













samen sterk voor werk





# JPA met Hibernate








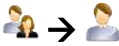
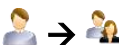




## Inhoudsopgave













<b>1</b>	<b>INLEIDING</b> 	<b>7</b>
1.1	Doelstelling.....	7
1.2	Vereiste voorkennis.....	7
1.3	Nodige software .....	7
1.4	ORM.....	7
1.5	Database.....	7
1.6	MySQL WorkBench.....	7
<b>2</b>	<b>JPA EN HIBERNATE INTEGREREN IN JE PROJECT</b> 	<b>8</b>
2.1	Eclipse Maven Project .....	8
2.2	StringUtils.....	9
2.3	Connection pool .....	9
2.4	persistence.xml .....	9
2.5	JPA initialiseren en de interface EntityManagerFactory .....	10
2.6	Samenvatting.....	11
2.7	De beginpagina.....	11
2.8	De website starten .....	11
2.9	Van EntityManagerFactory tot de database .....	12
<b>3</b>	<b>ENTITY</b> 	<b>13</b>
3.1	De database table en de bijbehorende entity class .....	13
3.2	Default constructor .....	14
3.3	Entity classes vermelden in persistence.xml.....	14
3.4	Extra JPA annotations.....	14
3.4.1	@Column .....	14
3.4.2	@Transient .....	14
3.4.3	@Temporal .....	14
<b>4</b>	<b>ENTITYMANAGER</b> 	<b>15</b>

4.1	Samenvatting.....	15
<b>5</b>	<b>ENTITY ZOEKEN VIA DE PRIMARY KEY</b>  .....	<b>16</b>
<b>6</b>	<b>ENUM</b>  .....	<b>18</b>
6.1	De enum .....	18
6.2	De enum voorstellen als een int kolom.....	18
6.3	De enum voorstellen als een varchar kolom of enum kolom .....	18
<b>7</b>	<b>JPA PROJECT</b> .....	<b>19</b>
7.1	Configuratie.....	19
7.2	Controle.....	20
7.3	De Project Explorer .....	20
7.4	De Data Source Explorer .....	20
<b>8</b>	<b>DALI</b> .....	<b>21</b>
8.1	Installatie.....	21
8.2	Gebruik .....	21
8.2.1	Grafische voorstelling .....	21
8.2.2	Code completion.....	21
<b>9</b>	<b>TRANSACTIES EN DE SERVICE LAYER</b>  .....	<b>22</b>
9.1	Transacties.....	22
9.2	Service layer .....	22
9.3	Voorbeeld .....	23
<b>10</b>	<b>ENTITY TOEVOEGEN</b>  .....	<b>24</b>
10.1	@GeneratedValue.....	24
10.2	Sequence .....	24
10.3	Voorbeeld.....	24
<b>11</b>	<b>ENTITY VERWIJDEREN</b>  .....	<b>28</b>
<b>12</b>	<b>ENTITY WIJZIGEN</b>  .....	<b>29</b>

<b>13</b>	<b>LEVENSCYCLI (LIFE CYCLE) VAN EEN ENTITY</b> 	<b>31</b>
<b>14</b>	<b>ENTITYMANAGER ALS THREADLOCAL VARIABLE</b> 	<b>32</b>
14.1	ThreadLocal variabele	32
14.2	EntityManager als ThreadLocal variabele	32
14.3	Repository layer	33
14.3.1	AbstractRepository	33
14.3.2	DocentRepository	34
14.4	De service layer	34
14.4.1	AbstractService	34
14.4.2	DocentService	34
14.5	CRUD operaties	35
<b>15</b>	<b>JPQL</b> 	<b>36</b>
15.1	Algemeen	36
15.2	Alle entities vragen	36
15.3	Sorteren	37
15.4	Selecteren	38
15.4.1	Positional parameters	38
15.4.2	Named parameters	39
15.5	Pagineren	40
15.6	Één kolom lezen	41
15.7	Meerdere kolommen lezen	42
15.8	Aggregate functions	43
15.9	group by	43
15.10	Named queries	45
15.10.1	Named queries in entity classes	45
15.10.2	Named query oproepen	45
15.10.3	Named queries in orm.xml	45
<b>16</b>	<b>BULK UPDATES EN BULK DELETES</b>	<b>47</b>
<b>17</b>	<b>INHERITANCE</b> 	<b>49</b>
17.1	Inheritance nabootsen in de database	49
17.2	Table per class hierarchy	49
17.2.1	Database	49

17.2.2	Voordelen van table per class hierarchy .....	49
17.2.3	Nadelen van table per class hierarchy .....	49
17.2.4	De base class .....	49
17.2.5	De afgeleide classses .....	50
17.2.6	Dali .....	50
17.2.7	Polymorphic queries .....	50
17.3	Table per subclass .....	52
17.3.1	Voordelen van table per subclass .....	53
17.3.2	Nadelen van table per subclass .....	53
17.3.3	Annotations bij de base class .....	53
17.3.4	Annotations bij de derived classes .....	53
17.4	Table per concrete class .....	54
17.4.1	Voordelen van table per concrete class .....	54
17.4.2	Nadelen van table per concrete class .....	54
17.4.3	Annotations .....	54
<b>18</b>	<b>VALUE OBJECTS </b> .....	<b>55</b>
18.1	Immutable value objecten .....	55
18.2	Database .....	57
18.3	JPA en value object classes .....	57
18.4	De value object class .....	57
18.5	De entity class die de value object class gebruikt .....	58
18.6	Service layer en repository layer .....	58
18.7	CRUD operaties .....	58
18.7.1	Create .....	58
18.7.2	Read .....	58
18.7.3	Update .....	58
18.7.4	Delete .....	59
18.8	JPQL .....	59
18.9	Value object classes detecteren .....	61
18.10	Aggregate .....	61
<b>19</b>	<b>EEN VERZAMELING VALUE OBJECTEN MET EEN BASISTYPE  </b> .....	<b>62</b>
19.1	De verzameling value objecten lezen uit de database .....	63
19.2	Value object toevoegen aan de verzameling .....	64
19.3	Value object verwijderen uit de verzameling .....	65
<b>20</b>	<b>EEN VERZAMELING VALUE OBJECTEN MET EEN EIGEN TYPE  </b> .....	<b>67</b>

20.1	De value object class .....	67
20.2	De verzameling value objecten in de entity class .....	68
20.3	Verzameling value objecten lezen uit de database.....	68
20.4	Value object toevoegen aan de verzameling .....	69
20.5	Value object verwijderen uit de verzameling .....	69
20.6	Een entity verwijderen .....	69
20.7	Een value object in de verzameling wijzigen.....	69
<b>21</b>	<b>MANY-TO-ONE ASSOCIATIE</b>  .....	<b>70</b>
21.1	De Java code.....	70
21.2	Eager loading.....	72
21.3	Lazy loading .....	72
<b>22</b>	<b>ONE-TO-MANY ASSOCIATIE</b>  .....	<b>73</b>
22.1	De associatie tussen de entity classes.....	73
22.2	De relatie tussen de bijbehorende tables .....	73
22.3	De Java code.....	73
22.4	One-to-many is lazy loading.....	73
22.5	De interface Set en de methods equals en hashCode .....	74
22.5.1	De methods equals en hashCode .....	74
<b>23</b>	<b>DE METHODS EQUALS EN HASHCODE LATEN GENEREREN</b> .....	<b>76</b>
<b>24</b>	<b>BIDIRECTIONELE ASSOCIATIE</b>  -  .....	<b>77</b>
24.1	De entity class aan de many kant van de associatie .....	77
24.2	De entity class aan de one kant van de associatie .....	77
24.3	De associatievariabelen bijwerken.....	77
<b>25</b>	<b>MANY-TO-MANY ASSOCIATIE</b>  -  .....	<b>80</b>
25.1	Database.....	80
25.2	Java code .....	80
25.3	Een bidirectionele @ManyToMany.....	82

<b>26</b>	<b>ONE-TO-ONE ASSOCIATIE</b>  .....	<b>83</b>
26.1	Database.....	83
26.2	Java.....	83
<b>27</b>	<b>ASSOCIATIES VAN VALUE OBJECTEN NAAR ENTITIES</b>  →  .....	<b>85</b>
<b>28</b>	<b>N + 1 PROBLEEM, JOIN FETCH QUERIES, ENTITY GRAPHS</b>  .....	<b>86</b>
28.1	N + 1 probleem.....	86
28.2	Join fetch .....	86
28.3	Entity graph .....	87
28.3.1	@NamedEntityGraph.....	87
28.3.2	De oproep van de named query .....	87
28.3.3	@NamedEntityGraphs .....	88
28.3.4	Meer dan één geassocieerde entity .....	88
28.3.5	De behoefte naar een geassocieerde entity van een geassocieerde entity.....	88
28.3.6	Code verbetering .....	88
<b>29</b>	<b>CASCADE</b>  .....	<b>89</b>
<b>30</b>	<b>ASSOCIATIES IN JPQL CONDITIES</b>  .....	<b>90</b>
30.1	Voorbeelden.....	90
30.2	Praktisch voorbeeld.....	90
<b>31</b>	<b>MULTI-USER EN RECORD LOCKING</b>  →  →  ←  ←  .....	<b>92</b>
31.1	Pessimistic record locking .....	92
31.2	Optimistic record locking .....	93
31.2.1	Versie kolom met een geheel getal .....	93
31.2.2	Versie kolom met een timestamp.....	95
31.3	Pessimistic record locking en optimistic record locking.....	96
<b>32</b>	<b>STAPPENPLAN</b>  .....	<b>97</b>
<b>33</b>	<b>HERHALINGSOEFENINGEN</b> .....	<b>99</b>
<b>34</b>	<b>COLOFON</b> .....	<b>100</b>

# 1 INLEIDING

## 1.1 Doelstelling

Je leert werken met JPA (Java Persistence API) versie 2.1.

JPA is een Java standaard waarmee je een relationele database aanspreekt.

Persistence betekent: een Java object opslaan als een record in een database.

## 1.2 Vereiste voorkennis

- Java
- SQL
- JDBC
- Servlets/JSP
- Maven

## 1.3 Nodige software

- Een JDK (Java Developer Kit) met versie 8 of hoger.
- Een JPA implementatie.

JPA is een specificatie: een document dat de werking van JPA beschrijft, een verzameling Java interfaces, Java annotations en utility classes.

Elke firma of organisatie kan de JPA specificatie implementeren.

Ze implementeren de JPA interfaces en verwerken de JPA annotations.

Er bestaan meerdere JPA implementaties. De belangrijkste zijn

- Hibernate (van de firma Red Hat)
- EclipseLink (van de organisatie Eclipse)
- Apache OpenJPA (van de organisatie Apache)

We gebruiken in de cursus de populairste implementatie: Hibernate.

- Een relationele database.  
Je kan met JPA de populaire relationele databases aanspreken.  
Je gebruikt in de cursus MySQL ([www.mysql.com](http://www.mysql.com)) en de MySQL Workbench.
- Een JDBC driver die hoort bij de relationele database.
- Tomcat 8.
- Eclipse IDE for Java EE Developers (versie Neon).

## 1.4 ORM


JPA is een ORM (object-relational mapping) library. Een ORM library helpt je om:

- Java objecten te bewaren als records in database tables.
- Records in database tables te lezen als Java objecten.

## 1.5 Database

De applicatie in de cursus werkt samen met de database fietsacademy.

Je maakt de database met het script `FietsAcademy.sql` uit het theoriemateriaal:

1. Je start de MySQL Workbench en je logt in op de Local instance.
2. Je kiest in het menu File de opdracht Open SQL Script en je opent `FietsAcademy.sql`.
1. Je voert dit script uit met de knop .

## 1.6 MySQL WorkBench

Deze cursus bevat enkele scripts die *alle* records van een table wijzigen.

Standaard weigert de MySQL WorkBench zo'n queries uit te voeren. Je wijzigt deze instelling:

- Je kiest in het menu Edit de opdracht Preferences.
- Je kiest links SQL Editor.
- Je verwijdert rechts het vinkje bij "Safe Updates" en je kiest OK.



## 2 JPA EN HIBERNATE INTEGREREN IN JE PROJECT

### 2.1 Eclipse Maven Project

1. Je kiest File, New, Dynamic Web Project.
2. Je tikt fietsacademy bij Project Name.
3. Je kiest 3.1 bij Dynamic web module version.
4. Je kiest <None> bij Target runtime en je kiest Next.
5. Je selecteert src bij Source folders on build path en je kiest Remove.  
Bij Maven is de naam van de source folder niet src.
6. Je kiest Add Folder, je tikt src/main/java en je kiest OK.  
src/main/java is de standaard Maven folder voor Java sources.
7. Je kiest Add Folder, je tikt src/main/resources en je kiest OK.  
src/main/resources is de standaard Maven folder voor configuratiebestanden.
8. Je tikt target/classes bij Default output folder en je kiest Next.  
target/classes is de standaard Maven folder voor gecompileerde bestanden.
9. Je tikt src/main/webapp bij Content Directory.  
src/main/webapp is de standaard Maven folder voor het webgedeelte van de website.
10. Je laat het vinkje afstaan bij Generate web.xml deployment descriptor.  
Je hebt web.xml niet nodig in de website. Je kiest Finish.
11. Je klikt met de rechtermuisknop op het project in de Project Explorer  
en je kiest Configure, Convert to Maven Project.
12. Je tikt be.vdab bij Group Id en je kiest Finish.
13. Je voegt aan pom.xml dependencies toe, voor </project>:

```
<dependencies>
  <dependency> <!-- servlets -->
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
  </dependency>
  <dependency> <!-- JSP's -->
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>jsp-api</artifactId>
    <version>[2.2,]</version>
    <scope>provided</scope>
  </dependency>
  <dependency> <!-- JSTL -->
    <groupId>jstl</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
    <scope>runtime</scope>
  </dependency>
  <dependency> <!-- Hibernate implementatie van JPA -->
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>[5.2.9, 5.2.99]</version>
  </dependency>
</dependencies>
```

14. Je slaat pom.xml op.

Je kopieert de inhoud van de theoriemateriaalfolder webapp naar src/main/webapp.

## 2.2 StringUtils

Je moet in de rest van de cursus regelmatig nagaan of een String naar een Long kan geconverteerd worden. Je moet soms ook nagaan of een String naar een BigDecimal kan geconverteerd worden.

Je maakt daartoe een package `be.vdab.util` en daarin de class `StringUtils`:

```
package be.vdab.util;
import java.math.BigDecimal;
public class StringUtils {
    public static boolean isLong(String string) {
        try {
            Long.parseLong(string);
            return true;
        } catch (NumberFormatException ex) {
            return false;
        }
    }
    public static boolean isBigDecimal(String string) {
        try {
            new BigDecimal(string);
            return true;
        } catch (NullPointerException | NumberFormatException ex) {
            return false;
        }
    }
}
```

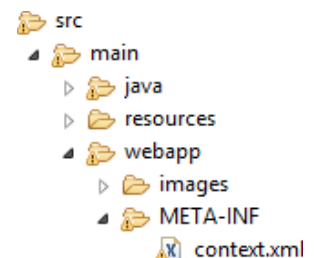
## 2.3 Connection pool

Java EE application servers en de meeste Java webserver (Tomcat, ...) bevatten een ingebakken connection pool. JPA kan die gebruiken.

Je plaatst het JAR bestand met de JDBC driver in de directory `lib` van Tomcat.

Je definieert in `src/main/webapp/META-INF` een connection pool in het bestand `context.xml`:

```
<?xml version='1.0' encoding='UTF-8'?>
<Context path='/fietsacademy'>
  <Resource name='jdbc/fietsacademy'
    type='javax.sql.DataSource'
    username='cursist' password='cursist'
    driverClassName='com.mysql.jdbc.Driver'
    url='jdbc:mysql://localhost/fietsacademy?useSSL=false'
    closeMethod='close'/>
</Context>
```

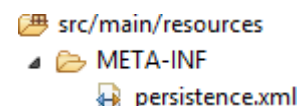


Deze connection pool is aanspreekbaar met de JNDI naam `java:/comp/env/jdbc/fietsacademy`.

## 2.4 persistence.xml

Je maakt in `src/main/resources` de folder `META-INF` en daarin `persistence.xml`. Dit is het configuratiebestand van JPA:

```
<?xml version='1.0' encoding='UTF-8'?>
<persistence version='2.1'
  xmlns='http://xmlns.jcp.org/xml/ns/persistence'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:schemaLocation='http://xmlns.jcp.org/xml/ns/persistence
    http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/persistence/persistence_2_1.xsd'>
  <persistence-unit name='fietsacademy'>
    <non-jta-data-source>
      java:/comp/env/jdbc/fietsacademy
    </non-jta-data-source>
```



```

<properties>
  <property name='hibernate.show_sql' value='true'/>
  <property name='hibernate.format_sql' value='true'/>
  <property name='hibernate.use_sql_comments' value='true'/>
</properties>
</persistence-unit>
</persistence>

```

③  
④  
⑤  
⑥

- (1) Er is een element persistence-unit per database die je aanspreekt.  
name bevat een unieke persistence-unit naam. Die mag verschillen van de database naam.
- (2) Je tikt bij non-jta-data-source de JNDI naam van de te gebruiken connection pool.
- (3) Het element properties omringt extra JPA eigenschappen.  
Elk element property beschrijft één eigenschap, met een name en een value.  
Alle JPA implementaties verwerken de standaard JPA eigenschappen.  
Hun naam begint met javax.persistence.  
Ons bestand bevat geen standaard JPA eigenschappen.  
Één JPA implementatie verwerkt eigenschappen waarvan het begin van de naam verschilt van javax.persistence. Enkel Hibernate verwerkt bijvoorbeeld eigenschappen waarvan de naam begint met hibernate. Dit zijn optionele eigenschappen, waarmee je bijvoorbeeld diagnostische informatie over SQL statements toont in logbestanden.
- (4) Als deze property true is, zie je als developer de SQL statements die Hibernate naar de database stuurt. Dit is interessant voor performance tuning.
- (5) Als deze property true is, zie je elk van de SQL statements bij (4) over meerdere regels, wat de leesbaarheid van zo'n SQL statement bevordert.
- (6) Als deze property true is, zie je commentaar in de SQL statements bij (4).  
Je krijgt met die commentaar inzicht waarom Hibernate de SQL statements uitvoert.

## 2.5 JPA initialiseren en de interface EntityManagerFactory

Je initialiseert JPA één keer in de applicatie, omdat deze initialisatie een eindje kan duren. JPA leest tijdens de initialisatie persistence.xml en onthoudt de informatie in een object. Dit object implementeert de interface EntityManagerFactory.

We noemen dit object de EntityManagerFactory.

- Je onthoudt de EntityManagerFactory in een static variabele.  
Zo blijft hij gedurende de ganse levensduur van de applicatie in het geheugen.
- De EntityManagerFactory is thread-safe.
- Je voert op het einde van de applicatie de EntityManagerFactory method close uit.  
Die method sluit de databaseconnecties in de connection pool.

Je maakt een package be.vdab.filters en daarin de class JPAFilter:

```

package be.vdab.filters;
// enkele imports ...
@WebFilter("*.htm")
public class JPAFilter implements Filter {
  private static final EntityManagerFactory entityManagerFactory
    = Persistence.createEntityManagerFactory("fietsacademy");
  @Override
  public void init(FilterConfig config) throws ServletException {
    // geen code nodig hier
  }
  @Override
  public void doFilter(ServletRequest request, ServletResponse response,
    FilterChain chain) throws ServletException, IOException {
    request.setCharacterEncoding("UTF-8");
    chain.doFilter(request, response);
  }
}

```

①  
②  
③

```

@Override
public void destroy() {
    entityManagerFactory.close();
}
}

```

④  
⑤

- (1) Deze variabele is de EntityManagerFactory.  
Een **static** variabele blijft in RAM gedurende de levensduur van de website.
- (2) Je maakt een EntityManagerFactory object met de static Persistence method `createEntityManagerFactory`.  
Je geeft de name mee van het element `persistence-unit` uit `persistence.xml`.
- (3) Je geeft aan dat request parameters uitgedrukt zijn in UTF-8.  
Je verwerkt zo request parameters met 'vreemde tekens' op een correcte manier.
- (4) De webserver roept deze method op bij het stoppen van de website.
- (5) Je sluit de EntityManagerFactory.

## 2.6 Samenvatting



De JPAFilter

bevat een



EntityManagerFactory.

## 2.7 De beginpagina

Je maakt een package `be.vdab.servlets`.

Je maakt daarin een servlet die GET requests verwerkt naar `index.htm`:

```

package be.vdab.servlets;
// enkele imports
@WebServlet("/index.htm")
public class IndexServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private static final String VIEW = "/WEB-INF/JSP/index.jsp";
    @Override
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        request.getRequestDispatcher(VIEW).forward(request, response);
    }
}

```

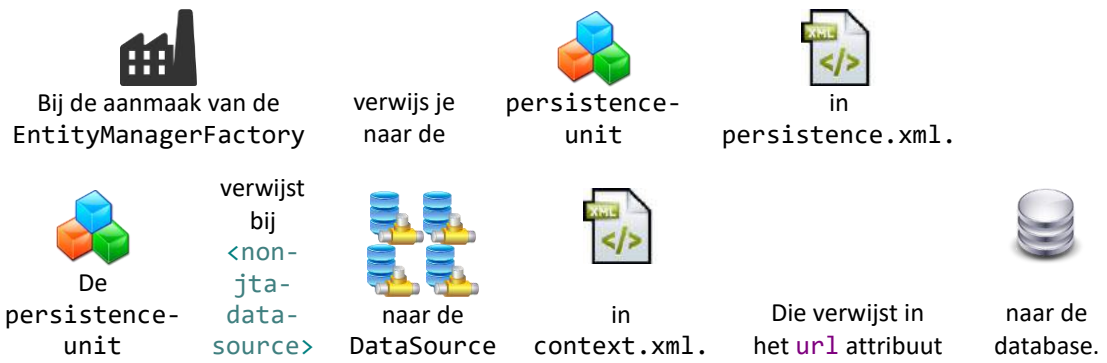
## 2.8 De website starten

Je start de website. Je ziet in het Eclipse venster Console diagnostische boodschappen, die beginnen met INFO. Boodschappen die beginnen met SEVERE wijzen op een mislukte JPA initialisatie. Mogelijke oorzaken:

- De database draait niet of bevat de database `fietsacademy` niet.
- `context.xml` niet op de juiste plaats (`src/main/webapp/META-INF`).  
Opgepast hierbij, je project heeft twee META-INF directories !
- Tikfouten in het hoofdlettergevoelige `context.xml`.
- `persistence.xml` niet op de juiste plaats (`src/main/resources/META-INF`).  
Opgepast hierbij, je project heeft twee META-INF directories !
- Tikfouten in het hoofdlettergevoelige `persistence.xml`.
- Tikfout in de string in volgende regel in JPAFilter:  

```
private static final EntityManagerFactory entityManagerFactory
    = Persistence.createEntityManagerFactory("fietsacademy");
```

## 2.9 Van EntityManagerFactory tot de database



Alles voor de keuken: zie takenbundel

### 3 ENTITY

Een entity class is een Java class die dezelfde data voorstelt als een table uit een database.  
Een object van een entity class heet een entity.

Je tikt in een entity class JPA annotations, die mapping informatie beschrijven.

Mapping informatie beschrijft de verbanden tussen de class en de bijbehorende database table.

#### 3.1 De database table en de bijbehorende entity class

De table docenten

De entity class Docent

docenten	
id	INT
voornaam	VARCHAR(30)
familienaam	VARCHAR(30)
wedde	DECIMAL(10,2)
rijksRegisterNr	BIGINT

<<entity>> Docent	
-id:	long
-voornaam:	String
-familienaam:	String
-wedde:	BigDecimal
-rijksRegisterNr:	long

De class bevat  
een private variabele  
per kolom uit de table docenten.

Je maakt een package `be.vdab.entities`. Je maakt daarin de entity class `Docent`:

```
package be.vdab.entities;
// enkele imports (vooral uit javax.persistence) ...
@Entity
@Table(name = "docenten")
public class Docent implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    private long id;
    private String voornaam;
    private String familienaam;
    private BigDecimal wedde;
    private long rijksRegisterNr;
    // je maakt getters voor de private variabelen, behalve voor serialVersionUID
    public String getNaam() {
        return voornaam + ' ' + familienaam;
    }
}
```

JPA stelt een aantal voorwaarden aan een entity class:

- (1) Je tikt `@Entity` juist voor de entity class.
- (2) Je tikt `@Table` voor de entity class, met de table naam.  
Je mag `@Table` weglaten als de table naam gelijk is aan de class naam.
- (3) JPA raadt aan dat de class `Serializable` implementeert.  
Dit is niet noodzakelijk voor de samenwerking met de database.  
Het is wel noodzakelijk als je objecten via serialization naar een binair bestand zou wegschrijven of over het netwerk zou transporteren met serialization.
- (4) Je tikt `@Id` voor de private variabele die hoort bij de primary key kolom.
- (5) JPA associeert een private variabele met een table kolom met dezelfde naam.  
JPA associeert dus de variabele `voornaam` met een kolom `voornaam`.

### 3.2 Default constructor

Een entity class moet een default constructor (een constructor zonder parameters) hebben. JPA roept die default constructor op als JPA een entity maakt op basis van een gelezen record. Bij Docent maakt de compiler een default constructor, omdat je zelf geen constructors schreef. Als je liever hebt dat zo weinig mogelijk classes deze constructor kunnen gebruiken, maak je die constructor **protected** in plaats van **public**.

### 3.3 Entity classes vermelden in persistence.xml

Je vermeldt elke entity class in persistence.xml. Je tikt voor <properties>:

```
<class>be.vdab.entities.Docent</class>
```



Opmerking: je moet bij sommige JPA implementaties (zoals Hibernate) entity classes niet vermelden in persistence.xml. We doen dit toch, zodat je eventueel een andere JPA implementatie kan uitproberen.

### 3.4 Extra JPA annotations

Je kan ook volgende annotations nodig hebben.

#### 3.4.1 @Column

Je tikt @Column voor een private variabele als de naam van de bijbehorende table kolom verschilt van de naam van die private variabele.

Je tikt bij de parameter name de naam van de kolom: @Column(name = "kolomnaam")

#### 3.4.2 @Transient

Je tikt @Transient voor een private variabele als die private variabele geen bijbehorende kolom heeft in de database table.

#### 3.4.3 @Temporal

Je tikt @Temporal voor een private Date variabele.

Je geeft als parameter het type aan van de bijbehorende table kolom.

@Temporal annotation	Kolomtype	Kolominhoud
@Temporal(TemporalType.DATE)	Date	enkel een datum
@Temporal(TemporalType.TIME)	Time	enkel een tijd
@Temporal(TemporalType.TIMESTAMP)	DateTime	een datum én een tijd



Opmerking: je moet @Temporal niet tikken bij de types LocalDate, LocalTime en LocalDateTime uit Java 8.



Artikel: zie takenbundel

## 4 ENTITYMANAGER

EntityManager is een interface uit JPA. Je gebruikt een EntityManager object om:

- Entities als nieuwe records toe te voegen aan de database.
- Records te lezen als entities.
- Records te wijzigen die bij entities horen.
- Records te verwijderen die bij entities horen.
- Transacties te beheren.

Een EntityManager heeft volgende eigenschappen:

- Je maakt hem aan met de EntityManagerFactory method `createEntityManager`.
- Een EntityManager is niet thread-safe.
- Nadat je een EntityManager gebruikte, voer je zijn `close` method uit in een **finally** blok. De `close` method sluit de databaseconnectie die de EntityManager gebruikte. Als je de `close` method niet oproept, zijn zeer snel alle connecties van de connection pool “in gebruik” en blokkeert de connection pool !

Je hebt op meerdere plaatsen in de website een EntityManager nodig.

Je voegt daartoe een static method toe aan JPAFilter:

```
public static EntityManager getEntityManager() {
    return entityManagerFactory.createEntityManager();
}
```

Gezien de method static is, kan je ze overal in de website oproepen als:

```
EntityManager entityManager = JPAFilter.getEntityManager();
```

### 4.1 Samenvatting



De EntityManagerFactory

maakt een



EntityManager.

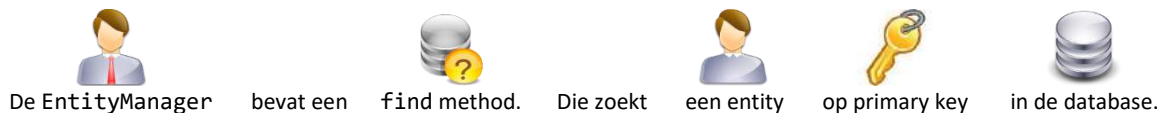
Die beheert



Entities.



## 5 ENTITY ZOEKEN VIA DE PRIMARY KEY



Je geeft aan de find method twee parameters mee:

- De entity class die hoort bij de table waarin je een record zoekt.
- De primary key waarde die je zoekt.

Het returntype van de find method is de entity class die je als eerste parameter meegaf.

- Als het record bestaat, geeft de find method je een entity terug.  
JPA vulde de private variabelen van die entity met de bijbehorende kolomwaarden.
- Als het record niet bestaat, geeft de find method **null** terug.

Je maakt als voorbeeld een pagina waarin de gebruiker een docentnummer intikt.

Jij zoekt daarna de bijbehorende docent en je toont de docent data.

Je maakt de class DocentRepository:

```
package be.vdab.repositories;
// enkele imports ...
public class DocentRepository {
    public Optional<Docent> read(long id) {
        EntityManager entityManager = JPAFilter.getEntityManager();
        try {
            return Optional.ofNullable(entityManager.find(Docent.class, id));
        } finally {
            entityManager.close();
        }
    }
}
```

①  
②

- (1) Je vraagt een EntityManager aan de servlet filter JPAFilter.
- (2) Je zoekt een Docent entity op de primary key met de find method.  
De 1° parameter is het type van de op te zoeken entity.  
De 2° parameter is de primary key waarde van de op te zoeken entity.

Je maakt een package be.vdab.servlets.docenten.

Als je veel servlets hebt, is het onoverzichtelijk alle servlets in één package te plaatsen

- Je maakt daarom in deze applicatie een package be.vdab.servlets.docenten met daarin enkel servlets die vooral met docenten werken.
- Je maakt verder in de cursus een package be.vdab.servlets.campussen met daarin de servlets die vooral met campussen werken.

ZoekenServlet verwerkt GET requests naar /docenten/zoeken.htm:

```
package be.vdab.servlets.docenten;
// enkele imports ...
@WebServlet("/docenten/zoeken.htm")
public class ZoekenServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private static final String VIEW = "/WEB-INF/JSP/docenten/zoeken.jsp";
    private final transient DocentRepository docentRepository
        = new DocentRepository();
}
```

```

@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    if (request.getQueryString() != null) {
        String id = request.getParameter("id");
        if (StringUtils.isLong(id)) {
            docentRepository.read(Long.parseLong(id))
                .ifPresent(docent -> request.setAttribute("docent", docent));
        } else {
            request.setAttribute("fouten",
                Collections.singletonMap("id", "tik een getal"));
            // singletonMap maakt intern een Map met één entry (key=id,
            // value=tik een getal) en geeft die Map terug als returnwaarde
        }
    }
    request.getRequestDispatcher(VIEW).forward(request, response);
}
}

```

Je maakt zoeken.jsp in src/main/webapp/WEB-INF/JSP/docenten:

```

<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%>
<%@taglib uri='http://java.sun.com/jsp/jstl/core' prefix='c'%>
<%@taglib uri='http://java.sun.com/jsp/jstl/fmt' prefix='fmt'%>
<%@taglib uri='http://vdab.be/tags' prefix='v' %>
<!doctype html>
<html lang='nl'>
<head>
<v:head title='${empty docent ? "Docent zoeken" : docent.naam}'/>
</head>
<body>
    <v:menu/>
    <h1>Docent zoeken</h1>
    <form>
        <label>Nummer:<span>${fouten.id}</span>
        <input name='id' value='${param.id}'
            required autofocus type='number' min='1'></label>
        <input type='submit' value='Zoeken'>
    </form>
    <c:if test='${not empty param and empty fouten and empty docent}'>
        Docent niet gevonden
    </c:if>
    <c:if test='${not empty docent}'>
        ${docent.naam}, wedde: &euro; <fmt:formatNumber value='${docent.wedde}'/>
    </c:if>
</body>
</html>

```

Je commit de sources en je publiceert op GitHub.

Je kan Docenten, Zoeken op nummer uitproberen.



Zoeken op nummer: zie takenbundel

## 6 ENUM

Het type van een private variabele van een entity class kan een enum zijn.

Je leert hier hoe JPA die enum in de database vertaalt naar:

- het kolomtype int
- of het kolomtype varchar (of enum bij databases die dit ondersteunen, zoals MySQL)

### 6.1 De enum

Je maakt een package `be.vdab.enums`. Je maakt daarin een enum `Geslacht`:

```
package be.vdab.enums;
public enum Geslacht {
    MAN, VROUW
}
```

Je houdt per docent het geslacht bij, met een private variabele in `Docent`:

```
private Geslacht geslacht; // je maakt een getter voor de variabele
```

### 6.2 De enum voorstellen als een int kolom

JPA gaat er standaard van uit dat bij een enum een int kolom hoort:

- De kolom waarde 0 hoort bij de eerste beschreven waarde in de enum `Geslacht`: `MAN`.
- De kolom waarde 1 hoort bij de tweede beschreven waarde: `VROUW`.
- ...

Je voert het script `GeslachtAlsInt.sql` uit.

Dit voegt aan de table `docenten` een int kolom `geslacht` toe en vult die met 0 (bij mannen), of 1 (bij vrouwen).

 `geslacht INT`

Als JPA een record omzet naar een `Docent` entity, vult JPA de variabele `geslacht`

- met `MAN` (als de kolom 0 bevat)
- of `VROUW` (als de kolom 1 bevat)

Je toont het teken ♂ of ♀ met een regel in `zoeken.jsp` na `<c:if>`:

```
${docent.geslacht == 'MAN' ? '&#x2642;' : '&#x2640;'}>
```

Je kan [Docenten](#), [Zoeken op nummer](#) uitproberen.

### 6.3 De enum voorstellen als een varchar kolom of enum kolom

Je voert het script `GeslachtAlsString.sql` uit.

Dit wijzigt het type van de kolom `geslacht` naar enum en vult de kolom met `MAN` (bij mannen) of `VROUW` (bij vrouwen).

 `geslacht ENUM('MAN','VROUW')`

Je tik in `Docent` `@Enumerated` voor de variabele `geslacht`

```
@Enumerated(EnumType.STRING)
private Geslacht geslacht;
```

Je geeft zo aan dat bij de variabele een varchar kolom hoort:

- De kolom waarde `MAN` hoort bij de enum waarde `MAN`.
- De kolom waarde `VROUW` hoort bij de enum waarde `VROUW`.

Je commit de sources en je publiceert op GitHub.

Je kan [Docenten](#), [Zoeken op nummer](#) uitproberen.



## 7 JPA PROJECT

### 7.1 Configuratie

Je tikt JPA annotations bij entity classes.

Bij het opstarten van je applicatie controleert JPA of de nodige annotations aanwezig zijn en of de annotation parameters correct zijn. JPA werpt een exception als er fouten zijn.

Je zal nu je project converteren naar een JPA project. Eclipse controleert dan reeds bij het tikken van je Java code of de nodige JPA annotations aanwezig zijn en of de annotation parameters correct zijn. Dit is handiger dan exceptions bij het starten van de applicatie.

1. Je klikt met de rechtermuisknop op het project in de Project Explorer en je kiest Configure, Convert to JPA Project.
2. Je kiest Next.
3. Je kiest EclipseLink 2.5.x bij Platform.
4. Je kiest User Library bij Type.
5. Je klikt op de knop  (Download Library ...).
6. Je kiest EclipseLink 2.5.2 en je kiest Next.
7. Je plaats een vinkje bij I accept the terms of this license en je kiest Finish.  
Opmerking: je mag de stappen 5 tot en met 7 bij een volgend project overslaan.
8. Je verwijdert het vinkje bij Include libraries with this application.  
Je wil de EclipseLink libraries die de annotations controleren niet opnemen in je applicatie.  
Je applicatie zou twee JPA implementaties bevatten: Hibernate en EclipseLinks.  
De kans dat die met mekaar in conflict komen is groot.
9. Je definieert een verbinding naar de database met Add connection...
10. Je kiest MySQL bij Connection Profile Types.
11. Je tikt een naam voor de databaseverbinding bij Name: fietsacademy en je kiest Next.
12. Je moet aangeven met welke JDBC driver Eclipse de database kan openen.  
Je klikt daartoe rechts op de knop  (New Driver Definition).
13. Je kiest MySQL JDBC Driver 5.1 en je kiest het tabblad JAR List.
14. Je kiest het jar bestand in de lijst en je kiest de knop Edit JAR/Zip.
15. Je duidt het JAR bestand aan met de JDBC driver voor MySQL en je kiest Open.  
Je hebt dit bestand normaal al in de lib folder van je Tomcat folder geplaatst.
16. Je kies OK.  
Opmerking: je mag de stappen 12 tot en met 16 bij een volgend project overslaan.
17. Je tikt fietsacademy bij Database.
18. Je vervangt bij URL het woord database door fietsacademy.
19. Je tikt het paswoord van de gebruiker root bij Password.
20. Je plaatst een vinkje bij Save password.  
Eclipse zal dan niet voortdurend dit paswoord opnieuw vragen.
21. Je plaatst een vinkje bij Connect every time the workbench is started.
22. Je kiest de knop Test Connection.  
Je ziet de boodschap Ping succeeded als Eclipse een databaseverbinding kan maken.  
Anders zie je de boodschap Ping failed en moet je de invoervakken corrigeren.
23. Je kiest de knop Finish.
24. Je kiest Annotated classes must be listed in persistence.xml.
25. Je kiest Finish.

## 7.2 Controle

Je ziet nu met twee voorbeelden hoe Eclipse de JPA annotations controleert:

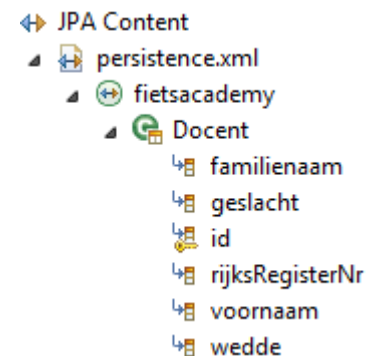
- Je plaatst in Docent de regel `@Id` in commentaar en je slaat de source op. Je ziet bij `@Entity` een foutmelding `The entity has no primary key attribute defined.` Deze foutmelding verdwijnt als je `@Id` uit commentaar haalt en de source opslaat.
- Je wijzigt docenten in `@Table` naar dosenten en je slaat de source op. Je ziet bij `@Table` een foutmelding `Table "dosenten" cannot be resolved.` Deze foutmelding verdwijnt als je dosenten wijzigt naar docenten en de source opslaat.

## 7.3 De Project Explorer

Je project bevat in de Project Explorer een extra onderdeel JPA Content.

Als je dit onderdeel openklapt zie je de entity class(es) die je registreerde in `persistence.xml`, met de class attributen. Het attribuut dat hoort bij de primary key krijgt een apart icoon.

Als je een class of attribuut dubbelklikt, opent Eclipse de source van deze class (tenzij de source al open stond).



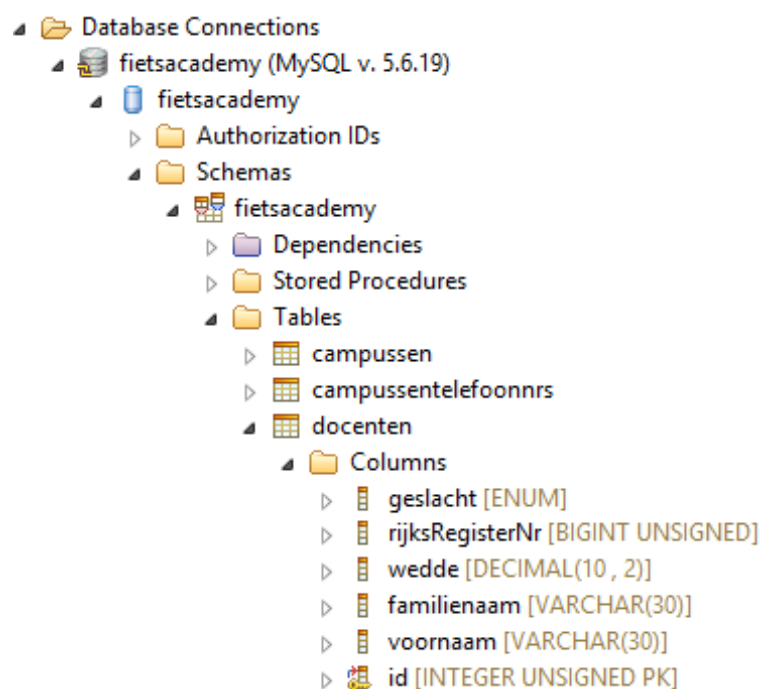
## 7.4 De Data Source Explorer

Je kan de database ook inkijken met Eclipse, in het venster Data Source Explorer (tabbladen onder in Eclipse).

Als je Database Connections openklapt, zie je de database(s) waarvoor je een connectie definieerde, hun tabellen en kolommen.

Je kan de data van een tabel ook zien en bijwerken door met de rechtermuisknop te klikken op de table en de opdracht Data, Edit te kiezen.

Je bewaar de wijzigingen met de knop  in de Eclipse toolbar.



JPA project: zie takenbundel

## 8 DALI

Dali is een plugin voor Eclipse waarmee je grafische voorstellingen krijgt van je entity classes.

### 8.1 Installatie

1. Je kiest in het menu Help de opdracht Install New Software.
2. Je kiest bij Work with voor --All Available Sites --.
3. Je tikt dali in het tekstvak 'type filter tekst' en je drukt Enter.
4. Je plaatst een vinkje bij Dali Java Persistence Tools - JPA Diagram Editor.
5. Je kiest Next.
6. Je kiest Next.
7. Je kiest I accept the terms of the licence agreement en je kiest Finish.
8. Je herstart Eclipse.

### 8.2 Gebruik

#### 8.2.1 Grafische voorstelling

1. Je klikt met de rechtermuisknop op het project onderdeel JPA Content in de Project Explorer en je kiest de opdracht Open Diagram.
2. Je klikt in de achtergrond van het diagram en je kiest de opdracht Show all Persistent Types.
3. Je ziet een grafische voorstelling van de entity class Docent.



4. Je slaat het diagram op.

#### 8.2.2 Code completion

Dali helpt je ook bij het tikken van tabelnamen in je Java classes. Je probeert dit uit:

1. Je verwijdert in Docent bij @Table het woord docenten.
2. Je plaats de cursor tussen de dubbele aanhalingstekens.
3. Je drukt Ctrl + spatie.
4. Je ziet een lijst met tabelnamen uit de database.
5. Je kiest docenten.

## 9 TRANSACTIES EN DE SERVICE LAYER

### 9.1 Transacties

Je mag bij JPA records lezen in een transactie, maar je mag dit ook zonder transactie.

Je moet records toevoegen, wijzigen of verwijderen in een transactie.

Zoniet werpt JPA een `TransactionRequiredException`.

De JPA interface `EntityManagerTransaction` stelt een transactie voor.

Elke transactie is geassocieerd met een `EntityManager`.

Je beheert de transactie van een `EntityManager` met de method `getTransaction`:

- transactie starten `entityManager.getTransaction().begin();`
- commit `entityManager.getTransaction().commit();`
- rollback `entityManager.getTransaction().rollback();`

Nadat je op een `EntityManager` een transactie start, behoren de records die je met die `EntityManager` leest, toevoegt, wijzigt of verwijdert tot die transactie.



De `EntityManager` bevat een `getTransaction` method.  Deze beheert transacties.

### 9.2 Service layer

Transactiebeheer in een servlet is niet ideaal. Een servlet bevat al veel code: request parameters valideren, request attributen aanmaken, de JSP oproepen ...

Transactiebeheer in een repository class is ook niet ideaal.

Één repository method class stelt één database opdracht voor, niet een groep opdrachten.

Een voorbeeld: geld overschrijven van één rekening naar een andere rekening.

1. Je roept de `RekeningRepository` method `update` op.  
Die verlaagt in de table rekeningen het saldo van de ene rekening.
2. Je roept de `RekeningRepository` method `update` nog eens op.  
Die verhoogt in de table rekeningen het saldo van de andere rekening.
3. Je roept de `VerrichtingRepository` method `create` op.  
Die voegt een record toe aan de table verrichtingen.

Deze handelingen moeten tot één transactie behoren:

als één handeling mislukt, moet de database al deze handelingen ongedaan maken.

De repository methods mogen daarom niet elk een aparte transactie starten.

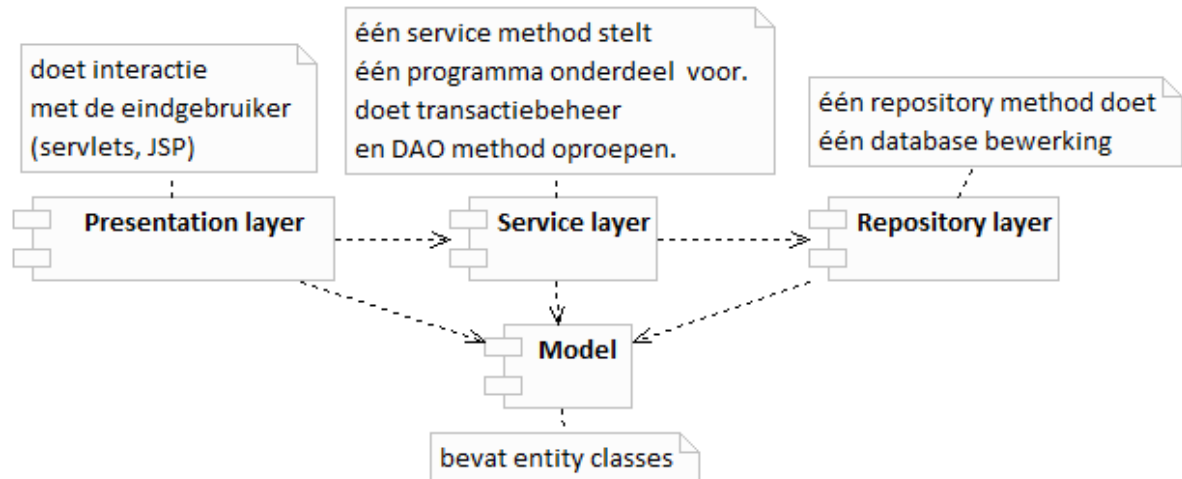
Java ontwikkelaars doen het transactiebeheer in een extra soort classes: service classes.

Er is één service class per entity class.

Je doet volgende stappen om een use case (programma onderdeel) uit te voeren

1. Je roept in de servlet methods `doGet` of `doPost` een service method op.
2. Die service method maakt een `EntityManager` en start daarop een transactie.
3. De service method roept één of meerdere repository methods op.  
Die doen hun handelingen binnen de transactie die de service method gestart heeft.  
Ze krijgen daartoe de `EntityManager` van de service method mee als parameter.
4. De service method doet een commit of rollback op de transactie.

Zo'n applicatie bevat meerdere layers (lagen):



De presentation layer roept per use case de service layer op, die de repository layer en de entities oproept.

### 9.3 Voorbeeld

DocentZoekenServlet gebruikt DocentService, die op zijn beurt DocentRepository gebruikt.

De service layer lijkt in dit voorbeeld overbodig. De use case kan later vergroten.

Je leest bijvoorbeeld data uit meerdere tables om één docent te tonen.

De service method roept dan methods uit meerdere repository classes op.

Je maakt DocentService in een package `be.vdab.services`:

```

package be.vdab.services;
// enkele imports ...
public class DocentService {
    private final DocentRepository docentRepository = new DocentRepository(); ❶
    public Optional<Docent> read(long id) {
        EntityManager entityManager = JPAFilter.getEntityManager();             ❷
        try {
            return docentRepository.read(id, entityManager);                    ❸
        } finally {
            entityManager.close();                                             ❹
        }
    }
}

```

- (1) DocentService gebruikt DocentRepository.
- (2) Je vraagt een EntityManager aan de servlet filter JPAFilter.
- (3) Je roept de repository layer op en geeft de EntityManager mee.
- (4) Je sluit de EntityManager.

Je wijzigt in DocentRepository de method `read`:

```

public Optional<Docent> read(long id, EntityManager entityManager) {
    return Optional.ofNullable(entityManager.find(Docent.class, id));
}

```

Je vervangt in ZoekenServlet

- DocentRepository door DocentService
- docentRepository door docentService

Je commit de sources en je publiceert op GitHub.

Je kan [Docenten, Zoeken op nummer](#) uitproberen.



## 10 ENTITY TOEVOEGEN



De EntityManager bevat een persist method. Die voegt een entity toe aan de database.

Je geeft aan de method persist de toe te voegen entity mee als parameter.

De method stuurt een SQL insert statement naar de database.

### 10.1 @GeneratedValue

Je tikt @GeneratedValue voor de private variabele id in de class Docent.

Je geeft hiermee aan dat de database (en niet je applicatie) de bijbehorende kolom invult.

Als de database hierbij autonummering gebruikt (zoals hier het geval is), plaats je de parameter strategy op IDENTITY: @GeneratedValue(strategy=GenerationType.IDENTITY)

JPA vult, na het toevoegen van een nieuw record,

de private variabele id met het getal in de autonumber kolom id.

### 10.2 Sequence

Sommige databasemerken, zoals Oracle, kennen geen autonumber kolom, maar wel een sequence.

Dit is ook een automatische nummering, bijgehouden door de database. Je kan deze nummering gebruiken om de primary key van nieuwe records in één of meerdere tables in te vullen.

Je kan bijvoorbeeld één sequence delen door de table klanten en de table leveranciers.

Als je een klant toevoegt, daarna een leverancier en daarna nog een klant,

krijgt de 1° klant het nummer 1, de leverancier het nummer 2 en de 2° klant het nummer 3.

MySQL kent geen sequence.

Je maakt met volgend SQL statement een sequence met de naam klantleverancierid:

**create sequence** klantleverancierid

Je voegt met volgend SQL statement een nieuw record toe aan de tabel klanten:

**insert into** klanten(id, naam)

**values** (klantleverancierid.nextval, 'Smits')

❶

(1) Als je op een sequence nextval uitvoert, geeft die sequence je een volgend volgnummer.

Je vervangt in je entity class dan GeneratedValue(strategy=GenerationType.IDENTITY) door

@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="myGenerator")

❶

@SequenceGenerator(name="myGenerator", sequenceName="klantleverancierid")

❷

(1) Je wijzigt de strategy naar Sequence en je tikt bij generator een vrij te kiezen naam.

(2) Je herhaalt deze naam bij name en je tikt bij sequenceName de naam van de sequence in de Oracle database.

### 10.3 Voorbeeld

Je maakt als voorbeeld een onderdeel waarmee de gebruiker een docent kan toevoegen.

Je voegt een method toe aan DocentRepository:

```
public void create(Docent docent, EntityManager entityManager) {
    entityManager.persist(docent);
}
```

Je voegt een method toe aan DocentService:

```
public void create(Docent docent) {
    EntityManager entityManager = JPAFilter.getEntityManager();
    entityManager.getTransaction().begin();
    try {
        docentRepository.create(docent, entityManager);
        entityManager.getTransaction().commit();
    }
```

```

    } catch (PersistenceException ex) {
        entityManager.getTransaction().rollback();
        throw ex;
    } finally {
        entityManager.close();
    }
}

```

Je voegt code toe aan Docent:

```

public Docent(String voornaam, String familienaaam, BigDecimal wedde,
    Geslacht geslacht, long rijksRegisterNr) {
    setVoornaam(voornaam);
    setFamilienaaam(familienaaam);
    setWedde(wedde);
    setGeslacht(geslacht);
    setRijksRegisterNr(rijksRegisterNr);
}

protected Docent() {}// default constructor is vereiste voor JPA

public static boolean isVoornaamValid(String voornaam) {
    return voornaam != null && ! voornaam.trim().isEmpty();
}

public static boolean isFamilienaaamValid(String familienaaam) {
    return familienaaam != null && ! familienaaam.trim().isEmpty();
}

public static boolean isWeddeValid(BigDecimal wedde) {
    return wedde != null && wedde.compareTo(BigDecimal.ZERO) >= 0;
}

public static boolean isRijksRegisterNrValid(long rijksRegisterNr) {
    long getal = rijksRegisterNr / 100;
    if (rijksRegisterNr / 1_000_000_000 < 50) {
        getal += 2_000_000_000;
    }
    return rijksRegisterNr % 100 == 97 - getal % 97;
}

public void setVoornaam(String voornaam) {
    if ( ! isVoornaamValid(voornaam)) {
        throw new IllegalArgumentException();
    }
    this.voornaam = voornaam;
}

public void setFamilienaaam(String familienaaam) {
    if ( ! isFamilienaaamValid(familienaaam)) {
        throw new IllegalArgumentException();
    }
    this.familienaaam = familienaaam;
}

public void setWedde(BigDecimal wedde) {
    if ( ! isWeddeValid(wedde)) {
        throw new IllegalArgumentException();
    }
    this.wedde = wedde;
}

public void setGeslacht(Geslacht geslacht) {
    this.geslacht = geslacht;
}

public void setRijksRegisterNr(long rijksRegisterNr) {
    if ( ! isRijksRegisterNrValid(rijksRegisterNr)) {
        throw new IllegalArgumentException();
    }
    this.rijksRegisterNr = rijksRegisterNr;
}

```

```
}
```

ToevoegenServlet verwerkt GET en POST requests naar /docenten/toevoegen.htm:

```
package be.vdab.servlets.docenten;
// enkele imports ...
@WebServlet("/docenten/toevoegen.htm")
public class ToevoegenServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private static final String VIEW = "/WEB-INF/JSP/docenten/toevoegen.jsp";
    private static final String REDIRECT_URL = "%s/docenten/zoeken.htm?id=%d";
    private final transient DocentService docentService = new DocentService();
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        request.getRequestDispatcher(VIEW).forward(request, response);
    }
    @Override
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        Map<String, String> fouten = new HashMap<>();
        String voornaam = request.getParameter("voornaam");
        if ( ! Docent.isVoornaamValid(voornaam)) {
            fouten.put("voornaam", "verplicht");
        }
        String familienaam = request.getParameter("familienaam");
        if ( ! Docent.isFamilienaamValid(familienaam)) {
            fouten.put("familienaam", "verplicht");
        }
        String weddeString = request.getParameter("wedde");
        BigDecimal wedde = null;
        if (StringUtils.isBigDecimal(weddeString)) {
            wedde = new BigDecimal(weddeString);
            if ( ! Docent.isWeddeValid(wedde)) {
                fouten.put("wedde", "tik een positief getal of 0");
            }
        } else {
            fouten.put("wedde", "tik een positief getal of 0");
        }
        String geslacht = request.getParameter("geslacht");
        if (geslacht == null) {
            fouten.put("geslacht", "verplicht");
        }
        String rijksRegisterNrString = request.getParameter("rijksregisternr");
        long rijksRegisterNr = 0;
        if (StringUtils.isLong(rijksRegisterNrString)) {
            rijksRegisterNr = Long.parseLong(rijksRegisterNrString);
            if ( ! Docent.isRijksRegisterNrValid(rijksRegisterNr)) {
                fouten.put("rijksregisternr", "verkeerde cijfers");
            }
        } else {
            fouten.put("rijksregisternr", "verkeerde cijfers");
        }
        if (fouten.isEmpty()) {
            Docent docent =
                new Docent(voornaam, familienaam, wedde, Geslacht.valueOf(geslacht),
                    rijksRegisterNr);
            docentService.create(docent);
            response.sendRedirect(response.encodeRedirectURL(String.format(
                REDIRECT_URL, request.getContextPath(), docent.getId())));
        }
    }
}
```

```

    } else {
        request.setAttribute("fouten", fouten);
        request.getRequestDispatcher(VIEW).forward(request, response);
    }
}
}

```

Je maakt toevoegen.jsp in WEB-INF/JSP/docenten:

```

<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%>
<%@taglib uri='http://java.sun.com/jsp/jstl/core' prefix='c'%>
<%@taglib uri='http://vdab.be/tags' prefix='v' %>
<!doctype html>
<html lang='nl'>
    <head>
        <v:head title='Docent toevoegen'/>
    </head>
    <body>
        <v:menu/>
        <h1>Docent toevoegen</h1>
        <form method='post' id='toevoegform'>
            <label>Voornaam:<span>${fouten.voornaam}</span>
            <input name='voornaam' value='${param.voornaam}'
                autofocus required></label>
            <label>Familiennaam:<span>${fouten.familiennaam}</span>
            <input name='familiennaam' value='${param.familiennaam}' required></label>
            <label>Wedde:<span>${fouten.wedde}</span>
            <input name='wedde' value='${param.wedde}' required type='number'
                min='0' step='0.01'></label>
            <div><label><span>${fouten.geslacht}</span>
                <input type='radio' name='geslacht' value='MAN'
                    ${param.geslacht=='MAN' ? 'checked' : ''}>Man</label></div>
            <div><label><input type='radio' name='geslacht' value='VROUW'
                ${param.geslacht=='VROUW' ? 'checked' : ''}>Vrouw</label></div>
            <label>Rijksregisternummer:<span>${fouten.rijksregisternr}
            </span>
            <input name='rijksregisternr' value='${param.rijksregisternr}' required
                type='number' min='10000000000' max='99999999999'></label>
            <input type='submit' value='Toevoegen' id='toevoegknop'>
        </form>
        <script>
            document.getElementById('toevoegform').onsubmit = function() {
                document.getElementById('toevoegknop').disabled = true;
            };
        </script>
    </body>
</html>

```

Je commit de sources en je publiceert op GitHub.

Je kan Docenten, Toevoegen uitproberen.

Je kan een docent met een bepaald rijksregisternummer geen twee keer toevoegen:

de table docenten bevat een unieke sleutel op de kolom rijksregisternr.

Je lost dit probleem verder in de cursus op een mooie manier op.

Je ziet in het Eclipse venster Console het SQL insert statement dat JPA verstuurt.



Toevoegen: zie takenbundel

## 11 ENTITY VERWIJDEREN



De EntityManager bevat een remove method. Die verwijdert een entity uit de database. Je maakt als voorbeeld een onderdeel waarmee de gebruiker een docent kan verwijderen.

Je voegt een method toe aan DocentRepository:

```
public void delete(long id, EntityManager entityManager) {
    read(id, entityManager)
    .ifPresent(docent -> entityManager.remove(docent));
}
```

**1**  
**2**

- (1) Je verwijdert een entity in twee stappen: je leest eerst de te verwijderen entity.
- (2) Je voert de EntityManager method remove uit en geeft die entity mee als parameter.

Je voegt een method toe aan DocentService:

```
public void delete(long id) {
    EntityManager entityManager = JPAFilter.getEntityManager();
    entityManager.getTransaction().begin();
    try {
        docentRepository.delete(id, entityManager);
        entityManager.getTransaction().commit();
    } catch (PersistenceException ex) {
        entityManager.getTransaction().rollback();
        throw ex;
    } finally {
        entityManager.close();
    }
}
```

Je voegt aan zoeken.jsp een button Verwijderen toe, voor </c:if>:

```
<h2>Acties</h2>
<c:url value='/docenten/verwijderen.htm' var='verwijderURL'>
    <c:param name='id' value='${docent.id}'/>
</c:url>
<form action='${verwijderURL}' method='post'>
    <input type='submit' value='Verwijderen'>
</form>
```

DocentVerwijderenServlet verwerkt POST requests naar /docenten/verwijderen.htm:

```
package be.vdab.servlets.docenten;
// enkele imports ...
@WebServlet("/docenten/verwijderen.htm")
public class DocentVerwijderenServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private final transient DocentService docentService = new DocentService();
    @Override
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        docentService.delete(Long.parseLong(request.getParameter("id")));
        response.sendRedirect(response.encodeRedirectURL(request.getContextPath()));
    }
}
```

## 12 ENTITY WIJZIGEN

Je wijzigt een entity in twee stappen:

1. Je leest de te wijzigen entity.
2. Je wijzigt private variabelen van die entity.

JPA stuurt, bij de commit op de transactie, automatisch een update statement naar de database en wijzigt hiermee het record dat bij de gewijzigde entity hoort.

Je hoeft dus zelf geen update statement naar de database te sturen !

Je kan in één transactie veel entities lezen en slechts enkele van die entities wijzigen.

JPA stuurt enkel voor de gewijzigde entities update statements naar de database.

Je maakt als voorbeeld een onderdeel waarmee de gebruiker één docent opslag geeft.

Je voegt een method toe aan Docent:

```
public void opslag(BigDecimal percentage) {
    BigDecimal factor =
        BigDecimal.ONE.add(percentage.divide(BigDecimal.valueOf(100)));
    wedde = wedde.multiply(factor).setScale(2, RoundingMode.HALF_UP);
}
```

Je voegt een method toe aan DocentService:

```
public void opslag(long id, BigDecimal percentage) {
    EntityManager entityManager = JPAFilter.getEntityManager();
    entityManager.getTransaction().begin();
    try {
        docentRepository.read(id, entityManager)
            .ifPresent(docent -> docent.opslag(percentage));
        entityManager.getTransaction().commit();
    } catch (PersistenceException ex) {
        entityManager.getTransaction().rollback();
        throw ex;
    } finally {
        entityManager.close();
    }
}
```

Je voegt aan zoeken.jsp een button Opslag toe, voor de laatste `</c:if>`:

```
<c:url value='/docenten/opslag.htm' var='opslagURL'>
    <c:param name='id' value='${docent.id}' />
</c:url>
<a href='${opslagURL}' class='knop'><input type="button" value='Opslag'></a>
```

OpslagServlet verwerkt GET en POST requests naar /docenten/opslag.htm:

```
package be.vdab.servlets.docenten;
// enkele imports ...
@WebServlet("/docenten/opslag.htm")
public class OpslagServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private static final String VIEW = "/WEB-INF/JSP/docenten/opslag.jsp";
    private static final String REDIRECT_URL = "%s/docenten/zoeken.htm?id=%d";
    private final transient DocentService docentService = new DocentService();
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        request.getRequestDispatcher(VIEW).forward(request, response);
    }
}
```

```

@Override
protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
    Map<String, String> fouten = new HashMap<>();
    String percentageString = request.getParameter("percentage");
    if (StringUtils.isBigDecimal(percentagesString)) {
        BigDecimal percentage = new BigDecimal(percentagesString);
        if (percentage.compareTo(BigDecimal.ZERO) <= 0) {
            fouten.put("percentage", " tik een positief getal");
        }
        else {
            long id = Long.parseLong(request.getParameter("id"));
            docentService.opslag(id, percentage);
            response.sendRedirect(response.encodeRedirectURL(
                String.format(REDIRECT_URL, request.getContextPath(), id)));
        }
    } else {
        fouten.put("percentage", "tik een positief getal");
    }
    if ( ! fouten.isEmpty()) {
        request.setAttribute("fouten", fouten);
        request.getRequestDispatcher(VIEW).forward(request, response);
    }
}
}

```

Je maakt opslag.jsp in WEB-INF/JSP/docenten:

```

<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%>
<%@taglib uri='http://vdab.be/tags' prefix='v'%>
<!doctype html>
<html lang='nl'>
<head>
<v:head title='Opslag'/>
</head>
<body>
<v:menu/>
<h1>Opslag</h1>
<form method='post' id='opslagform'>
<label>Percentage: <span>${fouten.percentage}</span>
<input name='percentage' value='${param.percentage}'
    type='number' min='0.01' step='0.01' autofocus required>
</label>
<input type='submit' value='Opslag' id='opslagknop'>
</form>
<script>
    document.getElementById('opslagform').onsubmit = function() {
        document.getElementById('opslagknop').disabled = true;
    };
</script>
</body>
</html>

```

Je commit de sources en je publiceert op GitHub.

Je kan dit via [Docenten](#), [Zoeken](#) uitproberen.

Je ziet daarna in het Eclipse venster Console het SQL update statement dat JPA verstuurt.



De EntityManager

stuurt bij de



commit



per gewijzigde  
entity



een update  
statement



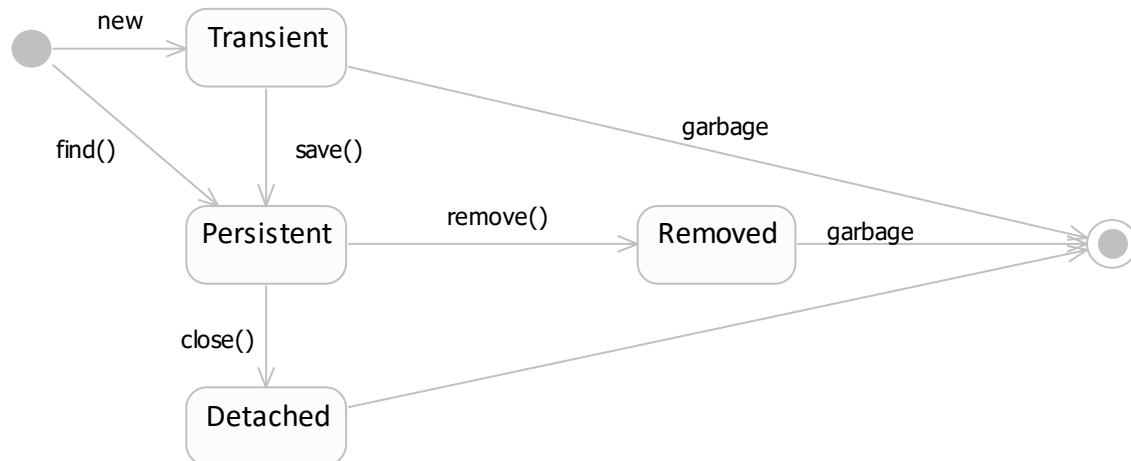
naar de  
database.



## 13 LEVENSCYCLI (LIFE CYCLE) VAN EEN ENTITY ☯

Je ziet in het volgende diagram de verschillende levenscycli (afgeronde rechthoeken) van een entity. Je ziet ook de belangrijkste overgangen (pijlen) tussen deze cycli.

- De linkse cirkel is het begin van een entity object: het is er nog niet.
- De rechtse cirkel is het einde van het entity object: het is uit het geheugen verwijderd door de Java garbage collector.



### Transient

Je hebt een entity aangemaakt (met het `new` keyword van Java), maar je hebt deze entity nog niet toegevoegd aan de database.

### Persistent

De EntityManager beheert de entity.

Dit kan een nieuwe entity zijn die je met de `save` method toevoegde aan de database, of een entity die je met de `find` method gelezen hebt uit de database.

JPA zal alle wijzigingen, die je op een persistent entity uitvoert, ook vastleggen in de database.

### Detached

De EntityManager beheert de entity niet meer.

Dit gebeurt als je de EntityManager sluit met de `close` method.

Wijzigingen die je doet op een detached entity, zal JPA niet meer vastleggen in de database.

### Removed

Je hebt het record, dat hoort bij een entity, verwijderd in de database met de `remove` method. De entity bevindt zich daarna nog in het geheugen.



## 14 ENTITYMANAGER ALS THREADLOCAL VARIABLEE

De code van de service layer is wat omslachtig en repetitief: elke service method

- geeft de EntityManager door aan de repository method.
- doet een rollback op de transactie, als een exception optreedt.
- sluit de EntityManager.

Je vereenvoudigt dit, door de EntityManager als een ThreadLocal variabele bij te houden. Je leert eerst een ThreadLocal variabele kennen.

### 14.1 ThreadLocal variabele

Een ThreadLocal variabele biedt, aan elke thread die de variabele gebruikt, een aparte kopie van de data in de variabele.

Elke thread ziet in de variabele enkel zijn eigen kopie, en moet geen rekening houden met andere threads die hun kopie lezen of wijzigen.

Een thread spreekt een ThreadLocal variabele op volgende manier aan:

- De thread vult zijn kopie van de data met de method set.
- De thread leest zijn kopie van de data met de method get.
- De thread verwijdert zijn kopie van de data met de method remove.

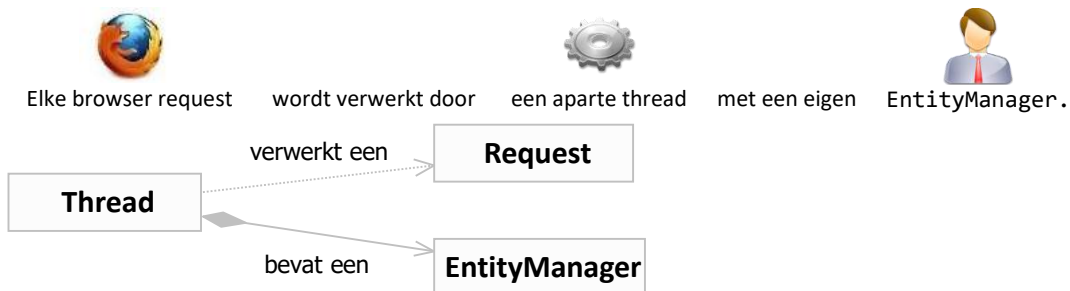
### 14.2 EntityManager als ThreadLocal variabele

De website bevat één variabele entityManagers van het type ThreadLocal<EntityManager>.

De threads, waarmee de webserver browser requests verwerkt, plaatsen elk hun eigen EntityManager in die variabele: je mag één EntityManager maar met één thread aanspreken.

Als een request binnenkomt, maak je (in de huidige thread) een EntityManager, en je plaatst die in de variabele entityManagers.

Tijdens het verwerken van de request loop je door de servlet, service, repository en JSP. Je kan in al die lagen de EntityManager van de huidige thread aanspreken in de variabele entityManagers.



Op het einde van de request verwerking sluit je de EntityManager van de huidige thread, en je verwijdert die EntityManager in de variabele entityManagers.

Je schrijft deze verantwoordelijkheden in de servlet filter JPFilter:

```

package be.vdab.filters;
// enkele imports ...
@WebFilter("*.htm")
public class JPFilter implements Filter {
    private static final EntityManagerFactory entityManagerFactory
        = Persistence.createEntityManagerFactory("fietsacademy");
    private static final ThreadLocal<EntityManager> entityManagers
        = new ThreadLocal<>();
    @Override
    public void init(FilterConfig config) throws ServletException {
    }
}
    
```

①

②

```

@Override
public void doFilter(ServletRequest request, ServletResponse response,
    FilterChain chain) throws IOException, ServletException {
    entityManagers.set(entityManagerFactory.createEntityManager());
    try {
        request.setCharacterEncoding("UTF-8");
        chain.doFilter(request, response);
    } finally {
        entityManagers.get().close();
        entityManagers.remove();
    }
}

public static EntityManager getEntityManager() {
    return entityManagers.get();
}

@Override
public void destroy() {
    entityManagerFactory.close();
}
}

```

- (1) De URL's van alle servlets eindigen op .htm.  
De method `doFilter` van deze filter wordt dus per request naar een servlet uitgevoerd.
- (2) Je maakt een `ThreadLocal` object. Gezien de variabele `static` is, blijft het object in het geheugen gedurende de levensduur van de website.
- (3) Deze method verwerkt een request.
- (4) Je maakt een `EntityManager` en plaatst die in `entityManagers`.  
Als gelijktijdige threads dit doen, hebben ze elk een `EntityManager` in `entityManagers`.
- (5) Je geeft de request door aan de servlet waarvoor hij bestemd is.
- (6) Nadat de servlet (en de bijbehorende JSP) de request verwerkt heeft komt de uitvoering van de code hier terecht. Je leest de `EntityManager` van de huidige thread en je sluit die `EntityManager`.
- (7) Je verwijdert de `EntityManager` van de huidige thread uit `entityManagers`.
- (8) Je haalt de `EntityManager` vanaf nu op uit de `ThreadLocal` variabele.

### 14.3 Repository layer

De `DocentRepository` methods moeten de `EntityManager` niet meer van de `DocentService` krijgen. Ze halen de `EntityManager` uit de `ThreadLocal` variabele.

#### 14.3.1 AbstractRepository

Je maakt in `be.vdab.repositories` een class `AbstractRepository`, waarvan alle repository classes erven. Deze class bevat code die je in elke concrete repository class nodig hebt.

```

package be.vdab.repositories;
// enkele imports ...
abstract class AbstractRepository {
    EntityManager getEntityManager() {
        return JPAFilter.getEntityManager();
    }
}

```

- (1) Je kan deze method aanspreken vanuit repository classes.
- (2) Je leest de `EntityManager` van de huidige thread uit de `ThreadLocal` variabele van `JPAFilter`.

### 14.3.2 DocentRepository

```
package be.vdab.repositories;
import be.vdab.entities.Docent;
// enkele imports ...
public class DocentRepository extends AbstractRepository {
    public Optional<Docent> read(long id) {
        return Optional.ofNullable(getEntityManager().find(Docent.class, id));
    }
    public void create(Docent docent) {
        getEntityManager().persist(docent);
    }
    public void delete(long id) {
        read(id).ifPresent(docent -> getEntityManager().remove(docent));
    }
}
```

## 14.4 De service layer

### 14.4.1 AbstractService

Je maakt in be.vdab.services een class AbstractService, waarvan alle service classes erven. Deze class bevat code die je in elke concrete service class nodig hebt.

```
package be.vdab.services;
// enkele imports ...
abstract class AbstractService {
    private EntityManager getEntityManager() {
        return JPAFilter.getEntityManager();
    }
    void beginTransaction() {
        getEntityManager().getTransaction().begin();
    }
    void commit() {
        getEntityManager().getTransaction().commit();
    }
    void rollback() {
        getEntityManager().getTransaction().rollback();
    }
}
```

### 14.4.2 DocentService

Je erft DocentService van AbstractService:

```
public class DocentService extends AbstractService {
    ...
}
```

De methods in DocentService worden veel korter

```
public Optional<Docent> read(long id) {
    return docentRepository.read(id);
}
public void create(Docent docent) {
    beginTransaction();
    try {
        docentRepository.create(docent);
        commit();
    } catch (PersistenceException ex) {
        rollback();
        throw ex;
    }
}
```

```
public void delete(long id) {
    beginTransaction();
    try {
        docentRepository.delete(id);
        commit();
    }
    catch (PersistenceException ex) {
        rollback();
        throw ex;
    }
}

public void opslag(long id, BigDecimal percentage) {
    beginTransaction();
    try {
        docentRepository.read(id).ifPresent(docent -> docent.opslag(percentage));
        commit();
    }
    catch (PersistenceException ex) {
        rollback();
        throw ex;
    }
}
```

## 14.5 CRUD operaties

CRUD operaties zijn de handelingen die je al kan uitvoeren op een entity:



**Create** Een entity toevoegen aan de database.



**Read** Een entity lezen aan de hand van zijn primary key waarde.



**Update** Een entity wijzigen in de database.



**Delete** Een entity verwijderen uit de database.



**ThreadLocal:** zie takenbundel

## 15 JPQL 🔎

### 15.1 Algemeen

Je leest tot nu één entity uit de database, aan de hand van de primary key, met de EntityManager method `find`.

Je schrijft voor andere lees operaties een query, in JPQL (Java Persistence Query Language). JPQL lijkt op SQL.

De interface TypedQuery stelt een JPQL query voor.

Je maakt een TypedQuery met de EntityManager method `createQuery`.

Je geeft aan die method twee parameters mee:

- een String met de JPQL query.
- Het type entities die de JPQL query teruggeeft.

```
TypedQuery<Docent> query =
    getEntityManager().createQuery("een JPQL query op docenten", Docent.class);
```

Je voert een TypedQuery uit met zijn method `getResultList`.

De method geeft je een List met de gevraagde entities:

```
List<Docent> docenten = query.getResultList();
```



De `createQuery` method maakt een TypedQuery. Die zoekt entities in de database.

### 15.2 Alle entities vragen

De eerste JPQL is eenvoudig: je leest alle docenten: `select d from Docent d`

Je tikt na `from` de naam van een entity class.

JPA zoekt dan records in de table docenten die bij de entity class hoort.

Je geeft aan Docent een alias `d` (`from Docent d`). en vermeldt die na `select`.

JPA vertaalt de JPQL query naar volgend SQL select statement:

```
select docent0_.id as id_, docent0_.voornaam as voornaam0_,
    docent0_.familienaam as familien2_0_, docent0_.wedde as wedde0_
from
    docenten docent0_
```

Je voegt een method toe aan DocentRepository:

```
public List<Docent> findByWeddeBetween(BigDecimal van, BigDecimal tot) {
    // je gebruikt van en tot later ...
    return getEntityManager().createQuery("select d from Docent d", Docent.class)
        .getResultList();
}
```

Je voegt een method toe aan DocentService:

```
public List<Docent> findByWeddeBetween(BigDecimal van, BigDecimal tot) {
    return docentRepository.findByWeddeBetween(van, tot);
}
```

VanTotWeddeServlet verwerkt GET requests naar `/docenten/vantotwedde.htm`:

```
package be.vdab.servlets.docenten;
// enkele imports ...
@WebServlet("/docenten/vantotwedde.htm")
public class VanTotWeddeServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private static final String VIEW = "/WEB-INF/JSP/docenten/vantotwedde.jsp";
    private final transient DocentService docentService = new DocentService();
```

```

@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    // voorlopig dummy waarden voor vanWedde en totWedde meegeven:
    request.setAttribute("docenten",
        docentService.findByWeddeBetween(null, null));
    request.getRequestDispatcher(VIEW).forward(request, response);
}
}

```

Je maakt vantotwedde.jsp in WEB-INF/JSP/docenten:

```

<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%>
<%@taglib uri='http://java.sun.com/jsp/jstl/core' prefix='c'%>
<%@taglib uri='http://java.sun.com/jsp/jstl/fmt' prefix='fmt'%>
<%@taglib uri='http://vdab.be/tags' prefix='v'%>
<!doctype html>
<html lang='nl'>
    <head>
        <v:head title='Docenten van tot wedde'/>
        <style>
            td:first-child, td:last-child {
                text-align:right;
            }
        </style>
    </head>
    <body>
        <v:menu/>
        <h1>Docenten van tot wedde</h1>
        <table>
            <thead>
                <tr><th>Nummer</th><th>Naam</th><th>Wedde</th></tr>
            </thead>
            <tbody>
                <c:forEach items='${docenten}' var='docent'>
                    <tr>
                        <td>${docent.id}</td>
                        <td>${docent.naam}</td>
                        <td><fmt:formatNumber value='${docent.wedde}'
                            minFractionDigits='2' maxFractionDigits='2'/></td>
                    </tr>
                </c:forEach>
            </tbody>
        </table>
    </body>
</html>

```

Je kan de website uitproberen.

### 15.3 Sorteren

Je sorteert records met **order by**.

Je tikt na **order by** geen kolomnamen, maar namen van bijbehorende private variabelen.

Je tikt voor elke variabele de alias, die je aan de entity gaf bij **from**, en een punt.

Je wijzigt in class DocentRepository in de method findByWeddeBetween de query:

**select** d **from** Docent d **order by** d.wedde **desc**, d.voornaam, d.familienaam

Je kan de website uitproberen.

## 15.4 Selecteren

Je leest slechts een deel van de entities met **where**.

Je tikt na **where** één of meerdere voorwaarden waaraan de te lezen entities moeten voldoen.

Je kan voorwaarden combineren met **and**, **or** (en eventueel **not**).

Je tikt in de voorwaarden geen kolomnamen, maar de namen van private variabelen.

Je ziet hier voorbeelden van queries met een **where** onderdeel:

```
select d from Docent d where d.wedde = 2200
```

docenten met een wedde gelijk aan 2200.

```
select d from Docent d where d.wedde >= 2000
```

docenten met een wedde vanaf 2000.

```
select d from Docent d where d.wedde * 12 >= 20000
```

docenten met een wedde \* 12 vanaf 20000.

```
select d from Docent d where d.wedde between 2000 and 2200
```

docenten met een wedde tussen 2000 en 2200 (grenzen inbegrepen).

```
select d from Docent d where d.familienaam like 'D%'
```

docenten met een familienaam die begint met D of d.

```
select d from Docent d where d.wedde is null
```

docenten met een niet-ingevulde wedde.

```
select d from Docent d where d.voornaam in ('Jules', 'Cleopatra')
```

docenten met een voornaam gelijk aan Jules of Cleopatra.

Je kan in **where** ook een subquery gebruiken (zoals in SQL)

```
select d from Docent d where d.wedde = (select max(dd.wedde) from Docent dd)
```

docenten met een wedde gelijk aan de grootste wedde van alle docenten.

Je kan ook een gecorreleerde subquery gebruiken (zoals in SQL)

```
select d from Docent d where d.wedde =  
(select max(dd.wedde) from Docent dd where dd.voornaam = d.voornaam)
```

docenten die de grootste wedde hebben van docenten met dezelfde voornaam.

Je wijzigt de query in de DocentRepository method `findByWeddeBetween`:

```
select d from Docent d where d.wedde between 2000 and 2200 order by d.wedde, d.id
```

Je ziet in de website de docenten met een wedde tussen 2000 en 2200.

### 15.4.1 Positional parameters

Je duidt een parameter in een query aan met een vraagteken, gevolgd door het volgnummer van de parameter. Zo'n parameters heten positional parameters.

Je geeft de parameters een waarde voor je de query uitvoert, met de TypedQuery method `setParameter(parameterVolgnummer, parameterWaarde)`. De volgnummers beginnen vanaf 1. De method geeft je als returnwaarde terug dezelfde TypedQuery.

Je wijzigt in DocentRepository de method `findByWeddeBetween`:

```
return getEntityManager().createQuery(  
    "select d from Docent d where d.wedde between ?1 and ?2  
    order by d.wedde, d.id", Docent.class)  
    .setParameter(1, van)  
    .setParameter(2, tot)  
    .getResultList();
```

Je tikt code in `vantotwedde.jsp`, voor `<table>`:

```
<form>  
  <label>Van:<span>${fouten.van}</span>  
  <input name='van' value='${param.van}' type='number' min='0' step='0.01'  
    required autofocus></label>
```

```

<label>Tot:<span>${fouten.tot}</span>
<input name='tot' type='number' value='${param.tot}' min='0' step='0.01'
    required></label>
<input type='submit' value='Zoeken'>
</form>
<c:if test="${not empty param and empty fouten and empty docenten}">
Geen docenten gevonden
</c:if>
<c:if test="${not empty docenten}">

```

Je tikt </c:if> na </table>.

Je wijzigt in VanTotWeddeServlet de code in method doGet:

```

if (request.getQueryString() != null) {
    Map<String, String> fouten = new HashMap<>();
    String van = request.getParameter("van");
    if ( ! StringUtils.isBigDecimal(van)) {
        fouten.put("van", "tik een getal");
    }
    String tot = request.getParameter("tot");
    if ( ! StringUtils.isBigDecimal(tot)) {
        fouten.put("tot", "tik een getal");
    }
    if (fouten.isEmpty()) {
        request.setAttribute("docenten", docentService.findByWeddeBetween(
            new BigDecimal(van), new BigDecimal(tot)));
    }
    else {
        request.setAttribute("fouten", fouten);
    }
}
request.getRequestDispatcher(VIEW).forward(request, response);

```

Je kan de website uitproberen.

#### 15.4.2 Named parameters

Naast positional parameters bestaan ook named parameters.

Je stelt een parameter dan voor met een naam, voorafgegaan door het teken :

Je geeft de parameters een waarde voor je de query uitvoert, met de TypedQuery method setParameter(parameterNaam, parameterwaarde). Je tikt geen : voor de naam.

Je wijzigt in DocentRepository de method findByWeddeBetween:

```

return getEntityManager().createQuery(
    "select d from Docent d where d.wedde between :van and :tot
    order by d.wedde, d.id", Docent.class)
    .setParameter("van", van)
    .setParameter("tot", tot)
    .getResultList();

```

Je kan de website terug uitproberen.

Named parameters zijn beter dan positional parameters. Je queries worden:

- ⊕ leesbaar en zelfdocumenterend.
- ⊕ onderhoudbaar.

Als je de query **select d from Docent d where d.voornaam=? and d.familienaam=?**

wijzigt in **select d from Docent d where d.familienaam=? and d.voornaam=?**

geeft dit niet meer het correcte resultaat, tenzij je de volgorde in de setParameter methods (waarmee je de parameters invult) wijzigt.





Als je een query hebt met het **in** sleutelwoord en daarin een parameter

**select** d **from** Docent d **where** d.voornaam **in** (:voornamen)

vul je deze parameter met een List of Set:

```
.setParameter("voornamen", Arrays.asList("Jules", "Cleopatra"));
```

Deze query geeft de docenten terug met Jules of Cleopatra als voornaam.

## 15.5 Pagineren

Als de gebruiker een zoekopdracht uitvoert die veel records teruggeeft, is het de gewoonte die records niet in één keer te tonen, maar te pagineren.

- Je toont de eerste 20 gevonden records.
- Als de gebruiker → aanklikt, toon je de volgende 20 records.
- Als de gebruiker ← aanklikt, toon je de vorige 20 records.

TypedQuery bevat hiertoe twee methods:

- `setFirstResult(int i)`  
Je geeft aan vanaf het hoeveelste record, uit alle gevonden records van de query, je records wenst te krijgen. De nummering begint bij nul.
- `setMaxResults(int i)`  
Je geeft aan hoeveel records je wenst, vanaf het record aangeduid via `setFirstResult`.

Je krijgt met volgende method oproepen 20 records vanaf het record 40:

```
.setFirstResult(39)
.setMaxResults(20);
```

Je toont → enkel als er een volgende pagina is. Je vraagt 21 records (één te veel) om dit te weten.

- Als JPA minder dan 21 records teruggeeft, is er geen volgende pagina.
- Als JPA 21 records teruggeeft, is er een volgende pagina. Je toont het laatste record niet.

Je wijzigt in `DocentRepository` de method `findByWeddeBetween`:

```
public List<Docent> findByWeddeBetween(BigDecimal van, BigDecimal tot,
    int vanafRij, int aantalRijen) {
    return getEntityManager().createQuery(
        "select d from Docent d where d.wedde between :van and :tot
        order by d.wedde, d.id", Docent.class)
        .setParameter("van", van)
        .setParameter("tot", tot)
        .setFirstResult(vanafRij)
        .setMaxResults(aantalRijen)
        .getResultList();
}
```

Je wijzigt in `DocentService` de method `findByWeddeBetween`:

```
public List<Docent> findByWeddeBetween(
    BigDecimal van, BigDecimal tot, int vanafRij, int aantalRijen) {
    return docentRepository.findByWeddeBetween(van, tot, vanafRij, aantalRijen);
}
```

Je voegt aan `VanTotWeddeServlet` een constante toe:

```
private static final int AANTAL_RIJEN = 20;
```

Je wijzigt in de method `doGet` de opdracht `request.setAttribute("docenten", ...);`:

```
int vanafRij = request.getParameter("vanafRij") == null ? 0 :
    Integer.parseInt(request.getParameter("vanafRij"));
request.setAttribute("vanafRij", vanafRij);
request.setAttribute("aantalRijen", AANTAL_RIJEN);
List<Docent> docenten =
    docentService.findByWeddeBetween(van, tot, vanafRij, AANTAL_RIJEN + 1);
if (docenten.size() <= AANTAL_RIJEN) {
    request.setAttribute("laatstePagina", true);
} else {
    docenten.remove(AANTAL_RIJEN);
}
request.setAttribute("docenten", docenten);
```

Je voegt code toe aan `vantotwedde.jsp`, na `</table>`:

```
<c:if test='${vanafRij != 0}'>
  <c:url value='' var='vorigePaginaURL'>
    <c:param name='van' value='${param.van}'/>
    <c:param name='tot' value='${param.tot}'/>
    <c:param name='vanafRij' value='${vanafRij - aantalRijen}'/>
  </c:url>
  <a href="<c:out value='${vorigePaginaURL}'/>" title='vorige pagina'
    class='pagineren'>&larr;</a>
</c:if>
<c:if test='${empty laatstePagina}'>
  <c:url value='' var='volgendePaginaURL'>
    <c:param name='van' value='${param.van}'/>
    <c:param name='tot' value='${param.tot}'/>
    <c:param name='vanafRij' value='${vanafRij + aantalRijen}'/>
  </c:url>
  <a href="<c:out value='${volgendePaginaURL}'/>" title='volgende pagina'
    class='pagineren'>&rarr;</a>
</c:if>
```

Je kan de website terug uitproberen.

## 15.6 Één kolom lezen

Je leest tot nu per record *alle* kolommen. Je kan ook één kolom (of enkele kolommen) lezen.

Je doet dit enkel als het lezen van alle kolommen de performantie benadeelt.

De table `docenten` bevat slechts enkele kolommen. Als je per record alle kolommen leest, krijg je geen performantieproblemen. Je zal als voorbeeld toch enkel de kolom `Voornaam` lezen.

Je vermeldt in je query bij **`select`** de alias die je aan de class geeft, een punt en de naam van de private variabele die hoort bij de kolom die je wil lezen:

```
select d.voornaam from Docent d
```

Het resultaat van de query is een `List<String>`: de variabele `voornaam` is ook een `String`.

Je voegt een method toe aan `DocentRepository`:

```
public List<String> findVoornamen() {
  return getEntityManager().createQuery(
    "select d.voornaam from Docent d", String.class).getResultList();
}
```

Je voegt een method toe aan `DocentService`:

```
public List<String> findVoornamen() {
  return docentRepository.findVoornamen();
}
```

`VoornamenServlet` verwerkt GET requests naar `/docenten/voornamen.htm`:

```
package be.vdab.servlets.docenten;
// enkele imports ...
@WebServlet("/docenten/voornamen.htm")
public class VoornamenServlet extends HttpServlet {
  private static final long serialVersionUID = 1L;
  private static final String VIEW = "/WEB-INF/JSP/docenten/voornamen.jsp";
  private final transient DocentService docentService = new DocentService();
  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    request.setAttribute("voornamen", docentService.findVoornamen());
    request.getRequestDispatcher(VIEW).forward(request, response);
  }
}
```

Je maakt voornamen.jsp in WEB-INF/JSP/docenten:

```
<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%>
<%@taglib uri='http://java.sun.com/jsp/jstl/core' prefix='c'%>
<%@taglib uri='http://vdab.be/tags' prefix='v'%>
<!doctype html>
<html lang='nl'>
<head>
<v:head title='Docenten voornamen' />
</head>
<body>
  <v:menu />
  <h1>Docenten voornamen</h1>
  <ul class='zonderbolletjes'>
    <c:forEach items='${voornamen}' var='voornaam'>
      <li style='font-size:${voornaam.length() mod 3 + 1}em;'>${voornaam}</li>
    </c:forEach>
  </ul>
</body>
</html>
```

Je kan de website uitproberen.

## 15.7 Meerdere kolommen lezen

Volgende query geeft een lijst met het nummer en de voornaam van elke docent:

```
select new be.vdab.valueobjects.VoornaamEnId(d.id, d.voornaam) from Docent d
```

Het query resultaat is een List met VoornaamEnId objecten.

Dit is een class die jij schrijft. Je geeft die class een constructor met

- een **long** parameter (die het id binnenkrijgt van JPA)
- een **String** parameter (die de voornaam binnenkrijgt van JPA)

Je maakt een package be.vdab.valueobjects en daarin de class VoornaamEnId:

```
package be.vdab.valueobjects;
public class VoornaamEnId {
  private final long id;
  private final String voornaam;
  public VoornaamEnId(long id, String voornaam) {
    this.id = id;
    this.voornaam = voornaam;
  }
  // je maakt getters voor de private variabelen
}
```

Je wijzigt in DocentRepository de method findVoornamen:

```
public List<VoornaamEnId> findVoornamen() {
  return getEntityManager().createQuery(
    "select new be.vdab.valueobjects.VoornaamEnId(d.id, d.voornaam) from Docent d",
    VoornaamEnId.class).getResultList();
}
```

Je wijzigt in de DocentService method findVoornamen String naar VoornaamEnId.

Je vervangt in voornamen.jsp de <c:forEach> ... </c:forEach>:

```
<c:forEach items='${voornamen}' var='voornaamEnId'>
  <c:url value='/docenten/zoeken.htm' var='docentURL'>
    <c:param name='id' value='${voornaamEnId.id}' />
  </c:url>
  <li style='font-size:${voornaamEnId.voornaam.length() mod 3 + 1}em;'>
    <a href='${docentURL}'>${voornaamEnId.voornaam}</a></li>
</c:forEach>
```

Je kan de website uitproberen.

## 15.8 Aggregate functions

JSQL bevat de aggregate functions `count()`, `min()`, `max()`, `avg()` en `sum()`.

- `select max(d.wedde) from Docent d`  
geeft een `BigDecimal` met de grootste docent wedde van alle docenten.
- `select max(d.wedde), min(d.wedde) from Docent d`  
geeft een array met twee `BigDecimal`s: de grootste en de kleinste wedde van alle docenten.

Deze queries geven één rij met één of meerdere kolommen terug.

Je voert zo'n query uit met de method `getSingleResult` Deze geeft je die rij.

Je voegt een method toe aan `DocentRepository`:

```
public BigDecimal findMaxWedde() {
    return getEntityManager().createQuery(
        "select max(d.wedde) from Docent d", BigDecimal.class).getSingleResult();
}
```

Je voegt een method toe aan `DocentService`:

```
public BigDecimal findMaxWedde() {
    return docentRepository.findMaxWedde();
}
```

Als de gebruiker `vantotwedde.jsp` opent, vul je het vak tot met de maximale wedde.

Je vervangt daartoe in `VanTotWeddeServlet` `if (request.getQueryString() != null)` { door  
`if (request.getQueryString() == null)` {  
     `request.setAttribute("tot", docentService.findMaxWedde());`  
`}` `else` {

Je gebruikt in `vantotwedde.jsp` deze waarde in de value van invoervak tot:

```
${empty tot ? param.tot : tot}
```

Je kan de website uitproberen.

## 15.9 group by

Je maakt met `group by` samenvattingen.

Volgende query geeft een lijst met per wedde het aantal docenten met die wedde:

```
select new be.vdab.valueobjects.AantalDocentenPerWedde(d.wedde, count(d))
from Docent d
group by d.wedde
```

Het query resultaat is een `List` met `AantalDocentenPerWedde` objecten.

Dit is een class die jij schrijft. Je geeft die class een constructor met twee parameters:

- een `BigDecimal` (die de wedde binnenkrijgt)
- een `long` (die het aantal docenten binnenkrijgt)

```
package be.vdab.valueobjects;
import java.math.BigDecimal;
public class AantalDocentenPerWedde {
    private final BigDecimal wedde;
    private final long aantal;
    public AantalDocentenPerWedde(BigDecimal wedde, long aantal) {
        this.wedde = wedde;
        this.aantal = aantal;
    }
    // je maakt getters voor de private variabelen
}
```

Je voegt een method toe aan `DocentRepository`:

```
public List<AantalDocentenPerWedde> findAantalDocentenPerWedde() {
    return getEntityManager().createQuery(
        "select new be.vdab.valueobjects.AantalDocentenPerWedde(d.wedde, count(d))" +
        "from Docent d group by d.wedde",
        AantalDocentenPerWedde.class).getResultList();
}
```

Je voegt een method toe aan DocentService:

```
public List<AantalDocentenPerWedde> findAantalDocentenPerWedde() {
    return docentRepository.findAantalDocentenPerWedde();
}
```

AantalPerWeddeServlet verwerkt GET requests naar /docenten/aantalperwedde.htm:

```
package be.vdab.servlets.docenten;
// enkele imports ...
@WebServlet("/docenten/aantalperwedde.htm")
public class AantalPerWeddeServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private static final String VIEW = "/WEB-INF/JSP/docenten/aantalperwedde.jsp";
    private final transient DocentService docentService = new DocentService();
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        request.setAttribute("weddesEnAantallen",
            docentService.findAantalDocentenPerWedde());
        request.getRequestDispatcher(VIEW).forward(request, response);
    }
}
```

Je maakt aantalperwedde.jsp in WEB-INF/JSP/docenten:

```
<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%>
<%@taglib uri='http://java.sun.com/jsp/jstl/core' prefix='c'%>
<%@taglib uri='http://java.sun.com/jsp/jstl/fmt' prefix='fmt'%>
<%@taglib uri='http://vdab.be/tags' prefix='v'%>
<!doctype html>
<html lang='nl'>
<head><v:head title='Aantal docenten per wedde'/></head>
<style>
td:first-child {
    padding-right:0.5em;
    text-align:right;
}
td:nth-child(2) div {
    background: linear-gradient(to right, wheat, orange);
    padding-left: 0.5em;
}
</style>
<body>
    <v:menu/>
    <h1>Aantal docenten per wedde</h1>
    <table>
        <thead>
            <tr><th>Wedde</th><th>Aantal docenten</th></tr>
        </thead>
        <tbody>
            <c:forEach items='${weddesEnAantallen}' var='weddeEnAantal'>
                <tr>
                    <td><fmt:formatNumber value='${weddeEnAantal.wedde}' minFractionDigits='2'
                        maxFractionDigits='2'/></td>
                    <td><div style='width:${weddeEnAantal.aantal}em'>
                        ${weddeEnAantal.aantal}</div></td>
                </tr>
            </c:forEach>
        </tbody>
    </table>
</body>
</html>
```

Je kan de website uitproberen.

## 15.10 Named queries

Je maakt queries tot nu met de EntityManager method `createQuery`. Bij elke uitvoering van zo'n query controleert JPA de syntax van de query en vertaalt JPA de query naar SQL. JPA controleert named queries slechts één keer (bij de EntityManagerFactory initialisatie) op syntax en vertaalt ze één keer naar SQL statements. JPA onthoudt deze SQL statements in het interne geheugen gedurende de levensduur van de applicatie.

Named queries hebben voordelen:

- ⊕ als de query syntaxfouten bevat, merk je dit vroeg (bij het starten van de applicatie) en niet laat (tijdens de eerste uitvoering van de query).
- ⊕ JPA doet bij elke query uitvoering geen syntaxcontrole en vertaling naar SQL meer, en voert de query dus snel uit.

### 15.10.1 Named queries in entity classes

Je kan in één entity class meerdere named queries schrijven.

Je definieert één named query met `@NamedQuery`, met twee parameters:

- `name` Een unieke named query naam binnen je applicatie.
- `query` De JPQL query van de named query.

```
@NamedQuery(name = "naamVanDeQuery", query = "JPQL van de query")
```

Je kan meerdere named queries tikken bij één entity class.

Je verzamelt dan meerdere `@NamedQuery` annotations in één array via `{ en }`.

Je geeft deze array als parameter aan `@NamedQueries`. Je tikt dit voor de entity class:

```
@NamedQueries({
    @NamedQuery(name = "naamEersteQuery", query = "JPQL eerste query"),
    @NamedQuery(name = "naamTweedeQuery", query = "JPQL tweede query") })
```

Je maakt als voorbeeld één named query, voor de class `Docent`:

```
@NamedQuery(name = "Docent.findByWeddeBetween",
    query = "select d from Docent d where d.wedde between :van and :tot
    order by d.wedde, d.id")
```

### 15.10.2 Named query oproepen

Je roept een named query op met de EntityManager method `createNamedQuery`.

Je geeft twee parameters mee:

- De naam van de named query.
- Het type entities die de query teruggeeft.

De method geeft je een `TypedQuery`.

Je vervangt in `DocentRepository` method `findByWeddeBetween` `createQuery(...)` door:  
`.createNamedQuery("Docent.findByWeddeBetween", Docent.class)`

Je kan de website terug uitproberen.

### 15.10.3 Named queries in orm.xml

Je kan named queries ook schrijven in `src/main/resources/META-INF/orm.xml`.

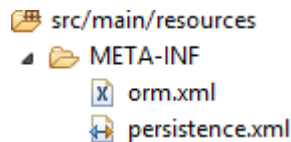
Dit heeft extra voordelen

- ⊕ Je kan een query in XML gemakkelijk over meerdere regels schrijven. Dit bevordert de leesbaarheid van de query.
- ⊕ Je kan de query wijzigen nadat de applicatie af is, zonder source code te compileren.
- ⊕ Eclipse controleert de query op tikfouten.

Je verwijdert in `Docent` de regel met `@NamedQuery`.

Je maakt orm.xml

in src/main/resources/META-INF:



```
<?xml version="1.0" encoding="UTF-8" ?>
<entity-mappings xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
    http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd" version="2.1">
  <named-query name='Docent.findByWeddeBetween'>
    <query>
      select d from Docent d
      where d.wedde between :van and :tot
      order by d.wedde, d.id
    </query>
  </named-query>
</entity-mappings>
```



Als je in de query Ctrl + spatie tikt na d. zie je de Docent attributen.

Nadat je een query getikt hebt sla je orm.xml op.

Je klikt daarna met de rechtermuisknop in orm.xml en je kiest de opdracht Validate.

Als de query tikfouten bevat, zie je foutmeldingen.

Je kan meerdere named queries onder mekaar schrijven.

De oproep van de named query blijft dezelfde.

Je commit de sources en je publiceert op GitHub.

Je kan de website uitproberen.



Zoeken op naam: zie takenbundel

## 16 BULK UPDATES EN BULK DELETES

Soms wijzig of verwijder je niet één record, maar *meerdere* records in één keer.

Je doet dit met de JPQL keywords **update** en **delete**. Deze sturen voor de groep wijzigingen of verwijderingen één update of delete SQL statement naar de database.

- Een bulk update query die de wedde van docenten met 10% verhoogt:  
**update** Docent d **set** d.wedde = d.wedde \* 1.1
- Een bulk delete query die docenten met een wedde tot en met 2000 verwijdert:  
**delete** Docent d **where** d.wedde <= 2000

Bulk updates en bulk delete queries kunnen ook parameters bevatten:

**update** Docent d **set** d.wedde = d.wedde \* :factor **where** d.wedde <= :totEnMetWedde

Je voert een bulk update of bulk delete uit met de Query method `executeUpdate`.

De EntityManager method `createNamedQuery` geeft je een Query object.

Je geeft bij de method oproep één parameter mee: de naam van de named query.

De method `executeUpdate` geeft je een int terug. Deze bevat:

- bij een bulk update het aantal gewijzigde records
- bij een bulk delete het aantal verwijderde records

Je maakt als voorbeeld een onderdeel waarmee alle docenten opslag krijgen.

Je voegt een query toe aan `orm.xml`:

```
<named-query name='Docent.algemeneOpslag'>
  <query>
    update Docent d
    set d.wedde = d.wedde * :factor
  </query>
</named-query>
```

Je voegt een method toe aan `DocentRepository`:

```
public void algemeneOpslag(BigDecimal factor) {
    getEntityManager().createNamedQuery("Docent.algemeneOpslag")
        .setParameter("factor", factor)
        .executeUpdate();
}
```

Je voegt een method toe aan `DocentService`:

```
public void algemeneOpslag(BigDecimal percentage) {
    BigDecimal factor = BigDecimal.ONE.add(
        percentage.divide(BigDecimal.valueOf(100)));
    try {
        beginTransaction();
        docentRepository.algemeneOpslag(factor);
        commit();
    } catch (PersistenceException ex) {
        rollback();
        throw ex;
    }
}
```

`AlgemeneOpslagServlet` verwerkt GET en POST requests naar `/docenten/algemeneopslag.htm`:

```
package be.vdab.servlets.docenten;
// enkele imports ...
@WebServlet("/docenten/algemeneopslag.htm")
public class AlgemeneOpslagServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private static final String VIEW = "/WEB-INF/JSP/docenten/algemeneopslag.jsp";
    private final transient DocentService docentService = new DocentService();
```



```

@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    request.getRequestDispatcher(VIEW).forward(request, response);
}

@Override
protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
    Map<String, String> fouten = new HashMap<>();
    String percentageString = request.getParameter("percentage");
    if (StringUtils.isBigDecimal(percentageString)) {
        BigDecimal percentage = new BigDecimal(percentageString);
        if (percentage.compareTo(BigDecimal.ZERO) <= 0) {
            fouten.put("percentage", "tik een positief getal");
        }
        else {
            docentService.algemeneOpslag(percentage);
        }
    }
    else {
        fouten.put("percentage", "tik een positief getal");
    }
    if (fouten.isEmpty()) {
        response.sendRedirect(response.encodeRedirectURL(request.getContextPath()));
    }
    else {
        request.setAttribute("fouten", fouten);
        request.getRequestDispatcher(VIEW).forward(request, response);
    }
}
}

```

Je maakt algemeneopslag.jsp in WEB-INF/JSP/docenten:

```

<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%>
<%@taglib uri='http://vdab.be/tags' prefix='v'%>
<!doctype html>
<html lang='nl'>
<head>
<v:head title='Algemene opslag docenten' />
</head>
<body>
    <v:menu/>
    <h1>Algemene opslag docenten</h1>
    <form method='post' id='opslagform'>
        <label>Percentage:<span>${fouten.percentage}</span>
        <input name='percentage' value='${param.percentage}' type='number'
            min='0.01' step='0.01' autofocus></label>
        <input type='submit' value='Opslag' id='submitknop'>
    </form>
    <script>
    document.getElementById('opslagform').onsubmit = function() {
        document.getElementById('submitknop').disabled = true;
    };
    </script>
</body>
</html>

```

Je commit de sources en je publiceert op GitHub.

Je kan de website uitproberen



Prijsverhoging: zie takenbundel

## 17 INHERITANCE 📦

### 17.1 Inheritance nabootsen in de database

Inheritance bestaat tussen Java classes, maar niet tussen database tables.

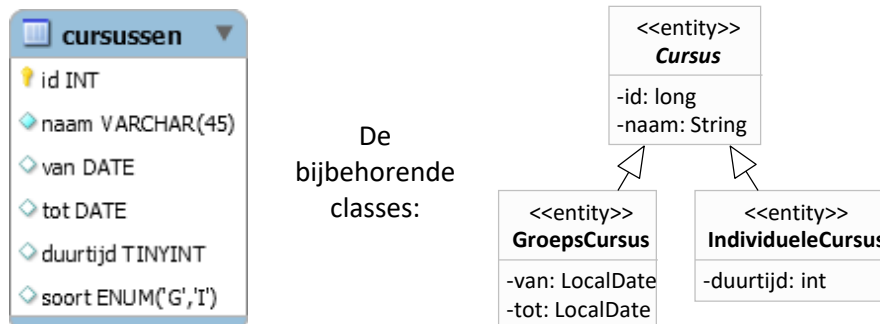
Je kan inheritance in de database nabootsen op drie manieren, die je hier leert:

- Table per class hierarchy.
- Table per subclass.
- Table per concrete class.

### 17.2 Table per class hierarchy

#### 17.2.1 Database

Je voert het script `TablePerClassHierarchy.sql` uit. Dit maakt een table cursussen:



Er is één table voor alle classes uit de inheritance hierarchy.

Deze table bevat kolommen voor alle attributen van alle classes van de hierarchy.

De kolom soort is een discriminator kolom.

Die geeft het type entity aan dat een record voorstelt. De kolom soort bevat:

- G als het record een GroepsCursus voorstelt.
- I als het record een IndividueleCursus voorstelt.

#### 17.2.2 Voordelen van table per class hierarchy

- ➕ Als je een cursus toevoegt, stuur je één **insert** statement naar één table. Je wijzigt of verwijdert een cursus ook met één statement naar één table.
- ➕ Als je een attribuut toevoegt aan een class, moet je maar één kolom toevoegen aan één table.

#### 17.2.3 Nadelen van table per class hierarchy

- ➖ De table bevat veel kolommen als de classes veel attributen hebben.
- ➖ Je kan sommige kolommen niet instellen als verplicht in te vullen. Je kan bijvoorbeeld de kolom Van niet instellen als verplicht in te vullen: je laat deze kolom leeg als je een IndividueleCursus toevoegt.

#### 17.2.4 De base class

```
package be.vdab.entities;
// enkele imports ...
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@Table(name = "cursussen")
@DiscriminatorColumn(name = "soort")
public abstract class Cursus implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String naam;
```

❶  
❷  
❸

```

@Override
public String toString() {
    return naam;
}
}

```

- (1) Je tikt `@Inheritance` bij de hoogste class in de inheritance hiërarchie.  
Je tikt bij strategy op welke manier je inheritance nabootst in de database.  
Je gebruikt bij "table per class hiërarchie" de waarde `SINGLE_TABLE`.
- (2) Je tikt bij de hoogste class uit de inheritance hiërarchie  
de naam van de table die hoort bij de objecten uit de inheritance hiërarchie.
- (3) Je tikt bij `@DiscriminatorColumn` de naam van discriminator kolom.

Als je bij `@Table` de fout `Table cursussen cannot be resolved` krijgt, doe je volgende stappen:

1. Je klappt in de Data Source Explorer niveau's van je database open tot je de map Tables ziet.
2. Je klikt met de rechtermuisknop op Tables en je kiest Refresh
3. Je kiest in het menu Project de opdracht Clean



Opmerking: je maakt voor de discriminator kolom (soort) geen bijbehorende variabele.

### 17.2.5 De afgeleide classes

```

package be.vdab.entities;
// enkele imports ...
@Entity
@DiscriminatorValue("G")
public class GroepsCursus extends Cursus {
    private static final long serialVersionUID = 1L;
    private LocalDate van;
    private LocalDate tot;
}

```

- (1) Je tikt bij `@DiscriminatorValue` de waarde in de discriminator kolom (Soort) als een record hoort bij een entity van de huidige entity class (GroepsCursus).

```

package be.vdab.entities;
// enkele imports ...
@Entity
@DiscriminatorValue("I")
public class IndividueleCursus extends Cursus {
    private static final long serialVersionUID = 1L;
    private int duurtijd;
}

```

Je vermeldt deze classes in `persistence.xml`, na `</class>`:

```

<class>be.vdab.entities.Cursus</class>
<class>be.vdab.entities.GroepsCursus</class>
<class>be.vdab.entities.IndividueleCursus</class>

```

### 17.2.6 Dali

Je klikt met de rechtermuisknop in het Dali diagram en je kiest Show all Persistence Types.  
Je ziet de drie nieuwe classes en hun inheritance verbanden.

### 17.2.7 Polymorphic queries

Dit betekent: als je in het **from** deel van een query een entity class vermeldt, bevat het resultaat van de query ook objecten van afgeleide classes.

JPQL query	Type objecten in resultaat
<b>select</b> g <b>from</b> GroepsCursus g	GroepsCursus
<b>select</b> c <b>from</b> Cursus c	Cursus, GroepsCursus en IndividueleCursus

Voorbeeld: de gebruiker tikt een woord. Je toont de cursussen met dit woord in hun naam.

Je voegt een query toe aan orm.xml:

```
<named-query name='Cursus.findByNaamContains'>
  <query>
    select c from Cursus c where c.naam like :zoals order by c.naam
  </query>
</named-query>
```

Je maakt een class CursusRepository:

```
package be.vdab.repositories;
// enkele imports ...
public class CursusRepository extends AbstractRepository {
    public List<Cursus> findByNaamContains(String woord) {
        return getEntityManager().createNamedQuery("Cursus.findByNaamContains",
            Cursus.class)
            .setParameter("zoals", '%' + woord + '%')
            .getResultList();
    }
}
```

Je maakt een class CursusService:

```
package be.vdab.services;
// enkele imports ...
public class CursusService extends AbstractService {
    private final CursusRepository cursusRepository = new CursusRepository();
    public List<Cursus> findByNaamContains(String woord) {
        return cursusRepository.findByNaamContains(woord);
    }
}
```

MetNaamServlet verwerkt GET requests naar /cursussen/metnaam.htm:

```
package be.vdab.servlets.cursussen;
// enkele imports ...
@WebServlet("/cursussen/metnaam.htm")
public class MetNaamServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private static final String VIEW = "/WEB-INF/JSP/cursussen/metnaam.jsp";
    private final transient CursusService cursusService = new CursusService();
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String woord = request.getParameter("woord");
        if (woord != null) {
            if (woord.trim().isEmpty()) {
                request.setAttribute("fouten",
                    Collections.singletonMap("woord", "verplicht"));
            } else {
                request.setAttribute("cursussen",
                    cursusService.findByNaamContains(woord));
            }
        }
        request.getRequestDispatcher(VIEW).forward(request, response);
    }
}
```

Je maakt metnaam.jsp in WEB-INF/JSP/cursussen:

```
<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%>
<%@taglib uri='http://java.sun.com/jsp/jstl/core' prefix='c'%>
<%@taglib uri='http://vdab.be/tags' prefix='v' %>
<!doctype html>
<html lang='nl'>
  <head><v:head title='Cursussen zoeken op naam'></v:head></head>
  <body>
    <v:menu/>
    <h1>Cursussen zoeken op naam</h1>
    <form>
      <label>Woord:<span>${fouten.woord}</span>
      <input name='woord' value='${param.woord}' autofocus required
        type='search'></label>
      <input type='submit' value='Zoeken'>
    </form>
    <c:if test='${not empty param and empty fouten and empty cursussen}'>
      Geen cursussen gevonden
    </c:if>
    <c:if test='${not empty cursussen}'>
      <ul>
        <c:forEach items='${cursussen}' var='cursus'>
          <c:set var='soortCursus' value='${cursus['class'].simpleName}'/>
          <li>${cursus}
            <img src='<c:url value="/images/${soortCursus}.png"/>'
              alt='${soortCursus}' title='${soortCursus}'></li>
        </c:forEach>
      </ul>
    </c:if>
  </body>
</html>
```

- (1) De JSP vertaalt de EL expressie "\${cursus['class'].simpleName}" naar `cursus.getClass().getSimpleName()`. Dit geeft een String terug. Als `cursus` een `GroepsCursus` object bevat, is dit `GroepsCursus`. Als `cursus` een `IndividueleCursus` object bevat, is dit `IndividueleCursus`.

Je kan de website uitproberen.

### 17.3 Table per subclass

Je voert het script `TablePerSubclass.sql` uit. Dit maakt een nieuwe versie van de table `cursussen` en voegt de tables `groepscursussen` en `individuelecursussen` toe:



De database bevat een table per class uit de inheritance hierarchy.

Elke table bevat enkel kolommen voor de attributen in de bijbehorende class.

De primary key van een table die hoort bij een derived class is tegelijk een foreign key die verwijst naar de primary key van de table die hoort bij de hoogste class uit de inheritance hiërarchy.

- `id` is in de table `cursussen` een autonumber kolom, maar niet in de tables `groepscursussen` en `individuelecursussen`.
- `id` is in de tables `groepscursussen` en `individuelecursussen` primary key en tegelijk foreign key verwijzend naar `id` in de table `cursussen`.

JPA voegt een individuele cursus toe met volgende stappen:

- `insert into cursussen(naam) values ("een individuelecursus")`
- JPA haalt de waarde op van de kolom `id` in dit nieuwe record.  
We veronderstellen dat deze waarde bijvoorbeeld 7 is.
- `insert into individuelecursussen(id, duurtijd) values (7, 12);`

### 17.3.1 Voordelen van table per subclass

- ⊕ Als je een attribuut toevoegt aan één van de classes, moet je maar één kolom toevoegen aan één table.
- ⊕ Je kan elke kolom als verplicht in te vullen kolom aanduiden als je dit wenst.

### 17.3.2 Nadelen van table per subclass

- ⊖ Als je een entity toevoegt, stuurt JPA `insert` statements naar *meerdere* tables. Ook bij het wijzigen of verwijderen van een klassikale cursus of zelfstudie cursus stuurt JPA statements naar *meerdere* tables. Dit benadeelt de performantie.
- ⊖ Om entities te lezen, doet het `select` statement een `join` van meerdere tables. Dit benadeelt de performantie.

Je doet volgende stappen, zodat Eclipse een correct beeld krijgt van de nieuwe tables:

1. Je klapt in de Data Source Explorer niveau's van je database open tot je de map Tables ziet.
2. Je klikt met de rechtermuisknop op Tables en je kiest Refresh.
3. Je kiest in het menu Project de opdracht Clean.

### 17.3.3 Annotations bij de base class

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@Table(name = "cursussen")
public abstract class Cursus implements Serializable {
    ...
}
```

1

- (1) Bij table per class hiërarchy plaats je strategy op JOINED.

Je kan bij `@Entity` onterecht de foutmelding `Discriminator column "DTYPE" cannot be resolved on table "cursussen"` krijgen. Je mag deze foutmelding negeren.

### 17.3.4 Annotations bij de derived classes

```
@Entity
@Table(name = "groepscursussen")
public class GroepsCursus extends Cursus {
    ...
}

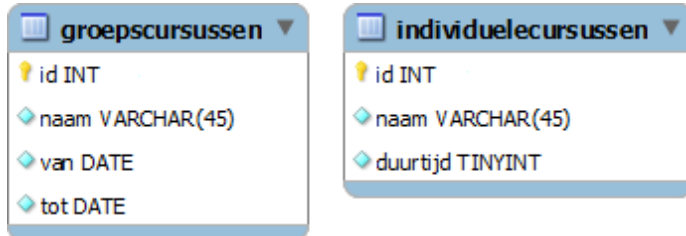
@Entity
@Table(name = "individuelecursussen")
public class IndividueleCursus extends Cursus {
    ...
}
```

De rest van je Java code wijzigt niet. Je kan de website terug uitproberen.

## 17.4 Table per concrete class

De database bevat enkel tables voor de laagste classes uit de inheritance hiërarchy (classes waarvan geen andere classes erven).

Je voert het script `TablePerConcreteClass.sql` uit. Dit verwijdert de table cursussen en voegt nieuwe versies van de tables groeps cursussen en individuele cursussen toe.



### 17.4.1 Voordelen van table per concrete class

- ➕ De tables bevatten geen overbodige kolommen.  
De table groeps cursussen bevat enkel kolommen met data over groeps cursussen.
- ➕ Als je een entity toevoegt, stuurt JPA een **insert** statement naar één table.  
Als je bijvoorbeeld een groeps cursus toevoegt, stuurt JPA een **insert** statement naar de table groeps cursussen. Ook het wijzigen of verwijderen van een cursus doet JPA met één statement naar één table.

### 17.4.2 Nadelen van table per concrete class

- ➖ Als je een attribuut toevoegt aan de base class (Cursus), moet je aan meerdere tables een kolom toevoegen.  
Als je bijvoorbeeld prijs toevoegt aan Cursus, moet je een kolom Prijs toevoegen aan de table groeps cursussen én aan de table individuele cursussen.
- ➖ Records in groeps cursussen mogen niet dezelfde primary key waarde hebben als records in individuele cursussen. De hier toegepaste oplossing: de autonummering van groeps cursussen begint bij 1, die van individuele cursussen bij 1000000.

Je doet volgende stappen, zodat Eclipse een correct beeld krijgt van de nieuwe tables:

1. Je klappt in de Data Source Explorer niveau's van je database open tot je de map Tables ziet.
2. Je klikt met de rechtermuisknop op Tables en je kiest Refresh.
3. Je kiest in het menu Project de opdracht Clean.

### 17.4.3 Annotations

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Cursus implements Serializable {
    ...
}
```

❶

- (1) Als je inheritance nabootst met table per concrete class, plaats je strategy op TABLE\_PER\_CLASS.

Je verwijdert `@GeneratedValue(strategy = GenerationType.IDENTITY)`.

Zoniet denkt JPA dat de autonummering van beide tables bij 1 begint en krijg je een fout bij het starten van de applicatie.

GroepsCursus en IndividueleCursus annotations blijven dezelfde.



De rest van je Java code wijzigt ook niet. Je commit de sources en je publiceert op GitHub. Je kan de website terug uitproberen.



Food en non-food artikels: zie takenbundel

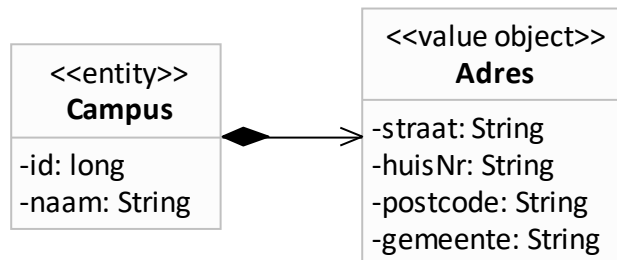
## 18 VALUE OBJECTS

Twee soorten objecten stellen dingen voor uit de werkelijkheid: Entities en Value objecten:

Entity 	Value object 
Bevat een identiteit attribuut (dat elke entity uniek voorstelt).	Bevat geen identiteit attribuut (dat elk value object uniek voorstelt)..
Heeft een zelfstandige levensduur: een entity verdwijnt niet op het moment dat een ander object verdwijnt.	Heeft geen zelfstandige levensduur: een value object verdwijnt op het moment dat een ander object verdwijnt.

Je gebruikt als voorbeeld de entity class Campus en de value object class Adres.

Er is ook een verband tussen deze classes: één campus heeft één adres



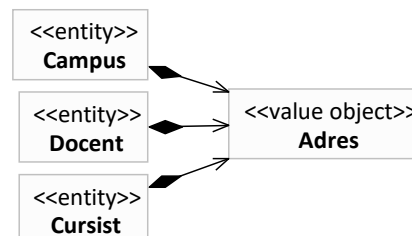
Campus is een entity:

- Campus bevat een attribuut id dat elke campus uniek voorstelt.
- Een campus verdwijnt niet op het moment dat een ander object verdwijnt.

Adres is een value object:

- Adres bevat geen attribuut dat elk adres uniek voorstelt.
- Als we een campus uit het systeem verwijderen, hebben we het adres van die campus ook niet meer nodig en verdwijnt dit adres dus.

Value object classes zijn herbruikbaar: ook een docent heeft een adres, ook een cursist heeft een adres, ....



### 18.1 Immutable value objecten

Je kan een class schrijven als mutable of immutable:

<b>Mutable</b>	De class bevat methods (bvb. setters) om na de aanmaak van een object van die class attributen te wijzigen.
<b>Immutable</b>	De class bevat een constructor met een parameter per attribuut. Je kan na het aanmaken van een object de attributen enkel lezen, niet wijzigen. De class bevat dus bijvoorbeeld geen setters.

Men raadt aan value objecten immutable te maken.

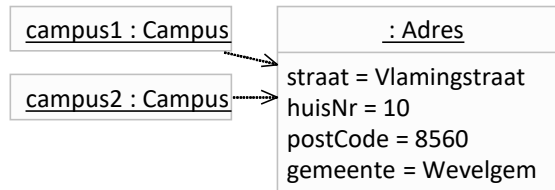
Je ziet in volgend voorbeeld wat er verkeerd loopt als een value object mutable is

```

Campus campus1 = new Campus();
campus1.setAdres(new Adres("Vlamingstraat", "10", "8560", "Wevelgem"));
Campus campus2 = new Campus();
campus2.setAdres(campus1.getAdres()); // de campussen delen hetzelfde gebouw
  
```



De Campus objecten campus1 en campus2 verwijzen op dit moment naar hetzelfde Adres object.

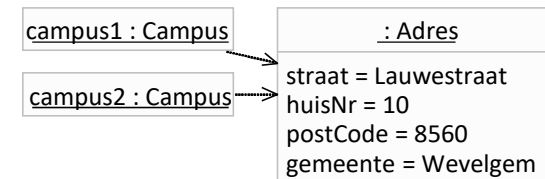


```
campus2.getAdres().setStraat("Lauwestraat"); // campus2 verhuist
```

Je hebt de straat van het Adres object gewijzigd waar campus2 naar verwijst.

Je beseft daarbij niet dat campus1 naar hetzelfde Adres object verwijst.

Je hebt dus 'per ongeluk' ook het adres van campus1 gewijzigd!

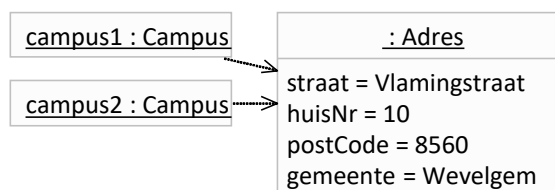


Je vermijdt deze fout door Adres immutable te maken (geen setters):

```

Campus campus1 = new Campus();
campus1.setAdres(new Adres("Vlamingstraat", "10", "8560", "Wevelgem"));
Campus campus2 = new Campus();
campus2.setAdres(campus1.getAdres()); // de campussen delen hetzelfde gebouw
  
```

De Campus objecten campus1 en campus2 verwijzen op dit moment naar hetzelfde Adres object



```

// campus2 verhuist.
// Je kan de straat van het bestaande adres niet wijzigen (immutable)
// Je maakt dus een nieuw Adres object dat je verbindt met campus2.
// Het oorspronkelijke Adres object (verbonden aan campus1) blijft intact.
campus2.setAdres(new Adres("Lauwestraat", "10", "8560", "Wevelgem"));
  
```



Opmerking: de standaard Java libraries bevatten ook immutable value objecten:

- String

De Java documentatie zegt:

Strings are constant; their values cannot be changed after they are created ... Because String objects are immutable they can be shared.

- BigDecimal

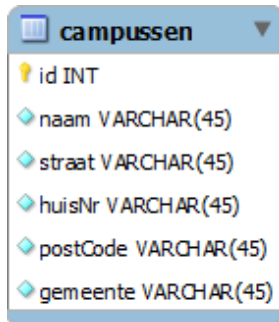
De Java documentatie zegt:

Immutable, arbitrary-precision signed decimal numbers.



## 18.2 Database

Als één entity één bijbehorend value object heeft, bevat het record dat bij de entity hoort ook kolommen voor de attributen van het bijbehorende value object.



De table campussen bevat campus data, waaronder ook adres data.

Het kan dus dat je dezelfde data:

- in het intern geheugen in meerdere classes opsplijt (om tot de herbruikbare class Adres te komen).
- in de database in één table bijhoudt.

De kans dat meerdere campussen hetzelfde adres hebben is zeer klein. Het is dus zinloos Straat, HuisNr, PostCode en Gemeente af te splitsen in een aparte table adressen. Dit zou de performantie benadelen.

## 18.3 JPA en value object classes

JPA stelt een aantal voorwaarden aan een value object class:

- Je tikt `@Embeddable` voor de class.
- Elke private variabele heeft een bijbehorende kolom in de table. Als dit niet het geval is, schrijf je voor de variabele `@Transient`. Als de naam van de kolom verschilt van de naam van de private variabele, schrijf je voor de variabele `@Column`, waarbij je de kolomnaam vermeldt.
- De class heeft een default constructor. Als je liever hebt dat zo weinig mogelijk classes deze constructor kunnen gebruiken, mag je er **protected** bij tikken.
- Je vermeldt de class in `persistence.xml`.
- JPA raadt aan dat de class de interface `Serializable` implementeert. Zo kan je een value object als session attribuut onthouden in een website.

JPA stelt een voorwaarde als je een value object class gebruikt in een entity class:

- Je schrijft (in de entity class) `@Embedded` voor de private variabele waarvan het type een value object class is.

## 18.4 De value object class

```
package be.vdab.valueobjects;
// enkele imports ...

@Embeddable
public class Adres implements Serializable {
    private static final long serialVersionUID = 1L;
    private String straat;
    private String huisNr;
    private String postcode;
    private String gemeente;
    // je maakt voor straat, huisNr, postcode en gemeente getters, geen setters
    public Adres(
        String straat, String huisNr, String postcode, String gemeente) {
        this.straat = straat;
        this.huisNr = huisNr;
        this.postcode = postcode;
        this.gemeente = gemeente;
    }
    protected Adres() {
    }
}
```

- (1) Je tikt `@Embeddable` voor een value object class.
- (2) Als je geen JPA gebruikt, kan je bij private variabelen **final** tikken. De compiler controleert dan dat je de variabele enkel bij zijn declaratie of in de constructor invult. JPA werkt echter niet samen met **final** variabelen.

- (3) De constructor bevat een parameter per private variabele.
- (4) Een value object class moet bij JPA een default constructor hebben.

Je vermeldt de class in persistence.xml, voor <properties>:

```
<class>be.vdab.valueobjects.Adres</class>
```

## 18.5 De entity class die de value object class gebruikt

```
package be.vdab.entities;
// enkele imports ...
@Entity
@Table(name="campussen")
public class Campus implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String naam;
    @Embedded
    private Adres adres;
    // je maakt getters en setters voor naam en adres
    public Campus(String naam, Adres adres) {
        setNaam(naam);
        setAdres(adres);
    }
    protected Campus() {}
}
```

- (1) Je tikt @Embedded voor een variabele met als type een value object class.

Je vermeldt de class in persistence.xml, voor <properties>:

```
<class>be.vdab.entities.Campus</class>
```

Je klikt met de rechtermuisknop in het Dali diagram en je kiest Show all Persistence Types. Je ziet de twee nieuwe classes en hun compositie verband.

## 18.6 Service layer en repository layer

Value objecten hebben geen eigen service layer of repository layer. Ze worden mee uit de database gelezen als je de entity leest die bij het value object hoort. Ze worden mee naar de database geschreven als je de entity, die bij het value object hoort, naar de database schrijft.

## 18.7 CRUD operaties

Je kan CRUD operaties toepassen op een entity die een value object bevat zonder nieuwe kennis. Je vindt hier onder voorbeeldcode, zonder service layer en repository layer.

### 18.7.1 Create

```
entityManager.getTransaction().begin();
Adres adres = new Adres("straat X", "huisnr X", "postcode X", "gemeente X");
Campus campus = new Campus("naam X", adres);
entityManager.persist(campus);
// JPA stuurt een insert statement naar de table campussen
entityManager.getTransaction().commit();
```

### 18.7.2 Read

```
Campus campus = entityManager.find(Campus.class, 1L);
// JPA stuurt een select statement naar de table campussen,
// maakt met de data uit dit record een Campus object en verbonden Adres object
String gemeente = campus.getAdres().getGemeente();
```

### 18.7.3 Update

```
entityManager.getTransaction().begin();
Campus campus = entityManager.find(Campus.class, 1L);
```

```
Adres nieuwAdres = new Adres("straat Y", "huisnr Y", "postcode Y", "gemeente Y");
campus.setAdres(nieuwAdres);
entityManager.getTransaction().commit();
// JPA stuurt een update statement naar de table campussen
// om het record met primary key waarde 1 te wijzigen.
```

#### 18.7.4 Delete

```
entityManager.getTransaction().begin();
Campus campus = entityManager.find(Campus.class, 1L);
entityManager.remove(campus);
entityManager.getTransaction().commit();
// JPA stuurt een delete statement naar de table campussen
// om het record met primary key waarde 1 te verwijderen
```

### 18.8 JPQL

Je kan in het **where** deel en **order by** deel van een query verwijzen naar attributen van een value object die bij een entity horen.

Voorbeeld: je leest de campussen uit West-Vlaanderen, gesorteerd op gemeente:

```
select c
from Campus c
where c.adres.postcode between '8000' and '8999'
order by c.adres.gemeente
```

Voorbeeld: De gebruiker tikt de naam van een gemeente. Jij toont de campussen in die gemeente.

Je voegt queries toe aan orm.xml:

```
<named-query name='Campus.findByGemeente'>
  <query>
    select c from Campus c where c.adres.gemeente = :gemeente order by c.naam
  </query>
</named-query>
<named-query name='Campus.findAll'> <!-- voor later in de cursus -->
  <query>
    select c from Campus c order by c.naam
  </query>
</named-query>
```

Je maakt een class CampusRepository:

```
package be.vdab.repositories;
// enkele imports ...
public class CampusRepository extends AbstractRepository {
    public List<Campus> findByGemeente(String gemeente) {
        return getEntityManager()
            .createNamedQuery("Campus.findByGemeente", Campus.class)
            .setParameter("gemeente", gemeente)
            .getResultList();
    }
    public List<Campus> findAll() { // voor later in de cursus
        return getEntityManager().createNamedQuery("Campus.findAll", Campus.class)
            .getResultList();
    }
    public Optional<Campus> read(long id) { // voor later in de cursus
        return Optional.ofNullable(getEntityManager().find(Campus.class, id));
    }
}
```

Je maakt een class CampusService:

```
package be.vdab.services;
// enkele imports ...
public class CampusService extends AbstractService {
    private final CampusRepository campusRepository = new CampusRepository();
    public List<Campus> findByGemeente(String gemeente) {
        return campusRepository.findByGemeente(gemeente);
    }
    public List<Campus> findAll() { // voor later in de cursus
        return campusRepository.findAll();
    }
    public Optional<Campus> read(long id) { // voor later in de cursus
        return campusRepository.read(id);
    }
}
```

InGemeenteServlet verwerkt GET requests naar /campussen/ingemeente.htm:

```
package be.vdab.servlets.campussen;
// enkele imports ...
@WebServlet("/campussen/ingemeente.htm")
public class InGemeenteServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private static final String VIEW = "/WEB-INF/JSP/campussen/ingemeente.jsp";
    private final transient CampusService campusService = new CampusService();
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String gemeente = request.getParameter("gemeente");
        if (gemeente != null) {
            if (gemeente.trim().isEmpty()) {
                request.setAttribute("fouten",
                    Collections.singletonMap("gemeente", "verplicht"));
            } else {
                request.setAttribute("campussen",
                    campusService.findByGemeente(gemeente));
            }
        }
        request.getRequestDispatcher(VIEW).forward(request, response);
    }
}
```

Je maakt ingemeente.jsp in WEB-INF/JSP/campussen:

```
<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%>
<%@taglib uri='http://java.sun.com/jsp/jstl/core' prefix='c'%>
<%@taglib uri='http://vdab.be/tags' prefix='v' %>
<!doctype html>
<html lang='nl'>
    <head>
        <v:head title='Campussen in
            ${empty param.gemeente ? "een gemeente" : param.gemeente}'></v:head>
        </head>
        <body>
            <v:menu/>
            <h1>Campussen in
                ${empty param.gemeente ? 'een gemeente' : param.gemeente}</h1>
            <form>
                <label>Gemeente: <span>${fouten.gemeente}</span>
                    <input name='gemeente' value='${param.gemeente}' autofocus required
                        type='search'></label>
                <input type='submit' value='Zoeken'>
            </form>
        </body>
    </html>
```

```

</form>
<c:if test='${not empty param and empty fouten and empty campussen}'>
    Geen campussen gevonden
</c:if>
<c:if test="${not empty campussen}">
    <ul>
        <c:forEach items='${campussen}' var='campus'>
            <li>${campus.naam}: ${campus.adres.straat} ${campus.adres.huisNr}</li>
        </c:forEach>
    </ul>
</c:if>
</body>
</html>

```

Je commit de sources en je publiceert op GitHub. Je kan de website uitproberen.

## 18.9 Value object classes detecteren

Met volgende tip detecteert je value object classes die je toevoegt aan je applicatie.

Als je een groep kolommen vindt in één of meerdere databasetables, onderzoek je of voor die groep een groepsnaam bestaat. Als dit zo is, maak je een value object class met die groepsnaam. Deze class bevat per kolom uit die groep een private variabele.

Fictief voorbeeld:

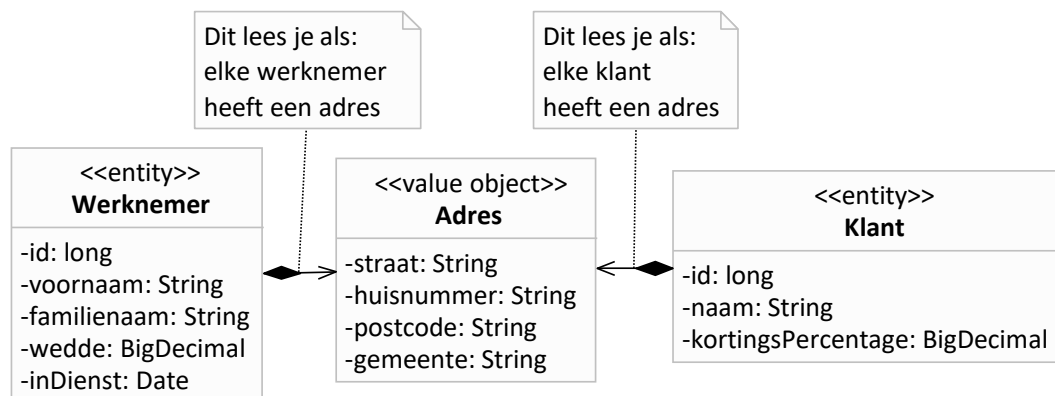
Werknemers	Klanten
id INT	id INT
voornaam VARCHAR(45)	naam VARCHAR(45)
familienaam VARCHAR(45)	straat VARCHAR(45)
straat VARCHAR(45)	huisnummer VARCHAR(45)
huisnummer VARCHAR(45)	postcode VARCHAR(45)
postcode VARCHAR(45)	gemeente VARCHAR(45)
gemeente VARCHAR(45)	kortingsPercentage DECIMAL(10,2)
wedde DECIMAL(10,2)	
inDienst DATE	

Je vindt in de table Werknemers én in de table Klanten eenzelfde groep kolommen: straat, huisnummer, postcode, gemeente.

Voor die groep bestaat een groepsnaam: Adres.

Je maakt dus in je applicatie een value object class Adres

De Java classes zullen er als volgt uitzien:



## 18.10 Aggregate

De samenhang van een entity en zijn bijbehorende value objecten(en) heet een aggregate.

In het voorbeeld hierboven zijn er twee aggregates:

- De werknemer en zijn adres.
- De klant en zijn adres.

## 19 EEN VERZAMELING VALUE OBJECTEN MET EEN BASISTYPE

De multiplicititeit tussen de entiteit Campus en het value object Adres is één: één Campus heeft één Adres.

Bij meervoudige multiplicititeit bevat de entity *een verzameling* value objecten. Deze verzameling bevindt zich in een andere table dan de tables met de entities.

Het type van de value objecten in de verzameling kan:

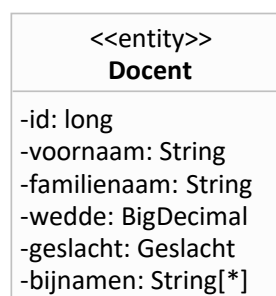
- een basistype zijn: Byte, Short, Integer, Long, Float, Double, Boolean, Char, String, BigDecimal, Date
- een eigen geschreven class zijn.

Je leert in dit hoofdstuk werken met basistypes.

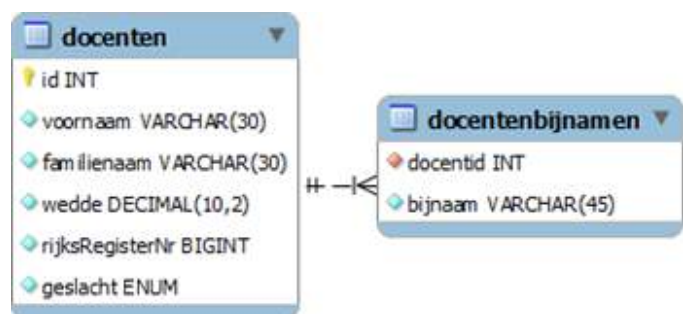
Je leert in het volgend hoofdstuk werken met een eigen geschreven class.

Voorbeeld: een docent heeft een verzameling bijnamen (van het basistype String):

De entity class Docent



De tables in de database



Docent is een entity:

- Docent bevat een attribuut id dat elke docent uniek voorstelt.
- Een docent verdwijnt niet op het moment dat een ander object verdwijnt.

De String array bijnamen is een verzameling value objecten:

- Een bijnaam bevat geen attribuut dat elke bijnaam uniek voorstelt.
- Als we een docent uit het systeem verwijderen, hebben we de bijnamen van die docent ook niet meer nodig en verdwijnen die dus.

Je definieert in de entity class Docent de verzameling bijnamen als een Set<String>:

```

@ElementCollection
@CollectionTable(name = "docentenbijnamen",
    joinColumns = @JoinColumn(name = "docentid") )
@Column(name = "Bijnaam")
private Set<String> bijnamen;
  
```

①  
②  
③  
④  
⑤

- (1) Je tikt @ElementCollection bij een variabele (zie 5) die een verzameling value objecten voorstelt.
- (2) Je geeft met @CollectionTable de naam van de table (docentenbijnamen) aan die de value objecten bevat.
- (3) Je geeft met @JoinColumn een kolom in deze table aan. Je kiest de foreign key kolom die verwijst naar primary kolom in de table (docenten) die hoort bij de huidige entity class (Docent). Je vult met deze @JoinColumn de parameter joinColumns van @CollectionTable.
- (4) Je geeft met @Column de naam van de kolom aan die hoort bij de value objecten in de verzameling.
- (5) JPA ondersteunt de types List, Set en Map uit de package java.util.

Je voegt aan de geparametriseerde constructor volgende regel toe:

```
bijnamen = new HashSet<>();
```



JPA stelt zelf geen vereisten aan een getter voor de verzameling.

Los van JPA raden we aan de getter van een verzameling als volgt te schrijven:

```
public Set<String> getBijnamen() {  
    return bijnamen; ❶  
    return Collections.unmodifiableSet(bijnamen); ❷  
}
```

- (1) Je leest met de method `getBijnamen` de bijnamen van een Docent  
 Als je `return bijnamen;` schrijft, kan je met `getBijnamen` per ongeluk een bijnaam toevoegen aan de docent: `docent.getBijnamen().add("Polle pap");`  
 Je kan ook per ongeluk een bijnaam verwijderen:  
`docent.getBijnamen().remove("Polle pap");`
- (2) Je verhindert dit met de static `Collections` method `unmodifiableSet`.  
 Je geeft een `Set` mee. Je krijgt een `Set` terug met dezelfde elementen.  
 Als je op die `Set` `add` of `remove` uitvoert, krijg je een `UnsupportedOperationException`.  
 De getter geeft zo een read-only voorstelling van de `Set`.

### 19.1 De verzameling value objecten lezen uit de database

Je gebruikt geen method om de verzameling value objecten uit de database te lezen.

JPA leest de verzameling value objecten zelf uit de database

wanneer je de verzameling voor de eerste keer aanspreekt in je Java code of in een JSP.

Wanneer je een entity uit de database leest,

leest JPA niet onmiddellijk de bijbehorende verzameling value objecten.

Bij de opdracht Docent `docent = entityManager.find(Docent.class, 1L);`

leest JPA nog niet de bijbehorende records uit de table `docentenbijnamen`.

Pas als je in het object `docent` de variabele `bijnamen` aanspreekt (bijvoorbeeld via de method `getBijnamen`), leest JPA de juiste records uit de table `docentenbijnamen`.

JPA maakt van de kolom `Bijnaam` in deze records een verzameling `String` objecten.

JPA wijst de variabele `bijnamen` in het object `docent` naar deze verzameling.

Het zo laat mogelijk ophalen van de value objecten die bij een entity horen, heet lazy loading.

Doel is de performantie van de applicatie hoog te houden: als je enkel docent data nodig hebt, en niet de bijbehorende bijnamen, leest JPA geen records uit de table `docentenbijnamen`.

Er is een vereiste: JPA moet de value objecten lezen met dezelfde `EntityManager` waarmee JPA de bijbehorende entity gelezen hebt. Anders krijg je een exception.

Je hebt aan deze vereiste voldaan: `JPAFilter` houdt gedurende de volledige verwerking van een browser request één `EntityManager` ter beschikking als `ThreadLocal` variabele.

Zelfs als je de bijnamen pas aanspreekt in de JSP, is de `EntityManager` waarmee je de campus hebt gelezen nog als `ThreadLocal` variabele ter beschikking.

Je toont in `zoeken.jsp` de bijnamen van een docent. Je tikt volgende regels voor `<h2>`:

```
<c:if test='${not empty docent.bijnamen}'>  
    <h2>Bijnamen</h2>  
    <ul><c:forEach items='${docent.bijnamen}' var='bijnaam'> ❶  
        <li>${bijnaam}</li>  
    </c:forEach></ul>  
</c:if>
```

- (1) Je spreekt voor het eerst de bijnamen van de docent aan.  
 JPA leest dan die bijnamen met een `select` opdracht naar de table `docentenbijnamen`

Je kan de website uitproberen.



## 19.2 Value object toevoegen aan de verzameling

Je voegt een method toe aan Docent:

```
public void addBijnaam(String bijnaam) {
    bijnamen.add(bijnaam);
}
```

Als je binnen een transactie value objecten toevoegt aan de verzameling, stuurt JPA automatisch, bij een commit op de transactie, per toegevoegd value object een SQL **insert** statement naar de table die bij de value objecten hoort.

### Voorbeeldcodefragment 1:

```
entityManager.getTransaction().begin();
Docent nieuweDocent = new Docent(...);
nieuweDocent.addBijnaam(...);
nieuweDocent.addBijnaam(...);
entityManager.persist(nieuweDocent);
entityManager.getTransaction().commit();
```

❶

- (1) JPA stuurt één **insert** statement naar de table docenten en twee **insert** statements naar de table docentenbijnamen.

### Voorbeeldcodefragment 2:

```
entityManager.getTransaction().begin();
Docent bestaandeDocent = entityManager.find(Docent.class, 1L);
bestaandeDocent.addBijnaam(...);
entityManager.getTransaction().commit();
```

❶

- (1) JPA stuurt één **insert** statement naar de table docentenbijnamen.

Je breidt het voorbeeld uit: de gebruiker kan bijnamen toevoegen.

Je voegt een method toe aan DocentService:

```
public void bijnaamToevoegen(long id, String bijnaam) {
    beginTransaction();
    try {
        docentRepository.read(id).ifPresent(docent->docent.addBijnaam(bijnaam));
        commit();
    } catch (PersistenceException ex) {
        rollback();
        throw ex;
    }
}
```

Je voegt code toe aan ZoekenServlet:

```
private static final String REDIRECT_URL = "%s/docenten/zoeken.htm?id=%d";
@Override
protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
    long id = Long.parseLong(request.getParameter("id"));
    String bijnaam = request.getParameter("bijnaam");
    if (bijnaam == null || bijnaam.trim().isEmpty()) {
        request.setAttribute("fouten",
            Collections.singletonMap("bijnaam", "verplicht"));
        docentService.read(id)
            .ifPresent(docent -> request.setAttribute("docent", docent));
        request.getRequestDispatcher(VIEW).forward(request, response);
    } else {
        docentService.bijnaamToevoegen(id, bijnaam);
        response.sendRedirect(response.encodeRedirectURL(
            String.format(REDIRECT_URL, request.getContextPath(), id)));
    }
}
```

Je voegt code toe in zoeken.jsp, voor de tweede <h2>:

```
<form method='post' id='toevoegform'>
  <label>Bijnaam: <span>${fouten.bijnaam}</span>
  <input name='bijnaam' value='${param.bijnaam}' required></label>
  <input type='submit' value='Toevoegen' id='toevoegknop'>
</form>
<script>
  document.getElementById('toevoegform').onsubmit = function() {
    document.getElementById('toevoegknop').disabled = true;
  };
</script>
```

Je kan de website uitproberen.

### 19.3 Value object verwijderen uit de verzameling

Je voegt een method toe aan Docent:

```
public void removeBijnaam(String bijnaam) {
    bijnamen.remove(bijnaam);
}
```

Als je binnen een transactie één of meerdere value objecten verwijdert uit de verzameling, stuurt JPA, bij een commit op de transactie, per verwijderd value object een SQL **delete** statement naar de table die bij de value objecten hoort.

Voorbeeldcodefragment:

```
entityManager.getTransaction().begin();
Docent bestaandeDocent = entityManager.find(Docent.class, 1L);
bestaandeDocent.removeBijnaam(...);
entityManager.getTransaction().commit();
```

JPA stuurt één **delete** statement naar de table docentenbijnamen.

Je breidt het voorbeeld uit. De gebruiker kan bijnamen verwijderen.

Je voegt een method toe aan DocentService:

```
public void bijnamenVerwijderen(long id, String[] bijnamen) {
    beginTransaction();
    try {
        docentRepository.read(id)
        .ifPresent(docent ->
            Arrays.stream(bijnamen).forEach(bijnaam->docent.removeBijnaam(bijnaam)));
        commit();
    } catch (PersistenceException ex) {
        rollback();
        throw ex;
    }
}
```

Je voegt code toe in zoeken.jsp, na de eerste </h2>:

```
<form method='post'>
```

Je wijzigt de regel <li>\${bijnaam}</li>

```
<li><label>${bijnaam}
<input type='checkbox' name='bijnaam' value='${bijnaam}'></label></li>
```

Je tikt na </ul>

```
<input type='submit' value='Bijnamen verwijderen' name='verwijderen'></form>
```

Je voegt een method toe in ZoekenServlet:

```
private void bijnamenVerwijderen(HttpServletRequest request,
    HttpServletResponse response, long id) throws IOException {
    String[] bijnamen = request.getParameterValues("bijnaam");
    if (bijnamen != null) {
        docentService.bijnamenVerwijderen(id, bijnamen);
    }
    response.sendRedirect(response.encodeRedirectURL(
        String.format(REDIRECT_URL, request.getContextPath(), id)));
}
```

Je verplaatst alle code binnen de method doPost (behalve de eerste regel) naar een nieuwe method bijnamenToevoegen. Deze method heeft volgende signatuur:

```
private void bijnamenToevoegen(HttpServletRequest request,
    HttpServletResponse response, long id) throws IOException, ServletException {
    ...
}
```

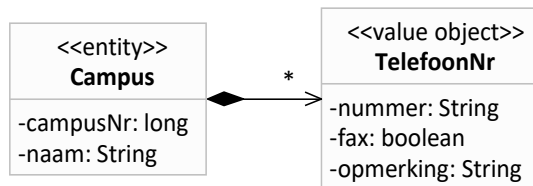
Je wijzigt de code in de method doPost:

```
long id = Long.parseLong(request.getParameter("id"));
if (request.getParameter("verwijderen") == null) {
    bijnamenToevoegen(request, response, id);
} else {
    bijnamenVerwijderen(request, response, id);
}
```

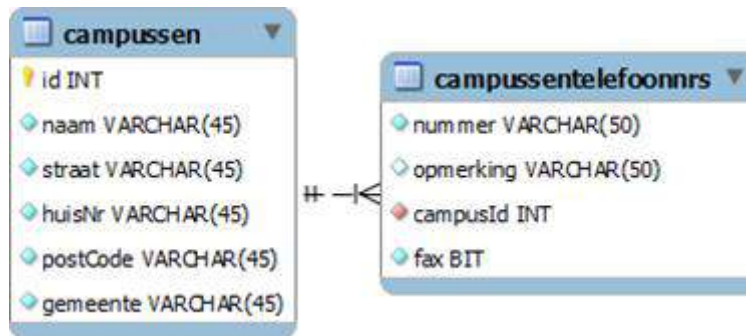
Je commit de sources en je publiceert op GitHub. Je kan de website uitproberen.

## 20 EEN VERZAMELING VALUE OBJECTEN MET EEN EIGEN TYPE

Het type van de value objecten in de verzameling kan een eigen geschreven class zijn.  
Voorbeeld: een Campus entity heeft meerdere TelefoonNr value objecten.



Als we een campus uit het systeem verwijderen, hebben we de telefoonnummers van die campus ook niet meer nodig en verdwijnen die dus.



De bijbehorende tables.

### 20.1 De value object class

```

package be.vdab.valueobjects;
// enkele imports
@Embeddable
public class TelefoonNr implements Serializable {
    private static final long serialVersionUID = 1L;
    private String nummer;
    private boolean fax;
    private String opmerking;
    public TelefoonNr(String nummer, boolean fax, String opmerking) {
        this.nummer = nummer;
        this.fax = fax;
        this.opmerking = opmerking; // je maakt getters voor nummer,fax,opmerking
    }
    protected TelefoonNr() {} // default constructor voor JPA
    @Override
    public String toString() {
        return nummer;
    }
    @Override
    public boolean equals(Object obj) {
        if ( ! (obj instanceof TelefoonNr)) {
            return false;
        }
        TelefoonNr telefoonNr = (TelefoonNr) obj;
        return nummer.equalsIgnoreCase(telefoonNr.nummer);
    }
    @Override
    public int hashCode() {
        return nummer.toUpperCase().hashCode();
    }
}
  
```

- (1) Je stelt straks in de class Campus de verzameling telefoonnummers van de campus voor als een Set<TelefoonNr>. Deze Set laat geen TelefoonNr objecten met hetzelfde nummer toe. Je baseert daartoe de equals method op het nummer.
- (2) Je baseert de method hashCode ook op het nummer.

De class TelefoonNr bevat geen variabele die verwijst naar het bijbehorende Campus object. Op die manier is de class TelefoonNr herbruikbaar in andere entity classes: Klant, ...

Je vermeldt de class in persistence.xml, voor <properties>:

```
<class>be.vdab.valueobjects.TelefoonNr</class>
```

## 20.2 De verzameling value objecten in de entity class

Je drukt de composition uit met extra code in Campus:

```
@ElementCollection
@CollectionTable(name = "campussentelefoonnr",
    joinColumns = @JoinColumn(name = "campusid"))
@OrderBy("fax")
private Set<TelefoonNr> telefoonNrs;
```

❶  
❷  
❸  
❹

- (1) Je tikt @ElementCollection bij een variabele met een verzameling value objecten.
- (2) Je geeft met @CollectionTable de naam van de table (campussentelefoonnr) aan die de value objecten bevat.
- (3) Je geeft met @JoinColumn een kolom in deze table aan.  
Je kiest de foreign key kolom die verwijst naar primary kolom in de table (campussen) die hoort bij de huidige entity class (Campus).  
Je vult met deze @JoinColumn de parameter joinColumns van @CollectionTable.
- (4) Je definieert met de optionele @OrderBy de volgorde waarmee JPA de value objecten moet lezen uit de database. Je vermeldt de naam van één of meerdere private variabelen (gescheiden door komma) die horen bij de kolom waarop je wil sorteren.  
Je kan omgekeerd sorteren met desc na een private variabele.

Je voegt een regel toe aan de geparametriseerde constructor:

```
telefoonNrs = new LinkedHashSet<>();
```

Je voegt methods toe:

```
public Set<TelefoonNr> getTelefoonNrs() {
    return Collections.unmodifiableSet(telefoonNrs);
}
public void add(TelefoonNr telefoonNr) {
    telefoonNrs.add(telefoonNr);
}
public void remove(TelefoonNr telefoonNr) {
    telefoonNrs.remove(telefoonNr);
}
```

Je klikt met de rechtermuisknop in het Dali diagram en je kiest Show all Persistence Types.

Je ziet de nieuwe class TelefoonNr en zijn compositie verband met de class Campus.

## 20.3 Verzameling value objecten lezen uit de database

JPA gebruikt ook hier lazy loading: JPA leest de verzameling value objecten uit de database op het moment dat je de verzameling voor de eerste keer aanspreekt.

Je voegt code toe aan ingemeente.jsp, na </li>:

```
<dl>
<c:forEach items="${campus.telefoonNrs}" var="telefoonNr">
    <dt>${telefoonNr.fax ? "Fax" : "Telefoon"}</dt>
    <dd>${telefoonNr.nummer} ${telefoonNr.opmerking}</dd>
</c:forEach>
</dl>
```

❶

- (1) Je spreekt hier voor het eerst de telefoonnummers aan. JPA leest die op dit moment met een select opdracht naar de table campussentelefoonnr.

Je commit de sources en je publiceert op GitHub. Je kan de website uitproberen.

## 20.4 Value object toevoegen aan de verzameling

Het toevoegen van een telefoonnummer aan een campus werkt zoals het toevoegen van een bijnaam aan een docent.

Als je binnen een transactie één of meerdere `TelefoonNr` objecten toevoegt aan de verzameling `telefoonNrs` in een `Campus` entity, stuurt JPA, bij een commit op de transactie, per toegevoegd value object een SQL **insert** statement naar de table `campussentelefoonnrs`.

## 20.5 Value object verwijderen uit de verzameling

Het verwijderen van een telefoonnummer uit een campus werkt zoals het verwijderen van een bijnaam uit een docent.

Als je binnen een transactie één of meerdere `TelefoonNr` objecten verwijdert uit de verzameling `telefoonNrs` in een `Campus` entity, stuurt JPA, bij een commit op de transactie, per verwijderd value object een SQL **delete** statement naar de table `campussentelefoonnrs` hoort.

## 20.6 Een entity verwijderen

Als je een entity verwijdert, verwijdert JPA de records die horen bij de verzameling value objecten.

- Als jij een docent verwijdert, verwijdert JPA ook de bijbehorende records in de table `docentenbijnamen`.
- Als jij een campus verwijdert, verwijdert JPA ook de bijbehorende records in de table `campussentelefoonnrs`.

## 20.7 Een value object in de verzameling wijzigen

Je kan een value object niet wijzigen als het immutable is.

Je wijzigt één telefoonnummer met volgende stappen.

1. Je verwijdert het te wijzigen telefoonnummer uit de verzameling.
2. Je voegt een telefoonnummer met de wijzigingen toe aan de verzameling.

Je kan een value object wel wijzigen als het mutable is.

JPA wijzigt bij de commit van de transactie het record dat bij het value object hoort.

Je detecteert met volgende tip een verzameling value objecten:

Als in een één op veel relatie tussen de tables A en B, bij het verwijderen van een record uit de table A de gerelateerde records uit de table B beter ook verwijderd worden, stel je de records uit de table B voor als een verzameling value objecten die behoren tot een entity die hoort bij table A.



Kortingen: zie takenbundel

## 21 MANY-TO-ONE ASSOCIATIE →

Je leert vanaf dit hoofdstuk werken met associaties tussen entities.

Je leert in dit hoofdstuk de eerste soort associatie: many-to-one.

Je voegt aan de class Docent een many-to-one associatie toe naar de class Campus.

- De multipliciteit is aan de kant van Docent \* en aan de kant van Campus 1: meerdere docenten horen bij één campus.
- Het is voorlopig een gerichte associatie:
  - Je weet welke campus bij een docent hoort.
  - Je weet niet welke docenten bij een campus horen.

Later in de cursus wordt dit een bidirectionele associatie, zodat je ook weet welke docenten bij een campus horen.

Een bidirectionele associatie is in gebruik handiger dan een gerichte associatie.

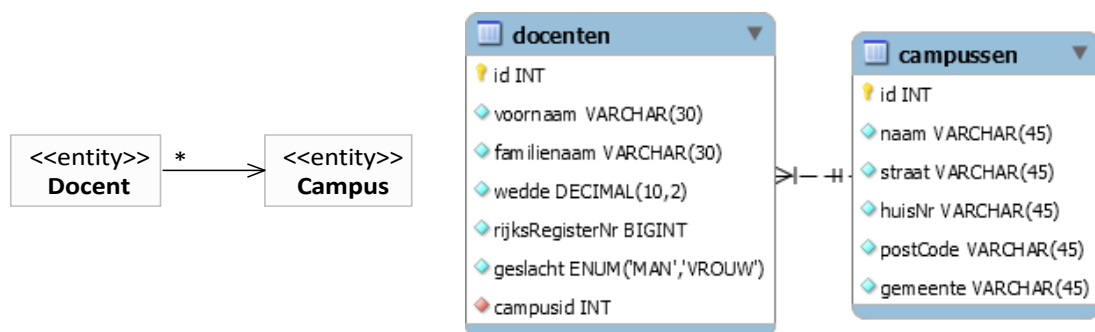
De code van een bidirectionele associatie is echter complexer dan die van een gerichte associatie.

Je voert het script CampusIdToevoegenAanDocenten.sql uit.

Dit voegt aan de table docenten een kolom campusid toe.

Deze is een foreign key die verwijst naar de primary key kolom id in de table campussen.

Er is nu een veel op één relatie tussen de table docenten en de table campussen:



Je doet volgende stappen, zodat Eclipse een correct beeld krijgt van de nieuwe tables:

1. Je klappt in de Data Source Explorer niveau's van je database open tot je de map Tables ziet.
2. Je klikt met de rechtermuisknop op Tables en je kiest Refresh.
3. Je kiest in het menu Project de opdracht Clean.

### 21.1 De Java code

Je definieert de associatie in de class Docent met een private variabele van het type Campus. Elk Docent object heeft zo een reference naar het bijbehorend Campus object. Dit is een groot verschil met de databasevoorstelling.



Als je in de class Docent de associatie zou uitdrukken met long campusId, kan je met die variabele enkel het campusnummer van een docent weten.

Als je de associatie uitdrukt met Campus campus, kan je met die variabele elke campus eigenschap van een docent weten, bijvoorbeeld campus.getNaam();

Je voegt code toe aan Docent:

```
@ManyToOne(optional = false)
@JoinColumn(name = "campusid")
private Campus campus;
// je maakt ook een getter en setter voor de variabele campus
```

①  
②

- (1) Je tikt `@ManyToOne` bij een variabele die een many-to-one associatie voorstelt. De foreign key kolom `campusId`, die bij deze associatie hoort, is in de database gedefinieerd als verplicht in te vullen. Je plaatst dan de parameter `optional` op `false`. JPA controleert dan voor het toevoegen of wijzigen van een record dat deze kolom wel degelijk ingevuld is en werpt een exception als dit niet het geval is.
- (2) De table `docenten` hoort bij de huidige class `Docent`. Je duidt met `@JoinColumn` de kolom `campusid` in deze table aan. Je kiest de foreign key kolom die verwijst naar de table `campussen` die hoort bij de geassocieerde entity (`Campus`).

Je ziet in het Dali diagram de associatie tussen de class `Docent` en de class `Campus`.

Voorbeeld gebruik van deze associatie:

je breidt de pagina uit waarin je een docent toevoegt.

De gebruiker kiest een campus voor de nieuwe docent uit een lijst met campussen



Je kent een campus toe aan een docent met volgende stappen:

1. Je maakt een nieuw `Docent` object of je leest een `Docent` object uit de database.
2. Je leest het `Campus` object dat je met het `Docent` object wil associëren.
3. Je associeert het campus object met het docent object: `docent.setCampus(campus)`;
4. JPA vult, bij de commit van de transactie, in het docenten record in de kolom `campusid` het nummer van de campus die je met het `Docent` object associeerde.

Je voegt een private variabele toe aan `ToevoegenServlet`:

```
private final transient CampusService campusService = new CampusService();
```

Je voegt een regel toe als eerste in de method `doGet`:

```
request.setAttribute("campussen", campusService.findAll());
```

Je voegt regels toe in de method `doPost`, voor `if (fouten.isEmpty()) {`:

```
String campusId = request.getParameter("campussen");
if (campusId == null) {
    fouten.put("campussen", "verplicht");
}
```

Je voegt een regel toe na `Docent docent = new Docent(...`:

```
campusService.read(Long.parseLong(campusId))
    .ifPresent(campus -> docent.setCampus(campus));
```

Je voegt een regel toe na `request.setAttribute("fouten", fouten)`:

```
request.setAttribute("campussen", campusService.findAll());
```

Je toont in `toevoegen.jsp` de lijst van campussen, voor `<input type='submit' ...>`:

```
<label>Campus: <span>${fouten.campussen}</span>
<select name='campussen' size='${campussen.size()}' required>
    <c:forEach items='${campussen}' var='campus'>
        <option value='${campus.id}'
            ${campus.id == param.campussen ? 'selected' : ''}>
            ${campus.naam} (${campus.adres.gemeente})</option>
    </c:forEach>
</select>
</label>
```

Je kan de website uitproberen.



## 21.2 Eager loading

JPA gebruikt op een many-to-one associatie default eager loading: als JPA een entity leest uit de many kant van de associatie (Docent), leest JPA tegelijk de bijbehorende entity uit de one kant van de associatie (Campus).

Je ziet dit in de use case “Docent zoeken” Je zoekt een docent.

Je ziet in het Eclipse venster Console dat JPA in zijn SQL select statement een record uit de table docenten én het bijbehorend record uit de table campussen leest:

```
select
...
  from
    docenten docent0_
  left outer join
    campussen campus1_
    on docent0_.campusid = campus1_.id
  where
    docent0_.id = ?
```

Je vraagt in deze use case enkel de docent te lezen, maar via de left outer join leest JPA tegelijk de bijbehorende campus.

Dit is interessant als je in de use case de campus data zou nodig hebben.

Dit is in deze use case echter niet het geval, en benadeelt de performantie.

Het performantieprobleem vergroot als je niet één, maar meerdere docenten leest.

Je ziet dit in de use case “Docenten van tot wedde”

Je ziet in het Eclipse venster Console de SQL select statements die JPA naar de database stuurt

Je ziet eerst een select statement die records zoekt in de table docenten.

Je ziet daarna meerdere select statements die elk één record zoeken in de table campussen.

JPA zoekt hiermee de campussen die horen bij de gelezen docenten.

In deze use case heb je echter de campus data niet nodig.

De overtoollige select statements in de table campussen benadelen de performantie.

## 21.3 Lazy loading

Je lost het performantieprobleem op door eager loading te vervangen door lazy loading.

Dit betekent: als JPA een entity te leest uit de many kant van de associatie (Docent), leest JPA niet onmiddellijk de bijbehorende entity uit de one kant van de associatie (Campus)

Pas als je in Java code of in een JSP de bijbehorende entity uit de one kant aanspreekt, leest JPA deze entity uit de database.

Je stelt lazy loading in met de parameter fetch van @ManyToOne.

Je wijzigt in Docent de regel @ManyToOne bij de variabele campus :

```
@ManyToOne(fetch = FetchType.LAZY, optional = false)
```

Je commit de sources en je publiceert op GitHub.

Als je nu de use cases “Docent zoeken” of “Docenten van tot wedde” uitvoert, stuurt JPA enkel een select statement naar de table docenten.



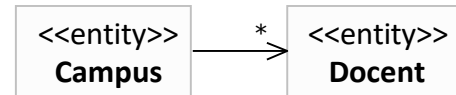
Opmerking: het lijkt bizar dat eager loading de default is, terwijl lazy loading interessanter is. Bij de definitie van de JPA standaard hadden sommige firma's het moeilijk lazy loading in hun JPA implementatie uit te werken. Dit is de (wat jammerlijke) reden waarom eager loading de default is.

## 22 ONE-TO-MANY ASSOCIATIE →

### 22.1 De associatie tussen de entity classes

Je voegt aan de class Campus een one-to-many associatie toe naar de class Docent.

- De multipliciteit is aan de kant van Campus 1 en aan de kant van Docent \*:  
bij één campus horen meerdere docenten.
- Het is een gerichte associatie:
  - Je weet welke docenten bij een campus horen.
  - Je weet niet welke campus bij een docent hoort.



Je plaatst daartoe de many-to-one associatie in de class Docent in commentaar:

- Je plaatst de variabele campus en bijbehorende annotations in commentaar.
- Je plaatst de method `getCampus` in commentaar.
- Je plaatst de method `setCampus` in commentaar.

Later in de cursus wordt dit een bidirectionele associatie, zodat je ook terug weet welke campus bij een docent hoort.

### 22.2 De relatie tussen de bijbehorende tables

De veel op één relatie tussen de table docenten en de table campussen is tegelijk een één op veel relatie tussen de table campussen en de table docenten.

Een relatie tussen tables is altijd bidirectioneel. De database blijft dus dezelfde.

### 22.3 De Java code

Je definieert de associatie in Campus met een private `Set<Docent>` variabele:

elk Campus object heeft een verzameling references naar de bijbehorende Docent objecten.

```

@OneToMany
@JoinColumn(name = "campusid")
@OrderBy("voornaam, familienaam")
private Set<Docent> docenten;
public Set<Docent> getDocenten() {
    return Collections.unmodifiableSet(docenten);
}
public void add(Docent docent) {
    docenten.add(docent);
}
public void remove(Docent docent) {
    docenten.remove(docent);
}
  
```

①  
②  
③

- (1) Je tikt `@OneToMany` bij een variabele die een one-to-many associatie voorstelt.
- (2) Bij de variabele `docenten` hoort de table `docenten`.  
Je definieert met `@JoinColumn` in die table de foreign key kolom (`campusid`) die verwijst naar de primary key van de table (`campussen`), die hoort bij de huidige class (`Campus`).
- (3) Je definieert met de optionele `@OrderBy` in welke volgorde JPA de Docent entities aan de many kant moet lezen uit de database. Je vermeldt de naam van één of meerdere private variabelen die horen bij de kolom waarop je wil sorteren.

een extra regel in de geparametriseerde constructor:

```
docenten = new LinkedHashSet<>();
```

### 22.4 One-to-many is lazy loading

JPA doet op een one-to-many associatie lazy loading.

Als je een Campus entity leest, leest JPA niet onmiddellijk de bijbehorende (Docent) entities.

Pas wanneer je in Java code of in een JSP de variabele `docenten` aanspreekt, leest JPA de Docent entities die bij de Campus entity horen.

## 22.5 De interface Set en de methods equals en hashCode

Objecten die je opneemt in een Set moeten correcte equals en hashCode methods hebben.

- equals moet **true** teruggeven als het object waarop je equals uitvoert (**this**) inhoudelijk gelijk is aan het object dat als parameter in de equals method binnenkomt.
- Objecten die volgens equals gelijk zijn, moeten hetzelfde getal teruggeven in hashCode.
- Als je hashCode uitvoert op objecten die volgens equals verschillend zijn, is het wenselijk (maar niet verplicht) dat je een verschillend getal krijgt. Dit maakt de Set performant.



Als je equals en hashCode baseert op de private variabele die hoort bij een automatisch gegenereerde primary key, verbreek je bovenstaande regels, en werkt de Set niet correct.

Dit probleem treedt op als je in Docent equals en hashCode baseert op id:

1. Je maakt een nieuw Docent object. id bevat 0. equals en hashCode werken op basis van 0.
2. Je associeert het Docent object met een Campus object: `campus.add(docent)`.  
Je voegt daarbij een reference naar het Docent object toe aan de `Set<Docent>` in campus.
3. Je slaat het Docent object op in de database. JPA vult de variabele id met het gegenereerde autonummer van het nieuw record. equals en hashCode werken nu op basis van het getal dat verschilt van 0. De `Set<Docent>` in het Campus object, die deze methods oproept, vindt het Docent object niet meer in zijn verzameling.

Een ander probleem treedt op als je meerdere nieuwe Docent objecten (nog niet opgeslagen in de database) toevoegt aan de `Set<Docent>` in het Campus object. Gezien al deze Docent objecten als id 0 hebben, zijn ze volgens hun equals en hashCode gelijk. Een Set laat geen gelijke objecten toe en laat dus ook niet meerdere nieuwe Docent objecten toe.

Je baseert equals en hashCode dus op één of meerdere variabelen die niet bij de automatisch gegenereerde primary key horen.

De waarde in deze variabele(n) moet bij elke entity van deze class uniek zijn.

Een verkeerde keuze is bijvoorbeeld de variabele voornaam.

De docenten **Peter** Van Petegem en **Peter** Van SantVliet zouden dezelfde zijn.

Ook de combinatie van voornaam én familienaam is niet uniek.

De variabele `rijksRegisterNr` is ideaal: elke docent heeft een uniek rijksregisternummer.

Zolang je het rijksregisternummer van een Docent object niet wijzigt terwijl dit object zich in een Set bevindt, werkt die Set correct.

### 22.5.1 De methods equals en hashCode

```
@Override
public boolean equals(Object obj) {
    if ( ! (obj instanceof Docent)) {
        return false;
    }
    return ((Docent) obj).rijksRegisterNr == rijksRegisterNr;
}

@Override
public int hashCode() {
    return Long.valueOf(rijksRegisterNr).hashCode();
}
```

Je maakt als voorbeeld een onderdeel waarmee de gebruikers de docenten per campus ziet.

Je toont de campussen als hyperlinks.

Als de gebruiker een campus aanklikt, toon je de docenten van die campus.

CampusDocentenServlet verwerkt GET requests naar /campussen/docenten.htm:

```
package be.vdab.servlets.campussen;
// enkele imports ...
@WebServlet("/campussen/docenten.htm")
public class CampusDocentenServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private static final String VIEW = "/WEB-INF/JSP/campussen/docenten.jsp";
    private final transient CampusService campusService = new CampusService();
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        request.setAttribute("campussen", campusService.findAll());
        String id = request.getParameter("id");
        if (id != null) {
            campusService.read(Long.parseLong(id))
                .ifPresent(campus -> request.setAttribute("campus", campus));
        }
        request.getRequestDispatcher(VIEW).forward(request, response);
    }
}
```

Je maakt docenten.jsp in WEB-INF/JSP/campussen:

```
<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%>
<%@taglib uri='http://java.sun.com/jsp/jstl/core' prefix='c'%>
<%@taglib uri='http://java.sun.com/jsp/jstl/fmt' prefix='fmt'%>
<%@taglib uri='http://vdab.be/tags' prefix='v'%>
<!doctype html>
<html lang='nl'>
<head>
    <v:head title='Docenten per campus' />
</head>
<body>
    <v:menu/>
    <h1>Docenten per campus</h1>
    <ul class='zonderbolletjes'>
        <c:forEach items='${campussen}' var='campus'>
            <c:url value='' var='url'>
                <c:param name='id' value='${campus.id}' />
            </c:url>
            <li><a href='${url}'>${campus.naam}</a></li>
        </c:forEach>
    </ul>
    <c:if test='${not empty campus}'>
        <h2>${campus.naam} (${campus.adres.gemeente})</h2>
        <dl>
            <c:forEach items='${campus.docenten}' var='docent'>
                <dt>${docent.naam}</dt>
                <dd>&euro; <fmt:formatNumber value='${docent.wedde}' maxFractionDigits='2'
                    minFractionDigits='2' /></dd>
            </c:forEach>
        </dl>
    </c:if>
</body>
</html>
```

Je commit de sources en je publiceert op GitHub. Je kan de website uitproberen.

## 23 DE METHODS EQUALS EN HASHCODE LATEN GENEREREN

Het is een goede gewoonte om de methods `equals` en `hashCode` te overriden in elke entity class en in elke value object class. Misschien verzamel je momenteel objecten van deze classes nu nog niet in een `Set`, maar doe je dit wel bij een uitbreiding of aanpassing van je applicatie.

Als de class nu reeds correcte `equals` en `hashCode` methods bevat, werkt die `Set` naar behoren: hij laat geen dubbele waarden toe.

De methods `equals` en `hashCode` zelf uitschrijven kan tijd vragen, zeker als deze methods moeten gebaseerd zijn op *meerdere* private variabelen.

Dit is het geval in de value object class `Adres`: twee `Adres` objecten verwijzen naar hetzelfde adres als hun straat, huisnummer, postcode én gemeente gelijk zijn.

Gelukkig kan je tijd sparen door deze methods door Eclipse toe te voegen aan de class:

1. Je opent de source van `Adres`.
2. Je kiest in het menu `Source` de opdracht `Generate hashCode() and equals()`.
3. Je laat de vinkjes staan bij de private variabelen waarop de methods `equals` en `hashCode` moeten gebaseerd zijn. Dit zijn hier *alle* private variabelen.
4. Je kiest bij `Insertion point` voor `Last member`.  
Je geeft zo aan dat Eclipse de gegenereerde methods `equals` en `hashCode` achteraan in de class moet toevoegen.
5. Je plaatst een vinkje bij `Use 'instanceof' to compare types`.  
Als je dit niet aanvinkt, kan je enkel `Adres` classes vergelijken, niet afgeleide classes van `Adres`. Gezien Hibernate intern soms met afgeleide classes van entities en value objecten werkt, is dit aanvinken belangrijk.
6. Je klikt op `OK`.

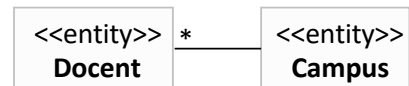


Baseer de methods `equals` en `hashCode` op een minimaal aantal private variabelen. In Docent was het voldoende deze methods enkel te baseren op de private variabele `rijksRegisterNr`: twee `Docent` objecten met eenzelfde `rijksRegisterNr` gaan over dezelfde docent.

## 24 BIDIRECTIONELE ASSOCIATIE

De associatie tussen Docent en Campus wordt bidirectioneel.

- Je weet welke campus bij een docent hoort én
- Je weet welke docenten bij een campus horen



### 24.1 De entity class aan de many kant van de associatie

Je haalt de private variabele campus, de bijbehorende JPA annotations, de bijbehorende getter en de bijbehorende setter uit commentaar.

We tonen hier nog eens de variabele en de bijbehorende JPA annotations:

```

@ManyToOne(fetch = FetchType.LAZY, optional = false)
@JoinColumn(name = "campusid")
private Campus campus;
  
```

### 24.2 De entity class aan de one kant van de associatie

De JPA annotations wijzigen aan de one kant bij een bidirectionele associatie.

```

@OneToMany(mappedBy = "campus")
@JoinColumn(name = "campusid")
@OrderBy("voornaam, familienaam")
private Set<Docent> docenten;
  
```

①  
②

- (1) Je voegt aan @OneToMany de parameter mappedBy toe.  
Je tikt daarbij de naam van de variabele (campus), die in de entity class aan de many kant van de associatie (Docent) de associatie voorstelt.
- (2) Je mag deze regel verwijderen.  
JPA vindt deze informatie reeds in @JoinColumn bij de variabele campus in de class Docent (de entity class aan de many-kant van de associatie).

### 24.3 De associatievariabelen bijwerken

Wanneer je een Docent object met een Campus object associeert, moet je:

- in het Docent object de variabele campus naar het Campus object laten wijzen.
- én in het Campus object aan de variabele docenten (een Set) een verwijzing toevoegen naar het Docent object.

Als je verkeerdelijk slechts één van de twee variabelen bijwerkt

- krijgt de rest van de applicatie een verkeerd beeld van de associatie.
- krijgt JPA een verkeerd beeld van de associatie en werkt de records niet correct bij.

Je tikt de code, waarmee je beide variabelen bijwerkt, in de entity classes zelf.

De andere lagen van je applicatie (repository, service, presentation), moeten dan deze verantwoordelijkheid niet op zich nemen.

Je moet er rekening mee houden dat je in een Servlet of in een class uit de service layer op twee manieren een Docent object met een Campus object kan associëren:

- `docent.setCampus(campus)` of
- `campus.add(docent)`

Je wijzigt in Campus de methods `add(Docent docent)` en `remove(Docent docent)`:

```

public void add(Docent docent) {
    docenten.add(docent);
    if (docent.getCampus() != this) { // als de andere kant nog niet bijgewerkt is
        docent.setCampus(this);      // werk je de andere kant bij
    }
}
  
```

```

public void remove(Docent docent) {
    docenten.remove(docent);
    if (docent.getCampus() == this) { // als de andere kant nog niet bijgewerkt is
        docent.setCampus(null);      // werk je de andere kant bij
    }
}

```

Je wijzigt in Docent de method setCampus:

```

public void setCampus(Campus campus) {
    if (this.campus != null && this.campus.getDocenten().contains(this)) {
        // als de andere kant nog niet bijgewerkt is
        this.campus.remove(this); // werk je de andere kant bij
    }
    this.campus = campus;
    if (campus != null && ! campus.getDocenten().contains(this)) {
        // als de andere kant nog niet bijgewerkt is
        campus.add(this); // werk je de andere kant bij
    }
}

```

Als de gebruiker een docent toevoegt, controleer je eerst of die docent nog niet bestaat.

Je maakt een class DocentBestaatAlException:

```

package be.vdab.exceptions;
public class DocentBestaatAlException extends RuntimeException {
    private static final long serialVersionUID = 1L;
}

```

Je maakt een query in orm.xml die zoekt of een docent met een rijksregisternummer al bestaat:

```

<named-query name='Docent.findByRijksRegisterNr'>
    <query>
        select d from Docent d where d.rijksRegisterNr = :rijksRegisterNr
    </query>
</named-query>

```

Je voegt een method toe aan DocentRepository:

```

public Optional<Docent> findByRijksRegisterNr(long rijksRegisterNr) {
    try {
        return Optional.of(getEntityManager()
            .createNamedQuery("Docent.findByRijksRegisterNr", Docent.class)
            .setParameter("rijksRegisterNr", rijksRegisterNr)
            .getSingleResult());
    } catch (NoResultException ex) {
        return Optional.empty();
    }
}

```

❶

- (1) Als de method getSingleResult geen record vindt, werpt ze een NoResultException. Je vangt deze fout op en je geeft **null** terug.

Je wijzigt in DocentService de method create:

```
public void create(Docent docent) {
    if (docentRepository.findByRijksRegisterNr(docent.getRijksRegisterNr())
        .isPresent()) {
        throw new DocentBestaatAlException();
    }
    beginTransaction();
    try {
        docentRepository.create(docent);
        commit();
    } catch (PersistenceException ex) {
        rollback();
        throw ex;
    }
}
```

Je wijzigt in ToevoegenServlet de **if** (fouten.isEmpty()) en de bijbehorende else:

```
if (fouten.isEmpty()) {
    Docent docent = new Docent(voornaam, familienaam, wedde,
        Geslacht.valueOf(geslacht), rijksRegisterNr);
    campusService.read(Long.parseLong(campusId))
        .ifPresent(campus -> docent.setCampus(campus));
    try {
        docentService.create(docent);
        response.sendRedirect(response.encodeRedirectURL(
            String.format(REDIRECT_URL, request.getContextPath(), docent.getId())));
    } catch (DocentBestaatAlException ex) {
        fouten.put("rijksregisternr", "bestaat al");
    }
}
if ( ! fouten.isEmpty()) {
    request.setAttribute("fouten", fouten);
    request.setAttribute("campussen", campusService.findAll());
    request.getRequestDispatcher(VIEW).forward(request, response);
}
```

Je commit de sources en je publiceert op GitHub. Je kan de website terug uitproberen.



## 25 MANY-TO-MANY ASSOCIATIE 🧑 - 🧑

### 25.1 Database

In een database kunnen twee tables geen directe veel-op-veel relatie hebben. Beide oorspronkelijke tables hebben een één op veel relatie met een tussentable.

De table verantwoordelijkheden beschrijft de verantwoordelijkheden van een docent.

Er is een veel-op-veel relatie tussen docenten en verantwoordelijkheden:

- één docent kan meerdere verantwoordelijkheden hebben
- meerdere docenten kunnen eenzelfde verantwoordelijkheid hebben



### 25.2 Java code

Dezelfde veel op veel associatie heeft geen tussenclass nodig in Java:



Je drukt deze bidirectionele associatie uit

- met een `Set<Verantwoordelijkheid>` variabele in de class `Docent`. Elk `Docent` object heeft zo een verzameling references naar de bijbehorende `Verantwoordelijkheid` objecten.
- met een `Set<Docent>` variabele in de class `Verantwoordelijkheid`. Elk `Verantwoordelijkheid` object heeft zo een verzameling references naar de bijbehorende `Docent` objecten.

Als je een `Docent` object met een `Verantwoordelijkheid` object associeert, moet je

- In het `Docent` object aan de variabele `verantwoordelijkheden` (een `Set`) een verwijzing naar het `Verantwoordelijkheid` object toevoegen.
- In het `Verantwoordelijkheid` object aan de variabele `docenten` (een `Set`) een verwijzing naar het `Docent` object toevoegen.

Als je verkeerdelijk slechts één van de twee variabelen bijwerkt

- krijgt de rest van de applicatie een verkeerd beeld van de associatie.
- krijgt JPA een verkeerd beeld van de associatie en wijzigt de records verkeerd.

Je maakt de class `Verantwoordelijkheid`:

```

package be.vdab.entities;
// enkele imports ...

@Entity
@Table(name = "verantwoordelijkheden")
public class Verantwoordelijkheid implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id; // je maakt zelf een getter
    private String naam; // je maakt zelf een getter
}
  
```

```

@Override
public boolean equals(Object obj) {
    if ( ! (obj instanceof Verantwoordelijkheid)) {
        return false;
    }
    return ((Verantwoordelijkheid) obj).naam.equalsIgnoreCase(this.naam);
}
@Override
public int hashCode() {
    return naam.toUpperCase().hashCode();
}
@ManyToMany
@JoinTable(
    name = "docentenverantwoordelijkheden",
    joinColumns = @JoinColumn(name = "verantwoordelijkheidid"),
    inverseJoinColumns = @JoinColumn(name = "docentid"))
private Set<Docent> docenten = new LinkedHashSet<>();
public void add(Docent docent) {
    docenten.add(docent);
    if ( ! docent.getVerantwoordelijkheden().contains(this)) {
        docent.add(this);
    }
}
public void remove(Docent docent) {
    docenten.remove(docent);
    if (docent.getVerantwoordelijkheden().contains(this)) {
        docent.remove(this);
    }
}
public Set<Docent> getDocenten() {
    return Collections.unmodifiableSet(docenten);
}
}

```

- (1) Je stelt straks in Docent de verantwoordelijkheden van de docent voor als een Set<Verantwoordelijkheid>. Deze Set laat geen Verantwoordelijkheid objecten met dezelfde naam toe. Je baseert daartoe de equals method op de naam.
- (2) Je baseert de method hashCode ook op de naam.
- (3) Je tikt @ManyToMany bij een variabele die een many-to-many associatie voorstelt.
- (4) Je definieert met @JoinTable de tussentable die hoort bij associatie.
- (5) Je tikt bij name de naam van de tussentable.
- (6) Je tikt bij joinColumns de naam van de kolom in de tussentable die de foreign key is naar de primary key van de table (verantwoordelijkheden) die hoort bij de huidige entity (Verantwoordelijkheid). Je vult joinColumns met een @JoinColumn.
- (7) Je tikt bij inverseJoinColumns de kolomnaam in de tussentable die de foreign key is naar de primary key van de table (docenten) die hoort bij de entity aan de andere associatie kant (Docent). Je vult inverseJoinColumns met een @JoinColumn.

Je vermeldt de class in persistence.xml, voor <properties>:

```
<class>be.vdab.entities.Verantwoordelijkheid</class>
```

Je voegt code toe aan Docent :

```
@ManyToMany(
    mappedBy = "docenten")
private Set<Verantwoordelijkheid> verantwoordelijkheden
    = new LinkedHashSet<>();
public void add(Verantwoordelijkheid verantwoordelijkheid) {
    verantwoordelijkheden.add(verantwoordelijkheid);
    if ( ! verantwoordelijkheid.getDocenten().contains(this)) {
        verantwoordelijkheid.add(this);
    }
}
public void remove(Verantwoordelijkheid verantwoordelijkheid) {
    verantwoordelijkheden.remove(verantwoordelijkheid);
    if (verantwoordelijkheid.getDocenten().contains(this)) {
        verantwoordelijkheid.remove(this);
    }
}
public Set<Verantwoordelijkheid> getVerantwoordelijkheden() {
    return Collections.unmodifiableSet(verantwoordelijkheden);
}
```

①  
②

- (1) Je tikt @ManyToMany bij een variabele die een many-to-many associatie voorstelt.
- (2) Je tikt bij de parameter mappedBy de variabele naam (docenten), die aan de andere associatie kant (Verantwoordelijkheid), de associatie voorstelt.

Je klikt met de rechtermuisknop in het Dali diagram en je kiest Show all Persistence Types. Je ziet de nieuwe class Verantwoordelijkheid en zijn associatie met de class Docent.

### 25.3 Een bidirectionele @ManyToMany

Je tikt bij een bidirectionele many-to-many associatie aan beide kanten @ManyToMany

- Je vermeldt bij één kant (bij ons Verantwoordelijkheid) alle detail van de tussentable.
- Je vermeldt bij de andere kant (in ons voorbeeld Docent) enkel de parameter mappedBy.

Het is niet belangrijk bij welke kant je de detail schrijft.

Je kan dus de detail van de tussentable ook schrijven in @ManyToMany in Docent :

```
@ManyToMany
@JoinTable(
    name = "docentenverantwoordelijkheden",
    joinColumns = @JoinColumn(name="docentId"),
    inverseJoinColumns = @JoinColumn(name="verantwoordelijkheidId"))
```

en enkel mappedBy in @ManyToMany in Verantwoordelijkheid:

```
@ManyToMany(mappedBy="verantwoordelijkheden")
```

Je voegt aan de geparametriseerde constructor van Docent een regel toe:

```
verantwoordelijkheden = new LinkedHashSet<>();
```

Je toont als voorbeeld in zoeken.jsp de verantwoordelijkheden van de docent.

Je tikt regels voor <h2>Acties</h2>:

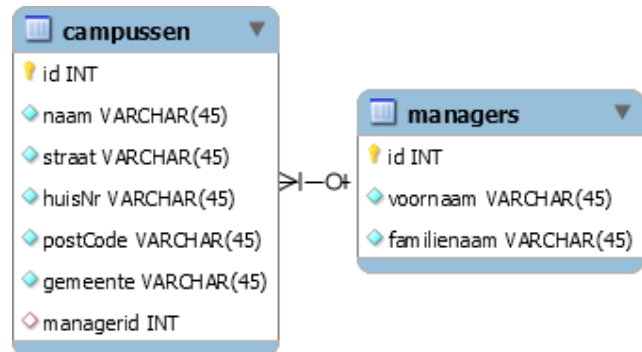
```
<c:if test='${not empty docent.verantwoordelijkheden}'>
    <h2>Verantwoordelijkheden</h2>
    <ul>
        <c:forEach items='${docent.verantwoordelijkheden}'
            var='verantwoordelijkheid'>
            <li>${verantwoordelijkheid.naam}</li>
        </c:forEach>
    </ul>
</c:if>
```

Je commit de sources en je publiceert op GitHub. Je kan de website uitproberen.

## 26 ONE-TO-ONE ASSOCIATIE

### 26.1 Database

Je voert het script  
 ManagerIdInCampussen.sql uit.  
 Dit maakt in de table campussen  
 een kolom managerid.  
 Deze is een foreign key  
 naar de table managers.  
 Er is nu een relatie tussen de table  
 campussen en de table managers.



Je doet volgende stappen, zodat Eclipse een correct beeld krijgt van de nieuwe tables:

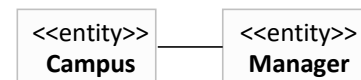
1. Je klapt in de Data Source Explorer niveau's van je database open tot je de map Tables ziet.
2. Je klikt met de rechtermuisknop op Tables en je kiest Refresh.
3. Je kiest in het menu Project de opdracht Clean.

### 26.2 Java

Je zal deze associatie beschrijven als bidirectioneel.

Je weet dan:

- welke manager bij een campus hoort.
- welke campus bij een manager hoort.



Je drukt deze associatie uit

- met een Manager variabele in de class Campus.  
Elk Campus object heeft zo een reference naar het bijbehorend Manager object.
- met een Campus variabele in de class Manager.  
Elk Manager object heeft zo een reference naar het bijbehorend Campus object.

Je voeg aan Campus code toe om de associatie uit te drukken:

```
@OneToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "managerid")
private Manager manager;
public Manager getManager() {
    return manager;
}
```

①  
②

- (1) Je tikt @OneToOne bij een variabele die een one-to-one associatie voorstelt.  
Een one-to-one associatie is default eager: als je een Campus entity leest uit de database, leest JPA tegelijk de bijbehorende Manager entity. Je kan overschakelen naar lazy loading door de parameter fetch op FetchType.LAZY te plaatsen.  
Als je nu een Campus entity leest uit de database, leest JPA de bijbehorende Manager pas wanneer je de manager van de campus voor de eerste keer aanspreekt.
- (2) De table campussen hoort bij de huidige entity (Campus).  
Je definieert met @JoinColumn de naam van de kolom in deze table die de foreign key bevat die verwijst naar de table (managers) die hoort bij de geassocieerde entity (Manager).

Je maakt een class Manager:

```
package be.vdab.entities;
// enkele imports
@Entity
@Table(name = "managers")
public class Manager implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String voornaam;
    private String familienaam;
    @OneToOne(
        fetch = FetchType.LAZY,
        mappedBy = "manager")
    private Campus campus;
    // Je maakt getters voor id, voornaam, familienaam en campus
}
```

❶  
❷  
❸

- (1) Je tikt @OneToOne bij een variabele die een one-to-one associatie voorstelt.
- (2) Ook deze one-to-one associatie is standaard eager: als je een Manager entity leest uit de database, leest JPA tegelijk de bijbehorende Campus entity. Je kan overschakelen naar lazy loading door de parameter fetch op FetchType.LAZY te plaatsen. Als je nu een Manager entity leest uit de database, leest JPA de bijbehorende Campus entity pas wanneer je de campus van de manager voor de eerste keer aanspreekt.
- (3) Je voegt aan @OneToOne de parameter mappedBy toe. Je tikt hierbij de variabele naam (manager), die aan de andere associatie kant (Campus) de associatie voorstelt.

Je vermeldt de class in persistence.xml, voor <properties>:

```
<class>be.vdab.entities.Manager</class>
```

Je klikt met de rechtermuisknop in het Dali diagram en je kiest Show all Persistence Types.

Je ziet de nieuwe class Manager en zijn associatie met de class Campus.

Je toont als voorbeeld in ingemeente.jsp per campus de bijbehorende manager, na </dl>:

```
<c:if test='${not empty campus.manager}'>
    Manager: ${campus.manager.voornaam} ${campus.manager.familienaam}
</c:if>
```

Je commit de sources en je publiceert op GitHub. Je kan de website uitproberen.

## 27 ASSOCIATIES VAN VALUE OBJECTEN NAAR ENTITIES

Je hebt nu geleerd hoe je associaties definieert van entities naar entities.

Je kan ook gerichte associaties definiëren van value objecten naar entities.

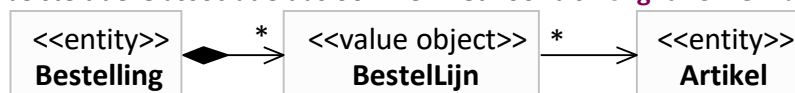
Je gebruikt bij de variabelen in de value object class, die verwijzen naar de entities, de JPA annotation `@ManyToOne` en `@ManyToOne`.

In het volgende voorbeeld bevat de class `BestelLijn` de regels

```
@ManyToOne(...)
```

```
private Artikel artikel;
```

Je stelt deze associatie dus ook hier niet voor als `long artikelid`;



Belangrijke beperkingen in JPA

- Je kan de composition tussen een entity en zijn value objects enkel uitdrukken als een gerichte associatie ( $\blacklozenge \rightarrow$ ), niet als een bidirectionele associatie. Je kan dus in `BestelLijn` geen `Bestelling` private variabele toevoegen en er JPA annotations voor schrijven.
- Je kan de associatie tussen een value object en een andere entity enkel uitdrukken als een gerichte associatie ( $\rightarrow$ ), niet als een bidirectionele associatie. Je kan dus in `Artikel` geen `List<BestelLijn>` private variabele toevoegen en er JPA annotations voor schrijven. Als je dit toch probeert, krijg je een exception bij de initialisatie van JPA.

Ervaring leert dat deze beperkingen niet hinderen in praktische applicaties

## 28 N + 1 PROBLEEM, JOIN FETCH QUERIES, ENTITY GRAPHS

### 28.1 N + 1 probleem



Het N + 1 probleem is een veel voorkomend performantieprobleem in JPA applicaties.

Je leert dit probleem kennen met extra code in `vantotwedde.jsp`: je toont per docent de campus.

- Je tikt `<th>Campus</th>` na `<th>Wedde</th>`.
- Je tikt `<td>${docent.campus.naam}</td>` voor de tweede `</tr>`.

Je voert de use case uit. Je ziet in het Eclipse venster Console dat JPA één select statement stuurt naar de table docenten, daarna veel gelijkaardige select statements naar de table campussen die elk één record lezen aan de hand van de primary key.

Deze overvloed van select statements is het N + 1 performantieprobleem:

- **1** staat voor het ene select statement naar de table docenten.  
JPA voert dit statement uit bij het uitvoeren van de named query `findByWeddeBetween`.  
Je vraagt in die named query records te lezen uit slechts één table: docenten
- **n** staat voor de meerdere gelijkaardige select statements naar de table campussen.  
JPA voert zo'n select statement uit telkens je in de JSP voor het eerst een bepaalde campus van een docent aanspreekt.

### 28.2 Join fetch

Als je zelf het SQL statement schrijft waarmee je docenten en de bijbehorende campussen leest, schrijf je één select statement waarin je de table docenten met de table campussen joint en zo in één performante query de docenten en hun bijbehorende campussen leest

```
select ... from docenten inner join campussen
on docenten.campusid = campussen.id
where ...
order by ...
```

Je kan een join ook in JPQL uitdrukken met de sleutelwoorden `join fetch`

Je wijzigt als voorbeeld in `orm.xml` de named query `Docent.findByWeddeBetween`

```
select d from Docent d join fetch d.campus
where d.wedde between :van and :tot
order by d.wedde, d.id
```

❶

- (1) Je tikt na de alias van de entity class (`d`) de sleutelwoorden `join fetch`.  
Je tikt daarna dezelfde alias, een punt en de naam van de variabele (`campus`) die in de entity class (`Docent`) naar de geassocieerde entity class (`Campus`) verwijst.

Je voert de use case opnieuw uit.

JPA vertaalt de named query naar een select statement. Je ziet dat JPA records leest uit de table docenten en (via een join in dit statement) de geassocieerde records uit de table campussen.

JPA voert de meerdere select statements naar de table campussen nu niet meer uit, wat de performantie sterk verbetert.

## 28.3 Entity graph

In JPA 2.1 werd het concept Entity graph geïntroduceerd als een alternatieve oplossing voor het N + 1 probleem, omdat join fetch volgende problemen heeft.

- Het is mogelijk dat je in een andere pagina van je website ook de docenten toont met een wedde tussen twee grenzen, maar zonder de bijbehorende campus data. Als je daarbij de named query `Docent.findByWeddeBetween` gebruikt krijg je een trage pagina, want die query leest ook campussen. Je maakt daarom voor die pagina een aparte named query `Docent.findByWeddeBetweenZonderCampussen`  

```
select d from Docent d
where d.wedde between :van and :tot
order by d.wedde, d.id
```

 Je bekomt op die manier veel sterk gelijkaardige queries, wat de onderhoudbaarheid en uitbreidbaarheid van de applicatie benadeelt.
- Je mag op de geassocieerde entities niet nog eens een join fetch doen. In onze query (`select d from Docent d join fetch d.campus ...`) zijn Campus entities de geassocieerde entities van Docent. Als je van die Campus entities ook de geassocieerde Manager entities wil lezen, mag je dit volgens de JPA standaard niet uitdrukken als `select d from Docent d join fetch d.campus c join fetch c.manager ...`  
 Als je in de JSP de manager van elke campus toont (`${docent.campus.manager.naam}`) verkrijg je met de named query zoals hij nu nog geschreven is weer het N + 1 probleem: gezien je de managers niet in de named query gelezen hebt, leest JPA ze één per één naarmate je ze aanspreekt in de JSP.

Bij Entity graphs druk je in de named query zelf geen join uit.

Je zal dit uitdrukken bij het oproepen van de named query.

Je mag de named query `Docent.findByWeddeBetween` dus terug wijzigen naar:

```
select d from Docent d
where d.wedde between :van and :tot
order by d.wedde, d.id
```

### 28.3.1 @NamedEntityGraph

Je kan in meer dan één named query de behoefte hebben om niet enkel de docenten te lezen, maar onmiddellijk ook de geassocieerde campussen (via een join).

Je drukt die behoefte één keer uit als een named entity graph, met `@NamedEntityGraph`.

Je schrijft `@NamedEntityGraph` voor de entity class die de behoefte heeft.

Je schrijft onze `@NamedEntityGraph` dus juist voor de class `Docent`:

```
@NamedEntityGraph(name = "Docent.metCampus",           ❶
    attributeNodes = @NamedAttributeNode("campus"))    ❷
```

- (1) Elke named entity graph moet een unieke naam hebben in je applicatie. De kans daartoe vergroot als je die naam begint met de naam van de huidige class.
- (2) Je vermeldt de naam van de private variabele `campus` in de huidige class `Docent`. Je drukt daarmee de behoefte uit dat bij het lezen van een `Docent` entity uit de database JPA direct ook de bijbehorende `Campus` entity moet lezen (via een join).

Je hebt deze behoefte nu als named entity graph gedefinieerd, maar JPA past die niet automatisch toe op elke named query die docenten leest. JPA zal dit pas doen als jij dit vraagt.

### 28.3.2 De oproep van de named query

Je tikt in de `DocentRepository` method `findByWeddeBetween` voor `.getResultList()`;

```
.setHint("javax.persistence.loadgraph",           ❶
    getEntityManager().createEntityGraph("Docent.metCampus"))  ❷
```

- (1) Je geeft met `setHint` een hint aan JPA. JPA past deze hint toe bij het uitvoeren van de named query. Elke hint heeft in JPA een naam. De hint om bij het uitvoeren van een named query rekening te houden met de behoefte beschreven met `@NamedEntityGraph` is `javax.persistence.loadgraph`.



- (2) Je zoekt de named entity graph die je met `@NamedEntityGraph` definieerde met de naam `Docent.metCampus`. JPA leest in die entity graph de behoefte om bij het lezen van een `Docent` direct de gerelateerde `Campus` te lezen en vertaalt de named query naar een SQL select statement met daarin
- ```
from docenten inner join campussen on docenten.campusid = campussen.id
```

Je voert de use case uit en ziet dat het N + 1 probleem vermeden is.

### 28.3.3 @NamedEntityGraphs

Als je bij een class meerdere `@NamedEntityGraph` annotations wil schrijven, moet je die tussen { en } verzamelen in een array en deze array gebruiken als parameter van `@NamedEntityGraphs`:

```
@NamedEntityGraphs({
    @NamedEntityGraph(...),
    @NamedEntityGraph(...)
})
```

### 28.3.4 Meer dan één geassocieerde entity

De named entity graph `Docent.metCampus` beschrijft de behoefte om met een `Docent` entity één geassocieerde entity (`Campus`) te lezen.

Je kan in een entity graph ook de behoefte beschrijven om daarnaast nog één of meerdere entities te lezen die met een `Docent` entity geassocieerd zijn.

Volgend voorbeeld drukt de behoefte uit om met een `Docent` entity direct de geassocieerde `Campus` entity én de geassocieerde `Verantwoordelijkheid` entities te lezen:

```
@NamedEntityGraph(name = "Docent.metCampusEnVerantwoordelijkheden",
    attributeNodes = {
        @NamedAttributeNode("campus"), @NamedAttributeNode("verantwoordelijkheden")})
```

Je vermeldt bij `attributeNodes` met { } één array van `@NamedAttributeNode` annotations.

### 28.3.5 De behoefte naar een geassocieerde entity van een geassocieerde entity

Je kan ook de behoefte hebben om een geassocieerde entity van een geassocieerde entity te lezen. Volgend voorbeeld drukt de behoefte uit om met een `Docent` entity direct de geassocieerde `Campus` entity én direct de geassocieerde `Manager` entity van die `Campus` entity te lezen:

```
@NamedEntityGraph(name = "Docent.metCampusEnManager",
    attributeNodes = @NamedAttributeNode(value = "campus",           ❶
        subgraph = "metManager"),                                  ❷
    subgraphs = @NamedSubgraph(name = "metManager",               ❸
        attributeNodes = @NamedAttributeNode("manager"))          ❹)
```

- (1) Je drukt de behoefte uit dat JPA bij het lezen van een `Docent` entity uit de database direct ook de geassocieerde `Campus` entity moet lezen.
- (2) Je verwijst naar bijkomende behoefte in verband met die geassocieerde `Campus` entity. Je verwijst naar de behoefte met de naam `metManager`. Die behoefte is bij (3) gedefinieerd.
- (3) Je drukt met `@NamedSubgraph` de behoefte van de geassocieerde `Campus` entity uit. De named subgraph naam moet enkel uniek zijn in de huidige `@NamedEntityGraph`, want je kan een named subgraph enkel in zijn `@NamedEntityGraph` gebruiken.
- (4) Je drukt de behoefte uit dat JPA ook de `Manager` van die `Campus` (via een extra join) direct moet lezen uit de database als hij een `Docent` leest.

### 28.3.6 Code verbetering

Je tikt de naam van de named entity graph (`Docent.metCampus`) in meerdere sources. Je hebt zo het risico op tikfouten die de compiler niet kan controleren en die leiden tot runtime fouten.

Je lost dit op:

- Je voegt een constante toe in `Docent`  

```
public static final String MET_CAMPUS = "Docent.metCampus";
```
- Je gebruikt deze constante: `@NamedEntityGraph(name = Docent.MET_CAMPUS, ...)`
- Je vervangt in `DocentRepository` `"Docent.metCampus"` door `Docent.MET_CAMPUS`


Je commit de sources en je publiceert op GitHub.

## 29 CASCADE

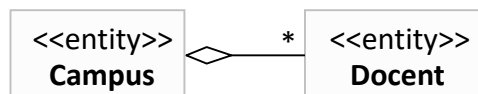
De levensduur van een value object is beperkt tot de levensduur van de bijbehorende entity. JPA doet daartoe enkele handelingen op value objecten van een entity:

- Als je een entity opslaat, slaat JPA de bijbehorende value objecten op.
- Als je een entity verwijdert, verwijdert JPA de bijbehorende value objecten.
- Als je een value object toevoegt aan de verzameling value objecten van een entity, voegt JPA een record toe dat hoort bij dit value object.
- Als je een value object verwijdert uit de verzameling value objecten van een entity, verwijdert JPA het record dat hoort bij dit value object.

Entities kunnen *uitzonderlijk* ook 'behoren' tot andere entities.

Je drukt dit in een class diagram uit met een aggregation (—).

Voorbeeld: een docent behoort tot een campus:



Je drukt een aggregation in Java code uit zoals een gewone associatie:

- de class Campus bevat een `Set<Docent>` variabele docenten
- de class Docent bevat een `Campus` variabele campus

De JPA annotations bij deze variabelen blijven ook dezelfde.

Een entity heeft standaard een zelfstandige levensduur, onafhankelijk van de levensduur van andere entities.

JPA volgt dit principe: als je een campus verwijdert, verwijdert JPA de bijbehorende docenten niet.

Je kan aan `@ManyToOne`, `@OneToMany`, `@ManyToMany` en `@OneToOne` een parameter `cascade` meegeven. JPA doet dan op de entities, waarbij je deze annotations schrijft, enkele handelingen die de levensduur van die entities bepaalt.

Je vult de parameter `cascade` met een waarde uit de enum `CascadeType`.

Je vindt hieronder de meest gebruikte waarden uit de enum `CascadeType`:

- `@OneToMany(cascade = CascadeType.PERSIST)`  
Als je nieuwe entities (van de many kant) toevoegt aan de collection in de entity (aan de one kant), voegt JPA records toe die bij deze nieuwe entities horen.
- `@OneToMany(cascade = CascadeType.REMOVE)`  
Als je een entity verwijdert (van de many kant) uit de collection in de entity (aan de one kant) verwijdert JPA de records die horen bij de verwijderde entity.
- `@OneToMany(cascade = CascadeType.ALL)`  
De combinatie van alle andere waarden.

Fictief voorbeeld:

als je in Campus bij de variabele docenten `@OneToMany` zou wijzigen naar `@OneToMany(cascade = CascadeType.REMOVE)`, geef je aan dat als je een campus verwijdert in de database, JPA ook de bijbehorende docenten moet verwijderen.

## 30 ASSOCIATIES IN JPQL CONDITIES 🧐

Je kan in condities van JPQL queries verwijzen naar de geassocieerde entiteiten van een entity.

### 30.1 Voorbeelden

Controleren of een geassocieerde entity bestaat in een one-to-one of many-to-one associatie.

Voorbeeld: welke campussen hebben geen manager:

```
select c from Campus c where c.manager = null
```

Het aantal geassocieerde entiteiten tellen in een one-to-many associatie met een meervoudige multipliciteit. Voorbeeld: welke campussen hebben meer dan 50 docenten:

```
select c from Campus c where size(c.docenten) > 50
```

Verwijzen naar een attribuut van een geassocieerde entity class.

Voorbeeld: welke campussen behoren bij managers met de familienaam Driesens.

Je verwijst dus vanuit een Campus entity naar een geassocieerde Manager entity en daarin naar het attribuut familienaam.

```
select c from Campus c where c.manager.familienaam = 'Driesens'
```

Een query parameter die zelf een entiteit is.

Voorbeeld: welke docenten hebben een familienaam die lijkt op een tekstpatroon en behoren tot een bepaalde campus. De tekenreeks en de campus zijn parameters.

```
select d from Docent d where d.familienaam like :patroon and d.campus = :campus
```

### 30.2 Praktisch voorbeeld

Je toont in docenten.jsp een hyperlink [Best betaalde](#).

Als de gebruiker deze aanklikt, toon je enkel de best betaalde docenten van de huidige campus.

Je voegt een query toe aan orm.xml:

```
<named-query name='Docent.findBestBetaaldeVanEenCampus'>
  <query>
    select d from Docent d where d.campus = :campus and d.wedde =
      (select max(dd.wedde) from Docent dd where dd.campus = :campus)
  </query>
</named-query>
```

Je voegt een method toe aan DocentRepository:

```
public List<Docent> findBestBetaaldeVanEenCampus(Campus campus) {
    return getEntityManager()
        .createNamedQuery("Docent.findBestBetaaldeVanEenCampus", Docent.class)
        .setParameter("campus", campus)
        .getResultList();
}
```

Je voegt een method toe aan DocentService:

```
private final CampusRepository campusRepository = new CampusRepository();
public List<Docent> findBestBetaaldeVanEenCampus(long id) {
    Optional<Campus> optionalCampus = campusRepository.read(id);
    if (optionalCampus.isPresent()) {
        return docentRepository.findBestBetaaldeVanEenCampus(optionalCampus.get());
    }
    return Collections.emptyList();
}
```

Je voegt een variabele toe aan CampusDocentenServlet:

```
private final transient DocentService docentService = new DocentService();
```

Je voegt code toe na request.setAttribute(...):

```
if (request.getParameter("bestbetaalde") != null) {
    request.setAttribute("docenten",
        docentService.findBestBetaaldeVanEenCampus(Long.parseLong(id)));
}
```

Je voegt code toe in `docenten.jsp`, na `</h2>`:

```
<c:url value='' var='bestBetaaldeUrl'>
  <c:param name='id' value='${campus.id}'/>
  <c:param name='bestbetaalde' value='true'/>
</c:url>
<a href='${bestBetaaldeUrl}'>Best betaalde</a>
```

Je wijzigt de laatste `<c:forEach ...>`:

```
<c:forEach items='${empty param.bestbetaalde ? campus.docenten : docenten}'
var='docent'>
```

Je commit de sources en je publiceert op GitHub. Je kan de website uitproberen.



Artikelgroepen: zie takenbundel



Artikellijst: zie takenbundel

## 31 MULTI-USER EN RECORD LOCKING → → ← ←

Je wijzigt één record met drie handelingen:

1. Je leest het record als een entity in het interne geheugen.
2. Je wijzigt deze entity in het interne geheugen.
3. JPA wijzigt het bijbehorende record bij een commit van de transactie.

Je hebt in een multi-user situatie het gevaar dat, tussen het lezen van het record en het wijzigen van het record, een andere gebruiker hetzelfde record wijzigde.

JPA overschrijft bij de commit de wijzigingen van die andere gebruiker !

Je kan dit op volgende manier zien:

1. Je plaatst een breakpoint in DocentService bij `commit()`;
2. Je voert het programma uit in debug mode en je geef docent 1 10% opslag.
3. Je komt bij het breakpoint.
4. Je tikt in de MySQL Workbench volgende opdracht en je voert hem daarna uit  
`update fietsacademy.docenten set familienaam = 'aangepast' where id = 1;`
5. Je laat het programma verder lopen.
6. Als je in de MySQL Workbench docent 1 opvraagt,  
zie je dat je applicatie de wijziging die je deed in de familienaam overschreef.

Je voorkomt dit probleem via pessimistic record locking of optimistic record locking.

Je leert beide methodes en daarna de voordelen en nadelen van beide methodes.

### 31.1 Pessimistic record locking

Je vergrendelt hierbij het record wanneer je het leest in de database.

Vanaf dan kan niemand anders het record wijzigen, verwijderen of vergrendelen.

JPA ontgrendelt het record op het einde van de transactie.

Je kan met de EntityManager method `find` een record vergrendelen bij het lezen met een derde parameter, die je invult met:

- `PESSIMISTIC_READ` (dit wordt in SQL `select ... lock in share mode`)  
Andere gebruikers kunnen het record lezen, maar niet wijzigen.
- of `PESSIMISTIC_WRITE` (dit wordt in SQL `select ... for update`)  
Andere gebruikers kunnen het record niet lezen met `PESSIMISTIC_READ` of `PESSIMISTIC_WRITE` en kunnen het record niet wijzigen.

Als het record dat jij wil vergrendelen, al vergrendeld is door een andere gebruiker, en die vergrendeling te lang duurt, werpt de `find` method een `PessimisticLockException`.

Je voegt code toe aan DocentRepository:

```
public Optional<Docent> readWithLock(long id) {
    return Optional.ofNullable(
        getEntityManager().find(Docent.class, id, LockModeType.PESSIMISTIC_WRITE));
}
```

Je vervangt in de DocentService class in de method `opslag`:

```
docentRepository.read(id)... door
docentRepository.readWithLock(id)...
```

Terwijl jij de opslag geeft, kan geen andere gebruiker hetzelfde record wijzigen

1. je plaatst een breakpoint (met een dubbele klik in de marge) in DocentService bij `commit()`;
2. Je voert het programma uit in debug mode en je geeft docent 1 10% opslag.
3. Je komt op het breakpoint.
4. Je tikt in de MySQL Workbench volgende opdracht en je voert hem daarna uit  
`update fietsacademy.docenten set familienaam = 'aangepast' where id = 1;`
5. Je ziet na enkele seconden de foutboodschap `Lock wait timeout exceeded`.
6. Je laat het programma verder lopen, tot de opslag is doorgevoerd.

7. Pas nu kan je met de MySQL Workbench hetzelfde record aanpassen door het **update** statement nog eens uit te voeren.



Opmerking 1: je kan een entity lezen zonder te vergrendelen

```
Docent docent = entityManager.find(Docent.class, 1L);
```

en deze entity later toch vergrendelen met de method lock van EntityManager  

```
entityManager.lock(docent, LockModeType.PESSIMISTIC_WRITE);
```

JPA werpt een PessimisticLockException als JPA het record al vergrendeld is.

Opmerking 2: Je vergrendelt de entities, die je leest met een JPQL query,

met de TypedQuery method setLockMode(LockModeType.PESSIMISTIC\_WRITE)



Bij pessimistic record locking    vergrendel je    een entity    tot het einde van de transactie

## 31.2 Optimistic record locking

De naam optimistic record locking is misleidend: je vergrendelt het te wijzigen record op geen enkel moment! Je doet wel volgende stappen

1. Je leest het te wijzigen record als een entity in het interne geheugen.
2. Je wijzigt deze entity in het interne geheugen.
3. JPA controleert bij de commit van de transactie of een andere gebruiker dit record wijzigde sedert het moment dat jij dit record gelezen hebt.

Als dit zo is, wijzigt JPA het record niet en werpt een OptimisticLockException.

JPA kan op twee manieren controleren of een andere gebruiker het record wijzigde:

- met een versie kolom met als type een geheel getal.
- met een versie kolom met als type timestamp.

Je ziet hier onder deze manieren, met hun voordelen en nadelen.

### 31.2.1 Versie kolom met een geheel getal

Hierbij bevat de table een gehele getal kolom die JPA als volgt gebruikt:

- Wanneer JPA het record leest, onthoudt JPA het getal in die kolom (bvb. 5).
- Elke applicatie die het record wijzigt, verhoogt het getal in die kolom.
- Wanneer JPA het record wijzigt, controleert JPA of het record ondertussen door een andere gebruiker werd gewijzigd:  

```
update docenten set voornaam=?, familienaam=?, wedde=?, geslacht=?,  
versie = versie + 1  
where id = ? and versie = ?
```

JPA vervangt het ? bij versie door het getal dat het record bevatte bij het lezen van het record (in ons voorbeeld 5).
- Als een andere gebruiker ondertussen hetzelfde record wijzigde, bevat de kolom Versie de waarde 6. Het update statement vindt geen record dat voldoet aan zijn where clause en wijzigt geen record.  

JPA weet zo dat een andere gebruiker het record gewijzigd heeft en werpt een OptimisticLockException.
- Als geen andere gebruiker ondertussen het record wijzigde, bevat de kolom Versie nog de oorspronkelijk gelezen waarde (5).  

Het update statement vindt een record dat voldoet aan zijn where clause en wijzigt dit record.  

JPA verhoogt hierbij het getal in de kolom Versie met 1.

Nadeel van een versie kolom met als type een geheel getal:

- ⊖ Alle applicaties (ook niet-JPA applicaties) moeten bij een recordwijziging het getal in de versie kolom verhogen.

Je voert het script `VersieAlsIntToevoegenAanDocenten.sql` uit.

Dit voegt aan de table `docenten` een `int` kolom `versie` toe.



Je doet volgende stappen, zodat Eclipse een correct beeld krijgt van de nieuwe tables:

1. Je klapt in de Data Source Explorer niveau's van je database open tot je de map Tables ziet.
2. Je klikt met de rechtermuisknop op Tables en je kiest Refresh.
3. Je kiest in het menu Project de opdracht Clean.

Je voegt een private variabele toe aan `Docent`:

```
@Version
```

```
private long versie;
```

❶

- (1) Je tikt `@Version` voor de private variabele die hoort bij de kolom die JPA kan gebruiken voor de versie controle.

Je maakt een class `RecordAangepastException`:

```
package be.vdab.exceptions;
```

```
public class RecordAangepastException extends RuntimeException {
    private static final long serialVersionUID = 1L;
}
```

Je wijzigt in `DocentService` de method `opslag`.

Pas hierbij op: je roept terug de `read` method op in plaats van de method `readWithLock`

```
public void opslag(long id, BigDecimal percentage) {
    beginTransaction();
    try {
        docentRepository.read(id).ifPresent(docent -> docent.opslag(percentage));❶
        commit();
    } catch (RollbackException ex) {
        if (ex.getCause() instanceof OptimisticLockException) {
            throw new RecordAangepastException();
        }
    } catch (PersistenceException ex) {
        rollback();
        throw ex;
    }
}
```

❷

❸

- (1) Je gebruikt bij optimistic record locking de method `find` van `EntityManager` zonder de derde parameter.
- (2) Als een andere gebruiker het record ondertussen wijzigde, werpt JPA een `RollbackException` bij een `commit` van de transactie.
- (3) Als de method `getCause` van deze `RollbackException` een `OptimisticLockException` teruggeeft, is de oorzaak van de exception dat een andere gebruiker het record wijzigde. Als de method `getCause` een ander type object teruggeeft, is de rollback veroorzaakt door een andere fout in de database.

Je vervangt in `OpslagServlet` de `if ... else ...` structuur:

```
if (fouten.isEmpty()) {
    long id = Long.parseLong(request.getParameter("id"));
    try {
        docentService.opslag(id, percentage);
        response.sendRedirect(response.encodeRedirectURL(
            String.format(REDIRECT_URL, request.getContextPath(), id)));
    }
    catch (RecordAangepastException ex) {
        fouten.put("percentage", "een andere gebruiker heeft deze docent gewijzigd");
    }
}
```



```

if ( ! fouten.isEmpty()) {
    request.setAttribute("fouten", fouten);
    request.getRequestDispatcher(VIEW).forward(request, response);
}

```

Je kan dit foutbericht op volgende manier zien

1. Je plaatst een breakpoint in DocentService op commit();
2. Je voert de programma uit in debug mode en je geeft docent 1 10% opslag.
3. Je komt op het breakpoint.
4. Je tikt in de MySQL WorkBench volgende opdracht en voert hem daarna uit:
 

```

update fietsacademy.docenten
set familienaam = 'aangepast', versie = versie + 1 where id = 1;

```
5. Je laat het programma verder lopen.

### 31.2.2 Versie kolom met een timestamp

Hierbij bevat de table een kolom van het type timestamp. Bij sommige databases heeft zo'n kolom precisie tot op de seconde, bij andere databases tot op de nanoseconde.

- Wanneer JPA het record leest, onthoudt JPA de datum-tijd in die kolom.
- Elke applicatie die het record wijzigt, vult in die kolom de systeemdatum en -tijd in. Een nog betere oplossing is dat de database zelf bij elke wijziging van het record de systeemdatum en -tijd invult in deze kolom. Als meerdere applicaties de records wijzigen, moet dan niet elk van de applicaties deze verantwoordelijkheid op zich nemen.
- Wanneer JPA het record wijzigt, controleert JPA of het record ondertussen door een andere gebruiker werd gewijzigd:
 

```

update docenten set voornaam=?, familienaam=?, wedde=?, geslacht=?
where id = ? and versie = ?

```

 JPA vervangt het ? bij Versie door de datum-tijd dat het record bevatte bij het lezen van het record. Als een andere gebruiker ondertussen hetzelfde record wijzigde, bevat de kolom Versie een andere waarde. Het update statement vindt geen record dat voldoet aan zijn where clause en wijzigt dus geen record. JPA weet zo dat een andere gebruiker het record gewijzigd heeft en werpt een OptimisticLockException. Als geen andere gebruiker het record wijzigde, bevat de kolom Versie de oorspronkelijk gelezen waarde. Het update statement vindt een record dat voldoet aan zijn where clause en wijzigt dit record. Als de database niet zelf bij elke recordwijziging de systeemdatum en -tijd invult in de kolom Versie, is het update statement dat JPA naar de database stuurt uitgebreider. JPA vult dan zelf de kolom Versie in met de systemdatum en -tijd.

Algemeen nadeel van een versiekolom van het type timestamp:

- ⊖ Een timestamp kolom heeft bij sommige databasemerken slechts een precisie tot op de seconde. Als een record meerdere keren per seconde wijzigt, kan je geen wijzigingen door andere gebruikers detecteren.

Nadeel van een versiekolom van het type timestamp,

als de applicaties zelf de kolom moeten bijwerken bij een recordwijziging:

- ⊖ Alle applicaties (ook niet-JPA applicaties) moeten bij een recordwijziging de systeemtijd invullen in de timestamp kolom.
- ⊖ Als de applicaties draaien op verschillende servers, moet de systeemtijd van die servers exact gelijk lopen, of je krijgt problemen.

Voordeel van een versiekolom van het type timestamp,

als de database zelf de kolom moeten bijwerken bij een recordwijziging:

- ⊕ De applicaties moeten het wijzigen van deze kolom niet op zich nemen.

Je voert het script VersieAlsTimestampToevoegenAanDocenten.sql uit.

Dit vervangt de kolom Versie

door een nieuwe kolom Versie van het type Timestamp

die de database zelf bij elke recordwijziging invult met de systeemtijd.

📌 versie TIMESTAMP



Je doet volgende stappen, zodat Eclipse een correct beeld krijgt van de nieuwe tables:

1. Je klappt in de Data Source Explorer niveau's van je database open tot je de map Tables ziet.
2. Je klikt met de rechtermuisknop op Tables en je kiest Refresh.
3. Je kiest in het menu Project de opdracht Clean.

Je wijzigt in Docent het type van de private variabele versie naar Timestamp (uit de package java.sql). Timestamp is afgeleid van Date.

- Date houdt de tijd bij tot op de seconde.
- Timestamp houdt de tijd bij tot op de nanoseconde.

**private** Timestamp **versie**;

Je commit de sources en je publiceert op GitHub.

Je kan de use case opslag terug uitvoeren.

Je kan het foutbericht op volgende manier zien:

1. Je plaatst een breakpoint in DocentService op commit();.
2. Je voert het programma uit in debug mode en je geeft docent 1 10% opslag.
3. Je komt op het breakpoint.
4. Je tikt in de MySQL Workbench volgende opdracht en voert hem daarna uit  
**update** fietsacademy.docenten **set** familienaam = 'aangepast' **where** id = 1;
5. Je laat het programma verder lopen.



### 31.3 Pessimistic record locking en optimistic record locking

- Pessimistic locking vraagt meer inspanning van de database dan optimistic locking: de database moet records vergrendelen en ontgrendelen. Dit benadeelt de performantie.
- Je kan bij de meeste databases geen pessimistic record locking doen op records die je leest in een subquery. Optimistic record locking lukt wel.
- Voor de beste optimistic record locking technieken (versie kolom, timestamp) moet je een kolom toevoegen aan de table. Bij pessimistic record locking hoeft dit niet.
- Als je bij optimistic record locking een versie kolom gebruikt die JPA wijzigt, werkt dit enkel als ook alle andere applicaties deze kolom aanpassen. Als je een record vergrendelt met pessimistic record locking, is dit record zowiezo vergrendeld voor alle applicaties.
- Bij pessimistic record locking weet je vroeg in een use case dat de recordwijziging niet zal lukken. Bij optimistic record locking weet je dit maar laat in de use case. Misschien moet je op dat moment andere handelingen van die use case ongedaan maken...

Als je bij sommige use cases toch twijfelt tussen pessimistic record locking en optimistic record locking, gebruik je volgende vuistregel.

- Je gebruikt pessimistic locking als de kans hoog is dat meerdere gebruikers op hetzelfde moment hetzelfde record wijzigen. Anders gebruik je optimistic record locking.

## 32 STAPPENPLAN

Je kan volgend stappenplan volgen als je een web applicatie maakt met JPA:

1. Converteer het project naar een JPA project.
2. Maak alle classes (Entities en Value objects) die personen en/of dingen uit de werkelijkheid voorstellen. Als je enkel Entities vindt en geen Value objects, heb je de werkelijkheid waarschijnlijk niet goed bestudeerd.

- a. Maak de Value object classes immutable (geen setters)
- b. Maak ook een Entity immutable als je die Entity enkel leest uit de database, niet toevoegt, wijzigt of verwijdert.

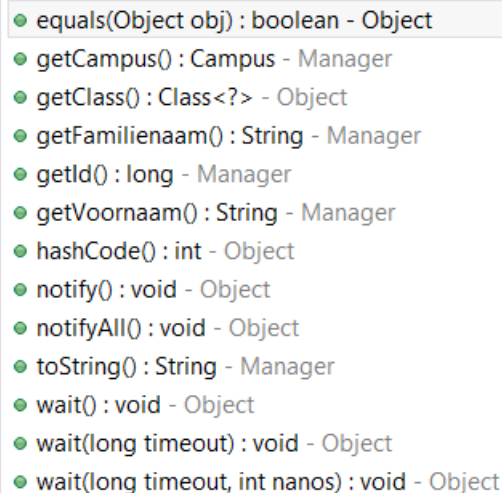
Zo'n Entity heeft voor zijn associaties naar andere Entities of Value objects enkel getters, geen setters, add methods, remove methods.

Dit heeft voordelen:

- i. Je bent productief, want je schrijft minder code.
- ii. De Entity class is kort en daarom leesbaar en met minder bugs.
- iii. Als je in een andere laag kijkt welke methods je op een Entity kan toepassen, is het lijstje kort en dus overzichtelijk.

```
Manager manager;
```

```
manager.
```





- equals(Object obj) : boolean - Object
- getCampus() : Campus - Manager
- getClass() : Class<?> - Object
- getFamilienaam() : String - Manager
- getId() : long - Manager
- getVoornaam() : String - Manager
- hashCode() : int - Object
- notify() : void - Object
- notifyAll() : void - Object
- toString() : String - Manager
- wait() : void - Object
- wait(long timeout) : void - Object
- wait(long timeout, int nanos) : void - Object

Press 'Ctrl+Space' to show Template Proposals

- c. Maak geen setters voor 'alleen-lezen' eigenschappen (bvb. autonumber kolom)
- d. Maak in elk van die classes de methods equals en hashCode.  
Als je in de applicatie een Set vult met objecten van die classes, zal die Set dan correct werken. Baseer de methods equals en hashCode niet op de private variabele die bij de primary key hoort, maar op een andere private variabele (of combinatie van private variabelen). Deze variabele (of combinatie van variabelen) moet een unieke waarde bevatten per object van de class).
- e. Beschrijf in Entities én in Value objects een relatie naar een Entity niet als een getal, maar als een reference variabele.

Deze reference variabele heeft als type die andere Entity.

 verkeerd	 correct
<pre>... public class Docent { ... private long campusNr; ... }</pre>	<pre>... public class Docent { ... @ManyToOne(optional = false) @JoinColumn(name = "campusid") private Campus campus; }</pre>

- f. Plaats een berekening die met een Entity of Value object te maken heeft in de class van die Entity of Value object, niet in Servlets, services of repositories  
 Eerste voordeel: een berekening in een Entity of Value object kan je vanuit alle andere onderdelen van je applicatie oproepen.  
 Tweede voordeel: als de berekening wijzigt, doe je die wijziging één keer (en niet meerdere keren als die berekening in meerdere JSP's gebeurt).  
 Voorbeeld: de berekening van een artikelprijs inclusief BTW op basis van de prijs exclusief BTW en de het % BTW:

```
...
public class Artikel {
    ...
    private BigDecimal prijsExclusief;
    private BigDecimal btwPercentage;
    public BigDecimal prijsInclusief() {
        return prijsExclusief.multiply(BigDecimal.ONE
            .add(btwPercentage.divide(BigDecimal.valueOf(100), 2,
                RoundingMode.HALF_UP)));
    }
}
```

- g. Voeg de Entity of Value object class toe aan persistence.xml
3. Voeg de JPAFilter met de EntityManager als ThreadLocal variabele toe.
  4. Doe volgende stappen per nieuwe pagina van de web applicatie
    - a. Voeg de repository classes en methods toe die je voor die pagina nodig hebt.
    - b. Voeg de Service classes en methods toe die je voor die pagina nodig hebt.  
 Als een method records toevoegt, wijzigt of verwijdert doe je dit in een transactie.  
 Als een method een record leest én daarna ook wijzigt,  
 pas je optimistic of pessimistic record locking toe.  
 Om een record te wijzigen volstaat het dit record te lezen en te wijzigen in het interne geheugen. JPA doet dezelfde wijziging in de database voor jou.
    - c. Voeg een Servlet toe  
 Als je een HttpSession variabele nodig hebt, onthoud je daarin geen Entities, maar identifiers van die Entities.
    - d. Voeg een JSP toe.
    - e. Test de pagina. Als die een N + 1 performantieprobleem heeft, los je dit probleem op met Named Entity Graphs

## 33 HERHALINGSOEFENINGEN



Muziek: zie takenbundel

## **34 COLOFON**

<b>Domeinexpertisemanager:</b>	Jean Smits
<b>Moduleverantwoordelijke:</b>	Hans Desmet
<b>Medewerkers:</b>	Hans Desmet
<b>Versie:</b>	16/8/2017
<b>Nummer dotatielijst:</b>	