Szakdolgozat

Bartha Zoltán

Debrecen

2022. április 10.

Debreceni Egxetem
Informatikai Kar

Rendeléskezelő applikáció Java SE-nel

Tiba Attila
tanársegéd

Bartha Zoltán
programterevző informatikus

Köszönetnyilvánítás

Köszönetnyilvánítás

Tartalomjegyzék

1 Bevezetés

A web egy gyorsan és dinamikusan fejlődő platform. A hagyományos asztali alkalmazások és natív mobilalkalmazások helyét folyamatosan veszik át a webapplikációk, illetve évről évre több teret nyernek a progresszív webalkalmazások. Számos önkéntes szabvány biztosítja rendeltetés szerű működését, melyeket például a W3C vagy az IETF felügyel, fejleszt. Emellett teret nyertek a de facto szabványok is, mint a JSON.

Az utóbbi tíz év alatt rohamos fejlődésen esett át a web. Ezen évtized alatt megnövekedett az igény weblapok, -alkalmazások és -szolgáltatások fejlesztésére, és ezzel együtt bővült a fejlesztők számára elérhető eszközök kínálata. Mint frontend, mint backend oldalon számos kiforrott, iparban használt és aktív közösséggel rendelkező megoldást találhatunk igényeink kielégítésére, és az opciók folyamatosan bővülnek.

A web nemcsak fejlesztői szempontból fejlődik. A végfelhasználók biztonságának és felhasználói élményének javítása egy konstans napirendi pont mind a szabványokat létrehozó testületek, mind a webfejlesztői közösség vezetői körében. Előbbire példa, hogy az új protokollokat, mint a QUIC, WebPush vagy WebRTC, már eleve a biztonságot és titkosítást szemelőtt tartva lettek tervezve, míg utóbbira példa lehet a Mozilla csoport törekvése a web biztonságosabbá és elérhetőbbé tételére [12]

A webes megoldásokra való átállással azonban új problémákba ütközhetünk. Az egyik ilyen a frontend és a backend közti kommunikáció. Amennyiben a backend oldalon megváltozik egy JSON séma, a frontend oldalán ezt a változást követnünk kell, másképp az alkalmazásunk nem fog rendeltetésszerűen működni. Erről a lépésről azonban hajlamosak lehetünk megfeledkezni, vagy nagyobb szabású projectek esetében megeshet, hogy nem is értesülünk róla.

Mivel lesz frontend és backend kódbázisunk, növekedni fog az alkalmazásunk mentális modelljének és valós implementációjának a komplexitása. Számos olyan forráskód szintű entitás jön létre egyszerre a két kódbázisban, melyek logikailag össze vannak kötve, vagy akár azonosak, mégsem tudjuk egyszerre megváltoztatni a kettőt. A frontend és a backend szoftver kód

szinten el van szeparálva, logikai szinten azonban függenek egymástól, és az inkonzisztenciákat nincs módunk észrevenni időben.

Ezekre a problémákra egy lehetséges megoldás az, hogy egyetlen, egységes kódbázisban létezik a frontend és a backend, így a frontend és a backend ugyanazokat szoftveres entitásokat használja, ami eddig két, elszeparált kódbázisban létezett.

A Java, illetve a JVM platform kiváló alapot nyújt stabil, ipari minőségű megoldások fejlesztésére. A Java nyelv fejlődése konzervatívnak és lassúnak tűnhet olyan ökoszisztémákhoz képest, mint a C# és a CLR platform.

A JVM ökoszisztéma azonban egy kiforrott alap, számos szabad és nyílt forrású megoldással, mint például a Spring, a Google Guava, az Apache Commons, és ezen megoldások választéka folyamatosan bővül. Számos kutatási területben élen jár, gondolhatunk itt a garbage collection-nel kapcsolatos technológiákra, JIT fordításra, optimalizációra, natív kód kezelésre.

A JVM nemcsak a Java nyelvnek ad otthont, hanem nyelvek egész családjának. Ilyen például a Kotlin, ami a mobil- és webfejlesztésben nyert szerepet az elmúlt években, vagy a Scala, ami főleg backend oldalon szerzett népszerűséget. Az említett nyelvek gyorsan fejlődnek, számos új nyelvi funkcióval bővülnek, és ezzel egy időben profitálnak a célplatform fejlődéséből. Emellett jó és kiforrott az interoperálási lehetőség Java kóddal, így a már meglévő Java könyvtárakat ezekben a nyelvekben is tudjuk használni.

Kiváló toolchain-nel rendelkezik, mint például az Apache Maven, a Gradle vagy az JetBrains IntelliJ. Ezek az eszközök ipari erősségű szoftverek, amelyek könnyen elérhetőek a fejlesztők számára, és segítik a fejlesztési folyamatot.

Azért választottam ezt a témát, mert a web egy globálisan elérhető, innovatív platform, ami lehetőséget nyújt érdekes, új technológiák kipróbálására, mint például a Java Spring Boot és Vaadin Flow keretrendszerek, amelyek ötvözésével áthidalható az fentebb taglalt "több kódbázis" probléma. A Java nyelv statikus, erősen típusos típusrendszere egy szigorú, de sok hibára figyelmeztető környezetet nyújt. Egy nagy kifejezőerővel rendelkező, modern, objektum orientált programozási nyelv, ami számos funkciót biztosít szoftvertervezési minták és komplex architektúrák felépítésére.

A szakdolgozat részeként egy rendelésfelvevő és -kezelő webalkalmazás jött létre éttermek részére, egy Java alapú megoldás formájában.

Az alkalmazás lefejlesztése módot ad a Java nyelvi elemeinek, szoftverfejlesztésben való

felhasználhatóságának megvizsgálására, szoftveres minták implementálására, illetve egy egységes, Java kódbázis létrehozására.

2 Tárgyalás

2.1. A program struktúrája, bemutatása

2.2. Tervezési minták

2.2.1. Factory

A *single responsibility* elvéből kiindulva, minden osztálynak csak és kizátólag egy felelősséggel kel rendelkeznie [10]. Ezt figyelmebe véve el kell választanunk az objektumok példányosítását maguktól az objektumoktól. Ebben segít a *Factory* minta.

A *Factory* mintának több variánsa van. Létezik olyan verziója, amiben a létrehozandó osztály egy metódusán keresztül hozzuk létre az új objektumokat [7]. A Spring keretrendszer bizonyos megkötései miatt egy ettől eltérő dizájnt használtunk. A Spring által nyújtott `@Autowired` annotációval tudjuk bizonyos objektumok létrehozásának szerepkörét átruházni a Spring-re. Ehhez azonban szükséges, hogy az osztály, amit példányosítani kívánunk, rendelkezzen egy argomentum nélküli konstruktorral. Vannak azonban olyan szolgáltatások, melyek működéséhez plusz információ, bizonyos paraméterek szükségesek.

Vegyük példaként a `RestaurantDetailsSetterService` -t.

```
1 public interface RestaurantDetailsSetterService {
2     void setName(String name);
3     void setDescription(String description);
4 }
```

Kódcsipet 2.1. RestaurantDetailsSetterService által definiált interfész

Ez a szolgáltatás felelős az egyes éttermek tulajdonságainak megváltoztatásáért, mint például az étterem neve vagy leírása. A metódusoknak nem adjuk át paraméterként, hogy melyik étterem tulajdonságait kívánjuk megváltoztatni, hiszen így minden egyes hívásnál meg kell bizonyosodnunk róla, hogy a helyes azonosítót adjuk át. Ehelyett az implementálandó osztály

felelőssége lesz ennek számontartása, egy privát mező formájában, amit egy, a konstruktorban paraméterként kapott értékkel inicializálunk. Pár ezt a megoldást is lehet helytelenül használni, például találomra kitalált azonosítót megadni a konstruktor paraméterének, azonban sokkal nehezebb. Csupán egy helyen kell ügyelnünk a paraméter helyességére, továbbá, mivel ezen paraméter érétke nem változik, érdemesebb eltárolni a szolgáltatás létrehozásakor, mintsem minden metódushívásnál újra megadni.

Mivel a szolgáltatás konstruktora rendelkezik egy paraméterrel, nem tudjuk az `@Autowired` annotáció segítségével létrehozni. Viszont képesek vagyunk definiálni egy factory osztályt, aminek egy megfelelő metódusa felel az objektum létrehozásáért.

```
1 public interface RestaurantDetailsSetterServiceFactory {
2     RestaurantDetailsSetterService get(long restaurantId);
3 }
```

Kódcsipet 2.2. A factory interfésze

A `get` metódus paraméterként megadunk minden kontextust a szolgáltatás létrehozásához.

```
1  @Component
2  public class DetailsServiceFactoryImpl implements DetailsServiceFactory {
3      @Autowired
4      private RestaurantRepository restaurantRepository;
5
6      @Override
7      public DetailsService get(long restaurantId) { return new
       DetailsServiceImpl(restaurantId); }
8
9      @AllArgsConstructor
10     private class DetailsServiceImpl implements DetailsService {
11         private long id;
12
13         // ...
14
15         @Override
16         public void setName(String name) {
17             // ..
18         }
19
20         @Override
21         public void setDescription(String description) {
22             // ..
23         }
24     }
25 }
```

Kódcsipet 2.3. Egy lehetséges factory implementáció

A factory osztály a kontextus (azaz az étterem azonosítója) nyújtásán kívül elérést biztosít az adatelérési réteghez. A `RestaurantDetailsSetterService` implementációja privát belső

osztályként szerepel a factory osztályban, ezzel növelve a factory osztály enkapszulációját. A factory minta ezen verziója a következő elemekből áll:

- egy létrehozandó szoftverkomponens interfésze (későbbiekben *cél*)
- az ezt létrehozó factory interfésze
- egy osztály (későbbiekben *host*), ami implementálja a factory interfészt, illetve más szolgáltatásokat vesz igénybe; ő maga is egy szolgáltatás
- host privát, belső osztálya, ami implementálja célt a host által nyújtott szolgáltatások és kontextus felhasználásával.

2.2.2. Builder

Amennyiben van egy komplex objektumunk, ami több komponenst is magában foglalhat, érdemes elválasztani az objektum létrehozását az objektum viselkedésének implementációjától [7] . Amennyiben az objektum módot ad

- egy egyszerú alapreprezentáció létrehozására
- ezen reprezentáció kibővítésére

létre tudunk hozni egy entitást, ami felügyeli ezen folyamatot. Ez az entitás, a *Builder*, definiál egy létrehozási folyamatot, mely folyamat több, eltérő reprezentációt is létre tud hozni, és az ezt egységesítő interfészt.

Egy példa a Java standard könyvtárából a `StringBuilder` . Segítségével képesek vagyunk egy stringet felépíteni kisebb szerkezeti egységek hozzáadásával, majd az így kapott karaktersorozatot megkapni. Az `insert` és `append` metódusai számos overload-dal rendelkeznek, így rugalmasabban tudjuk felépíteni a kívánt végeredményt.

Egy további példa a projectből a `NavBar` és a `NavBarBuilder` . A `NavBar` egy egységes navigációs sáv komponens, amelyet minden felhasználói csoport felülete használ. Egy opcionális címkéből, ami valamilyen információt ad a felületről, ahol megjelenik a sáv, illetve kattintható gombok sorozatából áll, melyek az alkalmazás egy megfelelő felületére navigálják a felhasználót. A `NavBar` egy komplex objektum. Ez nem viselkedésében tükröződik, hanem abban, hogy a megadható navigációs opciók számát nem tudjuk egyértelműen deifniálni, igény szerint változhat.

Egy lehetséges megoldás lenne, ha több konstruktort hoznánk létre az osztálynak, mind-

egyikben kezelve egy-egy lehetséges igényt:

- ne legyen címkéje, ne legyenek opciók
- legyen címkéje, de ne legyenek opciók
- ne legyen címkéje, legyen valamennyi opciója
- legyen címkéje, legyen valamennyi opciója

Ezen esetek számát le tudjuk csökkenteni, ha a konstruktor

- a címkét `Optional<String>` típusúként kezeli, azaz függetlenül attól, hogy kell-e címke vagy sem, a konstruktor rendelkezni fog ezzel a paraméterrel
- az opciókat változó hosszúságú paraméterlistaként adjuk meg, amely, abban az esetben, ha nincs navigációs opció, üres, másképp pedig az opciókat tartalmazza

```
1  public class NavBar extends HorizontalLayout {
2      public NavBar(Optional<String> label, NavOption... options) {
3          // ...
4      }
5
6      @Getter
7      @AllArgsConstructor
8      public static class NavOption {
9          private String label;
10         private Class<? extends Component> route;
11     }
12 }
```

Kódcsipet 2.4. A NavBar osztály implementációja

Ezzel lecsökkentettük az objektum létrehozásának módjainak számát, azonban ez a megoldás nem rugalmas. Az objektum kétrehozásához minden információra szükségünk van ami a címkét és az opciókat illeti, és lehetséges, hogy ezek nem állnak rendelkezésre egyszerre, vagy csak egy részük és ezek áthídalása jelentősen növeli a kódbázisunk komplecitását.

A `NavBarBuilder` bevezetésével a létrehozási folyamat jelentősen leegyszerűsödik. Definiálunk egy egységes interfészt, amely tartalmazza a `NavBar` létrehozásának lépéseit.

```
1 public interface NavBarBuilder {
2     NavBarBuilder addOption(NavBar.NavOption option);
3     NavBarBuilder setLabel(String label);
4     void reset();
5     NavBar build();
6 }
```

Kódcsipet 2.5. NavBarBuilder

Ezek a metódusok nem csak egyszerűbbé teszik a létrehozás folyamatát, hanem elrejtik a `NavBar` osztály valós implementációjának részleteit. A `NavBarBuilder` egy lehetséges

implementációját a 2.6 kódcsipetben láthatjuk.

```
1  public class NavBarBuilderImpl implements NavBarBuilder {
2      private List<NavBar.NavOption> options;
3      private String label;
4
5      public NavBarBuilderImpl() {
6          options = new ArrayList<>();
7          label = null;
8      }
9
10     @Override
11     public NavBarBuilder addOption(NavBar.NavOption option) {
12         this.options.add(option);
13         return this;
14     }
15
16     @Override
17     public NavBarBuilder setLabel(String label) {
18         this.label = label;
19         return this;
20     }
21
22     @Override
23     public void reset() {
24         options = new ArrayList<>();
25         label = null;
26     }
27
28     @Override
29     public NavBar build() {
30         return new NavBar(Optional.of(label), options.toArray(NavBar.
       NavOption[]::new));
31     }
32 }
```

Kódcsipet 2.6. A NavBarBuilder egy lehetséges implementációja

2.2.3. Adapter

Az *Adapter* minta felhasználásával egy osztály interfészét képesek vagyunk átalakítani valamilyen egyéb interfészre, amit a kliens definiált. Lehetővé teszi egyébként inkompatibilis interfészek és komponensek együttes használatát [7].

Az alkalmazás fejlesztése során felhasználtunk egy Vaadin add-on-t, a Crud UI Add-on-t, amely egy komponenst biztosít adatbázis műveletekhez szükséges felhasználói felületek létrehozására. Az add-on definiál egy interfészt, `CrudListener<T>` néven, melynek egy implementációját át kell adnunk a már említett komponens konstruktorának.

```
1  public interface CrudListener<T> extends Serializable {
2      Collection<T> findAll();
```

```
3       T add(T var1);
4       T update(T var1);
5       void delete(T var1);
6   }
```

Kódcsipet 2.7. A CrudListener interfész metódusai

Ezek a metódusok könnyen térképezhetőek adatbázis műveletekre, nem növelné jelentősen a komplexitás mértékét az alkalmazásunk szellemi modelljében, ha erre az interfészre hagyatkozna az adatelérési, illetve szolgáltatás réteg, mint egységes interfész. Azonban ez a dizájn erős kapcsolatot hozna létre a már említett rétegek, illetve egy külső, UI könyvtár között, amelynek szerepét számos okból kifolyólag átveheti valamilyen más megoldás, más interfésszel, ami ismét nem lesz kompatibilis az általunk fejlesztett szoftverentitások interfészével.

```
1 public interface EntityService<T> {
2       void add(T t);
3       Collection<T> getAll();
4       void update(T t);
5       void remove(T t);
6   }
```

Kódcsipet 2.8. Egységes interfész adatbázis entitásokat kezelő szolgáltatásoknak

Ahhoz, hogy a szolgáltatás réteg osztályainak állandó, más szoftveres komponensektől független interfészt tudjunk meghatározni, és mégis képesek legyünk ezen szolgáltalásokat használni a Crud UI-al, be kell vezetnünk egy közvetítő osztályt a két fél közé.

Mivel a Java nem támogatja a több osztályból való öröklődést, ezért csupán az `Adapter` minta *objektum adapter* variánsát tudjuk felhasználi. Ebben a verzióban létrehozunk egy osztályt, ami adapterként fog viselkedni a két interfész között, ami implementálja a célinterfészt (esetünkben a `CrudListener` -t), illetve rendelkezik egy mezővel, ami a kiinduló interfész egy példánya (esetünkben ez a `EntityService`). Ezen mezőt a konstruktor paramétereként kapott példánnyal inicializáljuk. Az adapter osztály a kiinduló interfész metódusait felhasználva implementálja a célinterfész metódusait.

```
1  @AllArgsConstructor
2  public class EntityServiceAdapter<T> implements CrudListener<T> {
3      private final EntityService<T> service;
4
5      @Override
6      public Collection<T> findAll() { return service.getAll(); }
7
8      @Override
9      public T add(T t) {
10         service.add(t);
11         return t;
12     }
```

```
13
14      @Override
15      public T update(T t) {
16          service.update(t);
17          return t;
18      }
19
20      @Override
21      public void delete(T t) { service.remove(t); }
22  }
```

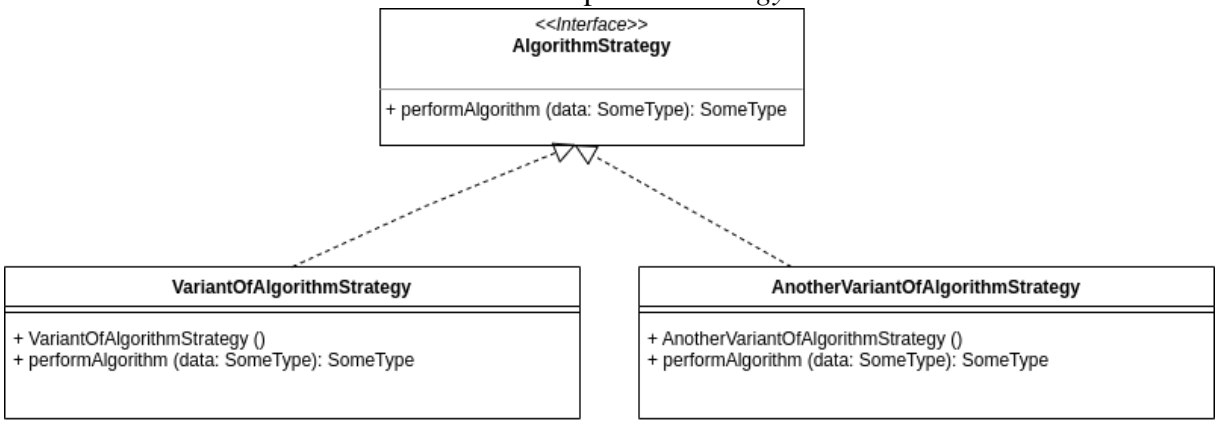
Kódcsipet 2.9. Az adapter osztály

A későbbiekben amikor a célinterfész egy példányára van szükségünk, példányosítjuk az adapter osztályt, a kiinduló interfész valamely implementációjával.

2.2.4. Strategy

A *Strategy* viselkedési minta lehetővé teszi, hogy definiálva egy általános algoritmus interfészét, algoritmusok egész családját vagyunk képesek létrehozni, melyek mindegyike egy lehetséges, érvényes implementációját enkapszulálja az algoritmusnak, és kölcsönösen felcserélhetővé teszi őket. A minta lehetővé teszi az algoritmus interfészének és valós implementációinak elválasztását, ami azt eredményezi, hogy ezek bármikor különbözhetnek kliensektől, amelyek felhasználják, anélkül, hogy a külvilág számára észlelhető viselkedésük inkonzisztens lenne [7].

2.1. ábra. Általános példa a *Strategy* mintára



Az alkalmazás fejlesztése során a publikus funkcióként elérhető keresés funkció implementálásában használtuk fel. Az általános algoritmus kívánt működése az, hogy termékek egy halmazából, előre meghatározott keresési szempontok alapján meghatározza azokat a termékeket,

amelyek a keresési szempontok szerint megfelelnek bizonyos kritériumoknak.

Ezen algoritmus köré kiépíthünk egy strategy mintát, ahol a mintában szereplő általános algoritmust a következő interfészben definiáljuk:

```
1 @AllArgsConstructor
2 @Getter
3 public class SearchProperties {
4     private String name;
5     private long minPrice;
6     private long maxPrice;
7 }
8
9 @FunctionalInterface
10 public interface SearchStrategy {
11     List<Item> filterSearch(List<Item> hayStack, SearchProperties
      properties);
12 }
```

Kódcsipet 2.10. Egységes interface az algoritmus elérésére

Láthatjuk, hogy egy egységes interface mellett, amelyen keresztül elérjük az algoritmus éppen aktuális implementációját, definiáljuk azt is, hogy milyen szempontok szerint történik a keresés. Alternatív megoldás lehetett volna, ha a `filterSearch` nem kéri ezeket paraméterként, hanem az egyes implementáló osztályok saját maguk definiálják az általuk vizsgált szempontokat, de ez a megoldás nem lenne rugalmas.

Bár így lehetséges, hogy egy implementáció olyan keresési szempontot is kap, amely számára irreleváns, az egyes implementációk egymás között teljesen felcserélhetőek.

Jelen formájában az alkalmazás csak egyfajta keresést támogat. Akkor fogadunk el egy terméket találatként, ha

- `name` mezőjének rész-string-je a `properties` `name` mezője
- `priceInHuf` mezőjének értéke nagyobb vagy egyenlő a `properties` minPrice mezőjénél
- `priceInHuf` mezőjének értéke kisebb vagy egyenlő a `properties` maxPrice mezőjénél

Ezen definíció mentén a következő képpen definiáltuk az algoritmus implementációját:

```
1 public class FilterSearchStrategy implements SearchStrategy{
2     @Override
3     public List<Item> filterSearch(List<Item> hayStack, SearchProperties
      properties) {
4         return hayStack
5                 .stream()
6                 .filter(item -> item.getPriceInHuf() >= properties.
      getMinPrice())
7                 .filter(item -> item.getPriceInHuf() <= properties.
      getMaxPrice())
```

```
8                   .filter(item -> item.getName().contains(properties.getName
    ()))
9                   .toList();
10      }
11  }
```

Kódcsipet 2.11. A SearchStrategy egy lehetséges implementációja

Mivel a keresési szempontok is egységesítésre kerültek, így a felhasználói felület tervezésekor hagyazhatunk ezekre a szempontokra, egységesen vagyunk képesek validálni őket.

Amennyiben más fajta keresési algoritmus szeretnénk implementálni, például egy olyan megoldást, amiben a éppen vizsgált elem `name` mezője illeszkedik-e az erre vonatkozó keresési feltételre valamilyen *fuzzy search*-ön [8] alapuló algoritmussal vizsgáljuk.

2.3. Használt könyvtárak és keretrendszerek

2.3.1. Lombok

A Lombok könyvtár, aminek célja a repetatív kódrészletek (úgynevezett boilerplate kód) írásának elkerülése, a fejlesztői élmény javítása. A legtöbb esetben nem terjeszti ki a Java funkcióit, hanem már meglévő funkciók használatát teszi kényelmesebbé.

Bizonyos keretrendszerek disziplínái megkövetelik például a getter-ek és setter-ek, bizonyos contructor-ok definiálását. Ilyen esetben hasznlhatjuk a Lombok, ami a build folyamatunkba beépülve, valid Java bytecode-ot generál automatikusan azokban az osztályokban, ahol bevezettük az annotációit.

Amennyiben kíváncsiak vagyunk arra, hogy a Lombok milyen transzformációkat hajt végre a kódunkon, vagy egyszerűen csak meg akarunk válni tőle, és eltávolítani a függőségink közül, a kódbázisunkat pedig megtisztítani a Lombok annotációktól, abban az esetben erre is van lehetőség.

A delombok nevű eszköz előállítja azokat az osztályok forráskódját, amikben Lombok annotációt használtunk, eltávolítva az annotációkat és helyükre a velük ekvivalens kód kerül, amelyet eddig a Lombok generált.

@NoArgsConstructor, @AllArgsConstructor

A `@NoArgsConstructor`-t egy osztályra helyezve egy argomentumok nélküli konstruktort fog generálni. Amennyiben ez nem lehetséges, például egy `final` mező miatt, a generálási folyamat egy fordítási hibát fog eredményezni [2]. Ez megkerülhető úgy, ha az annotáció `force` paraméterének `true` értéket adunk meg. Ezzel elérjük, hogy a `final` mezők is inicializálva lesznek `0`, `false` vagy `null` értékkel, azonban olyan mezők esetében, a `@NonNull` annotációval végzünk null vizsgálatot, ez a vizsgálat nem fog legenerálódni, megtörténni.

```
1  @NoArgsConstructor
2  class NoArgsConstructorClass {
3      private String string;
4      public int integer;
5  }
6
7  @NoArgsConstructor(force = true)
8  class NoArgsConstructorClassWithNonNullField {
9      private @NonNull String string;
10     public int integer;
11 }
```

Kódcsipet 2.12. @NoArgsConstructor

```
1  class NoArgsConstructorClass {
2      private String string;
3      public int integer;
4
5      public NoArgsConstructorClass()
6      {
7      }
8  }
9
10 class NoArgsConstructorClassWithNonNullField {
11     private @NonNull String string;
12     public int integer;
13
14     public NoArgsConstructorClassWithNonNullField()
15     {
16         this.string = null;
17     }
18 }
```

Kódcsipet 2.13. @NoArgsConstructor delombok után

A project elkészítése során Spring Data JPA entitás osztályokat láttunk el `@NoArgsConstructor` annotációval, mivel a JPA megköveteli egy ilyen konstruktor létezését.

A `@AllArgsConstructor` egy konstruktort generál egy osztálynak, annak egy-egy mezőjéhez tartozó egy paraméterrel. Amennyiben egy mező el van látva a `@NonNull` annotációval, a generált konstruktor egy null check-et fog végezni azon a mezőn.

```
@AllArgsConstructor
class AllArgsConstructorClass {
    private String string;
    public int integer;
    @NonNull
    protected Double number;
}
```

Kódcsipet 2.14. @AllArgsConstructor

```
class AllArgsConstructorClass {
    private String string;
    public int integer;
    @NonNull
    protected Double number;

    public AllArgsConstructorClass(String string, int integer, @NonNull
    Double number)
    {
        this.string = string;
        this.integer = integer;
        this.number = number;
    }
}
```

Kódcsipet 2.15. @AllArgsConstructor delombok után

Az `@AllArgsConstructor` hátránya, hogy csak azok a mezők szerepelnek a konstruktor paraméterei között, amelyeket az osztályban deklaráltunk, azaz a szülő osztály mezőihez tartozó argomentumok nem szerepelnek a gyermek osztály generált konstruktorában. Ennek ellenére használata eredményes volt a projectben, például DTO-k és adattagokkal rendelkező enumok definniálásánál.

@Getter, @Setter

Egy osztály bármilyen mezőjét annotálhatjuk a `@Getter`-rel vagy `@Setter`-rel, a Lombok automatikusan generálni fog egy alapértelmezett getter/setter metódust.

A `foo` mező alapértelmezett getter-je a `getFoo` nevű metódus, illetve `boolean` típusú mező esetében `isFoo`, ami a `foo` mezőt téríti vissza. Ezen mező alapértelmezett setter-je egy `setFoo` nevű, egy paraméteres metódus, amely paraméter típusa azonos a mező típusával.

A generált metódus publikus láthatóságú lesz, amennyiben ezt nem írjuk felül az annotációban elhelyezett `AccessLevel` értékkel. Ennek lehetségek értékei `PUBLIC`, `PROTECTED`, `PACKAGE` és `PRIVATE`, melyek rendre a Java nyelv láthatóságainak felelelnek meg.

Ezeket az annotációkat nemcsak mezőkön, hanem osztályokon is alkalmazhatjuk. Ebben az esetben az annotációhatása ekvivalens azzal, h az osztály minden, nem statikus mezőjét annotál-

tuk volna. Kivételt képeznek azok a mezők, amelyeket menuálisan annotálunk és láthatóságnak `AccessLevel.NONE` -t határozunk meg.

```
1 @Getter
2 class GetterSetterClass {
3     int foo;
4     @Setter
5     double bar;
6     @Setter(AccessLevel.PROTECTED)
7     String baz;
8     @Getter(AccessLevel.NONE)
9     int bat;
10 }
```

Kódcsipet 2.16. @Getter, @Setter

```
1 class GetterSetterClass {
2     int foo;
3     double bar;
4     String baz;
5     int bat;
6
7     public int getFoo() { return this.foo; }
8
9     public double getBar() { return this.bar; }
10
11    public String getBaz() { return this.baz; }
12
13    public void setBar(double bar) { this.bar = bar; }
14
15    protected void setBaz(String baz) { this.baz = baz; }
16 }
```

Kódcsipet 2.17. @Getter, @Setter delombok után

@ToString

Bármely osztály annotálható `@ToString` -el, amely felülírja az osztály `toString` metódusát, egy, a Lombok által generált implementációval. Alapértelmezetten, ezen implementáció által visszaadott string tartalmazza az osztály nevét, követve a az osztály nem statikus mezőinek nevével és ezek értékeivel, a deklarálásuk sorrendjében.

Amenability nem akarjuk, hogy minden mező megjelenjen a metódus output-jában, jelezhetjük a nem kívánt mezőket a `@ToString.Exclude` annotációval, felsorolhatjuk a tartalmazni kívánt mezők nevét a `@ToString` metódus `includeFieldNames` paraméterében, vagy a `onlyExplicitlyIncluded` paraméternek megadott `true` értékkel és a tartalmazni kívánt mezőkön elhelyezett `@ToString.Include` annotációval. Amennyiben a `callSuper` praméternek `true` értéket adunk, a visszaadott string tartalmazni fogja a szülő osztály `toString`

metódusának output-ját

@Builder

A `@Builder` annotációval annotált `Foo` osztályhoz, konstruktorhoz vagy metódushoz (a későbbiekben ez *target*) generálódik egy belső, statikus, `FooBuilder` nevű osztály (a későbbiekben ez *builder*).

A builder osztály tartalmaz target egy-egy paraméteréhez vagy mezőjéhez tartozó, privát, nem statikus, nem `final` mezőt, egy package private no-args konstruktort. Builder minden mezője rendelkezik egy setter-szerű metódussal, ami a mező értékéhez a paraméterként kapott értéket rendeli, és a builder példányt adja vissza, ezzel elhetővé téve a metódushívások egymásba láncolását. Builder rendelkezik egy `build` metódussal, amely meghívásakor meghívódik target (amennyiben target osztály, annak egy megfelelő konstruktora), builder mezőinek értékével, és ugyanazt a típust adja vlssza, mint target (osztály esetében ez target).

A target-et tartalmazó osztályban (amennyiben target osztály, ez target) generálódik egy `builder` metódus, ami builder egy új példányát hozza létre.

```
1 @Builder
2 class BuilderClass {
3     int i;
4     double d;
5 }
```

Kódcsipet 2.18. @Builder

```
1 class BuilderClass {
2     int i;
3     double d;
4 
5     BuilderClass(int i, double d) {
6         this.i = i;
7         this.d = d;
8     }
9 
10    public static BuilderClassBuilder builder() { return new
      BuilderClassBuilder(); }
11 
12    public static class BuilderClassBuilder {
13        private int i;
14        private double d;
15 
16        BuilderClassBuilder() {}
17 
18        public BuilderClassBuilder i(int i) {
19            this.i = i;
20            return this;
```

```
21          }
22
23          public BuilderClassBuilder d(double d) {
24              this.d = d;
25              return this;
26          }
27
28          public BuilderClass build() { return new BuilderClass(i, d); }
29
30          public String toString() { // ...
31          }
32      }
33  }
```

Kódcsipet 2.19. @Builder delombok után

@Slf4j

A `@Log` annotáció számos variánssal rendelkezik. Ezen variánsok egy-egy log kezelő megoldáshoz készülnek, mivel számos ilyen megoldás van a JVM platformon, egy annotáció családról beszéhetünk. Közös tulajdonsága a család tagjainak, hogy az annotáció elhelyezése után elérhető egy `log` nevű objektum, ami az adott logolási megoldás logger példánya. Az `@Slf4j` az azonos nevű loggert teszi elérhetővé.

A fejlesztés során hasznosnak bizonyult, mivel a kódbázis tisztább és kisebb lett, a kód kevesebb zajt tartalmaz.

```
1 @Slf4j
2 class Slf4jClass {}
```

Kódcsipet 2.20. @Slf4j

```
1 class Slf4jClass {
2     private static final Logger log = org.slf4j.LoggerFactory.getLogger(
      Slf4jClass.class);
3 }
```

Kódcsipet 2.21. @Slf4j delombok után

2.3.2. Spring

A Spring keretrendszer a Java, illetve Jakarta EE alkalmazásfejlesztés de facto sztenderdjévé vált 2002-es megjelenése óta. A framework egyik legfontosabb funkciója a *dependency injection* modellje, ami segíti a gyors alkalmazásfejlesztést, továbbá átláthatóbb, tisztább kódot eredményez. A Srping modulárisan van felépítve, amelyek olyan szolgltatásokat nyújtanak, mint az adatelérés, tesztelés és web integráció. Fejlesztőként nem vagyunk kényszerítve, hogy

a keretrendszer által kínált összes komponenst egyszerre használjuk. A moduláris modell lehetővé teszi, hogy csupán a szükséges elemeket tartalmazza projectünk, attól függően, hogy az éppen aktuálisan fejlesztett alkalmazás mit igényel [4].

A Spring portfólióhoz számos más project tartozik, mint például a Spring Security, Spring Data vagy a Spring Boot, melyek mindegyike a Spring framework által nyújtott infrastruktúrára épül. Ezek célja rendre az autentikáció és autorizáció, az adatelérés és a Spring alkalmazások létrehozásának egyszerűbbé, elérhetőbbé tétele.

Spring Security

Az alkalmazásban az autentikációt és autorizációt Spring Security segítségével oldottuk meg.

```
1  @EnableWebSecurity
2  @Configuration
3  public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
4      // ...
5      @Override
6      protected void configure(HttpSecurity http) throws Exception {
7          http.
8              // ...
9              .requestMatchers(SecurityUtils::isFrameworkInternalRequest).
       permitAll()
10             .antMatchers("/").permitAll()
11             .antMatchers("/api/**").permitAll()
12             .antMatchers("/app/public/**").permitAll()
13             .antMatchers("/app/restaurant/**").hasRole("RESTAURANT")
14             .antMatchers("/app/end-user/**").hasRole("END_USER")
15             // ...
16     }
17     // ,,,
18 }
```

Kódcsipet 2.22. Spring Security konfiguráció

Láthatjuk, hogy van lehetőségünk Java kóddal is konfigurálni a Spring Security-t, a kódcsipetben látható módon. Képesek vagyunk URL minták megadásával meghatározni, hogy milyen felhasználói jogkör szükséges az adott erőforrás eléréséhez. Az alkalmazás könnyű bővíthetőségét segíti, ha a végpontok URL-jének meghatározásakor figyelembe vesszük, hogy milyen funkcióhoz és szerepkörhöz tartoznak, ezzel szemantikus jelentőséget adva az URL-eknek és egyúttal egy réteg alapú architektúrát hozunk létre. Ebben az esetben

- a Vaadin framework beéső működéséhez szükséges erőforrások autentikáció nélkül elérhető

- a gyökér URL, ahova az oldal felkeresése esetén először érkezünk publikusan elérhető
- bármely read-only API autentikáció nélkül elérhető
- az applikáció publikus funkciói autentikáció nélkül elérhetőek
- az applikáció éttermekhez tartozó funkciói eléréséhez *RESTAURANT* szerepkörrel kell rendelkeznie a felhasználónak
- az applikáció végfelhasználókhoz tartozó funkciói eléréséhez *END_USER* szerepkörrel kell rendelkeznie a felhasználónak

Amennyiben valamely szerepkör funkcióját szeretnénk bővíteni, például egy GraphQL alapú API-t létrehozni, vagy valamilyen új lehetőséggel augmentálni a az éttermeket, ezt könnyen és egszerűen megtehetjük ezen módszer bővíthetősége miatt.

Spring Boot

A Spring által biztosított dependency injection model az `@Autowired` annotáció segítségével érjük el. Ezzel a megoldással csak a Sprning által kezelt entitásokat tudjuk injektálni, azaz olyan osztályokat, amelyeknek van no-args-konstruktora, illetve el van látva valamilyen Spring által nyújtott *stereotype annotációval*. Ezek kötül a legfontosabbak a `@Component` , `@Repository` és a `@Service` . Ez utóbbi kettő kiterjeszti a `@Component` szerepkörét.

Egy `@Component` -el annotált osztályokat automatikusan detektálja a Spring, mint általa kezelt komponens. A `@Repository` annyiban terjeszti ki ezt a viselkedési módot, hogy az adatelérési rétegből érkező, perzisztenciához kapcsolódó, platform specifikus hibákat elkapja és a Spring egységes kivételeként dobja tovább. A `@Service` jelzi a Spring számára, hogy az annotált osztály a szolgáltatás rétegbe tartozik, és üzleti logikát tartalmazhat, de jelenleg ennek nincsen szemantikus jelentősége. A Spring nem kényszeríti ránk ezen annotációk szerepkörüknek megfelelő idiomatikus használatát, azonban a jövőben érkezhetnek olyan változtatások a keretrendszerben, amely feltételezi az annotációk előírt használatát.

Az `@Autowired` annotációt osztályok mezőin, konstruktorokon, illetve setter metódusokon helyezhetjük el.

```
1 public class FooService {
2     @Autowired
3     private final FooRepository fooRepository;
4 }
```

Kódcsipet 2.23. Mezőn elhelyezett `@Autowired`

```
1 public class FooService {
2     private FooRepository fooRepository;
3
4     @Autowired
5     public void setFooRepository (FooRepository fooRepository) {
6         this.fooRepository = fooRepository;
7     }
8 }
```

Kódcsipet 2.24. Setter-en elhelyezett `@Autowired`

```
1 public class FooService {
2     private final FooRepository fooRepository;
3
4     @Autowired
5     public FooService(FooRepository fooRepository) { this.fooRepository =
      fooRepository; }
6 }
```

Kódcsipet 2.25. Konstruktoron elhelyezett `@Autowired`

A konstruktor alapú megoldás preferálandó, amennyiben szeretnénk, ha az injektált komponenshez tartozó mező `final` lenne, vagy ha csak az objektum létrejöttekor van szükségünk rá.

Az `@Autowired` típus alapján rezolválja az inkektálásra alkalmas osztályokat. Amennyiben több osztály is alkalmas erre, valamilyen módon jeleznünk kell a Spring felé, hogy melyik implementációt kívánjuk használni.

Ezt megtehetjük úgy, hogy a komponens deklarálásakor egy egyedi azonosítóval látjuk el, majd később erre hivatkozunk.

```
1 @Component("fooComponent")
2 public class FooComponent implements SomeComponent {
3     // ..
4 }
5
6 public class FooService {
7     @Autowired
8     @Qualifier("fooComponent")
9     private SomeComponent component;
10 }
```

Kódcsipet 2.26. Azonosítóval ellátott komponens

Egy alternatív megoldás ha az injektált mező neveként az injektálni kívánt implementáció nevét használjuk.

```
1 @Component("fooComponent")
2 public class FooComponent implements SomeComponent {
3     // ..
4 }
5
```

```
6 public class FooService {
7     @Autowired
8     private SomeComponent fooComponent;
9 }
```

Kódcsipet 2.27. Osztály neve, mint azonosító

2.3.3. Spring Boot Starter Data JPA

A Spring Boot Starter Data JPA egyszerűbbé teszi az adatbázisokkal való kommunikációt, egyszerűbbé teszi az adatelérési réteg kialakítását, mindezt oly módon, hogy a megoldás átlátható, könnyen búvíthető legyen. Amennyiben úgy döntünk, hogy nem kívánunk mi magunk adatbázis sémákat, entitásokat, lekéréseket létrehozni, hanem ezt a feladatot a Spring Data JPA-nak delegáljuk, lehetőségünk van arra, hogy a JPA hozza létre az adattáblákat, lekérdezéseket és egyéb SQL parancsokat, és ezek eredményét kezelje.

Perzisztens entitások

Ahhoz, hogy perzisztálhassuk egy osztály egy példányát egy adatbázisban

- annotálnunk kell a `@Entity` annotációval
- rendelkeznie kell egy argomentumok nélküli konstruktorral
- minden perzisztálni kívánt mezőjének rendelkeznie kell publikus getter-el és setter-el
- egy mezőt meg kell jelölnünk az `@Id` annotációval, ami az adatbázisbeli azonosítója lesz

Opcionálisan annotálhatjuk az entitás osztályt a `@Table` annotációval, amelynek `name` paramétereként megadott string lesz az entitásokat tartalmazó tábla neve.

Az entitás egy-egy mezőjéhez tartozó oszlop neve a mező nevével ekvivalens, amennyiben ezt nem írjuk felül a mezőn elhelyezett `@Column` annotáció `name` paraméterének adott stringgel.

Ügyeljünk arra, hogy semmiképen sem használjunk SQL kulcsszavakat táblák vagy mezők neveként.

@Converter, AttributeConverter

Amennyiben egy olyan értéket akarunk perzisztálni, ami nem

- JPA entitás

- JPA entitások kollekciója
- primitív Java típusok és ezek wrapper osztályai, illetve `String`

biztosítanunk kell egy osztályt, ami definiál egy kölcsönös leképezést egy perzisztálható és a jelenlegi perzisztálni kívánt típus között.

Ennek eléréséhez definiálnunk kell egy osztályt, ami annotálva van a `@Converter` annotációval, illetve implementálja a `AttributeConverter<A, C>` generikus interfészt, ahol A az az entitás attribútum típus, amiből kiindulunk, és perzisztálni kívánjuk, C pedig az, amit valóban el tud tárolni az adatbázis.

Az `AttributeConverter<S, T>` interfésznek két definiált metódusa van:

- `C convertToDatabaseColumn(A attribute)` , ez konvertálja az entitásból kapott attribútumot egy, adatbázis által tárolható értékké
- `A convertToEntityAttribute(C column)` , ez konvertálja az adatbázis egy oszlopában szereplő értéket entitás attribútummá.

Az interfészt implementáló osztályt annotálnunk kell a `@Converter` annotációval annak érdekében, hogy alkalmazható legyen az átváltás. Ammenyiben az annotáció `autoApply` paraméterének igaz értéket adunk, a perzisztancia ellátójának (angolul *persistence provider*) muszály automatikusan alkalmazni a konvertert minden entitás minden perzisztált attribútumára, amire alkalmazható, kivéve, ahol az attribútumon elhelyezett `@Convert` annotáció ezt felülírja [1].

A project elkészítése során `enum` értékek konvertálására használtuk.

```
1 @Getter
2 @AllArgsConstructor
3 public enum DishType {
4     PIZZA("Pizza"),
5     // ....
6     private final String name;
7 }
```

Kódcsipet 2.28. DishType

```
1 @Converter(autoApply = true)
2 public class DishTypeConverter implements AttributeConverter<DishType,
      String> {
3     @Override
4     public String convertToDatabaseColumn(DishType dishType) {
5         if (dishType == null) { return null; }
6         return dishType.getName();
7     }
8
9     @Override
```

```
10      public DishType convertToEntityAttribute(String s) {
11          if (s == null) { return null; }
12          return DishType.valueOf(s);
13      }
14  }
```

Kódcsipet 2.29. DishTypeConverter

JPA kapcsolatok

A JPA kapcsolatoknak két fajtája van:

- egyirányú (*unidirectional*)
- kétirányú (*bidirectional*)

Két entitás közötti kapcsolat definiálásának nincs hatása az entitások adatbázisra való leképezésének, csupán azt definiálja, hogy milyen irányban tudjuk használni a kapcsolatot a domain modellünkben.

Objektumok esetében a kétirányú kapcsolat egy definiált fogalom, szemantikája jól meghatározható, a relációs adatbázisok esetében azonban ez a fogalom nem létezik, csupán egyirányú kapcsolatok vannak (foreign key-ek formájában). Így a Hibernate, ami az objektumok relációkra való leképzést végzi, két egyirányú kapcsolattal modellezi a kétirányú kapcsolatot. A Hibernate a kapcsolat mindkét oldalának változásait követi, így például ha a kapcsolat pontosan egy résztvevője megváltozik, az az adatbázisban is meg fog változni, de ez a változás nem fog manifesztálódni a kapcsolat másik oldalán, azaz inkonzisztencia lép fel az adatbázisban. Ezt a problémát oldja meg a birtokló oldal, birtoklott oldal meghatározása.

Ha Hibernate csupán a birtokló oldal változásait követi nyomon, így ha a birtokló oldalon bármilyen változás történik a kapcsolatban, azt képes leképezni a kapcsolat másik felére, a kapcsolat birtokolt oldalán történő változtatásokat pedig nem követi. Ez a megoldás megoldja a kétirányú kapcsolatokból eredő inkonzisztenciákat. A birtokolt oldal könnyen felismerhető onnan, hogy értéket adunk a kapcsolatot jelző annotáció `mappedBy` paraméterének.

@OneToOne

Amennyiben egy mezőt a `@OneToOne` annotációval látunk el, egy egy-az-egyhez leképezést tudunk létrehozni két entitás között, azaz egy olyan leképezést, amelyben egy entitás egy példányához egy másik entitás legfeljebb egy példányát rendeljük. Az egy-az-egyhez leképezést több módon implementálhatjuk:

- *foreign key* használatával
- *közös primary key* használatával
- *join table* használatával

A project elkészítése során foreign key használatával implementáltuk a kapcsolatot. Ebben az implementációban a `@OneToOne` -nal annotált mezőt a `@JoinColumn` annotációval is el kell látni, amelynek `name` paraméterével adjuk meg a foreign key-t tartalmazó oszlop nevét. Ez az oszlop a kapcsolatban szereplő birtokló fél táblájában fog szerepelni.

Amennyiben a kapcsolat egyirányú, a birtokló fél egyértelműen meghatározható: az az entitás, amely a `@OneToOne` -nal annotált mezőt tartalmazza.

Ha a kapcsolatot kétirányúvá szeretnénk bővíteni, a egy-az-egyhez kapcsolat másik felét képző entitásban el kell látnunk a kapcsolat másik felének mezőjét a `@OneToOne` annotációval, amelynek `mappedBy` paraméterével adjuk meg, hogy a másik oldal melyik mezője hivatkozik rá.

```
1  // ...
2  @Entity
3  @Table(name = "restaurant_table")
4  public class RestaurantTable {
5      // ...
6      @OneToOne(cascade = CascadeType.ALL)
7      @JoinColumn(name = "restaurant_table_id")
8      private EndUser user;
9      // ...
10 }
11
12 // ...
13 @Entity
14 @Table(name = "end_users")
15 public class EndUser {
16     // ...
17     @OneToOne(mappedBy = "user")
18     private RestaurantTable table;
19     // ...
20 }
```

Kódcsipet 2.30. RestaurantTable mint birtokló, EndUser, mint birtokolt

@OneToMany, @ManyToOne

A `@OneToMany` és a `@ManyToOne` annotációkkal egy-a-többhöz és több-az-egyhez kapcsolatokat tudunk definiálni. Amennyiben külön-külön alkalmazzuk őket, ezek egyirányú kapcsolatokat határoznak meg, azonban ha egymással párban, két entitáson, akkor egy darab, kétirányú kapcsolatot definiál az entitások között.

A projectben például a `@ManyToOne` annotációval reprezentáltuk a rendelések és a pincérek közötti kapcsolatot, azaz egy pincérnek több, hozzá rendelt rendelése lehet, míg egy rendeléshez legfeljebb egy pincér rendelhető.

```
// ...
@Entity
@Table(name = "orders")
public class Order {
    // ...
    @ManyToOne
    @JoinColumn(name = "server_id")
    private Server server;
    // ...
}
```

Kódcsipet 2.31. Az Order entitás több-az-egyhez kapcsolatban van a Server-rel

@ManyToMany

A project során a rendelések és a rendelésekben szereplő termékek közötti kapcsolatot több-a-többhoz kapcsolattal írtuk le, mivel egy rendeléshez több termék tartozik, illetve egy termék több rendeléshez is hozzárendelhető.

Több féle módon implementálhatjuk:

- *join table* használatával
- egy új, közvetítő entitás létrehozásával, ami enkapszulálja a kapcsolatban résztvevő feleket (például egy ItemsOfOrder entitás, ami tartalmazná hogy melyik rendelés mely termékkel van asszociálva)
- *composite key* használatával

A project során *join table*-t felhasználva implementáltuk. A birtokló oldalon definiálni kell a kapcsolatokat tartalmazó tábla nevét, illetve hogy a kapcsolatban résztvevő felek id-jai ennek a táblának mely oszlopaiban szerepelnek

```
// ...
@Entity
@Table(name = "orders")
public class Order {
    // ...
    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(
            name = "order_items",
            joinColumns = @JoinColumn(name = "order_id"),
            inverseJoinColumns = @JoinColumn(name = "dish_id"))
    private List<Item> items;
    // ...
```

```
13 }
14
15 // ...
16 @Entity
17 public class Item {
18     // ...
19     @ManyToMany(mappedBy = "items")
20     Set<Order> orders;
21     // ...
22 }
```

Kódcsipet 2.32. Order, mint birtokló, Item, mint birtokolt

JpaRepository, @Repository

A Spring Data JPA által biztosított `JpaRepository` egyszerű, könnyen használható és kibővíthető megoldást nyújt az adatelérési réteg létrehozására. Csupán egy interface-t kell létrehoznunk, ami kiterjeszti a `JpaRepository<T, ID>` generikus interfészt, ahol a T a perzisztált entitás típusa, ID pedig ezen entitás azonosító mezőjének típusa [9]. A `JpaRepository` által definiált metódusok lehetővé teszik az alapvető DUCS adatbázis műveletek (azaz Delete, Update, Create, Select) használatát perzisztált entitásokra.

Amennyiben az alap műveleteken túlmutató lekérdezéseket szeretnénk definiálni, erre is van módunk. A `JpaRepository` -t kiterjesztő interfészben van módunk további metódusokat deklarálni, amelyek, ha nevük bizonyos szemantikai szabályokat követnek, interpretálva lesznek, mint SQL query-k. Az említett szemantikai szabályok az 5.1 táblázatban találhatóak.

A `@Repository` annotációval jelezzük a Spring keretrendszer felé, hogy az interfész egy *Repository*, azaz egy mechanizmus entitások tárolásra, kinyerésésre és keresésésre. A Spring 2.5-ös verzója óta a `@Component` annotáció egy specializációjaként is szolgál, azaz imlpementációi automatikusan detektálódnak [3].

2.3.4. Vaadin

A Vaadin keretrendszerek egy családja, ami a Flow illetve a Hilla (korábban Fusion) frontend keretrendszereket foglalja magában. Az általunk használt Flow framework lehetővé teszi reszpontív, interaktív felhasználói felületek létrehozását kizárólag Java-ban, komponensek segítségével. Számos beépített komponensel és funkcióval rendelkezik, illetve számos közösség által fejlesztett plug-in-ja van.

A beépített komponensek között találhatóak HTML-ből ismerős elemek, mint például a

- `Input`
- `TextArea`
- `Button`
- `Div`
- `H1, H2`
- `Hr`

Emellett biztosít olyan komponenseket, amelyeket nem tudunk megfeleltetni HTML elemeknek, azonban hasonló konstrukciókkal már találkozhattunk.

- `HorizontalLayout` , egy flex container, aminek `flex-direction` tulajdonsága `row` értékű
- `VerticalLayout` , egy flex container, aminek `flex-direction` tulajdonsága `column` értékű
- `PasswordField` , az `<input>` elem egy specializációja
- `NumberField` , az `<input>` elem egy specializációja

A Vaadin lehetőség nyújt továbbá *route*-ok definiálására és az ezek közötti navigációra.

```
1 @Route("app/restaurant/staff")
2 public class RestaurantStaffView extends RestaurantViewBase {
3     private final String[] crudProperties = {"name"};
4
5     @Autowired
6     public RestaurantStaffView(EntityServiceFactory<Server>
      serverEntityServiceFactory) {
7         super("Staff", "Staff");
8         // ...
9     }
10 }
```

Kódcsipet 2.33. a `app/restaurant/staff` route-hoz tartozó nézet definíciója

Ezek között a `UI` osztály `navigate` metódusával tudunk navigálni, úgy, hogy a metódusnak átadjuk paraméterként a cél route-hoz tartozó nézetet definiáló osztály referenciáját.

```
1 var backButton = new Button("Back to orders");
2 backButton.addClickListener(buttonClickEvent -> UI.getCurrent().navigate(
      RestaurantOrdersView.class));
```

Kódcsipet 2.34. Gombnyomásra visszanavigálunk a rendelésekhez

2.4. Felhasznált szabványok, ajánlások

2.4.1. UUID

Az univerzálisan egyedi azonosító (angolul *Universally unique identifier*) vagy UUID egy ITF szabvány, amit az RFC 4122 definiált. A fő motiváció használatára az, hogy nem szükséges egy központi szerv bevonása a létrejövő azonosítók adminisztrálására, így generálásuk teljes mértékben automatizálható. A UUID-k fix mérete miatt, ami 128 bit, jelentősen kisebb, mint a legtöb alternatív megoldás. Kompakt méretéből következik, hogy használata optimálisabb teljesítményt eredményez rendező és hasító algoritmusok, illetve adatbázisban való tárolás esetében. Az említett RFC-ben leírt generáló algoritmus akár másodpercenként 10 milló allokációt tud elvégezni gépenként, melyből kifolyólag a UUID-t tranzakciók azonosítójaként is használható.

A UUID azonosítók valójában nem *teljesen* egyediek, azaz van esély arra, hogy két generált azonosító ugyanazzal az értékkel fog rendelkezni, ütközni fognak. Azonban ennek a valószínűsége elenyésző. A UUID ütközést tekinthetjük a születésnap probléma egy speciális esetének. Annak az esélye, hogy egy populációban, egymástól függetlenül kiosztott x azonosító közül n-et kiválasztva p valószínűséggel legyen köztük legalább kettő egyező az

$$n = 0.5 + \sqrt{0.25 - 2 \times (\ln q) \times x}$$

formulával kiválóan lehet közelíteni [11], ahol $q = 1 - p$. Ebből adódóan ahhoz, hogy legalább 50%-os eséllyel generálódjon legalább két UUID,

$$n \approx 0.5 + \sqrt{0.25 + 2 \times (\ln 2) \times 2^{122}} \approx 2.71 \times 10^{18}$$

azonosítót kellene generálnunk. Ehhez, a másodpercenkénti 10 milló generált azonosítóból kiindulva megközelítóleg 8587 év szükséges, azaz ennyi idő suükséges ahhoz, hogy 50%-os valószínűséggel előidézzükn egy UUID ütközést.

Egy étterem egy, a rendszerbe felvett asztalához tartozó, automatikusan létrehozott végfelhasználói fiók "felhasználóneve" egy UUID, amit az említett fiók létrehozásakor generálunk. Ez egy optimális megoldás, hiszen az étteremnek nem kell "kitalálni" valamilyen egyedi azo-

nosítót a végfelhasználói fióknak, így gyorsabb és felhasználóbarátabb az asztalok a rendszerbe való felvezetésének folyamata. A UUID tulajdonságaiból kiindulva ezek az azonosítók ésszerű keretek között valóban egyediek lesznek.

2.4.2. REST API

Az állapotreprezentáció-transzfer (angolul *representational state transfer*) vagy REST nem egy konkrét, jól definiált standard, sokkal inkább egy architektúrális stílus, ami elterjedt technológiákat és szabványokat használ fel web alapú szolgáltatások tervezésére és implementálására [15].

Roy Thomas Fielding doktori disszertációban számos megkötést tett arra, hogy hogyan definiálható egy REST szolgáltatás architektúrája [6].

Kliens-szerver architektúra

Emögött a *separation-of-concerns* elv áll. Azzal, hogy elválasztjuk a felhasználói interfészt az adattárolás szerepkörétől, skálázhatóbbá és szélesebb körben portolhatóvá a felhasználói interfész. Emellett ez az elkülönülés megengedi, hogy külön a szerverkomponensek egymástól elkülönülve fejlődjenek.

Állapotmentes

A kliens és szerver közti kommunikációnak állapotmentesnek kell lennie, oly módon, hogy a klienstől érkező minden kérése tartalmazza az összes, a kérés feldolgozásához szükséges információt, nem hagyatkozhat bármilyen, a szerveren tárolt kontextusra.

Ez a döntés növelte a

- láthatóságot, egy megfigyelő (vagy *monitoring*) rendszernek csupán egy kérés alapján meg tudja határozni a kérés teljes természetét
- megbízhatóságot, mivel a rendszer könnyebben helyreáll egy részleges hiba után
- skálázhatóságot, mivel azzal, hogy nem tárol a szerver kérések között semmilyen állapotot, gyorsabban fel tudja szabadítani használt erőforrásait, illetve nem kell menedzselnie az erőforrásokat akár több kérésen keresztül

Ezzekkel szemben negatívumként említhetjük a romlott hálózati teljesítményt.

Gyorsítótár

Az előbb említett hátrányra válaszul vezessük be a modellünkbe gyprsítótárazhatóság fogalmát. Ez megköveteli hogy egy válaszban szereplő adat implicit vagy explicit módon meg legyen jelölve hogy gyorsítótárazható-e. Ha egy válasz gyorsítótárazható, akkor egy kliens oldali gyorsítótárnak módjában áll újra felhasználni azt az adatot későbbi, az eredeti kéréssel ekvivalens kérésekhez.

Egységes interfész

Azzal, hogy a REST szolgáltatások egy egységes interfészen kommunikálnka a klienssel, függetlenítjuk a klienset a szolgáltatás implementációjától. Annak érdekében, hogy egy interenet szintű REST szolgáltatásoknak egy egységes interfészt határozzunk meg, (azaz egy egyezményt kliens és a szolgáltatás között, ami definiálja kommunkációjuk formáját), ezt szabványok felhasználásával kell megtennünk.

- Erőforrás azonosítás: URI standard
- Erőforrás manipuláció: HTTP standard
- Önleíró üzenetek: MIME típusok
- HATEOAS: hyperlink-ek és URI template-ek

Réteg alapú rendszer

Hierarchikus rétegeket alakítunk ki azáltal, hogy az adott rétegek csak a velük közvetlenül kapcsolatban lévő rétegekről tudnak, csak ezekkel tudnak kommunikálni. Azzal, hogy limitáljuk az egyes rétegek tudását a rendszerről, csökken az egész rendszer komplexitása. A rétegek alapú architektúra adta lehetőségeket használhatjuk legacy szolgáltatások, komponensek szeparálására, illetve megkönnyíti az ezekről való átállást.

REST és HTTP

A REST szolgáltatások HTTP kérések fogadása és válaszok küldése útján bonyolítják le a kliens és szerver közti kommunikációt. A HTTP kérések és válaszok szemantikai jelentését az 5.2 ábrán láthatjuk.

2.4.3. BCrypt

A 90-es évek végén a mikroprocesszorok gyorsulása rohamosan növelte kibertámadások mögött álló számítási teljesítményt, míg számos autentikációs séma titkos, felhasználók által választott jelszavakon alapult, amelyek hossza és véletkenszerűsége közel konstans maradt. Niels Provos és David Mazières erre a problémára megoldásként egy jövőbiztos, a hardveres fejlődéssel lépést tartó jelszó sémát, és ennek részeként a BCrypt algoritmust [14].

Az algoritmust szándékosan lassúra és költségesre tervezték. Ez jó design-beli döntésnek bizonyult, mivel ezzel csökkenthető a *brute-force*, azaz nyers számításierőn alapuló támadások hatékonysága. A BCrypt által nyújtott védelem tovább növelhető azzal, ha az algoritmust többször lefuttatjuk, minden új körben az előzőből kapott értéket véve hash-elendő értéknek. Ezzel a technikával még tovább inkementálható a titkosítás hatékonysága.

Az projektben először SHA-256 algoritmussal titkosítottuk a felhasználók jelszavát. Az emögött álló érvek a gyors, hatékony működés, illetve az alacsony erőforrás igény voltak. A későbbiekben viszont pont ezekből az okokból kifolyólag, illetve mivel hatékonysága jelentős teljesítménybeli növekedést élvez, ha GPU-n implementáljuk. Ez a tulajdonsága jelentősen növeli a brute-force alapú támadások hatékonyságát [13]. A BCrypt algoritmus nem rendelkezik ezzel a tulajdonsággal, nehéz hatékonyan implementálni GPU-n. Azzal, hogy egy eleve lassabb, nem párhuzamosítható algoritmust használunk, amit könnyen tudunk skálázni a több körös titkosítással, nem rontjuk sem az alkalmazást rendeltetésszerűen használó felhasználók élményét, sem az alkalmazás hatékonyságát. Amennyiben egy jogosult személy kívánja magát autentikálni, az algoritmus futási ideje szinte elhanyagolható lesz.

2.4.4. Web Push API

Az alkalmazásban szeretnénk, hogyha az étterem kap valamilyen értesítést arról, amint új rendelés érkezik be, a kliens pedig, ehhez hasonlóan, kapjon valamilyen vizuálisan megjelenő visszajelzést a felhasználói felületen, ha az étterem feldolgozta a rendelést, vagy a rendelés elkészült; amennyiben pedig a felhasználók fizetni szeretnének, ezen szándékukat tudják jelezni oly módon, hogy az étterem azonnal értesüljön erről, és a tranzakció lebonyolításához szükséges minden információ rendelkezésükre álljon.

Ezen probléma megoldására valamilyen alkalmazáson belül eseménykezelő rendszert kell

létrehoznunk, ami alkalmas előre meghatározott üzenettípusok küldésére, ezekre való feliratkozásra (azaz bejövő üzenetek figyelésére), illetve ezen üzenetek kezelésére. Létrehoztunk egy sugárzó-feliratkozó (angolul *boradcaster-subscriber*) alapú event kezelő rendszert, ahol a feliratkozók egy callback megadásával képesek feliratkozni eventekre, a sugárzók pedig a kód bármely pontjából képesek eventeket sugározni, az event értelmezéséhez szükséges minden információval együtt.

Mivel ezek az eventek a szerveren jönnek létre, de hatásukat a klienseknél, azaz a felhasználói felületen fejtik ki, ezért megoldást kell találnunk arra, hogyan teremtsünk gyors kommunikációt a kettő között. Egy lehetséges megoldás a kliens oldali poll-ozás lenne. A kliens előre meghatároztt időközönként kérést indít a szerver felé valamilyen információért, például hogy feldolgozásra került-e egy rendelés, vagy elkészült-e, és a szervertől kapott válasz alapján cselekszik.

A probléma ezzel a megoldással, hogy nem hatékony. Amennyiben túl hosszú a kérés újraküldésének ideje, a kliens unreszponzívnak tűnhet, lassan reagál a szerver oldali változásokra. Amennyiben pedig ezen időintervallum túl rövid, a szerverünknek túl sok kérést kell kezelnie, és ezen kérések egy része "feleslges", hiszen nem történt állapotváltozás az előző kérés feldolgozása óta. Tpvábbá függetlenül attól, hogy milyen gyakran poll-ozunk, ez a folyamat jelentős hálózati forgalmat fog generálni és amennyiben a kapcsolat nem megfelelő erősségű, ez jelentősen ronthatja a felhasználói élményt és az alkalmazás hatékonyságát.

A megoldás abban rejlik, hogy a szerveren létrejövő változásokról (például egy rendelés állapota megváltozott) értesítést küldünk a klienseknek, akik ezt az értesítést fogadják és kezelik. Ezen rendszer létrehozásában a Vaadin által szolgáltatott *Push* funkcióra hagyatkoztunk, ami a *Push API*-on alapszik.

A Push API a W3C által előterjesztett megoldás [5], ami üzenetek küldését teszi lehetővé egy szerverről, még akkor is, amikor a fogadó fél, jellemzően egy web alkalmazás vagy felhasználói ágens inaktív. Ez a megoldás hatékonyan és megbízhatóan kézbesíti a küldött adatokat. A transzpotzációs folyamat technikai háttere az RFC 8030-ban definiált *web push protocol*-ra épül. A protokoll célzottan a valós idejű event-ek hatékony, erőforráskímélő transzportációjára lett megalkotva [16].

3 Összefoglaló

Összefoglaló

4 Irodalomjegyzék

[1] *@Converter dokumentációja*. 2022. URL: https://jakarta.ee/specifications/persistence/2.2/apidocs/javax/persistence/converter.

[2] *@NoArgsConstructor, @RequiredArgsConstructor, @AllArgsConstructor dokumentációja*. 2022. URL: https://projectlombok.org/features/constructor.

[3] *@Repository dokumentációja*. 2022. URL: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Repository.html.

[4] Balaji Varanasi; Maxim Bartkov. *Spring REST: Building Java Microservices and Cloud Applications*. Apress, 2022. ISBN: 1484274768; 9781484274767.

[5] Peter Beverloo és Martin Thomson. *Push API*. W3C Working Draft. https://www.w3.org/TR/2022/WI push-api-20220323/. W3C, 2022. márc.

[6] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000, 76–86. old.

[7] Erich Gamma és tsai. "Elements of Reusable Object-Oriented Software". *Design Patterns. massachusetts: Addison-Wesley Publishing Company* (1995).

[8] Patrick AV Hall és Geoff R Dowling. "Approximate string matching". *ACM computing surveys (CSUR)* 12.4 (1980), 381–402. old.

[9] *JpaRepository dokumentációja*. 2022. URL: https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html.

[10] Robert C Martin, James Newkirk és Robert S Koss. *Agile software development: principles, patterns, and practices*. 2. köt. Prentice Hall Upper Saddle River, NJ, 2003.

[11] Frank H Mathis. “A generalized birthday problem”. *SIAM review* 33.2 (1991), 265–270. old.

[12] “Mozilla’s vision for the evolution of the Web”. (2022). URL: https://webvision.mozilla.org/full/.

[13] Rohan Patra és Sandip Patra. “Cryptography: A Quantitative Analysis of the Effectiveness of Various Password Storage Techniques”. *Journal of Student Research* 10.3 (2021).

[14] Niels Provos és David Mazieres. “A Future-Adaptable Password Scheme.” *USENIX Annual Technical Conference, FREENIX Track*. 1999. köt. 1999, 81–91. old.

[15] Robert Richards. “Representational state transfer (rest)”. *Pro PHP XML and web services*. Springer, 2006, 633–672. old.

[16] M. Thomson, E. Damaggio és B. Raymor. *Generic Event Delivery Using HTTP Push*. RFC 8030. RFC Editor, 2016. dec.

5 Függelék

5.1. táblázat: JpaRepository kiterjesztéséhez használható kulcsszavak

Kulcsszó	Minta	JPQL kódcsipet
And	findByLastnameAndFirstname	… where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2
Is,Equals	findByFirstname findByFirstnameIs findByFirstnameEquals	… where x.firstname = 1?
Between	findByStartDateBetween	… where x.startDate between 1? and ?2
LessThan	findByAgeLessThan	… where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	… where x.age <= ?1
GreaterThan	findByAgeGreaterThan	… where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	… where x.age >= ?1
After	findByStartDateAfter	… where x.startDate > ?1
Before	findByStartDateBefore	… where x.startDate < ?1
IsNull	findByAgeIsNull	… where x.age is null
IsNotNull,NotNull	findByAge(Is)NotNull	… where x.age not null
Like	findByFirstnameLike	… where x.firstname like ?1
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	… where x.lastname <> ?1

In	findByAgeIn(Collection<Age> ages)	... where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> age)	... where x.age not in ?1
True	findByActiveTrue()	... where x.active = true
False	findByActiveFalse()	... where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	... where UPPER(x.firstame) = UPPER(?1)

5.2. táblázat: HTTP kérések és szerepük REST szolgáltatásokban

HTTP kérés	Szerepe	HTTP válasz kód
GET	erőforrás reprezentációnak kinyerése	200: az erőforrás megtalálható a szerveren
		404: az erőforrás nem található meg
		400: a kérés formázása nem megfelelő
POST	új erőforrás létrehozása	201: az erőforrás létrejött a szerveren
		200, 204: a létrejött erőforrás nem azonosítható URI-val
PUT	létező erőforrás állapotának frissítése; ha az nem létezik a szolgáltatás dönthet létrehozásáról	201: új erőforrás jött létre
		200, 204: egy létező erőforrás frissítése sikeres volt
DELETE	erőforrás törlése	200: az erőforrás törölve lett
		202: a kérés elfogadásra került, feldolgozásra vár
		204: az erőforrás törölve lett, státuszáról nem érkezett információ a válaszban
PATCH	egy erőforrás részleges frissítése	nincs ajánlás