helyen ahol vele kapcsolatba kerülhetünk, (például ha egy licitálásban részt vett), akkor ott ezt figyelnünk kell. Ezen vizsgálat sajnos, ahogy írtam korábban, nem készült már el, csupán egy fizikai törlése a terméknek.

Két fő funkcionalitást megvalósító metódus szerepel még, a licitálás, illetőleg a termék megvásárlása, amely természetesen csak a fix áras termékekre vonatkozik.

A licitálás megvalósító metódus:

```
@PostMapping("/bid/{entityId}/{price}")
public void placeBid(
                @PathVariable("entityId")int entityId,
                @PathVariable("price")int price,
                HttpServletResponse response)
{
        Useruser = userService.getCurrentUser();

        userService.checkIfActivated(user);

        if(productService.get(entityId) == null)
        {

        response.setStatus(HttpServletResponse.SC_NOT_FOUND);
        return;
        }

        productService.bid(entityId,price);
}
```

A fixáras vásárlást megvalósító metódus:

```
@PostMapping("/buy/{id}")
public void buy(
                @PathVariable("id")Integer id,
                HttpServletResponse response)
{

        Useruser = userService.getCurrentUser();

        userService.checkIfActivated(user);
```

```
        if(productService.get(id) == null)
        {
        response.setStatus(HttpServletResponse.SC_NOT_FOUND);
        return;
        }

        productService.buy(id);
}
```

Azért tartottam fontosnak e két kódrészlet beemelését a kódból, mert így picit jobban át tudok menni azon, hogy mit is valósítanak meg, és hogyan is. Sok helyen kezdőnek azzal a metódusok/függvények, hogy a felhasználó service-ből kérem le az aktuális felhasználót. Ezen lépés szükséges, hogy megtudjam, hogy az aktuális felhasználó, aktiválva van, vagy sem. Amennyiben nincsen, akkor egy `UserNotActivatedException` kivétel váltódik ki.

Amennyiben nem létezik a termék, amelyiken műveletet szeretnénk végrehajtani, akkor egy `Not Found (404)` állapotkódú `HTTP` választ kapunk. Ha minden a helyén van, akkor kezdődhet meg a vásárlás, licitálás szolgáltatása. A kép törlése itt valósul meg. Át szükséges adni a termék azonosítóját, illetve a kép azonosítóját.

Továbbá a képfeltöltés is található meg, amely a már megismert viselkedést valósítja meg.

3.4.6. Comment kontroller

A feltöltésénél annyival egészül ki, hogy van egy érvényesítő metódus, ami csupán annyit csinál, hogy ha a termék nem létezik, akkor kivétel váltódik ki. Törlés itt is szerepel, akárcsak a termék kontrolleren.

3.4.7. ProductFilter kontroller

A feltöltött termékekre valósíthatunk meg vele szűrést. A szűrés működésébe még melyebben bele fogok menni a 3.5. pontban. Ezen kontroller a csontváznak összesen három függvényét valósítja meg. Minden esetben egy `ProductFilter` DTO-t adok át, és a szerint tudom megkapni, a létező összes ilyen termék darabszámát, illetve az összes ilyen terméket lapozva.

3.4.8. Attribute, AttributeCore kontroller

Az attribútumok kezelését teszik lehetővé. Csak a csontvázban bemutatottokat valósítják meg.

3.5. Szolgáltatások

A szolgálatásokban megfogalmazott logika bemutatásaira szeretnék most bővebben kitérni.

3.5.1. Bázis szolgáltatás

Akárcsak az entitásoknál a bázis entitás, itt is gondoltam, hogy szükséges volna egy absztrakció, ami segítségével az azonos metódusokat és függvényeket megvalósíthatom egységesen.

Ám azt is észrevettem, hogy vannak olyan szolgáltatásaim, melyek nem használják ki teljes mértékben az ott felsoroltokat, ezért csak feleslegesen ott foglalják a helyet, így a bázis interfészt implementáltam egy absztrakt osztályból, így maguk a szolgáltatás implementációm ebből az absztrakt osztályból fognak örökölni, míg a bázis interfészt is megvalósítják.

Így már tényleg csak a szolgáltatásban valóban szükséges metódusok/függvények kerülnek tényleges implementálásra.

3.5.2. User szolgálatás

Ezen szolgáltatás valósítja meg főként a felhasználók regisztrációját, továbbá egyéb felhasználói műveleteket.

A csontvázban megfogalmazott műveletek implementálása a posztban a regisztrálásnak felel meg, míg a put a jelszó frissítésnek.

A regisztrációt megelőz egy ellenőrzés az átadott adatokra. Megvizsgálásra kerül a regisztrációhoz szükséges adatok létezése, majd ezt követően a felhasználónév és az e-mail adatbázisban való létezésének vizsgálata.

Végül az átadott szerepkörnek a létezése kerül megvizsgálásra.

Fontos itt a szerepkörnél kiemelni, hogy most megengedett bárki számára admin szintű szerepek regisztrálása is, ami nyilvánvalóan rossz, így ha ez termelési környezetbe kerülne ki, akkor ez a lehetőség tiltásra kerülne, tesztelési, fejlesztési célokra viszont megfelel ez a fajta kialakítás.

Ezt követően a két profil entitás (buyer, seller) hozzáadásra kerül sor, végül pedig egy aktivációs entitás létrehozásra.

Az utolsó művelet, ami végbemegy, az aktivációs e-mail küldése.

E-mail küldését egy utility-ként vettem fel.

```
@Override
public void sendSimpleMessage(String to,String subject,String text){
        MimeMessage message=emailSender.createMimeMessage();
        MimeMessageHelper helper=newMimeMessageHelper(message);

        try
        {
                helper.setTo(to);
                helper.setSubject(subject);
                helper.setText(text,true);
        }
        catch(MessagingException e)
        {
                throw new EmailSendingException(e.getLocalizedMessage());
        }

        emailSender.send(message);
}
```

Ezen metódust hívva tudok bármilyen szöveges e-mailt küldeni. Három paraméterre van csak szükség, egy `to` e-mail címre, jelen esetben ez a regisztrálandó fiók e-mail címe, egy tárgyra, ami itt a regisztráció, végül az üzenetet, amelyben a regisztráció tényét írjuk le és az aktivációs kóddal ellátott linket adjuk meg.

Az e-mail küldéséért a Spring Framework által szolgáltatott `JavaMailSender` interfészt felelős. Az `application.properties` fájlban használatához szükséges a következő paraméterek átadása:

`spring.mail.host=smtp.gmail.com`	SMTP szerver, jelen esetben Gmail
`spring.mail.port=587`	Port-ja
`spring.mail.username=felhasználónév`	Az SMTP szolgáltatónál regisztrált fiók felhasználó neve (nem kötelező, amennyiben a fiókhoz nem tartozik felhasználónév)
`spring.mail.password=jelszó`	Fiók jelszava
`spring.mail.properties.mail.smtp.starttls.enable=true`	TLS kapcsolódás bekapcsolása

`spring.mail.properties.mail.smtp.starttls.required=true`	TLS szükségesség
`spring.mail.properties.mail.smtp.auth=true`	Bejelentkezéses SMTP szolgálató

A jelszavakhoz `BCryptPasswordEncoder`-t használtam [7]. Ez egy, a Spring által ajánlott hash alapú titkosító algoritmus. A hash-elés azt jelentené, hogy a jelszót valamennyi iteráció múlva átalakítsuk egy olyan alfa-numerikus sorozattá, ami egy véletlen alfa-numerikus sorozatnak tűnik. Fontos kiemelni azt, hogy ugyanazon jelszó kétszeri hash-elése mindig ugyanazt a hash-elt jelszót adja.

Ha valakinek van egy nagy adatbázisa a gyakori jelszavakról, akkor azokról hash-t generálva valószínűleg megszerezné a mi adatbázisunkban felelhető jelszó hash-ek mindegyikét.

Így ha az adatbázisunkba illetéktelenek beférkőznek, akkor a felhasználóink jelszavát is megszerzik. A bcrypt azonban használ sót is a jelszavakhoz, ami azt jelenti, hogy nem csak a jelszót hash-eli, hanem a jelszót és a só sztringet együtt, ami már egy sokkal erősebb tárolható jelszót ad vissza. Bár még így is feltörhető volna brute force-al a hash-elt jelszavaink, ám a bcrypt nagy interációjának köszönhetően ez egy nem igazán járható út.

3.5.3. Buyer, Seller szolgáltatás

A bázis szolgáltatások közül nem kerül minden implementálásra, a bázisokon kívül pedig összesen egy, a kép mentése.

3.5.4. Product szolgáltatás

A termékek megvásárlása illetve a licitálás implementálásán kívül a kategóriák elérése, képek feltöltése, törlése is itt kerül megvalósítására.

Az osztály bemutatását a termékeken lévő kategóriák frissítésével kezdeném. A kategóriák hozzáadása úgy működik, hogy a kategóriák azonosítóját adom át a HTTP törzsben, és ha ezek között van olyan, amelyik nem szerepel a terméken akkor hozzáadom. A jelenlegi implementáció alatt nincsen abból hiba, ha olyan kategória azonosítót adok át, ami nem létezik. A törlésük úgy valósul meg, hogy az azonosítókat mínusz 1-el szorzom.

A vásárlást megvalósító metódus a következőképpen alakul:

```
@Override
public void buy(Integer id){
	Product entity=productRepository.findById(id);
	Buyer buyer=userService.getCurrentUser().getBuyer();

	if(entity.getType()!=ProductType.FixedPrice)
		throw new NotFixedPricedProductException(
				"The product is not fix priced");

	if(entity.getBuyer()!=null)
		throw new AlreadySoldException(
				"The product is already sold");

	if(!entity.getActive())
		throw new OverdueException(
				"The selling period has ended");

	entity.setBuyer(buyer);
	entity.setActive(false);
	productRepository.save(entity);

	buyer.addProduct(entity);

	buyerRepository.save(buyer);

	emailUtil.sendSimpleMessage(buyer.getUser().getEmail(),
	"You baught a product","You baught the product:
"+entity.getName());
}
```

Az azonosító alapján megszerezzük a terméket, majd az aktuális bejelentkezett felhasználó alapján a vásárló profil entitást. Amennyiben a termék nem fix áras, akkor kivétel eldobására kerül sor, illetve ha már nincsen hozzákapcsolt vevő, akkor szintén kivétel váltódik ki. A végső vizsgálat az aktivitásra vonatkozik, ahol is megnézzük, hogy nem-e járt-e még le a vásárlási időszak.

Végül a vásárlás tényéről egy e-mail kerül ki küldésre. A továbbfejlesztéshez megemlítendő, hogy itt is szükség volna egy a regisztrációhoz hasonló, a vásárlást tényét megerősítő e-mail ki küldésére, amiben arra kérjük a potenciális vásárlót, hogy igazolja egy kattintással, hogy valóban ő az, aki a vásárlást kezdeményezte.

A licitálás megvalósítása hasonló valamennyire a vásárláséhoz. Kialakítása a következőképpen működik:

```
@Override
public void bid(int entityId,int price){
        Product entity=productRepository.findById(entityId);
        Buyer buyer=userService.getCurrentUser().getBuyer();

        if(!entity.getActive()){
                throw new OverdueException(
                                "The selling period has ended");
        }

        if(entity.getType()!=ProductType.Bidding){
                throw new NotBiddingProductException(
                                "The product is not for bidding");
        }

        if(entity.getBiddings()!=null && !entity.getBiddings().isEmpty()
                        &&price<=entity.getBiddings()
                        .stream().mapToInt(x->x.getPrice())
                        .max().getAsInt()){
                throw new SmallerPriceException(
                                "The price must be higher");
        }

        Bid bid=new Bid();
        bid.setBuyer(buyer);
        bid.setPrice(price);
        bid.setProduct(entity);

        bidRepository.save(bid);

        entity.addBid(bid);
        productRepository.save(entity);
}
```

Átadásra kerül a termék azonosítója és egy ár. Itt is megvizsgálásra kerül, hogy tart-e még a vásárlási időszak, illetve, hogy a termék licitálós-e. Végül azt nézzük meg, hogy a licitet növelő érték az valóban növeli-e az aktuális értéket. Amennyiben nem, akkor arról szintén kivétel formájában tájékoztatjuk a klienst.

Ha minden rendben volt, akkor létrehozunk egy új bid entitást, beállítjuk neki a terméket, végül a terméknek a bid-et.

3.5.5. Adatbázis poll szolgálatás

Ez a szolgáltatás jelenleg csak egy szolgálatást hív meg, a termék deaktiváló szolgáltatást.

A feladata ennek a szolgáltatásnak az volna, hogy az adatbázist bizonyos időközönként meghívjuk valamely művelet végrehajtása érdekében. Nagyon fontos, hogy ezt ne a fő szálon tegyük meg, hiszen ez egy végtelen ciklus. A projekt indításakor ezt a szolgáltatást is el kell indítani. Az indítását már a felhasználó kontrollerben beharangozott `startPoll` fogja kezdeményezni, ami pusztán csak egyszer tud lefutni. Minden 5 másodpercben pedig bizonyos műveleteket fog végrehajtani az adatbázisunkon.

3.5.6. Termék deaktiváló szolgáltatás

Ez került tehát meghívásra az előző pontban érintett szolgáltatás által. A feladat az volna, hogy vizsgálni kell, hogy van-e olyan termék, amelyiknek lejárt az ideje, mert akkor őt deaktiválni kell. Ennek megoldásra első gondolatom az volt, hogy az adatbázisban létrehozok egy `event`-et (`trigger`), ami majd bizonyos időközönként lefut és a lejárt termékeken átbillenti az aktivációs flag-et hamisra. Ám hamar rájöttem, hogy ez az ötlet több sebből is vérzik.

Az ok pedig, amiért rossz ez a fajta implementáció, hogy így nem vagyok képes több nagyon fontos művelet végrehajtására. Többek között például e-mail küldés a licit végén a győztesnek.

Így aztán a szolgáltatás nem is csinál mást, mint átrohan a termékeken, megnézni, hogy az aktuális termék vég dátuma a mostani dátum előtt van-e, azaz már lejárt, és hogy aktív-e még, hiszen ha már nem aktív, akkor már korábban megtaláltuk, és deaktiváltuk. Ezen feltételek teljesülése mellett vizsgáljuk meg azt, hogy fix áras vagy licites-e. Amennyiben fix áras, akkor csupán átbillentjük az aktivitást jelző logikai értéket. Ám ha licitálás valósul meg, akkor megnézzük, hogy ki az, aki a legnagyobb licittel rendelkezik, hiszen akkor őt állítjuk be, mint vevőt a termékeknek, és neki küldünk e-mail-t a győzelem tényéről.

3.5.7. Komment szolgáltatás

A kommentelés mögötti logika megvalósítása. Csupán a bázis szolgálatásban szerepeltetett metódusokat, függvényeket valósítja meg.

3.5.8. Kép szolgáltatás

A bázis szolgálatások közül összesen csak kettőt valósít meg. Mivel nem kérünk vissza tömegesen képeket, csak egyenként, egy get by id alakú függvényen kívül másra nincs is szükség. A másik pedig, amit megvalósít, az az összes kép számának a lekérése.

3.5.9. Attribútum szolgáltatás

A metódus, amit kiemelnék itt, az a validálás, A többit azért nem, mert azok a bázis szolgáltatás általi metódusokat, függvényeket valósítják csupán meg.

Megvizsgáljuk, hogy a már meglévő attribútumlistán szereplő attribútumok valamelyikére szeretne-e értéket megadni-e vagy sem. Amennyiben nem létezik az az attribútumnév, akkor kivétel váltódik ki. Mivel az attribútumot egy termékhez kapcsoljuk, ezért fontos, hogy az azt használni hivatott termék létezzen.

Mint már az attribútum entitásnál kifejtettem, egy attribútum összesen három típus közül válogathat. Amennyiben az érték, amit átadni szeretnénk, nem felel meg annak a típusnak, amihez őt szeretnénk csatolni, akkor ott kivétel fog történni, mégpedig egy `NumericConversionException`.

3.5.10. Attribútum mag szolgáltatás

A bázis szolgáltatásban megfogalmazottakon kívül nem is hajt végre extra műveletet. Ezen szolgáltatás csupán, a post, put, get és delete műveleteket valósítja meg, természetesen megfelelően leképezve a DTO-król az értékeket az entitásokra, és vissza.

3.5.11. Termék szűrő szolgáltatás

Utoljára hagytam e szolgáltatás bemutatását, mert bár ez készült el utoljára, mégis inkább azért, mert ez az, ami talán a termékekre licitálásán, vásárlásán kívül a másik legfontosabb műveletet hatja végre, a szűrést.

A szűrést bár említettem korábban, úgy kellene valahogy elképzelni, hogy a kliens oldalon lesz egy kategória kiválasztó, amiben opcionálisan beklikkelgetem a megfelelő kategóriákat, majd opcionálisan megadok egy termék nevet (töredékszó), opcionálisan kiválaszthatok attribútumok közül olyanokat, amik érdekelnek, majd minden egyes kiválasztott attribútumhoz meg kell adni egy műveletet (például egyenlő, kisebb egyenlő) és egy értéket. A keresésre kattintva pedig azon termékek jelennének majd meg, amelyek teljesítik e feltételekéként.

Egy kis személyes megjegyzésem ehhez a szolgáltatáshoz az volna, hogy erre vagyok a legbüszkébb, mert minden, amit elképzeltem, az itt egyesül (attribútumok, kategóriák), felhasználásra kerül. Bár talán ez a legkomplexebb, ami megtalálható a szolgáltatásaim között, hiszen ez az, ami minden nagyobb szolgáltatás által adott értékeket használja, mégis ezt volt a legrövidebb elkészíteni, a könnyeden átlátható interfészes absztrakciónak köszönhetően.

Szerepel két numerikus érték összehasonlító függvény, amelyek igazat adnak, amennyiben a kiválasztott operáció, az átadott érték és a kiválasztott művelet a szűrőben átadott értékhez mérten teljesül.

Maga a szolgáltatás szíve a `getAll` függvény, mely először opcionálisan leszűri az összes terméket egy töredékszóra. Utána, ha vannak átadott kategóriák, akkor azok szerint szűr tovább.

Végül magukra az attribútumokra szűr. Működése a következőképpen alakul: Végigmegyek a szűrő magokon, majd azon belül a korábban már leszűrt termékeken. Az aktuális terméknek az attribútumain megyek bentebb tovább. Ezen attribútum magjának veszem a műveletét, majd a magban megfogalmazott típusnak megfelelően a szűrő által biztosított operációval megnézem, hogy az adott termék megfelel-e a biztosított követelményeknek.

Előfordulhat, hogy a termék valamely attribútumra vonatkozó feltételnek megfelelt, ezért őt elcsomagoltuk, mint jó terméket, de később belefuthatunk egy olyan attribútumába, ahol viszont elromlik, így ekkor őt ki kell venni ebből a listából. A legvégén leképezzük DTO listává a termék listát, és készen is vagyunk.

Mindezt megelőzi egy validálás a filter DTO-ra, amely megnézi, hogy az átadott kategóriák és attribútummagok léteznek-e.

A licitálás megvalósítása hasonló valamennyire a vásárláséhoz. Kialakítása a következőképpen működik:

```
@Override
public void bid(int entityId,int price){

        Product entity=productRepository.findById(entityId);
        Buyer buyer=userService.getCurrentUser().getBuyer();

        if(!entity.getActive()){
```

```
                throw new OverdueException(
                                "The selling period has ended");
        }

        if(entity.getType()!=ProductType.Bidding){
                throw new NotBiddingProductException(
                                "The product is not for bidding");
        }

        if(entity.getBiddings()!=null&&!entity.getBiddings().isEmpty()
                        &&price<=entity.getBiddings()
                        .stream().mapToInt(x->x.getPrice())
                        .max().getAsInt()){
                throw new SmallerPriceException(
                                "The price must be higher");
        }

        Bid bid=new Bid();
        bid.setBuyer(buyer);
        bid.setPrice(price);
        bid.setProduct(entity);

        bidRepository.save(bid);

        entity.addBid(bid);
        productRepository.save(entity);
}
```

Átadásra kerül a termék azonosítója és egy ár. Itt is megvizsgálásra kerül, hogy tart-e még a vásárlási időszak, illetve, hogy a termék licitálós-e. Végül azt nézzük meg, hogy a licitet növelő érték az valóban növeli-e az aktuális értéket. Amennyiben nem, akkor arról szintén kivétel formájában tájékoztatjuk a klienst.

Ha minden rendben volt, akkor létrehozunk egy új bid entitást, beállítjuk neki a terméket, végül a terméknek a bid-et.

Összegzés

Dolgozatomban bemutatásra kerültek először a felhasznált technológiák majd az elkészült projektem.

Minden részét a projektnek, amelyet fontosnak éreztem bemutatásra, megpróbáltam hiánytalanul prezentálni.

A projektem egy webáruház egy lehetőséges megvalósítását járja körül. A kigondolt áruházamban termékeket lehet felölteni, ezekere licitálni, vagy ha éppen fix árasak, akkor azokat megvásárolni. A termékekre lehet keresni egy szűrő szolgáltatás által.

Szűrni lehet termék névre, kategóriákra, vagy már bizonyos jellemzőkre, melyeket itt attribútumoknak neveztem el.

A fő ok, amiért ezt a projektet választottam, az volt, hogy egy nagyobb webalkalmazást alkothassak meg, és hogy megtanulhassam, hogy mik azok a fő komponensek, amik összerántásával egy kész, működőképes, esetleg termelés kész alkalmazást írhassunk. Nagyon élvezetesnek tartottam a projekt előkészületeket, a tervezés, fázisát.

Sok tapasztalatot gyűjtöttem az elkészítés során, és bár nem lett minden készen, amit tervbe vettem magnak, mégsincsen hiányérzetem, mert tudom, hogy a fontos képességeket már elsajátítottam, amikkel e hiányosságokat könnyedén ki tudnám küszöbölni.

A tanulást kiindulásként nagyon sok online kurzus megtekintése indította el, majd később, amikor már biztosabbnak éreztem tudásom, voltam képes hatékonyabban értelmezni a dokumentációkat.

Összességben a Spring Boot egy pozitív csalódás volt, amivel sikerült sok fontos fogalmat megértenem.

Szerencsémre a tutoriálok és a viszonylag könnyebben értelmezhető dokumentációkban kevésszer volt hiány.

A biztonság kérdése viszont nagyon nehézkes volt számomra, minthogy ahhoz való források elemzését nem tették könnyebbé a videó kurzusok, mert azok nem nagyon voltak fellelhetőek.

Köszönetnyilvánítás

Szeretnék köszönetet mondani témavezetőmnek Dr. Jeszenszky Péternek, amiért a tanított tárgyai által sikerült olyan szakmai alapot elsajátítanom amiért, annak idején a Debreceni Egyetem Informatika Karát választottam.

Továbbá szeretnék köszönetet mondani Fekete Gyulának, amiért cégüknél végezhetem el szakmai gyakorlatomat valamint, hogy a projektmunkák által tovább mélyíthettem tudásomat.

Végül pedig páromnak, Jenei Alexandrának, amiért türelmes volt velem, az egyetemen töltött éveim alatt, valamint a projektmunkák, illetve a szakdolgozat írása alatt is türelemmel és figyelemmel kísérte végig munkámat.

Irodalomjegyzék

[1] MvnRepository
URL: https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-test

[2] heartin. *Introduction and History of the Spring Framework*, 2015.
URL: *https://javajee.com/introduction-and-history-of-the-spring-framework*

[3] Sergiy Pylypets. *The Magic of Spring Data*, 2018.
URL*: https://dzone.com/articles/magic-of-spring-data*

[4] *Spring Data JPA - Reference Documentation*
URL: *https://docs.spring.io/spring-data/jpa/docs/current/reference/html/*

[5] *What is Object/Relational Mapping?*
URL: *http://hibernate.org/orm/what-is-an-orm/*

[6] *Hibernate - JPA Annotations*
URL: *http://www.techferry.com/articles/hibernate-jpa-annotations.html*

[7] Dan Arias, *Bcrypt*, 2018.
URL: *https://auth0.com/blog/hashing-in-action-understanding-bcrypt/*