

# **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 .....	3
2.	A felhasznált technológiák .....	4
2.1.	Spring Boot .....	4
2.1.1.	Történelme .....	4
2.1.2.	Kiindulás .....	5
2.1.3.	Függőségek befecskendezése .....	5
2.1.4.	Főbb szkópok .....	6
2.2.	Spring Data .....	8
2.2.1.	Annotáló konfiguráció .....	8
2.2.2.	Lekérdező metódusok .....	8
2.2.3.	Lekérdezés készítés .....	9
2.2.4.	@Query használata .....	10
2.3.	Hibernate .....	11
2.3.1.	ORM .....	11
2.3.2.	Annotációk .....	11
3.	A szoftver bemutatása .....	13
3.1.	A program főbb funkciói .....	13
3.2.	Entitások .....	15
3.2.1.	Bázis entitás .....	15
3.2.2.	User, Seller, Buyer entitások .....	16
3.3.	Adatbázis .....	22
3.3.1.	Diagram .....	22
3.3.2.	Leírás .....	22
3.4.	Kontrollerek .....	24
3.4.1.	Csontváz .....	24
3.4.2.	User kontroller .....	25
3.4.3.	Buyer, Seller kontroller .....	25
3.4.4.	Image kontroller .....	26
3.4.5.	Product kontroller .....	26
3.4.6.	Comment kontroller .....	28
3.4.7.	ProductFilter kontroller .....	29

3.4.8 Attribute, AttributeCore kontroller .....	29
3.5. Szolgáltatások.....	30
3.5.1. Bázis szolgáltatás.....	30
3.5.2. User szolgáltatás.....	30
3.5.3. Buyer, Seller szolgáltatás.....	32
3.5.4. Product szolgáltatás .....	32
3.5.5. Adatbázis poll szolgáltatás.....	35
3.5.6. Termék deaktiváló szolgáltatás .....	35
3.5.7. Komment szolgáltatás .....	36
2.5.4. Kép szolgáltatás.....	36
2.5.5. Attribútum szolgáltatás.....	36
2.5.6. Attribútum mag szolgáltatás .....	36
2.5.7. Termék szűrő szolgáltatás .....	36
Összegzés .....	39
Irodalomjegyzék .....	40
Köszönetnyilvánítás.....	41

## 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 Boot

Röviden szeretnék egy kis bemutatót írni az általam választott Framework-ről. A bemutatása során sokszor a [9] dokumentációt használom.

#### 2.1.1. Történelme

Az igény egy könnyebben használható Spring Framework-re 2012 októberében merült fel Mike Youngstrom által.

A könnyítésnek lehetőségét abban látta, hogy amennyire csak lehet, absztrahálni kell a Spring keretrendszert. Szerinte a servlet konténer használata az új fejlesztők számára egy olyan tanulási görbét von maga után, amit nem biztos, hogy szükséges volna rájuk helyezni.

A tanulási görbe alatt értendő pl. a `web.xml` és egyéb servlet konfiguráció, `war` mappa szerkezet, konténer implementációk (pl. portok, szálkészlet, stb.), komplex osztálybetöltő hierarchia stb.

A felsoroltak nem rendelkeznek egy egységes konfigurációval, helyette egy nem egységes, inkonzisztens módon adhatók meg.

Szerinte egyszerűsíthető volna a Spring keretrendszer, ha egybeágyaznák és egységesítenék a gyakori web konténereket egy közös Spring konténerben.

Bővebben a [1] github issue-ban fejti ki ötleteit.

A történelméről bővebben pedig a [4] cikkben olvashatunk.

### 2.1.2. Kiindulás

A projekt alapját előállíthatjuk a *Spring Initializer* segítségével. A projektben Maven-t használtam, így itt is azt fogom körül járni.

A Spring Boot, a Spring által elérhető nagyobb funkcionalitásokat néhány függőségbe gyűjtötte. Ezek hozzáadását követően sok dolgunk már nem akad.

Az Initializer által előállított projektben létrejön az `src/main/java/package/` mappában a `main` metódus, amin `@SpringBootApplication` annotáció szerepel.

Ez tulajdonképpen három nagyobb annotációt olvaszt magába:

<code>@Configuration</code>	Az osztályon, amelyiken alkalmazzák, ott ez az annotáció ezt az osztályt a Spring Java alapú konfigurációs osztállyá nevezi ki.
<code>@ComponentScan</code>	Komponens keresést indít el, így bármelyik web controller vagy bármelyik másik komponens megtalálásra kerül.
<code>@EnableAutoConfiguration</code>	Itt a Spring Boot egyik „varázslata”, ez az automata konfigurációt teszi lehetővé. Segítségével nem kell több oldalnyi konfigurációs kódot írni.

### 2.1.3. Függőségek befecskendezése

`@Autowired`: A függőségek befecskendezéséhez használjuk. Azokat az osztályokat, amiket ellátunk olyan annotációkkal, amelyeket a `@ComponentScan` megtalálhat, azokat itt befecskendezhetjük.

Maga a fecskendezés három módon történhet meg:

- mezőn
- konstruktoron
- setter metóduson.

Példa:

```
public class Osztalyom {  
    //1. Lehetőség  
    @Autowired  
    private Komponentens komponens;  
  
    //2. Lehetőség  
    @Autowired  
    public Osztalyom (Komponentens komponens){  
        this.komponens = komponens;  
    }  
  
    //3. Lehetőség  
    @Autowired  
    public void setKomponens (Komponentens komponens){  
        this.komponens = komponens;  
    }  
}
```

#### 2.1.4. Főbb szkópok

Bab szkópok segítségével adhatjuk meg az osztályok példányosításának módját. A Spring IOC konténert használhatjuk a bab szkópok általi példányosításához. Több fajta szkóp létezik: `singleton`, `prototype`, `request`, `session` és `global session`.

Bővebb leírással a [6] dokumentáció tud szolgálni.

##### 2.1.4.1. Egyke

Amikor egy bab egyke, akkor csupán egy példánya fog használatban lenni. Bármikor, amikor kérés irányul a Spring IOC konténer felé egy egyke bab eléréséhez, akkor mindig ugyanaz a példány kerül átadásra. Ezért aztán ha módosításra kerül egy egyke bab, akkor bármilyen új hivatkozás erre a babra az új változtatásokkal ellátott babot kapja, hiszen csupán egy példánya létezik.



Példa:

```
@Scope("singleton")
public class SingletonClass {

    private SingletonClass singletonClass;

    @Bean
    public SingletonClass getSingletonClass(){
        return singletonClass;
    }
}
```

#### 2.1.4.2. Prototípus

A prototípus szkóppal ellátott babok minden új kérés esetén új példányt hoznak létre. Egyke szkópot állapot nélküli babokra teszünk, míg az állapottal rendelkezőkre prototípust teszünk. Úgy használhatjuk, hogy a `@Scope` annotációnak a „prototype” karakterláncot adjuk át.

#### 2.1.4.3. Kérés

A kérés szkóp új babot csinál minden egyes HTTP kérésnél egy munkameneten belül. Még akkor is, ha 2 lekérés érkezik egy munkameneten belül szimultán módon. Minden kérés egy eltérő memória címet hivatkozik. Minden kérés függetlenül kezelt, és nincsenek egymásra hatással.

#### 2.1.4.4. Munkamenet

Ebben az esetben egy HTTP kérésen belül 2 ugyanolyan típusú babra való hivatkozás az egykében megismert viselkedést fogja tanúsítani, azaz nem fog új helyet lefoglalni az új hivatkozásra a babnak a kérésen belül, hanem a már meglévőt fogja használni.

## 2.2. Spring Data

Az egyik legelőrehaladottabb megoldást az adatfeldolgozásra a Spring Data Project biztosítja. Képes automatikusan generálni tárolókat az általunk átadott funkcionális interfészből. Bővebben a [2] cikkben olvashatunk róla.

Az absztrakció alapja a **Repository** interfész. Ebből származnak le olyan interfészek, mint pl. **JpaRepository**, **CrudRepository**, **MongoRepository**.

A bemutatásában sokszor fogok a dokumentációhoz [3] visszatérni.

### 2.2.1. Annotáló konfiguráció

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

Például:

```
@Autowired
private SzemelyRepository szemelyRepository;

//Illetve:
@Autowired
public Osztalyom (SzemelyRepository szemelyRepository){
    this.szemelyRepository = szemelyRepository;
}
```

### 2.2.2. Lekérdező metódusok

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

Egy példa egy ház interfész létrehozására:

```
interface HouseRepository extends Repository <House, Long> { ... }
```

A generikus paraméterei egy ház entitás, míg a másik a ház entitás egyedi azonosítójának a típusa.

Ezt követően egy lekérdező műveletet a következőképpen készíthetünk el:

```
interface HouseRepository extends Repository <House, Long> {  
    List <House> findByColor(String color);  
}
```

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

Például. életkorra a következő képpen nézne ki:

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

A leszűrt elemekre lehet használni a `distinct` műveletet is:

```
List <House> findDistinctHouseByColor(String color);
```

Rendezni is lehet a következőképpen:

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

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

Előfordulhat, hogy 1-nél több propertyre szeretnénk szűrni, ilyenkor egy összekapcsoló logikai névre van szükség: `Or/And`.

Például:

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

### 2.2.3. Lekérdezés készítés

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

A felépítése egy ilyen lekérdezésnek a következő módon alakul:  
{visszatérési típus} findBy {Entitás valamely propertyje}[and | or  
[{entitás valamely propertyje}[and | or]..] ]([ [típus  
property\_név],[típus property\_név]..]);

Fontos, hogy a paraméter lista elemszáma meg kell, egyezzen a lekérdezésben átadott tulajdonság nevek 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	<code>findByAgeGreaterThan</code>
Before	<code>findByStartDateBefore</code>
Containing	<code>findByFirstnameContaining</code>
Not	<code>findByLastnameNot</code>
NotIn	<code>findByAgeNotIn (Collection &lt;Age&gt; ages)</code>
True	<code>findByActiveTrue ()</code>

#### 2.2.4. @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 h from House h where h.color = ?1")
House findByColor(String color);
```

Ebben a példában a sztringben megjelenő „h” egy a House entitás egy példányát képviseli. Az „h” segítségével tudunk hivatkozni a benne található tulajdonságok egyikére.

Alkalmazható a Like kifejezés is a következő módon:

```
@Query("select h from House h where h.address like ?1")
List<House> findByAddressEndsWith(String address);
```

A % karaktert a Spring Data kiszedve, 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 h from House h where h.color = :color or h.age = :age")
User findByColorOrAge(@Param("color") String color, @Param("age")
Integer age);
```

## 2.3. Hibernate

A Hibernate [5] a JPA-nak az egyik legnépszerűbb megvalósítása. A Hibernate egy objektum-relációs le képezést megvalósító programkönyvtár.

### 2.3.1. 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 [7] 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. Támogatja az egyedi érték típusok le képezését is. Továbbá lehetőség van a következőkre:

- Felülírni az alapértelmezett SQL típust, amikor egy oszlopot képezünk le egy tulajdonságra.
- Leképezni Java Enumokat akárcsak a szokásos típusokat

### 2.3.2. Annotációk

A `javax.persistence` csomagot használva érhetjük el őket.

Az annotációk [8] segítségével tudjuk konfigurálni az entitásokat, és a kapcsolatot közöttük. Helyette használható XML konfiguráció.

Néhány annotáció:

<code>@Entity</code>	Az entitás babjainkat ezzel az annotációval kell, hogy elássuk.
<code>@Table (name=táblanév)</code>	Az átadott táblanév alapján köti az adatbázisban megtalálható táblát ezzel az entitással.
<code>@Id</code>	Jelezzük, hogy a property az egy egyedi azonosító.
<code>@Column (name=oszlopnév)</code>	Az átadott oszlopnév alapján köti az adatbázisban megtalálható tábla oszlopát ezen entitás proprietijén.
<code>@Lob</code>	Nagyméretű objektumok használata esetén kell megadni.
<code>@OneToOne</code>	Egy az egyhez kapcsolathoz használandó.
<code>@OneToMany</code>	Egy az többhöz kapcsolathoz használandó.
<code>@ManyToOne</code>	Több az egyhez kapcsolathoz használandó.
<code>@ManyToMany</code>	Több az 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) úgy, mint

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

A felhasználó élmény megteremtése érdekében szükséges lehetőséget teremteni egy olyan kapcsolati formára a potenciálisvevő 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 keresés működéséhez elengedhetetlenül szükséges a már említett kategóriák, illetve attribútumok. A kategóriák jelentenék a termék hova való besorolhatóságát, úgy, mint pl. 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 értem a termék komolyabb leírását. Attribútum alatt általában egy név-érték párt gondolunk. Itt is valamennyire hasonló a kettő, 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 rendszermemória és így tovább. Ez mind olyan leírása a terméknek, ami alapján tudunk szűrni ilyen tulajdonságokra, pl. ha a 3GB, vagy annál nagyobb rendszermemóriával rendelkező, zöld színű telefonokat akarunk megkapni akkor azt az implementált szűrőmmel megtehetjük.

Komolyabb használatára még a későbbiekben visszatérek.

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, úgy, mint a termékek feltöltése, licitálás, valamint a fix áras megvásárlás. Regisztrált felhasználó amennyiben elfelejtette jelszavát, könnyen kaphat újat.

A bejelentkezett felhasználó kap egy access token-t, amivel őt egyértelműen azonosítjuk, és bármilyen olyan funkció, ami belépést igényel, a token alapján megkapott felhasználó nevében tudja használni a web shop-ot.

A szoftver csupán kettő nagyobb szerepkört különböztet meg, az USER-t illetőleg, az ADMIN-t. Funkciós korlátozások ezek szerint természetesen szükségesek, úgy, mint a titkos felhasználói információk elrejtése, a kategóriák, és az attribútumok feltölthetősége csupán az ADMIN által.



## 3.2. Entitások

### 3.2.1. Bázis entitás

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

Tartalmazza az `Id`-t, amit el-láttam 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.

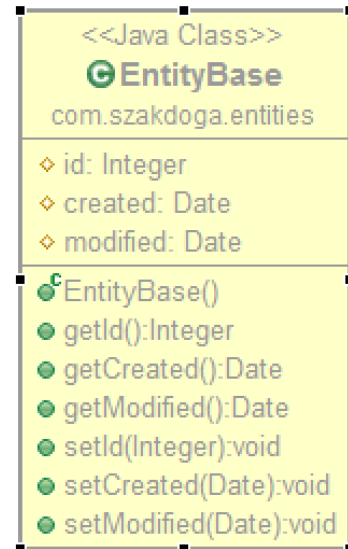
Az adatbázisnak a MySQL-t választottam, amiben van „auto increment”, így nagyon kényelmessé teszi az egyedi azonosítók automata generálását. A projekt elején az adatbázis kiválasztásánál az Oracle Database is szóba jött.

Felvettem két további fontos mezőt, a létrehozás dátumát illetve a módosítás dátumát.

Ezen kettőre a JSON ignorációt is felvettem, azért ha `Repository`-n keresztül kérnék le entitásokat, akkor a létrehozási, illetőleg 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.persistence` csomag által szolgáltatott `@MappedSuperClass` annotációt vettem igénybe.

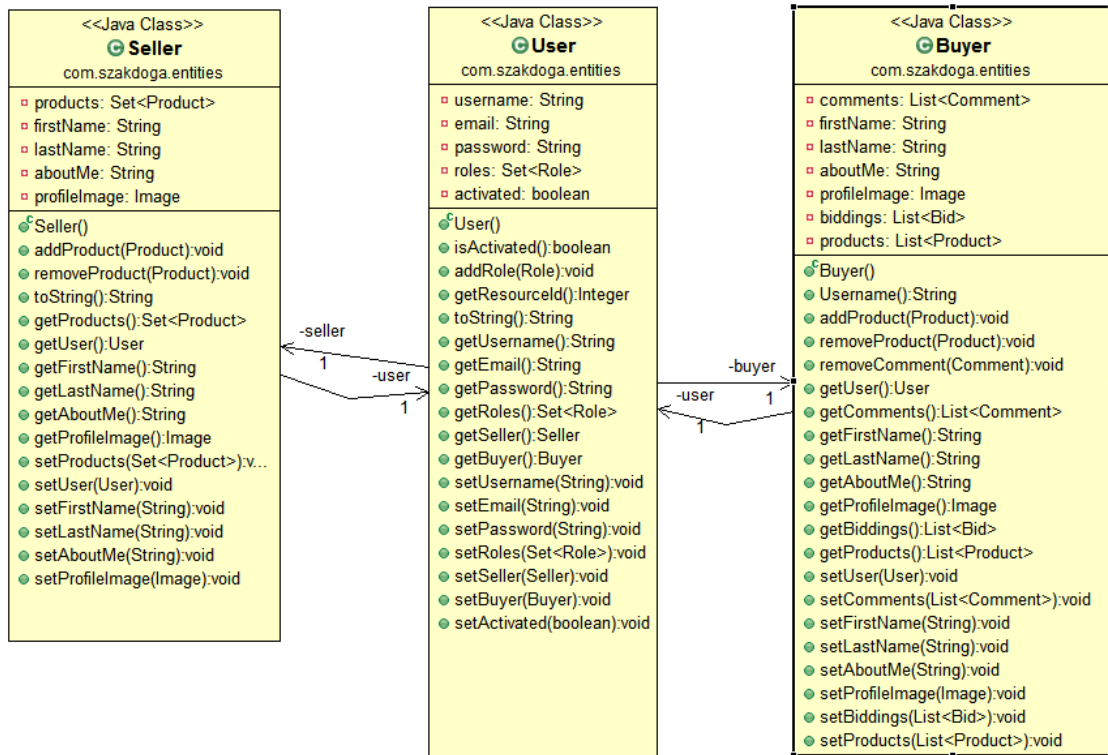
Beemeletem a projektembe a lombok nevű függőséget, ami nagyban megkönnyítette a munkámat, hiszen nem szükséges általa a getterek-setterek állandó legeneráltatása, így ezen boilerplate kódtól tisztábbak lettek az entitásaim, illetve a DTO-im.



1. ábra A Bázis entitás UML osztály diagramja

### 3.2.2. User, Seller, Buyer entitások

A bázis entitás után a második legnagyobb osztály, ami vagy közvetlen, vagy közvetett módon, de kapcsolatban áll ezen entitásokkal, főleg, hogy a **Seller** és a **Buyer** szülője a **User**.



2. ábra User, Seller, Buyer entitások

#### 3.2.2.1. User entitás

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

Előforduló asszociációs kapcsolatok:

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

Minthogy ez már olyan entitás, amint szeretnénk perzisztálni 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. Benne átadható `name` értéknek 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.

Rendelkezik továbbá egy aktiváltságot 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 az entitás ami az aktivációs karakterláncot tartalmazza az a `UserActivation` entitás. Rendelkezik továbbá még a lejárat 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 kapcsolatok:

<code>@OneToMany</code>	Buyer entitással
<code>@OneToOne</code>	User, Image entitásokkal

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

Az `aboutMe` mezőn egy `@Lob` annotációszerepel. 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 kapcsolatok:

@OneToMany	Product, Bid, Comment entitásokkal
@OneToOne	User, 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 megfogalmazására. 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 rá keresni, ha nem volna egy pár egyedi attribútum típus.

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

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

@ManyToOne	Product, 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	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 kapcsolatok:

@ManyToOne	Product, 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 idő intervallumot 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, akkor, ha 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.

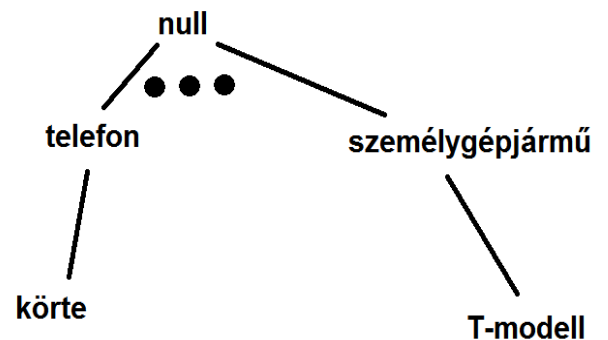
A munka elején, úgy képzeletem, hogy egy belső levelezéssel oldom meg a kommunikáció lehetőségét a vevők és az eladók között, ám rájöttem, hogy ezzel azt a potenciált veszítem el, hogy a vásárlók egy még átfogóbb képet kapjanak az adott termékről. Ezenfelül, ami még nagyon fontos, hogy az eladót is lehessen kommentálni, hiszen az ő munkássága az oldal presztízst javíthatja, de ugyan ilyen arányban rombolhatja is. Sajnálatos módon ez már nem készült el, de amennyiben ez a projekt valós környezetbe kerülne ki, egészen biztos, hogy az megvalósításra kerülne.

Előforduló asszociációs kapcsolatok:

@OneToMany	Comment, Attribute, Image entitásokkal
@ManyToOne	Buyer, entitással
@ManyToMany	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 fa-szerűen képzelhetők el. Ezt a faszerű kialakítást úgy gondoltam, hogy kiindulásként van egy nullértékű szülő kategória. Belőle származnak le az alkategóriák, minden alkategória egyes darabja egy újabb szülő kategória lehet, de nyilván már nullértékű nem.



3. ábra kategória entitás kép magyarázat

Előforduló asszociációs kapcsolatok:

@ManyToMany	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.

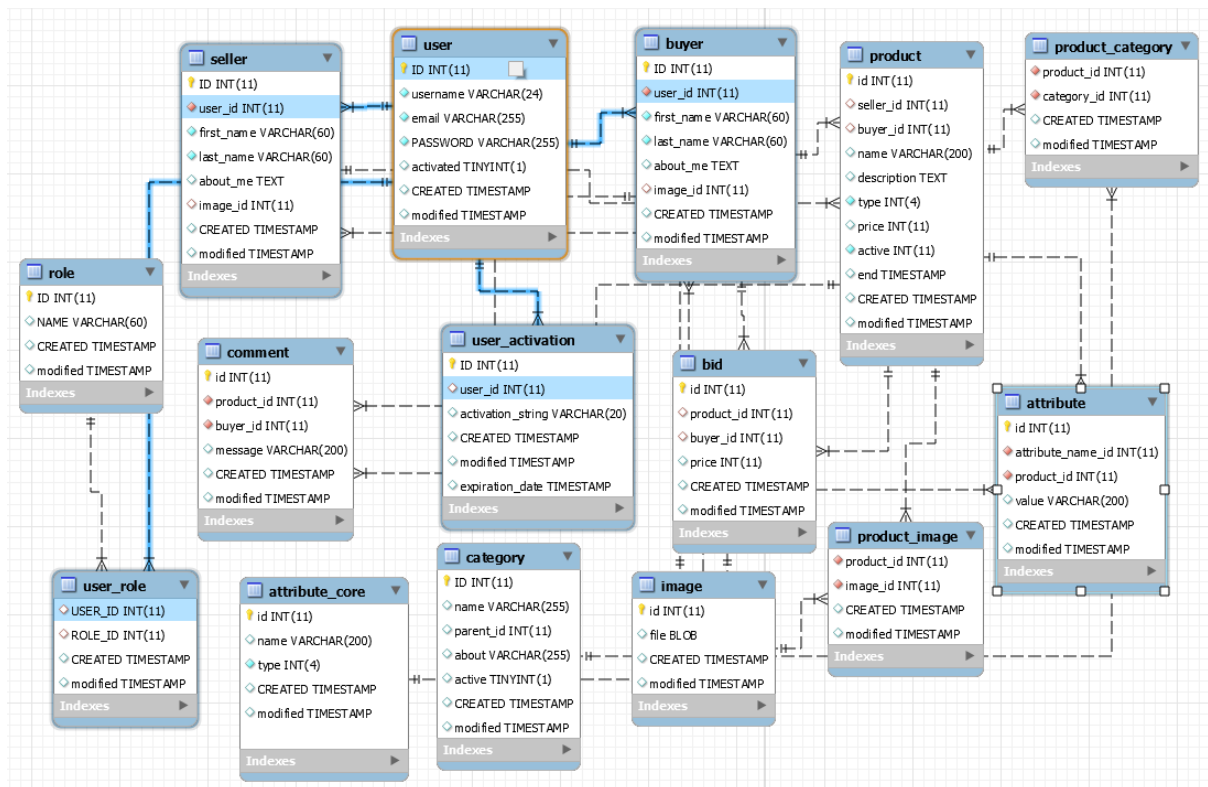
### **3.2.2.10. AccessTokenEntity, RefreshTokenEntity entitások**

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



#### 3.3.2. 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. Kevesebbnel nehéz volna kiszorgálni az alkalmazást, többel pedig redundancia léphet fel. Kiindulásnál ugyanis úgy alkottam meg a képek perzisztálását, hogy a profil táblákon volt egy blob tárolásra alkalmas oszlop, míg a termékek pedig már a kép entitással álltak kapcsolatban.

Ám kis átalakítás hatására mindenki a kép táblát használta, ez pedig azért is jó, mert így sokkal könnyedebb táblákat hozhattam létre. Maga 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 keríték sort. Többnyire azonos kialakítást képviselnek, a felesleges komplexitás elkerülése érdekében.

A kontrollerek implementálják a REST API-t, és többnyire JSON-t adnak vissza. Azon egyszerűsítés érdekében, hogy a controller 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 controllerjeim nagyrésze ugyanazon függvényeket/metódusokat tartalmazza így törekedtem arra, hogy azok ne térjenek el egymástól, így az egységességnek köszönhetően esetleges javítása, tovább fejlesztése sokkalta egyszerűbb.

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

Ezen csontváz a következőképpen fest:

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

<code>@GetMapping („/{id}”) public List &lt;DTO&gt;get (@PathVariable („id”) id, HttpServletResponse)</code>	Egyedi azonosító alapján való lekérés.
<code>@GetMapping („/all”) public List &lt;DTO&gt; get ()</code>	Összes létező entitás (DTO-vá képezve) lekérése.
<code>@GetMapping („/size”) public List &lt;DTO&gt;size ()</code>	Az összes létezőnek a darabszáma. Szükség lehet rá ha lapozva szeretnénk lekérni a már létezőket.
<code>@GetMapping („/all/{page}/{size}”) public List &lt;DTO&gt; getAll (@PathVariable („id”) page, @PathVariable („id”) size)</code>	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 műveletkeihez 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.

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

### 3.4.3. Buyer, Seller kontroller

Az egységes kialakítás végett mindkettő pusztán egy plusz metódust valósít meg, a profil kép feltöltésére vonatkozót. A feltöltött profil kép elérésére az **Image** kontrollert használok, 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 profil kép. 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 kép feltöltő metódusnál szükség volt a `GetMapping`-ben felvenni egy a `consumes` paramétert, hogy megmondjam, hogy milyen MIME média típusokat fogadok el a kienstől. Jelen esetben a `multipart/form-data`-át engedek meg. Ekkor a kép feltöltésekor meg kell adni a fájlnek egy nevet is, amit én a feltöltő metódusomban „image”-nek várok. A feltöltött képet pedig `MultipartFile`-ként kezelem a továbbiakban, amiből könnyen kinyerhető azon bájttömb, amit tudok majd tárolni a feltöltéskor.

#### 3.4.4. Image kontroller

Az előző pontban kifejtett okok miatt itt nem szerepel a felöltés, csupán az elérés. Továbbá a képek törlése is hasonlóan oda korlátozódik, ahol azokat felöltjük 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 lettek elhelyezve, ám ha jobban megnézzük, akkor látjuk, hogy bár az elérése valóban ki bővül a „profile”-al, ám ugyanazt a funkcionálisát valósítja meg, mint a sima kép elérés. Amennyiben nincsen profil kép feltöltve, akkor egy alapértelmezett kép kerül átadásra.

A függvények bájttömbként adják vissza a képeket, ám 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, 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. `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 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, így ha felveszünk hozzá egy flag-et amivel jelezzük, hogy ő kitiltott felhasználó, akkor minden helyen ahol vele kapcsolatba kerülhetünk, pl. ha egy licitálásban részt vett, akkor ott ezt figyelniünk kell, de így kapunk egy megoldást erre a problematikára. 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.

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

    userService.checkIfActivated(user);

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

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

```

@PostMapping("/buy/{id}")
public void buy(
    @PathVariable("id") Integer id,
    HttpServletResponse response)
{
    User user = 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 valósít 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-e, vagy sem. Amennyiben nincsen, akkor egy `UserNotActivatedException` kivétel váltódik ki.

Amennyiben a termék nem létezik, amelyiken szeretnénk műveletet 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 kép feltöltés is található meg, amely a már megismert viselkedést valósítja meg.

### 3.4.6. Comment kontroller

A feltöltésénél annyival egészül ki, hogy van egy érvényesítési 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 következő nagy 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 termékdarabszámát, összes a termékeket, illetve az összes ilyen terméket lapozva.

### **3.4.8 Attribute, AttributeCore kontroller**

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

### 3.5. Szolgáltatások

A szolgáltatásokban 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 implementációim 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 szolgáltatás

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 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 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 = new MimeMessageHelper(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:

<code>spring.mail.host=smtp.gmail.com</code>	SMTP szerver, jelen esetben Gmail
<code>spring.mail.port=587</code>	Port-ja
<code>spring.mail.username=felhasználónév</code>	Az SMTP szolgáltatónál regisztrált fiók felhasználó neve (nem kötelező amennyiben a fiókhoz nem tartozik felhasználónév)
<code>spring.mail.password=jelszó</code>	Fiók jelszava

<code>spring.mail.properties.mail.smtp.starttls.enable=true</code>	TLS kapcsolódás bekapcsolása
<code>spring.mail.properties.mail.smtp.starttls.required=true</code>	TLS szükségesség
<code>spring.mail.properties.mail.smtp.auth=true</code>	Bejelentkezési SMTP szolgáltató

A jelszavakhoz `BCryptPasswordEncoder`-t használtam. 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. Ám a [10] előnye a többi ismertebb hash algoritmussal szemben (MD5, SHA1), hogy lassú.

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

Így ha az adatbázisunkba illetéktelenek beférkőznek, akkor a felhasználóink jelszavát is megszerzik. A `bcrypt` azonban használ só-t 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 iterációjának 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 kezdeném a termékeken lévő kategóriák frissítésével. 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. Jelenlegi implementáció alatt nincsen abból hiba, ha olyan kategória azonosítót adok át, ami nem létezik. A törlésük úgy valósul meg, hogy az azonosítókat szorzom mínusz 1-el.

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.sendMessage(buyer.getUser().getEmail(),
        "You bought a product", "You bought the product: "+
entity.getName());
}
```

Az azonosító alapján megszerezünk 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 nem már van 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

mege erősítő 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 licitálás-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 szolgáltatás

Ez a szolgáltatás jelenleg csak egy szolgáltatást 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 másodpercen 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 gondoltam 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 pl. 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 átbillentyük 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 szolgáltatásban szerepeltetett metódusokat, függvényeket valósítja meg.

### **2.5.4. Kép szolgáltatás**

A bázis szolgáltatások 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.

### **2.5.5. 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 attribútum listán 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 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`.

### **2.5.6. 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 át mappolva a DTO-król az értékeket az entitásokra, és vissza.

### **2.5.7. 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 be-klikkelgetem a megfelelő kategóriákat, majd opcionálisan megadok egy termék nevet (töredékszó), opcionálisan kiválaszthatok attribútumok közül azokat, amik érdekelnek, majd minden egyes kiválasztott attribútumhoz meg kell adni egy műveletet (pl. 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, ami minden nagyobb szolgáltatás által adott értékeket használja, mégis ezt volt a legrövidebb elkészíteni, a könnyedén 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 az 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 megfelelet, 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 az termék listát, és készen is vagyunk.

Mindezt megelőz 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 került először a felhasznált technológiák majd az elkészült projekt.

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

A projekt egy web áruház egy lehetséges megvalósítását járja körül. A kigondolt áruházamban termékeket lehet felölteni, ezekre licitálni, vagy ha éppen fix áras, akkor azt megvásárolni. A termékekre lehet keresni egy szűrő szolgáltatás által.

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

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

Sok tapasztalatot gyűjtöttem az elkészítés során, és bár nem lett minden készen, amit tervbe vettem magnak, még sincsen 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 koncepciót megértenem.

Szerencsémre a tutorialok 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.

## Irodalomjegyzék

- [1] Mike Youngstrom, *Improved support for 'containerless' web application architectures*, 2012 október 17.  
<https://github.com/spring-projects/spring-framework/issues/14521>
- [2] Sergiy Pylypets, *The Magic of Spring Data*, 2018 október 19.  
<https://dzone.com/articles/magic-of-spring-data>
- [3] *Spring Data JPA - Reference Documentation*  
<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/>
- [4] *History of Spring Framework and Spring Boot*  
<https://www.quickprogrammingtips.com/spring-boot/history-of-spring-framework-and-spring-boot.html>  
Bean scopes
- [6] <https://docs.spring.io/spring/docs/3.0.0.M3/reference/html/ch04s04.html>
- [7] *What is Object/Relational Mapping?*  
<http://hibernate.org/orm/what-is-an-orm/>
- [8] *Hibernate - JPA Annotations*  
<http://www.techferry.com/articles/hibernate-jpa-annotations.html>
- [9] *Spring Framework Reference Documentation*  
<https://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/>
- [10] Dan Arias, *Bcrypt*, 2018 május 31.  
<https://auth0.com/blog/hashing-in-action-understanding-bcrypt/>

## **Köszönetnyilvánítás**

Szeretnék köszönetet mondani témavezetőmnek Dr. Jeszenszky Péternek, amiért a tanított tárgyai által sikerült olyan szakmai alapot elsajátítanom amért, annak idején a debreceni egyetem informatika karát választottam.

Továbbá szeretnék köszönetet mondani Fekete Gyulának, amiért cégüknél végezhettem 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.