














samen sterk voor werk



JPA met Hibernate























Inhoudsopgave


1	INLEIDING 	7
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
1.7	Applicatie lagen	8
2	PROJECT 	9
2.1	DataSource	9
3	ENTITY 	11
3.1	De database table en de bijbehorende entity class	11
3.2	Default constructor	12
3.3	Naming strategy	12
3.4	Extra JPA annotations.....	12
3.4.1	@Column	12
3.4.2	@Transient	12
4	ENTITYMANAGER 	13
4.1	Samenvatting.....	13
5	ENTITY ZOEKEN VIA DE PRIMARY KEY 	14
6	ENUM 	16
6.1	Enum	16
6.2	Voorstelling als int kolom.....	16
6.3	Voorstelling als varchar kolom of enum kolom	17

7	JPA PROJECT	18
7.1	Configuratie.....	18
7.2	Controle.....	19
7.3	Project Explorer	19
7.4	Data Source Explorer.....	19
7.5	persistence.xml	19
8	DALI.....	20
8.1	Installatie	20
8.2	Grafische voorstelling.....	20
8.3	Code completion	20
9	ENTITY TOEVOEGEN 	21
9.1	@GeneratedValue.....	21
9.2	Sequence	21
9.3	Voorbeeld.....	21
10	ENTITY VERWIJDEREN 	23
11	ENTITY WIJZIGEN 	24
12	LEVENSCYCLI (LIFE CYCLE) VAN EEN ENTITY 	28
13	JPQL 	29
13.1	Algemeen	29
13.2	Alle entities lezen	29
13.3	Sorteren.....	30
13.4	Selecteren.....	30
13.4.1	Named parameters.....	31
13.5	Één kolom lezen	31
13.6	Meerdere kolommen lezen	32
13.7	Aggregate functions	33
13.8	Group by.....	33

13.9	Named queries	34
13.9.1	Named queries in entity classes	34
13.9.2	Named query oproepen.....	35
13.9.3	Orm.xml	35
14	BULK UPDATES EN BULK DELETES.....	36
15	INHERITANCE 	37
15.1	Inheritance nabootsen in de database.....	37
15.2	Table per class hierarchy	37
15.2.1	Database	37
15.2.2	Voordelen	37
15.2.3	Nadelen.....	37
15.2.4	Base class	37
15.2.5	Derived classes	38
15.2.6	Dali	38
15.2.7	Repository layer	38
15.3	Polymorphic queries.....	40
15.4	Table per subclass	41
15.4.1	Voordelen	41
15.4.2	Nadelen.....	41
15.4.3	Base class annotations	41
15.4.4	Derived classes annotations : GroepsCursus	41
15.4.5	Derived classes annotations : IndividueleCursus	41
15.4.6	Testen	42
15.5	Table per concrete class	43
15.5.1	Voordelen	43
15.5.2	Nadelen.....	43
15.5.3	Annotations	43
16	VALUE OBJECTS 	46
16.1	Immutable	46
16.2	Database.....	48
16.3	Java	48
16.3.1	Value object: Adres.....	48
16.3.2	Entity die het value object gebruikt: Campus	48
16.4	Service layer en repository layer	49
16.5	CRUD operaties	49
16.5.1	Create	49
16.5.2	Read	49
16.5.3	Update	49
16.5.4	Delete	49

16.6	JPQL	49
16.7	Value object classes detecteren	51
16.8	Aggregate	51
16.9	Value object als command object	52
17	VERZAMELING VALUE OBJECTS MET EEN BASISTYPE 	53
17.1	Verzameling value objects lezen uit de database	55
17.2	Value object toevoegen aan de verzameling	55
17.3	Value object verwijderen uit de verzameling	56
18	VERZAMELING VALUE OBJECTS MET EEN EIGEN TYPE 	57
18.1	Value object: TelefoonNr	57
18.2	Entity met verzameling value objects: Campus	58
18.3	Verzameling value objects lezen uit de database	59
18.4	Value object toevoegen aan de verzameling	59
18.5	Value object verwijderen uit de verzameling	59
18.6	Entity verwijderen	59
18.7	Value object in de verzameling wijzigen	60
19	MANY-TO-ONE ASSOCIATIE  → 	61
19.1	Database	61
19.2	Java	61
19.3	Eager loading	63
19.4	Lazy loading	63
20	ONE-TO-MANY ASSOCIATIE  → 	65
20.1	Database	65
20.2	Java	65
20.3	Testen corrigeren	66
20.4	Set en de methods equals en hashCode	66
20.5	Equals en hashCode laten genereren	68

20.6	One-to-many is lazy loading.....	69
21	BIDIRECTIONELE ASSOCIATIE  - 	70
21.1	Docent (many kant).....	70
21.2	One kant: Campus	70
21.3	Testen corrigeren	70
21.4	Associatievariabelen bijwerken.....	71
22	MANY-TO-MANY ASSOCIATIE  - 	73
22.1	Database.....	73
22.2	Java.....	73
22.3	Bidirectionele @ManyToMany	76
22.4	Repository testen	76
23	ASSOCIATIES VAN VALUE OBJECTS NAAR ENTITIES  → 	78
24	N + 1 PROBLEEM, JOIN FETCH QUERIES, ENTITY GRAPHS 	79
24.1	N + 1 probleem.....	79
24.2	Join fetch	79
24.3	Entity graph	80
24.3.1	@NamedEntityGraph.....	80
24.3.2	De oproep van de named query	80
24.3.3	@NamedEntityGraphs	81
24.3.4	Meer dan één geassocieerde entity	81
24.3.5	De behoefte naar een geassocieerde entity van een geassocieerde entity.....	81
24.3.6	Refactoring	81
25	CASCADE 	82
26	ASSOCIATIES IN JPQL CONDITIES 	83
26.1	Voorbeelden.....	83
27	MULTI-USER EN RECORD LOCKING  →  →  ←  ← 	84
27.1	Pessimistic record locking	84
27.2	Optimistic record locking	85
27.2.1	Versie kolom met een geheel getal	85

27.2.2	Versie kolom met een timestamp.....	86
27.3	Pessimistic record locking en optimistic record locking.....	87
28	STAPPENPLAN 	88
29	HERHALINGSOEFENINGEN	90
30	COLOFON	91

1 INLEIDING 🌀

1.1 Doelstelling

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

JPA is een Java standaard. Je spreekt met JPA een relationele database aan.

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

1.2 Vereiste voorkennis

- Java
- SQL
- JDBC
- Spring fundamentals

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) en de MySQL Workbench.
- Tomcat.
- Eclipse IDE for Java EE Developers (versie Photon).

1.4 ORM

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

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

JPA gebruikt intern JDBC om de database aan te spreken.

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.
2. Je logt in op de Local instance.
3. Je kiest in het menu File de opdracht Open SQL Script.
4. Je opent `FietsAcademy.sql`.
5. 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.

1.7 Applicatie lagen

Je leerde in de cursus “Spring fundamentals” je applicatie opdelen in verschillende lagen. Elke laag heeft een verantwoordelijkheid:

- **Entities**
Stellen dingen of personen uit de werkelijkheid voor.
Deze laag blijft dezelfde als je JPA gebruikt.
Je leert hier dat je wel JPA annotations tikt bij de entities.
- **Repositories**
Spreken de database aan. De interfaces uit deze laag zijn gelijkaardig als je JPA gebruikt.
De code in de implementatieclasses wijzigt:
vroeger gebruikte je JDBC om de database aan te spreken, je zal dit nu doen met JPA.
- **Services**
Beheren transacties en roepen methods op van entities en repositories om een use case uit te werken. Deze laag blijft dezelfde als je JPA gebruikt.
- **Controllers**
Verwerken browser requests.
Roepen de Services om de requests te verwerken.
Roepen JSP's op om HTML naar de browser te sturen.
Deze laag blijft dezelfde als je JPA gebruikt.
- **JSP's**
Sturen HTML naar de browser. Deze laag blijft dezelfde als je JPA gebruikt.

Gezien de meeste lagen dezelfde blijven, zal je in deze cursus niet elke laag uitwerken.

Je zal altijd de Repositories laag uitwerken.

Je zal met integration tests aantonen dat deze Repositories laag correct werkt.

Deze tests zijn zo geschreven dat ze ook werken als de database geen records bevat.

2 PROJECT

Je maakt het project met de hulp van de website start.spring.io:

1. Je opent in je browser de website <http://start.spring.io>.
2. Je tikt `be.vdab` bij Group.
3. Je tikt `fietsacademy` bij Artifact.
4. Je tikt dependencies bij Search for dependencies, gescheiden door Enter: MySQL, JPA. Spring voegt onder andere de dependencies toe voor Hibernate, de populairste JPA implementatie.
5. Je kiest Generate Project.



Als je een website maakt, volg je de stappen in de cursus “Spring fundamentals”. Je kiest dan niet de JDBC dependency, maar de JPA dependency.

Je importeert het project in Eclipse:

1. Je kopieert de map uit het zip bestand naar je workspace map van Eclipse.
2. Je start Eclipse.
3. Je kiest in het menu File de opdracht Import.
4. Je kiest Maven, Existing Maven Projects.
5. Je kiest Next.
6. Je kiest Browse.
7. Je kiest de map `fietsacademy` en je kiest OK.
8. Je kiest Finish.



Als je een website maakt, volg je de stappen in de cursus “Spring fundamentals” om de JSP dependencies en de jsoup dependency aan `pom.xml` toe te voegen.

2.1 DataSource

Je definieert de databaseverbinding in `src/main/resources/application.properties`:

```
spring.datasource.url=jdbc:mysql://localhost/fietsacademy?useSSL=false
spring.datasource.username=cursist
spring.datasource.password=cursist
```

❶

- (1) JPA gebruikt intern JDBC om de database aan te spreken.
JPA gebruikt daarom de JDBC URL die je ook al leerde kennen in de JDBC cursus.

Spring maakt op basis van deze instellingen een DataSource.

Je maakt een test voor deze DataSource.

Je maakt in `src/test/java` een package `be.vdab.fietsacademy.repositories`.

Je maakt daarin de class `DataSourceTest`:

```
package be.vdab.fietsacademy.repositories;
// enkele imports
@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace = Replace.NONE)
public class DataSourceTest {
    @Test
    public void goedGeïnitieerd() {
    }
}
```

❶

❷

- (1) `@DataJpaTest` is vergelijkbaar met `@JdbcTest`.
`@DataJpaTest` maakt enkel een DataSource bean en een EntityManager bean.
Zo wordt de test snel uitgevoerd. Je leert EntityManager verder in de cursus kennen.

- (2) `@DataJpaTest` voert de tests standaard niet uit met de MySQL database beschreven in `application.properties`.
`@DataJpaTest` voert de tests standaard uit met een in-memory database.
Een in-memory database houdt de data bij in het interne geheugen, niet op de harde schijf.
Dit houdt in dat als je programma crasht of stopt je alle data verliest.
Dit is ontoelaatbaar in productie, gedurende een test kan het wel.
Het gebruik van een in-memory database versnelt je test.
Je zal in de cursus “Spring advanded” leren testen met een in-memory database.
Je test hier nog met je MySQL database.
Je vraagt hier aan Spring in je test de MySQL database niet te vervangen door een in-memory database.

Je voert deze test uit.



Je commit de sources. Je publiceert op GitHub.



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)
emailAdres	VARCHAR(60)

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

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

Je maakt een package `be.vdab.fietsacademy.entities` (entity classes moeten zich bevinden in een subpackage van de originele package van je project: `be.vdab.fietsacademy`)

Je maakt daarin de entity class `Docent`:

```
package be.vdab.fietsacademy.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 String emailAdres;
    // getters, behalve voor serialVersionUID
}
```

1
2
3
4
5

JPA stelt een aantal voorwaarden aan een entity class:

- (1) `@Entity` staat voor de entity class.
- (2) `@Table` duidt de table in de database aan die bij de entity hoort..
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) `@Id` staat 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`.



Als je een website maakt, tik je ook `@NumberFormat(...)` voor de variabele `wedde`.
Je voegt ook validation annotations toe als de gebruiker deze entity intikt.

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. De compiler maakt bij Docent een default constructor, omdat je zelf geen constructors schreef. Je maakt die constructor **protected** in plaats van **public** als je liever hebt dat zo weinig mogelijk classes deze constructor kunnen gebruiken.

3.3 Naming strategy

Standaard hoort bij JPA bij de private variabele emailAdres een kolom email_adres in de database (hoofdletters worden vervangen door kleine letters met voorafgaande _).

Met volgende regels in application.properties hoort bij private variabele emailAdres een kolom emailAdres in de database.

```
spring.jpa.hibernate.naming.physical-strategy=\norg.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
```

3.4 Extra JPA annotations

Je kan ook volgende annotations nodig hebben.

3.4.1 @Column

@Column staat voor een private variabele als de bijbehorende kolom naam 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

@Transient staat voor een private variabele die geen bijbehorende kolom heeft in de database.



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.

Spring Boot maakt een EntityManager als een bean die je kan injecteren in je repositories laag. Spring maakt deze bean zodra je de JPA dependency toevoegt aan je project.

Tijdens het initialiseren van de EntityManager worden ook je entity classes gecontroleerd

- Hebben ze een default constructor ?
- Duiden ze met @Id de variabele aan die bij de primary key hoort ?
- ...

Als aan deze regels niet voldaan wordt, werpt de initialisatie van de EntityManager een exceptie.

Gezien Spring tijdens DataSourceTest ook al probeert een EntityManager te maken, mislukt DataSourceTest als Spring de EntityManager niet kan maken.

Je plaatst in Docent de regel @Id even in commentaar.

Je voert de test uit. Hij mislukt. Als je scrollt in de Failure Trace in het tabblad JUnit zie je de tekst `No identity specified for entity: be.fietsacademy.entities.Docent`

Je haalt in Docent de regel @Id terug uit commentaar.

Je voert de test uit. Hij lukt.

4.1 Samenvatting



De EntityManager

beheert



Entities.

5 ENTITY ZOEKEN VIA DE PRIMARY KEY



De EntityManager

bevat een



find method.

Die zoekt



een entity



op primary key



in de database.

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 per entity een bijbehorende repository class.

Je maakt dus een package `be.vdab.fietsacademy.repositories`.

Je maakt daarin een interface `DocentRepository`:

```
package be.vdab.fietsacademy.repositories;
public interface DocentRepository {
    Optional<Docent> read(long id);
}
```

Je implementeert deze method in `JpaDocentRepository`:

```
package be.vdab.fietsacademy.repositories;
// enkele imports
@Repository
class JpaDocentRepository implements DocentRepository {
    @Override
    public Optional<Docent> read(long id) {
        throw new UnsupportedOperationException();
    }
}
```

❶

- (1) Je maakt (zie cursus JUnit) eerst een minimale implementatie, zodat je een test kan schrijven vooraleer je de echte implementatie maakt.

Je maakt de folder voor bestanden die bij testen horen maar geen Java sources zijn:

1. Je klikt met de rechtermuisknop op je project in de Project Explorer.
2. Je kiest New, Source Folder.
3. Je tikt `src/test/resources`.

Je maakt in deze folder een SQL bestand aan dat je zal gebruiken in een test:

1. Je klikt met de rechtermuisknop op de folder.
2. Je kiest New, Other, SQL Development, SQL File en je kiest Next.
3. Je tikt `insertDocent` en je kiest Finish.
4. Je tikt volgende SQL opdracht in dit bestand:


```
insert into docenten(voornaam, familienaam, wedde, emailadres)
values('testM', 'testM', 1000, 'testM@fietsacademy.be');
```
5. Je slaat het bestand op.

Je maakt een test voor `JpaDocentRepository`:

```
package be.vdab.fietsacademy.repositories;
// enkele imports
@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace = Replace.NONE)
@Sql("/insertDocent.sql")
@Import(JpaDocentRepository.class)
```

❶

```

public class JpaDocentRepositoryTest
    extends AbstractTransactionalJUnit4SpringContextTests {
    @Autowired
    private JpaDocentRepository repository;
    private long idVanTestMan() {
        return super.jdbcTemplate.queryForObject(
            "select id from docenten where voornaam = 'testM'", Long.class);
    }
    @Test
    public void read() {
        Docent docent = repository.read(idVanTestMan()).get();
        assertEquals("testM", docent.getVoornaam());
    }
    @Test
    public void readOnbestaandeDocent() {
        assertFalse(repository.read(-1).isPresent());
    }
}

```

- (1) @DataJpaTest maakt enkel een DataSource bean en een EntityManager bean.
Je maakt hier een bean van de te testen class: JpaDocentRepository.

Je voert de test uit. Hij mislukt.

Je implementeert de method in JpaDocentRepository:

```

package be.vdab.fietsacademy.repositories;
// enkele imports
@Repository
class JpaDocentRepository implements DocentRepository {
    private final EntityManager manager;
    JpaDocentRepository(EntityManager manager) {
        this.manager = manager;
    }
    @Override
    public Optional<Docent> read(long id) {
        return Optional.ofNullable(manager.find(Docent.class, id));
    }
}

```

- (1) 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.
De find method geeft de Docent entity terug als hij die vindt in de database.
De find method geeft null terug als hij de Docent entity niet vindt in de database.
Je leerde in de Lambda cursus dat als een method mogelijks de te zoeken waarde niet vindt
deze method best een Optional teruggeeft. JPA werd echter gemaakt voor Optional
werd toegevoegd aan Java. Je maakt je eigen method (de huidige method read) wel
"modern" door een Optional terug te geven aan de code die de read method oproept.

Je voert de test uit. Hij lukt.

JPA stuurt SQL statements naar de database.

Je ziet deze statements, tijdens het uitvoeren van de test, in het venster Console.

De waarden die JPA invult in de ? tekens in de SQL statements zie je echter niet.

Je voegt regels toe aan application.properties om dit te zien:

logging.level.org.hibernate.SQL=DEBUG

logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE



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 vertaalt in de database naar:

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

6.1 Enum

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

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

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

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

6.2 Voorstelling als 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 `IntGeslacht.sql` uit.

Dit voegt aan de table `docenten` een int kolom `geslacht` toe



en vult die met 0 (bij mannen), of 1 (bij vrouwen).

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 wijzigt `insertDocent.sql`:

```
insert into docenten(voornaam,familienaam,wedde,emailadres,geslacht)
values ('testM','testM',1000,'testM@fietsacademy.be',0);
insert into docenten(voornaam,familienaam,wedde,emailadres,geslacht)
values ('testV','testV',1000,'testV@fietsacademy.be',1);
```

Je slaat het bestand op.

Je maakt methods in `JpaDocentRepositoryTest`:

```
private long idVanTestVrouw() {
    return super.jdbcTemplate.queryForObject(
        "select id from docenten where voornaam='testV'", Long.class);
}

@Test
public void man() {
    assertEquals(Geslacht.MAN,
        repository.read(idVanTestMan()).get().getGeslacht());
}

@Test
public void vrouw() {
    assertEquals(Geslacht.VROUW,
        repository.read(idVanTestVrouw()).get().getGeslacht());
}
```

Je voert de test uit. Hij lukt.

6.3 Voorstelling als varchar kolom of enum kolom

Je voert het script `StringGeslacht.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 wijzigt `insertDocent.sql`:

- Je wijzigt in het 1° SQL statement de laatste 0 naar 'MAN'.
- Je wijzigt in het 2° SQL statement de laatste 1 naar 'VROUW'.

Je slaat het bestand op.

Je voert de test uit. Die lukt.



Je commit de sources. Je publiceert op GitHub.



7 JPA PROJECT

7.1 Configuratie

Je tikt JPA annotations bij entity classes.

JPA controleert, bij het maken van de EntityManager 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.

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 EclipseLink.
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 geeft de JDBC driver aan die 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 op je computer van in de cursus JDBC.
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. Je corrigeert dan de invoervakken.
23. Je kiest de knop Finish.
24. Je kiest Discover annotated classes automatically.
25. Je kiest Finish.



Het lijkt misschien vreemd dat je hier de JDBC driver beschreven hebt om connecties te maken en dat je ook de connectiegegevens beschreven hebt, terwijl deze reeds beschreven zijn in pom.xml en application.properties.

De configuratie die je hier beschreven hebt wordt echter niet gebruikt door je programma *at run time*, maar Eclipse *at design time* van je programma.

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 fout 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 fout verdwijnt als je dosenten wijzigt naar docenten en de source opslaat.

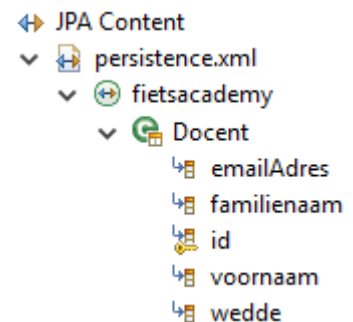
7.3 Project Explorer

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

Als je dit onderdeel openklapt zie je de entity class(es), 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).



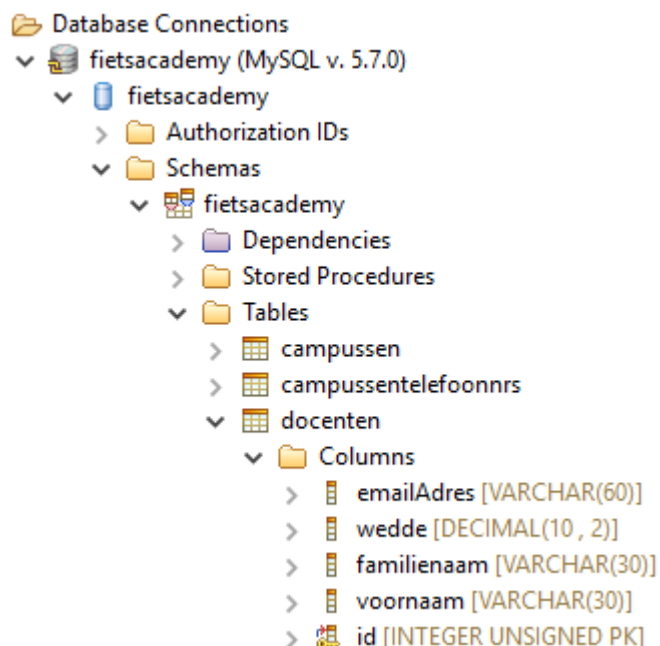
7.4 Data Source Explorer

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 bewaart de wijzigingen met de knop  in de Eclipse toolbar.



7.5 persistence.xml

Eclipse heeft in `src/main/java` een map `META-INF` gemaakt en daarin `persistence.xml`.

Je applicatie heeft dit bestand niet nodig, maar Eclipse wel.

Je moet echter één regelin dit bestand wijzigen, anders krijg je problemen in het hoofdstuk JPQL:

```
<persistence-unit name="default">
```

8 DALI

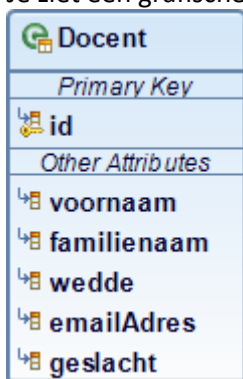
Dali is een plugin voor Eclipse waarmee je grafische voorstellingen krijgt van je entity classes.

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.
Je kiest hierbij de laatste versie.
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 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 met de rechtermuisknop 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.

8.3 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.



Je hebt Eclipse extensies geïnstalleerd en geconfigureerd om gemakkelijker te werken met je project waarin je JPA gebruikt. Maar je project blijft nog altijd een standaard Maven project dat je perfect met een andere IDE (bvb. NetBeans) kan openen.



JPA project - Dali: zie takenbundel

9 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.

9.1 @GeneratedValue

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

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

Als de database 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 autonummer kolom id.

9.2 Sequence

Sommige databasemerken, zoals Oracle, kennen geen autonummer 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 leert hier hoe zou je werken met een database die sequences ondersteunt.

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.

9.3 Voorbeeld

Je maakt een geparametriseerde constructor in Docent. Je maakt daarin geen parameter voor id.

Je maakt ook een default constructor. JPA heeft die nodig. Je maakt die constructor protected.

Je maakt een method declaratie in DocentRepository:

```
void create(Docent docent);
```

Je implementeert deze method in JpaDocentRepository. Je maakt zoals in de JUnit cursus een eerste minimale implementatie, zodat je er een test voor kan schrijven.

```
@Override
```

```
public void create(Docent docent) {
```

```
    throw new UnsupportedOperationException();
```

```
}
```

Je maakt een constante in JpaDocentRepositoryTest:

```
private static final String DOCENTEN = "docenten";
```

Je maakt een variabele in JpaDocentRepositoryTest:

```
private Docent docent;
```

Je maakt een method:

```
@Before
public void before() {
    docent = new Docent("test", "test", BigDecimal.TEN, "test@fietsacademy.be",
        Geslacht.MAN);
}
```

Je maakt een test:

```
@Test
public void create() {
    int aantalDocenten = super.countRowsInTable(DOCENTEN);
    repository.create(docent);
    assertEquals(aantalDocenten + 1, super.countRowsInTable("docenten"));
    assertNotEquals(0, docent.getId());
    assertEquals(1, super.countRowsInTableWhere(DOCENTEN,
        "id=" + docent.getId()));
}
```

Je voert de test uit. Hij mislukt.

Je implementeert de method in JpaDocentRepository:

```
@Override
public void create(Docent docent) {
    manager.persist(docent);
}
```

Je voert de test uit. Je doet dit gemakkelijk in het tabblad JUnit, onder in Eclipse.

Je klikt met de rechtermuisknop op create en je kiest Run.

De test lukt.



Je commit de sources. Je publiceert op GitHub.



Toevoegen: zie takenbundel

10 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 maakt een method declaratie in DocentRepository:

```
void delete(long id);
```

Je implementeert deze method in JpaDocentRepository:

```
@Override
public void delete(long id) {
    throw new UnsupportedOperationException();
}
```

Je maakt een variabele in JpaDocentRepositoryTest:

```
@Autowired
private EntityManager manager;
```

Je maakt een test:

```
@Test
public void delete() {
    long id = idVanTestMan();
    int aantalDocenten = super.countRowsInTable(DOCENTEN);
    repository.delete(id);
    manager.flush();
    assertEquals(aantalDocenten - 1, super.countRowsInTable(DOCENTEN));
    assertEquals(0, super.countRowsInTableWhere(DOCENTEN, "id=" + id));
}
```

- (1) Je hebt op de vorige regel de docent verwijderd.
De EntityManager voert verwijderingen niet onmiddellijk uit, maar spaart die op tot juist voor de commit van de transactie. Hij stuurt dan alle opgespaarde verwijderingen in één keer naar de database, via JDBC batch updates. Dit verhoogt de performantie.
In deze test is het wel nodig dat de docent onmiddellijk verwijderd wordt, zodat we de correcte werking van onze delete method kunnen testen.
Je voert de flush method uit die gevraagde verwijderingen onmiddellijk uitvoert.

Je voert de test uit. Hij mislukt.

Je implementeert de method in JpaDocentRepository:

```
@Override
public void delete(long id) {
    read(id)
        .ifPresent(docent -> manager.remove(docent));
}
```

- (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 voert de test uit. Hij lukt.

11 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 moet dus ook geen update(...) method maken in DocentRepository.

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 maakt een method in Docent :

```
public void opslag(BigDecimal percentage) {
    if (percentage.compareTo(BigDecimal.ZERO) <= 0 ) {
        throw new IllegalArgumentException();
    }
    BigDecimal factor =
        BigDecimal.ONE.add(percentage.divide(BigDecimal.valueOf(100)));
    wedde = wedde.multiply(factor, new MathContext(2, RoundingMode.HALF_UP));
}
```

Je maakt tests voor deze method:

```
package be.vdab.fietsacademy.entities;
// enkele imports
public class DocentTest {
    private final static BigDecimal ORIGINELE_WEDDE = BigDecimal.valueOf(200);
    private Docent docent1;
    @Before
    public void before() {
        docent1 = new Docent("test", "test", ORIGINELE_WEDDE,
            "test@fietsacademy.be", Geslacht.MAN);
    }
    @Test
    public void opslag() {
        docent1.opslag(BigDecimal.TEN);
        assertEquals(0, BigDecimal.valueOf(220).compareTo(docent1.getWedde()));
    }
    @Test(expected = NullPointerException.class)
    public void opslagMetNullKanNiet() {
        docent1.opslag(null);
    }
    @Test(expected = IllegalArgumentException.class)
    public void opslagMet0KanNiet() {
        docent1.opslag(BigDecimal.ZERO);
        assertEquals(0, ORIGINELE_WEDDE.compareTo(docent1.getWedde()));
    }
    @Test(expected = IllegalArgumentException.class)
    public void negatieveOpslagKanNiet() {
        docent1.opslag(BigDecimal.valueOf(-1));
        assertEquals(0, ORIGINELE_WEDDE.compareTo(docent1.getWedde()));
    }
}
```

Je voert de tests uit. Die lukken.

Je maakt een package `be.vdab.fietsacademy.exceptions`.

Je maakt daarin een class `DocentNietGevondenException`:

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

Je maakt een package `be.vdab.fietsacademy.services`.

Je maakt daarin een interface `DocentService`:

```
package be.vdab.fietsacademy.services;
import java.math.BigDecimal;
public interface DocentService {
    void opslag(long id, BigDecimal percentage);
}
```

Je implementeert deze method in een class `DefaultDocentService`:

```
package be.vdab.fietsacademy.services;
import java.math.BigDecimal;
@Service
class DefaultDocentService implements DocentService {
    private final DocentRepository docentRepository;
    DefaultDocentService(DocentRepository docentRepository) {
        this.docentRepository = docentRepository;
    }
    @Override
    public void opslag(long id, BigDecimal percentage) {
        throw new UnsupportedOperationException();
    }
}
```

Je schrijf unit tests voor deze method. Je test zoveel mogelijk functionaliteit met unit tests, omdat unit tests sneller uitvoeren dan integration tests.

```
package be.vdab.fietsacademy.services;
import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.when;
import static org.mockito.Mockito.verify;
// enkele andere imports
@RunWith(MockitoJUnitRunner.class)
public class DefaultDocentServiceTest {
    private DefaultDocentService service;
    @Mock
    private DocentRepository repository;
    private Docent docent;
    @Before
    public void before() {
        docent = new Docent("test", "test", BigDecimal.valueOf(100),
            "test@fietsacademy.be", Geslacht.MAN);
        when(repository.read(1)).thenReturn(Optional.of(docent));
        when(repository.read(-1)).thenReturn(Optional.empty());
        service = new DefaultDocentService(repository);
    }
    @Test
    public void opslag() {
        service.opslag(1, BigDecimal.TEN);
        assertEquals(0, BigDecimal.valueOf(110).compareTo(docent.getWedde()));
        verify(repository).read(1);
    }
}
```

```

@Test(expected = DocentNietGevondenException.class)
public void opslagVoorOnbestaandeDocent() {
    service.opslag(-1, BigDecimal.TEN);
    verify(repository).read(-1);
}
}

```

Je voert de tests uit. Ze mislukken.

Je maakt ook een integration test. Je test daarin wat je in de unit test niet kan testen: is de docent niet enkel gewijzigd in het interne geheugen, maar ook in de database ?

```

package be.vdab.fietsacademy.services;
// enkele imports
@RunWith(SpringRunner.class)
@SpringBootTest
@Sql("/insertDocent.sql")
public class DefaultDocentServiceIntegrationTest
    extends AbstractTransactionalJUnit4SpringContextTests {
    @Autowired
    private DefaultDocentService service;
    @Autowired
    private EntityManager manager;
    private long idVanTestMan() {
        return super.jdbcTemplate.queryForObject(
            "select id from docenten where voornaam='testM'", Long.class);
    }
    @Test
    public void opslag() {
        long id = idVanTestMan();
        service.opslag(id, BigDecimal.TEN);
        manager.flush();
        BigDecimal nieuweWedde = super.jdbcTemplate.queryForObject(
            "select wedde from docenten where id=?", BigDecimal.class, id);
        assertEquals(0, BigDecimal.valueOf(1_100).compareTo(nieuweWedde));
    }
}

```

- (1) Je hebt in deze test veel beans nodig: een DataSource bean, een JpaDocenRepository bean, een DefaultDocentService bean. Als je veel beans nodig hebt, wordt het complex deze één per één te laden met @Import. Je gebruikt @SpringBootTest als alternatief. Deze annotation laadt alle beans (DataSource, Repository bean, Service beans, Controller beans) en stelt ze ter beschikking van je test.
- (2) Je hebt op de vorige regel de docent gewijzigd. De EntityManager voert wijzigingen niet onmiddellijk uit, maar spaart die op tot juist voor de commit van de transactie. Hij stuurt dan alle opgespaarde wijzigingen in één keer naar de database, via JDBC batch updates. Dit verhoogt de performantie. In deze test is het wel nodig dat de docent onmiddellijk gewijzigd wordt, zodat we de correcte werking van onze opslag method kunnen testen. Je voert de flush method uit die gevraagde wijzigingen onmiddellijk uitvoert.

Je voert de tests uit. Ze mislukken.

Je implementeert de method in DefaultDocentService:

```
package be.vdab.fietsacademy.services;
// enkele imports
@Service
@Transactional(readOnly = true, isolation = Isolation.READ_COMMITTED)
class DefaultDocentService implements DocentService {
    private final DocentRepository docentRepository;
    DefaultDocentService(DocentRepository docentRepository) {
        this.docentRepository = docentRepository;
    }
    @Override
    @Transactional(readOnly = false, isolation = Isolation.READ_COMMITTED)
    public void opslag(long id, BigDecimal percentage) {
        Optional<Docent> optionalDocent = docentRepository.read(id);
        if (optionalDocent.isPresent()) {
            optionalDocent.get().opslag(percentage);
        } else {
            throw new DocentNietGevondenException();
        }
    }
}
```

Je voert de unit tests en de integration tests uit. Ze lukken.



Je commit de sources. Je publiceert op GitHub.



De EntityManager stuurt bij de commit per gewijzigde entity een update statement naar de database.

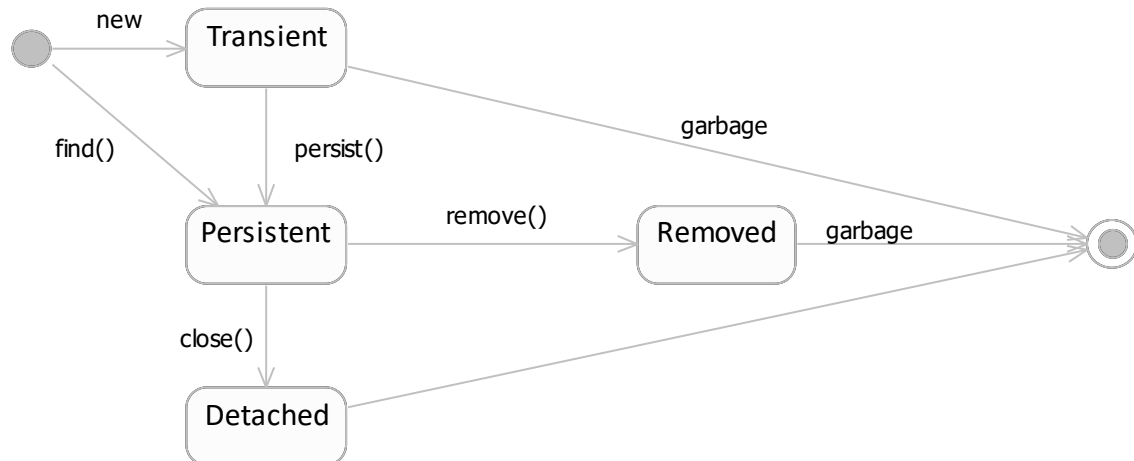
Je hebt nu de CRUD operaties gezien:

- Create (een entity toevoegen)
- Read (een entity lezen aan de hand van zijn primary key waarde)
- Update (een entity wijzigen)
- Delete (een entity verwijderen).

12 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 `persist` 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.

13 JPQL 🔭

13.1 Algemeen

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

Je maakt 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 = manager.createQuery("een JPQL query", 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.

13.2 Alle entities lezen

Je 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 familienaam0_, docent0_.wedde as wedde0_
from docenten docent0_
```

Je maakt een method in DocentRepository:

```
List<Docent> findAll();
```

Je implementeert de method in JpaDocentRepository:

```
@Override
public List<Docent> findAll() {
    throw new UnsupportedOperationException();
}
```

Je test deze method in JpaDocentRepositoryTest:

```
@Test
public void findAll() {
    List<Docent> docenten = repository.findAll();
    assertEquals(super.countRowsInTable(DOCENTEN), docenten.size());
}
```

Je voert de test uit. Hij mislukt.

Je implementeert code in de method `findAll` in JpaDocentRepository:

```
return manager.createQuery("select d from Docent d", Docent.class)
    .getResultList();
```

Je voert de test uit. Hij lukt.

13.3 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 sorteert de docenten op wedde.

Je voegt code toe aan de method `findAll` in `JpaDocentRepositoryTest`:

```
BigDecimal vorigeWedde = BigDecimal.ZERO;
for (Docent docent : docenten) {
    assertTrue(docent.getWedde().compareTo(vorigeWedde) >= 0);
    vorigeWedde = docent.getWedde();
}
```

Je voert de test uit. Hij mislukt.

Je wijzigt het JPQL statement in de method `findAll` in `JpaDocentRepository`:

```
select d from Docent d order by d.wedde
```

Je voert de test uit. Hij lukt.



Opmerking: JPQL sorteert standaard van klein naar groot.

Je kan ook sorteren van groot naar klein: **order by** d.wedde **desc**.

Je kan ook op meerdere waarden sorteren: **order by** d.wedde, d.familienaam.

JPA sorteert op wedde. JPA sorteert docenten met gelijke wedde op familienaam.

13.4 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 maakt een method declaratie in DocentRepository:

```
List<Docent> findByWeddeBetween(BigDecimal van, BigDecimal tot);
```

Je implementeert deze method in JpaDocentRepository:

```
@Override
public List<Docent> findByWeddeBetween(BigDecimal van, BigDecimal tot) {
    throw new UnsupportedOperationException();
}
```

Je maakt een test in JpaDocentRepositoryTest:

```
@Test
public void findByWeddeBetween() {
    BigDecimal duizend = BigDecimal.valueOf(1_000);
    BigDecimal tweeduizend = BigDecimal.valueOf(2_000);
    List<Docent> docenten = repository.findByWeddeBetween(duizend, tweeduizend);
    long aantalDocenten = super.countRowsInTableWhere(DOCCENTEN,
        "wedde between 1000 and 2000");
    assertEquals(aantalDocenten, docenten.size());
    docenten.forEach(docent -> {
        assertTrue(docent.getWedde().compareTo(duizend) >= 0);
        assertTrue(docent.getWedde().compareTo(tweeduizend) <= 0);
    });
}
```

Je voert de test uit. Hij mislukt.

13.4.1 Named parameters

Je stelt een parameter in een JPQL query 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 JpaDocentRepository de code in de method findByWeddeBetween:

```
return manager.createQuery(
    "select d from Docent d where d.wedde between :van and :tot", Docent.class)
    .setParameter("van", van)
    .setParameter("tot", tot)
    .getResultList();
```

Je voert de test uit. Hij lukt.



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.



Je kan in een JPQL query selecteren én sorteren:

```
select d from Docent d where d.wedde = 1000 order by d.familienaam
```

13.5 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.emailAdres from Docent d
```

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

Je maakt een method declaratie in DocentRepository:

```
List<String> findEmailAdressen();
```


Je implementeert deze method in JpaDocentRepository:

```
@Override
public List<String> findEmailAdressen() {
    throw new UnsupportedOperationException();
}
```

Je maakt een test in JpaDocentRepositoryTest:

```
@Test
public void findEmailAdressen() {
    List<String> adressen = repository.findEmailAdressen();
    long aantal = super.jdbcTemplate.queryForObject(
        "select count(distinct emailadres) from docenten", Long.class);
    assertEquals(aantal, adressen.size());
    adressen.forEach(adres -> assertTrue(adres.contains("@")));
}
```

Je voert de test uit. Hij mislukt.

Je implementeert de code in de method in JpaDocentRepository:

```
return manager.createQuery(
    "select d.emailAdres from Docent d", String.class).getResultList();
```

Je voert de test uit. Hij lukt.

13.6 Meerdere kolommen lezen

Volgende query geeft een lijst met het nummer en het email adres van elke docent:

```
select new be.vdab.fietsacademy.queryresults.IdEnEmailAdres(d.id, d.emailAdres)
from Docent d
```

Het query resultaat is een List met IdEnEmailAdres objecten.

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

- een **long** parameter (die het id binnenkrijgt van JPA)
- een **String** parameter (die het email adres binnenkrijgt van JPA)

Je maakt een package `be.vdab.fietsacademy.queryresult`.

Je maakt daarin de class `IdEnEmailAdres`:

```
package be.vdab.fietsacademy.queryresults;
public class IdEnEmailAdres {
    private final long id;
    private final String emailAdres;
    // een geparametriseerde constructor en getters voor de private variabelen
}
```

Je maakt een method declaratie in `DocentRepository`:

```
List<IdEnEmailAdres> findIdsEnEmailAdressen();
```

Je implementeert deze method in `JpaDocentRepository`:

```
@Override
public List<IdEnEmailAdres> findIdsEnEmailAdressen() {
    throw new UnsupportedOperationException();
}
```

Je maakt een test in `JpaDocentRepositoryTest`:

```
@Test
public void findIdsEnEmailAdressen() {
    List<IdEnEmailAdres> idsEnAdressen = repository.findIdsEnEmailAdressen();
    assertEquals(super.countRowsInTable(DOCENTEN), idsEnAdressen.size());
}
```

Je voert de test uit. Hij mislukt.

Je implementeert de code in de method in JpaDocentRepository:

```
return manager.createQuery(
    "select new be.vdab.fietsacademy.queryresults.IdEmailAdres(d.id, d.emailAdres)" +
    "from Docent d", IdEmailAdres.class).getResultList();
```

Je voert de test uit. Hij lukt.

13.7 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 BigDecimals: 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 getResult Deze geeft je die rij.

Je maakt een method declaratie in DocentRepository:

```
BigDecimal findGrootsteWedde();
```

Je implementeert deze method in JpaDocentRepository:

```
@Override
public BigDecimal findGrootsteWedde() {
    throw new UnsupportedOperationException();
}
```

Je maakt een test in JpaDocentRepositoryTest:

```
@Test
public void findGrootsteWedde() {
    BigDecimal grootste = repository.findGrootsteWedde();
    BigDecimal grootste2 = super.jdbcTemplate.queryForObject(
        "select max(wedde) from docenten", BigDecimal.class);
    assertEquals(0, grootste.compareTo(grootste2));
}
```

Je voert de test uit. Hij mislukt.

Je implementeert de code in de method in JpaDocentRepository:

```
return manager.createQuery(
    "select max(d.wedde) from Docent d", BigDecimal.class).getSingleResult();
```

Je voert de test uit. Hij lukt.

13.8 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.fietsacademy.queryresults.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 maakt. 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.fietsacademy.queryresults;
//enkele imports
public class AantalDocentenPerWedde {
    private final BigDecimal wedde;
    private final long aantal;
    // een geparametriseerde constructor en getters voor de private variabelen
}
```

Je maakt een method declaratie in DocentRepository:

```
List<AantalDocentenPerWedde> findAantalDocentenPerWedde();
```

Je implementeert deze method in JpaDocentRepository:

```
@Override
public List<AantalDocentenPerWedde> findAantalDocentenPerWedde() {
    throw new UnsupportedOperationException();
}
```

Je maakt een test in JpaDocentRepositoryTest:

```
@Test
public void findAantalDocentenPerWedde() {
    List<AantalDocentenPerWedde> aantalDocentenPerWedde =
        repository.findAantalDocentenPerWedde();
    long aantalUniekeWeddes = super.jdbcTemplate.queryForObject(
        "select count(distinct wedde) from docenten", Long.class);
    assertEquals(aantalUniekeWeddes, aantalDocentenPerWedde.size());
    long aantalDocentenMetWedde1000 = super.countRowsInTableWhere(DOCENTEN,
        "wedde = 1000");
    aantalDocentenPerWedde.stream()
        .filter(aantalPerWedde ->
            aantalPerWedde.getWedde().compareTo(BigDecimal.valueOf(1_000)) == 0)
        .forEach(aantalPerWedde ->
            assertEquals(aantalDocentenMetWedde1000, aantalPerWedde.getAantal()));
}
```

Je voert de test uit. Hij mislukt.

Je implementeert de code in de method in JpaDocentRepository:

```
return manager.createQuery(
    "select new be.vdab.fietsacademy.queryresults.AantalDocentenPerWedde(" +
    "d.wedde,count(d)) from Docent d group by d.wedde",
    AantalDocentenPerWedde.class).getResultList();
```

Je voert de test uit. Hij lukt.

13.9 Named queries

Je maakt queries tot nu met de EntityManager method createQuery.

JPA controleert bij *elke* uitvoering van de query de syntax en vertaalt de query naar SQL.

JPA controleert de syntax van een named query maar 1 keer en vertaalt hem maar 1 keer naar een SQL statement. JPA onthoudt dit SQL statement in het interne geheugen gedurende de levensduur van de applicatie.

Named queries hebben voordelen:

- ⊕ als de query syntaxfouten bevat, merk je dit vroeg (bij de start van de applicatie) en niet laat (bij de 1^o uitvoering van de query). Bij test driven development (zoals bij ons) merk je het nog vroeger: bij het mislukken van DataSourceTest.
- ⊕ JPA doet bij elke query uitvoering geen syntaxcontrole en vertaling naar SQL meer, en voert de query dus snel uit.

13.9.1 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, juist 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")
```

13.9.2 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 de JpaDocentRepository method findByWeddeBetween createQuery(...) door:
createNamedQuery("Docent.findByWeddeBetween", Docent.class)

Je voert de test uit. Hij lukt.

13.9.3 Orm.xml

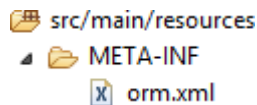
Je kan named queries ook schrijven in src/main/resources/META-INF/orm.xml.

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 verwijdt in Docent de regel met @NamedQuery.

Je maakt in src/main/resources
een map META-INF
en daarin orm.xml



```
<?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 voert de test uit. Hij lukt.



Je commit de sources. Je publiceert op GitHub.



Zoeken op naam: zie takenbundel

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

Je maakt een query in orm.xml:

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

Je maakt een method declaratie in DocentRepository:

```
int algemeneOpslag(BigDecimal percentage);
```

Je implementeert deze method declaratie in JpaDocentRepository:

```
@Override
public int algemeneOpslag(BigDecimal percentage) {
    throw new UnsupportedOperationException();
}
```

Je maakt een test in JpaDocentRepositoryTest:

```
@Test
public void algemeneOpslag() {
    int aantalAangepast = repository.algemeneOpslag(BigDecimal.TEN);
    assertEquals(super.countRowsInTable(DOCENTEN), aantalAangepast);
    BigDecimal nieuweWedde = super.jdbcTemplate.queryForObject(
        "select wedde from docenten where id=?", BigDecimal.class, idVanTestMan());
    assertEquals(0, BigDecimal.valueOf(1_100).compareTo(nieuweWedde));
}
```

Je voert de test uit. Hij mislukt.

Je implementeert de code in de method in JpaDocentRepository:

```
BigDecimal factor =
    BigDecimal.ONE.add(percentage.divide(BigDecimal.valueOf(100)));
return manager.createNamedQuery("Docent.algemeneOpslag")
    .setParameter("factor", factor)
    .executeUpdate();
```

❶

- (1) De method executeUpdate voert de bulk update uit en geeft het aantal gewijzigde records terug.

Je voert de test uit. Hij lukt.



Je commit de sources. Je publiceert op GitHub.



Prijsverhoging: zie takenbundel

15 INHERITANCE 🏠🏠🏠

15.1 Inheritance nabootsen in de database

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

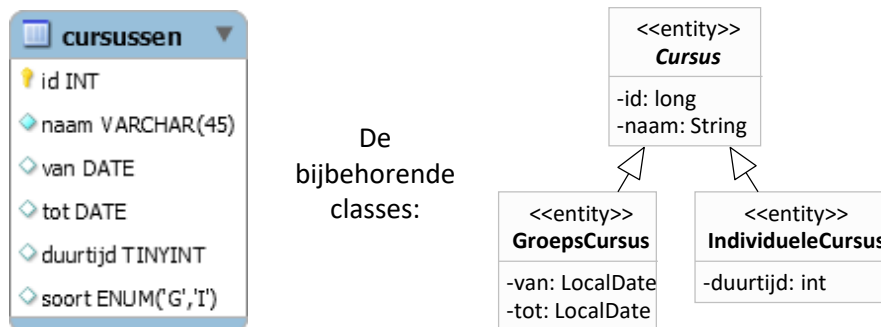
Je kan inheritance in de database nabootsen op drie manieren, die je hier leert:

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

15.2 Table per class hierarchy

15.2.1 Database

Je voert het script `Hierarchy.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.

Je zorgt er met volgende stappen voor dat ook Eclipse deze nieuwe table ziet:

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

15.2.2 Voordelen

- ➕ Als je een cursus toevoegt, stuurt JPA één **insert** statement naar één table.
JPA 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.

15.2.3 Nadelen

- ➖ 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.

15.2.4 Base class

```

package be.vdab.fietsacademy.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;
  
```

❶

❷

```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private long id;
private String naam;
// een geparametriseerde en een protected default constructor
// getters voor id en naam
}

```

- (1) @Inheritance staat bij de hoogste class in de inheritance hiërarchie. strategy duidt de manier aan waarmee je inheritance nabootst in de database. SINGLE_TABLE staat voor “table per class hiërarchy”.
- (2) @DiscriminatorColumn duidt de naam van discriminator kolom aan.



Opmerking: de discriminator kolom (soort) heeft geen bijbehorende variabele.

15.2.5 Derived classes

```

package be.vdab.fietsacademy.entities;
// enkele imports ...
@Entity
@DiscriminatorValue("G")
public class GroepsCursus extends Cursus {
    private static final long serialVersionUID = 1L;
    private LocalDate van;
    private LocalDate tot;
    // een geparametriseerde constructor en een protected default constructor
    // getters voor van en tot
}

```

- (1) @DiscriminatorValue duidt de waarde in de discriminator kolom (Soort) aan als een record hoort bij een entity van deze entity (GroepsCursus).

```

package be.vdab.fietsacademy.entities;
// enkele imports ...
@Entity
@DiscriminatorValue("I")
public class IndividueleCursus extends Cursus {
    private static final long serialVersionUID = 1L;
    private int duurtijd;
    // een geparametriseerde constructor en een protected default constructor
    // een getter voor duurtijd
}

```

15.2.6 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.

15.2.7 Repository layer

Je maakt een interface CursusRepository:

```

package be.vdab.fietsacademy.repositories;
// enkele imports
public interface CursusRepository {
    Optional<Cursus> read(long id);
    void create(Cursus cursus);
}

```

Je implementeert de interface in een class JpaCursusRepository:

```
package be.vdab.fietsacademy.repositories;
import be.vdab.fietsacademy.entities.Cursus;
// enkele imports
@Repository
public class JpaCursusRepository implements CursusRepository {
    @Override
    public Optional<Cursus> read(long id) {
        throw new UnsupportedOperationException();
    }
    @Override
    public void create(Cursus cursus) {
        throw new UnsupportedOperationException();
    }
}
```

Je maakt insertCursus.sql in src/test/resources:

```
insert into cursussen(naam, van, tot, soort)
values('testGroep', '2018-01-01', '2018-01-01', 'G');
insert into cursussen(naam, duurtijd, soort)
values('testIndividueel', 3, 'I');
```

Je maakt testen:

```
package be.vdab.fietsacademy.repositories;
// enkele imports
@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace = Replace.NONE)
@Import(JpaCursusRepository.class)
@Sql("/insertCursus.sql")
public class JpaCursusRepositoryTest
    extends AbstractTransactionalJUnit4SpringContextTests {
    private static final String CURSUSSEN = "cursussen";
    private static final LocalDate EEN_DATUM = LocalDate.of(2019, 1, 1);
    @Autowired
    private JpaCursusRepository repository;
    private long idVanTestGroepsCursus() {
        return super.jdbcTemplate.queryForObject(
            "select id from cursussen where naam='testGroep'", Long.class);
    }
    private long idVanTestIndividueleCursus() {
        return super.jdbcTemplate.queryForObject(
            "select id from cursussen where naam='testIndividueel'", Long.class);
    }
    @Test
    public void readGroepsCursus() {
        Optional<Cursus> optionalCursus = repository.read(idVanTestGroepsCursus());
        assertEquals("testGroep", ((GroepsCursus) optionalCursus.get()).getNaam());
    }
    @Test
    public void readIndividueleCursus() {
        Optional<Cursus> optionalCursus =
            repository.read(idVanTestIndividueleCursus());
        assertEquals("testIndividueel",
            ((IndividueleCursus) optionalCursus.get()).getNaam());
    }
}
```



```

@Test
public void createGroepsCursus() {
    int aantalGroepsCursussen = super.countRowsInTableWhere(CURSUSSEN,
        "soort='G'");
    GroepsCursus cursus = new GroepsCursus("testGroep2", EEN_DATUM, EEN_DATUM);
    repository.create(cursus);
    assertEquals(aantalGroepsCursussen + 1,
        super.countRowsInTableWhere(CURSUSSEN, "soort='G'"));
    assertEquals(1,
        super.countRowsInTableWhere(CURSUSSEN, "id=" + cursus.getId()));
}

@Test
public void createIndividueleCursus() {
    int aantalIndividueleCursussen = super.countRowsInTableWhere(CURSUSSEN,
        "soort='I'");
    IndividueleCursus cursus = new IndividueleCursus("testIndividueel2", 7);
    repository.create(cursus);
    assertEquals(aantalIndividueleCursussen + 1,
        super.countRowsInTableWhere(CURSUSSEN, "soort='I'"));
    assertEquals(1,
        super.countRowsInTableWhere(CURSUSSEN, "id=" + cursus.getId()));
}
}

```

Je voert de tests uit. Ze mislukken.

Je implementeert JpaCursusRepository:

```

package be.vdab.fietsacademy.repositories;
// enkele imports

@Repository
public class JpaCursusRepository implements CursusRepository {
    private final EntityManager manager;
    JpaCursusRepository(EntityManager manager) {
        this.manager = manager;
    }
    @Override
    public Optional<Cursus> read(long id) {
        return Optional.ofNullable(manager.find(Cursus.class, id));
    }
    @Override
    public void create(Cursus cursus) {
        manager.persist(cursus);
    }
}

```

Je voert de tests uit. Ze lukken.



Je commit de sources. Je publiceert op GitHub.

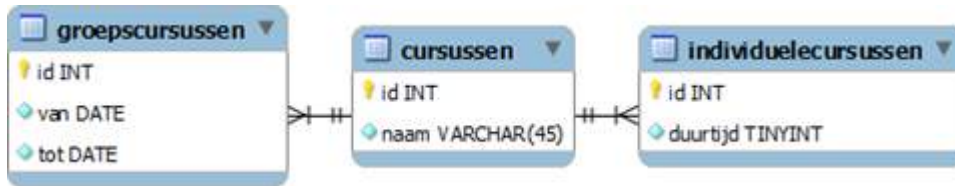
15.3 Polymorphic queries

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

JPQL query	Type objecten in resultaat
select g from GroepsCursus g	GroepsCursus
select c from Cursus c	Cursus, GroepsCursus en IndividueleCursus

15.4 Table per subclass

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



Je zorgt er voor dat ook Eclipse deze nieuwe tables kent:

1. Je klapt in de Data Source Explorer database niveau's open tot je de map Tables ziet.
2. Je klikt met de rechtermuisknop op Tables en je kiest Refresh

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);

15.4.1 Voordelen

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

15.4.2 Nadelen

- ⊖ Als je een entity toevoegt, stuurt JPA **insert** statements naar *meerdere* tables. Ook bij het wijzigen of verwijderen van een klassieke cursus of zelfstudiecursus 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.

15.4.3 Base class annotations

```

@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@Table(name = "cursussen")
  
```

- (1) Je plaatst bij table per subclass de 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.

15.4.4 Derived classes annotations : GroepsCursus

```

@Entity
@Table(name = "groepscursussen")
  
```

15.4.5 Derived classes annotations : IndividueleCursus

```

@Entity
@Table(name = "individuelecursussen")
  
```

15.4.6 Testen

Je wijzigt insertCursus.sql:

```
insert into cursussen(naam) values('testGroep');
insert into groepscursussen(id,van,tot)
values((select id from cursussen where naam='testGroep'),
'2018-01-01','2018-01-01');
insert into cursussen(naam) values('testIndividueel');
insert into individuelecursussen(id,duurtijd)
values((select id from cursussen where naam='testIndividueel'),3);
```

Je slaat het bestand op.

Je maakt constanten in JpaCursusRepositoryTest:

```
private static final String GROEPS_CURSUSSEN = "groepscursussen";
private static final String INDIVIDUELE_CURSUSSEN = "individuelecursussen";
```

Je wijzigt de methods createGroepsCursus en createIndividueleCursus:

```
@Test
public void createGroepsCursus() {
    int aantalRecordsInCursussen = super.countRowsInTable(CURSUSSEN);
    int aantalRecordsInGroepsCursussen = super.countRowsInTable(GROEPS_CURSUSSEN);
    GroepsCursus cursus = new GroepsCursus("testGroep2", EEN_DATUM, EEN_DATUM);
    repository.create(cursus);
    assertEquals(aantalRecordsInCursussen + 1, super.countRowsInTable(CURSUSSEN));
    assertEquals(aantalRecordsInGroepsCursussen + 1,
        super.countRowsInTable(GROEPS_CURSUSSEN));
    assertEquals(1,
        super.countRowsInTableWhere(CURSUSSEN, "id=" + cursus.getId()));
    assertEquals(1,
        super.countRowsInTableWhere(GROEPS_CURSUSSEN, "id=" + cursus.getId()));
}

@Test
public void createIndividueleCursus() {
    int aantalRecordsInCursussen = super.countRowsInTable(CURSUSSEN);
    int aantalRecordsInIndividueleCursussen =
        super.countRowsInTable(INDIVIDUELE_CURSUSSEN);
    IndividueleCursus cursus = new IndividueleCursus("testIndividueel2", 7);
    repository.create(cursus);
    assertEquals(aantalRecordsInCursussen + 1, super.countRowsInTable(CURSUSSEN));
    assertEquals(aantalRecordsInIndividueleCursussen + 1,
        super.countRowsInTable(INDIVIDUELE_CURSUSSEN));
    assertEquals(1,
        super.countRowsInTableWhere(CURSUSSEN, "id=" + cursus.getId()));
    assertEquals(1,
        super.countRowsInTableWhere(INDIVIDUELE_CURSUSSEN, "id=" + cursus.getId()));
}
```



Je commit de sources. Je publiceert op GitHub.

15.5 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 `ConcreteClass.sql` uit. Dit verwijdert de table cursussen en voegt nieuwe versies van de tables groepscursussen en individuelecursussen toe.



Je kan bij table per concrete class geen autonummer veld gebruiken als primary key.

Je zou records met hetzelfde nummer hebben in de table groepscursussen en in de table individuelecursussen. Table per concrete class laat dit niet toe.

Een oplossing is een sequence zijn, gedeeld door beide tables. MySQL kent echter geen sequence.

Je vult in beide tables de kolom id van een nieuw record met een UUID.

Dit is een string met hexadecimale tekens. Elke gegenereerde UUID is uniek over de hele wereld.

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

1. Je klapt in de Data Source Explorer database niveau's open tot je de map Tables ziet.
2. Je klikt met de rechtermuisknop op Tables en je kiest Refresh.

15.5.1 Voordelen

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

15.5.2 Nadelen

- Als je een attribuut toevoegt aan de base class, moet je aan meerdere tables een kolom toevoegen. Als je bijvoorbeeld prijs toevoegt aan Cursus, moet je een kolom Prijs toevoegen aan de table groepscursussen én aan de table individuelecursussen.
- Je kan geen autonummer gebruiken als primary key. Het alternatief (UUID) geeft primary keys die minder leesbaar zijn en wat trager zijn als je ze ook gebruikt als foreign key in andere tables.

15.5.3 Annotations

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

- (1) Je duidt "table per concrete class" aan met `TABLE_PER_CLASS`.

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

Je wijzigt het type van de variabele `id` naar `String`.

Je voegt aan de geparametriseerde constructor volgende opdracht toe:

```
id = UUID.randomUUID().toString(); // nieuwe unieke UUID toekennen
```

Je wijzigt het returntype van de method `getId()` naar `String`.

GroepsCursus en IndividueleCursus annotations blijven dezelfde.

Je wijzigt in `CursusRepository` in de method `read` het type van de parameter `id` naar `String`.

Je wijzigt in `JpaCursusRepository` in de method `read` het type van de parameter `id` naar `String`.

Je wijzigt insertCursus.sql:

```
insert into groepscursussen(id,naam,van,tot)
values(uuid(),'testGroep','2018-01-01','2018-01-01');
insert into individuelecursussen(id,naam,duurtijd)
values(uuid(),'testIndividueel',3);
```

Je slaat het bestand op.

Je verwijdert de constante CURSUSSEN in JpaCursusRepositoryTest.

Je wijzigt de methods idVanTestGroepsCursus en idVanTestIndividueleCursus:

```
private String idVanTestGroepsCursus() {
    return super.jdbcTemplate.queryForObject(
        "select id from groepscursussen where naam='testGroep'", String.class);
}
private String idVanTestIndividueleCursus() {
    return super.jdbcTemplate.queryForObject(
        "select id from individuelecursussen where naam='testIndividueel'",
        String.class);
}
```

Je maakt een variabele: @Autowired private EntityManager manager;

Je wijzigt de methods createGroepsCursus en createIndividueleCursus:

```
@Test
public void createGroepsCursus() {
    int aantalGroepsCursussen = super.countRowsInTable(GROEPS_CURSUSSEN);
    GroepsCursus cursus = new GroepsCursus("testGroep2", EEN_DATUM, EEN_DATUM);
    repository.create(cursus);
    manager.flush();
    assertEquals(aantalGroepsCursussen + 1,
        super.countRowsInTable(GROEPS_CURSUSSEN));
    assertEquals(1, super.countRowsInTableWhere(GROEPS_CURSUSSEN,
        "id='" + cursus.getId() + "'"));
}
@Test
public void createIndividueleCursus() {
    int aantalIndividueleCursussen=super.countRowsInTable(INDIVIDUELE_CURSUSSEN);
    IndividueleCursus cursus = new IndividueleCursus("testIndividueel2", 7);
    repository.create(cursus);
    manager.flush();
    assertEquals(aantalIndividueleCursussen + 1,
        super.countRowsInTable(INDIVIDUELE_CURSUSSEN));
    assertEquals(1, super.countRowsInTableWhere(INDIVIDUELE_CURSUSSEN,
        "id='" + cursus.getId() + "'"));
}
```

- (1) Je hebt op de vorige regel een cursus met een UUID primary key toegevoegd. De EntityManager voert deze toevoegingen niet onmiddellijk uit, maar spaart die op tot juist voor de commit van de transactie. Hij stuurt dan alle opgespaarde toevoegingen in één keer naar de database, via JDBC batch updates. Dit verhoogt de performantie. In deze test is het wel nodig dat de cursus onmiddellijk toegevoegd wordt, zodat we de correcte werking van onze create method kunnen testen. Je voert de flush method uit die gevraagde toevoegingen onmiddellijk uitvoert. Als je een autonumber primary key hebt, kan JPA geen JDBC batch updates gebruiken, omdat je daarbij de gegenereerde primary key waarden niet kan ophalen. JPA voegt dan een record toe zodra je de persist method uitvoert op de EntityManager.

Je voert de tests uit. Ze lukken.





Je commit de sources. Je publiceert op GitHub.



Food en non-food artikels: zie takenbundel

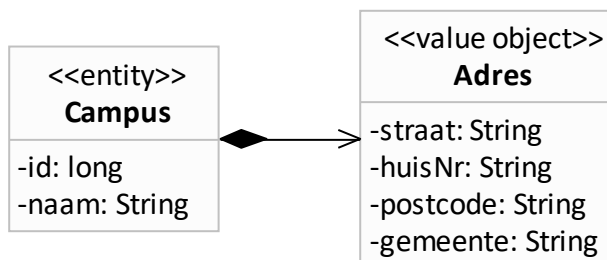
16 VALUE OBJECTS

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

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



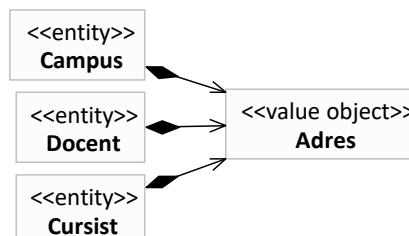
Campus is een entity:

- 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,



16.1 Immutable

Je kan een class schrijven als mutable of immutable:

Mutable	De class bevat methods (bvb. setters) om na de aanmaak van een object van die class attributen te wijzigen.
Immutable	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.

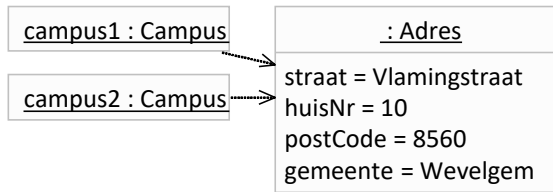
Men raadt aan value objects immutable te maken.

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

```

Adres adres = new Adres("Vlamingstraat", "10", "8560", "Wevelgem")
Campus campus1 = new Campus("Naam campus 1", adres);
// de campussen delen hetzelfde gebouw:
Campus campus2 = new Campus("Naam campus 2", adres);
  
```

campus1 en campus2 verwijzen naar hetzelfde Adres object.

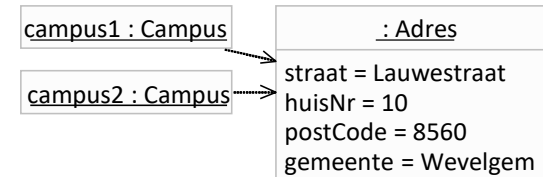


```
campus2.getAdres().setStraat("Lauwestraat"); // campus2 verhuist
```

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

Je besepte 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):

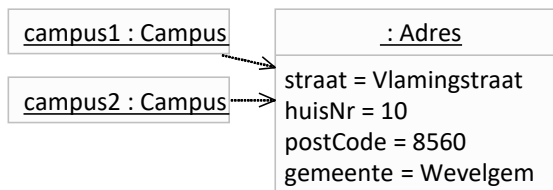
```
Adres adres = new Adres("Vlamingstraat", "10", "8560", "Wevelgem")
```

```
Campus campus1 = new Campus("Naam campus 1", adres);
```

```
// de campussen delen hetzelfde gebouw:
```

```
Campus campus2 = new Campus("Naam campus 2", adres);
```

campus1 en campus2 verwijzen 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 objects:

- 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

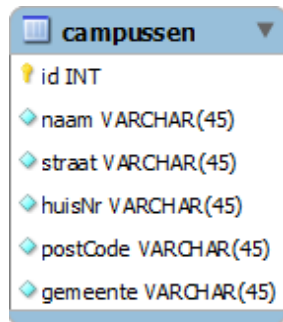
De Java documentatie zegt:

Immutable, arbitrary-precision signed decimal numbers.



16.2 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 opsplittst (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.

16.3 Java

16.3.1 Value object: Adres

Je maakt een package `be.vdab.fietsacademy.valueobjects` en daarin een class `Adres`:

```
package be.vdab.fietsacademy.valueobjects;
// enkele imports ...
@Embeddable ❶
public class Adres implements Serializable {
    private static final long serialVersionUID = 1L;
    private String straat; ❷
    private String huisNr;
    private String postcode;
    private String gemeente;
    // een geparametriseerde constructor en een protected default constructor ❸
    // getters voor straat, huisNr, postcode en gemeente
}
```

- (1) `@Embeddable` staat voor een value object class.
- (2) Je kan bij de variabelen `final` tikken als je geen JPA gebruikt. 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) Een value object class moet bij JPA een default constructor hebben. Voor JPA volstaat het deze constructor `protected` te maken.

16.3.2 Entity die het value object gebruikt: Campus

```
package be.vdab.fietsacademy.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 ❶
    private Adres adres;
    // een geparametriseerde constructor (zonder id parameter)
    // een protected default constructor en getters voor id, naam en adres
}
```

- (1) `@Embedded` staat voor een variabele met als type een value object 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.

16.4 Service layer en repository layer

Value objects 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 in de database opgeslagen als je de entity, die bij het value object hoort, in de database opslaat.

16.5 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.

16.5.1 Create

```
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
```

16.5.2 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();
```

16.5.3 Update

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

16.5.4 Delete

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

16.6 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
```

Je maakt een interface CampusRepository:

```
package be.vdab.fietsacademy.repositories;
// enkele imports
public interface CampusRepository {
    void create(Campus campus);
    Optional<Campus> read(long id);
}
```

Je implementeert deze interface in een class JpaCampusRepository:

```
package be.vdab.fietsacademy.repositories;
// enkele imports
@Repository
class JpaCampusRepository implements CampusRepository {
    @Override
    public void create(Campus campus) {
        throw new UnsupportedOperationException();
    }
}
```

```

@Override
public Optional<Campus> read(long id) {
    throw new UnsupportedOperationException();
}
}

```

Je maakt insertCampus.sql in src/test/resources:

```

insert into campussen(naam,straat,huisNr,postCode,gemeente)
values('test','test','test','test','test');

```

Je maakt tests:

```

package be.vdab.fietsacademy.repositories;
// enkele imports
@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace = Replace.NONE)
@Import(JpaCampusRepository.class)
@Sql("/insertCampus.sql")
public class JpaCampusRepositoryTest
    extends AbstractTransactionalJUnit4SpringContextTests {
    private static final String CAMPUSSEN = "campussen";
    @Autowired
    private JpaCampusRepository repository;
    private Campus campus;
    @Before
    public void before() {
        campus = new Campus("test", new Adres("test", "test", "test", "test"));
    }
    private long idVanTestCampus() {
        return super.jdbcTemplate.queryForObject(
            "select id from campussen where naam='test'", Long.class);
    }
    @Test
    public void read() {
        Campus campus = repository.read(idVanTestCampus()).get();
        assertEquals("test", campus.getNaam());
        assertEquals("test", campus.getAdres().getGemeente());
    }
    @Test
    public void create() {
        int aantalCampussen = super.countRowsInTable(CAMPUSSEN);
        repository.create(campus);
        assertEquals(aantalCampussen + 1, super.countRowsInTable(CAMPUSSEN));
        assertEquals(1, super.countRowsInTableWhere(CAMPUSSEN, "id="+campus.getId()));
    }
}

```

Je voert de tests uit. Ze mislukken.

Je implementeert de methods in JpaCampusRepository:

```

package be.vdab.fietsacademy.repositories;
// enkele imports
@Repository
class JpaCampusRepository implements CampusRepository {
    private final EntityManager manager;
    JpaCampusRepository(EntityManager manager) {
        this.manager = manager;
    }
}

```

```

@Override
public void create(Campus campus) {
    manager.persist(campus);
}

@Override
public Optional<Campus> read(long id) {
    return Optional.ofNullable(manager.find(Campus.class, id));
}
}

```

Je voert de tests uit. Ze lukken.



Je commit de sources. Je publiceert op GitHub.

16.7 Value object classes detecteren

Je detecteert met volgende tip value object classes.

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:

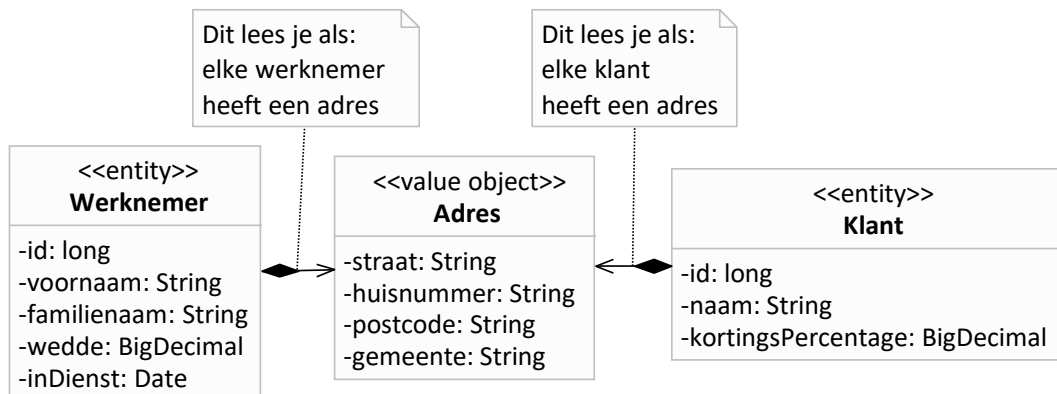
Werknemers	Klanten
id INT	id INT
voornaam VARCHAR(45)	naam VARCHAR(45)
familiennaam VARCHAR(45)	straat VARCHAR(45)
straat VARCHAR(45)	huisnummer VARCHAR(45)
huisnummer VARCHAR(45)	postcode VARCHAR(45)
postcode VARCHAR(45)	gemeente VARCHAR(45)
gemeente VARCHAR(45)	kortingsPercentage DECIMAL(10,2)
wedde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

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

Deze groep heet Adres.

Je maakt een value object class Adres

De Java classes zullen er als volgt uitzien:



16.8 Aggregate

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

In het voorbeeld hierboven zijn er twee aggregates:

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

16.9 Value object als command object

Je kan een value object gebruiken als command object in je website.

De bedoeling is dat de gebruiker de properties van het value object kan intikken.

Bij het submitten van de de form, maakt Spring een nieuw value object met de default constructor. Je value object class moet dan een *public* default constructor hebben.

Spring roept standaard ook setters op van je value object, om de gegevens die gebruiker intikte in te vullen in het value object. Je leerde echter dat setters afgeraden worden in een value object.

Spring biedt een oplossing, waarbij Spring de ingetikte gegevens *rechtstreeks* in de private variabelen van het command object invult en geen setters gebruikt.

Je controller bevat een method die de pagina met de form *toont* aan de gebruiker:

```
@GetMapping
ModelAndView toonForm() {
    return new ModelAndView(EEEN_JSP, new Adres());
}
```

Je controller bevat een method die de submit van de form verwerkt:

```
@PostMapping
ModelAndView verwerkForm(@Valid Adres adres, BindingResult bindingResult) {
    ...
}
```

Je voegt nog een method toe aan de controller:

```
@InitBinder("adres")
void initBinder(DataBinder binder) {
    binder.initDirectFieldAccess();
}
```

- (1) Je tikt `@InitBinder` voor een method.
Je vermeldt als parameter de naam waarmee je het command object aanbiedt aan de JSP.
- (2) De method naam is vrij te kiezen. De method moet een `DataBinder` parameter hebben.
De `DataBinder` is het object dat Spring intern gebruikt om ingetikte gegevens in te vullen in het command object.
- (3) Je voert op dit object de method `initDirectFieldAccess` uit. De `DataBinder` vult dan de ingetikte gegevens *rechtstreeks* in in de private variabelen van het command object.

Als je een entity gebruikt als command object en die entity bevat een value object (zoals Campus een Adres bevat), moet je ook een `initBinder` method voorzien:

```
@InitBinder("campus")
void initBinder(DataBinder binder) {
    binder.initDirectFieldAccess();
}
```

17 VERZAMELING VALUE OBJECTS MET EEN BASISTYPE

De multipliciteit tussen Campus en Adres is één: één Campus heeft één Adres.

Bij meervoudige multipliciteit bevat de entity *een verzameling* value objects.

Deze verzameling bevindt zich in een andere table dan de tables met de entities.

Het type van de value objects in de verzameling kan:

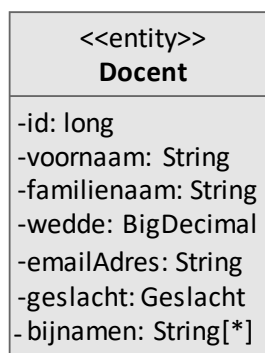
- een basistype zijn:
Byte, Short, Integer, Long, Float, Double, Boolean, Char, String, BigDecimal, Date
- een eigen geschreven class zijn.

Je leert in dit hoofdstuk werken met basistypes.

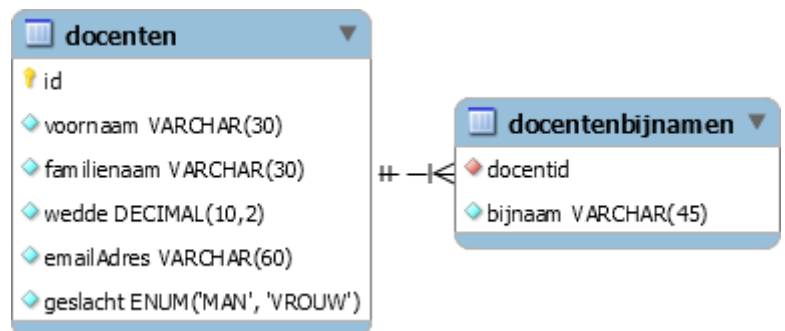
Je leert in het volgend hoofdstuk werken met een eigen geschreven class.

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 objects:

- 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 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) @ElementCollection staat voor een verzameling value objects (⑤).
- (2) @CollectionTable duidt de table naam aan die de value objects bevat.
- (3) @JoinColumn duidt een kolom in deze table aan.
Het is de foreign key kolom die verwijst naar primary kolom in de table (docenten) die hoort bij de huidige entity class (Docent).
Je vult met @JoinColumn de parameter joinColumns van @CollectionTable.
- (4) @Column duidt de kolom naam aan die hoort bij de value objects in de verzameling.
- (5) JPA ondersteunt de types List, Set en Map.

Je voegt een opdracht toe aan de geparametriseerde constructor:

```
this.bijnamen = new LinkedHashSet<>();
```

Je maakt methods die met deze Set samenwerken:

```

public Set<String> getBijnamen() {
    throw new UnsupportedOperationException();
}
  
```

```

public boolean addBijnaam(String bijnaam) {
    throw new UnsupportedOperationException();
}
public boolean removeBijnaam(String bijnaam) {
    throw new UnsupportedOperationException();
}

```

Je maakt tests in DocentTest:

```

@Test
public void eenNieuweDocentHeeftGeenBijnamen() {
    assertTrue(docent1.getBijnamen().isEmpty());
}
@Test
public void bijnaamToevoegen() {
    assertTrue(docent1.addBijnaam("test"));
    assertEquals(1, docent1.getBijnamen().size());
    assertTrue(docent1.getBijnamen().contains("test"));
}
@Test
public void tweeKeerDezelfdeBijnaamToevoegenKanNiet() {
    docent1.addBijnaam("test");
    assertFalse(docent1.addBijnaam("test"));
    assertEquals(1, docent1.getBijnamen().size());
}
@Test(expected = NullPointerException.class)
public void nullAlsBijnaamToevoegenKanNiet() {
    docent1.addBijnaam(null);
}
@Test(expected = IllegalArgumentException.class)
public void eenLegeBijnaamToevoegenKanNiet() {
    docent1.addBijnaam("");
}
@Test(expected = IllegalArgumentException.class)
public void eenBijnaamMetEnkelSpatiesToevoegenKanNiet() {
    docent1.addBijnaam(" ");
}
@Test
public void bijnaamVerwijderen() {
    docent1.addBijnaam("test");
    assertTrue(docent1.removeBijnaam("test"));
    assertTrue(docent1.getBijnamen().isEmpty());
}
@Test
public void eenBijnaamVerwijderenDieJeNietToevoegdeKanNiet() {
    docent1.addBijnaam("test");
    assertFalse(docent1.removeBijnaam("test2"));
    assertEquals(1, docent1.getBijnamen().size());
    assertTrue(docent1.getBijnamen().contains("test"));
}

```

Je voert de tests uit. Ze mislukken.

Je implementeert de methods in Docent:

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;
    return Collections.unmodifiableSet(bijnamen);
}

```

❶
❷

- (1) Je leest met de method `getBijnamen` de bijnamen van een Docent.
 Als je `return bijnamen;` tikt, 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`.

```
public boolean addBijnaam(String bijnaam) {
    if (bijnaam.trim().isEmpty()) {
        throw new IllegalArgumentException();
    }
    return bijnamen.add(bijnaam);
}
public boolean removeBijnaam(String bijnaam) {
    return bijnamen.remove(bijnaam);
}
```

Je voert de tests uit. Ze lukken.

17.1 Verzameling value objects lezen uit de database

De `EntityManager` heeft geen method om de verzameling value objects uit de database te lezen. JPA leest de verzameling value objects 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 objects.

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

17.2 Value object toevoegen aan de verzameling

Als je binnen een transactie value objects 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 objects hoort.

Voorbeeldcodefragment 1:

```
Docent docent = new Docent(...);
docent.addBijnaam(...);
docent.addBijnaam(...);
manager.persist(docent)
```

❶

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

Voorbeeldcodefragment 2:

```
Docent docent = manager.find(Docent.class, 1L);
docent.addBijnaam(...);
```

JPA stuurt op het einde van de transactie één `insert` statement naar de table `docentenbijnamen`.

17.3 Value object verwijderen uit de verzameling

Als je binnen een transactie één of meerdere value objects 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 objects hoort.

Voorbeeldcodefragment:

```
Docent docent = manager.find(Docent.class, 1L);
docent.removeBijnaam(...);
```

JPA stuurt op het einde van de transactie één **delete** statement naar de table docentenbijnamen.

Je voegt een opdracht toe aan insertDocent.sql:

```
insert into docentenbijnamen(docentid,bijnaam)
values ((select id from docenten where voornaam='testM'),'test');
```

Je slaat het bestand op.

Je maakt als voorbeeld twee testen in JpaDocentRepositoryTest:

```
@Test
public void bijnamenLezen() {
    Docent docent = repository.read(idVanTestMan()).get();
    assertEquals(1, docent.getBijnamen().size());
    assertTrue(docent.getBijnamen().contains("test"));
}

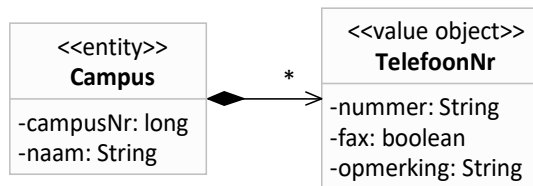
@Test
public void bijnaamToevoegen() {
    repository.create(docent);
    docent.addBijnaam("test");
    manager.flush();
    assertEquals("test", super.jdbcTemplate.queryForObject(
        "select bijnaam from docentenbijnamen where docentid=?", String.class,
        docent.getId()));
}
```



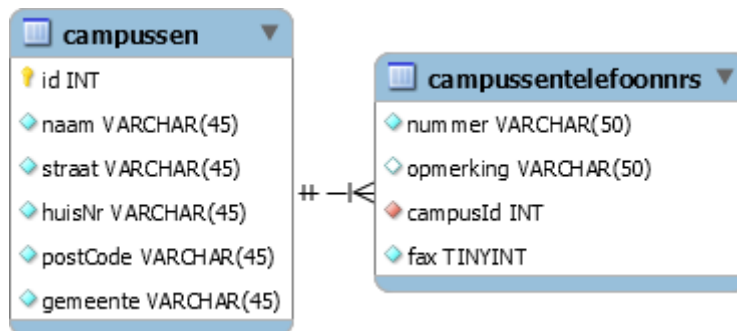
Je commit de sources. Je publiceert op GitHub.

18 VERZAMELING VALUE OBJECTS MET EEN EIGEN TYPE

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



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.

18.1 Value object: TelefoonNr

```

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

- (1) Je stelt straks in Campus de verzameling telefoonnummers 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.

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

Je maakt testen voor equals en hashCode:

```
package be.vdab.fietsacademy.valueobjects;
// enkele imports
public class TelefoonNrTest {
    private TelefoonNr nummer1, nogEensNummer1, nummer2;
    @Before
    public void before() {
        nummer1 = new TelefoonNr("1", false, "");
        nogEensNummer1 = new TelefoonNr("1", false, "");
        nummer2 = new TelefoonNr("2", false, "");
    }
    @Test
    public void telefoonNrsZijnGelijkAlsHunNummersGelijkZijn() {
        assertEquals(nummer1, nogEensNummer1);
    }
    @Test
    public void telefoonNrsZijnVerschillendAlsHunNummersVerschillen() {
        assertNotEquals(nummer1, nummer2);
    }
    @Test
    public void eenTelefoonNrVerschiltVanNull() {
        assertNotEquals(nummer1, null);
    }
    @Test
    public void eenTelefoonNrVerschiltVanEenAnderTypeObject() {
        assertNotEquals(nummer1, "");
    }
    @Test
    public void gelijkeTelefoonNrsGevenDezelfdeHashCode() {
        assertEquals(nummer1.hashCode(), nogEensNummer1.hashCode());
    }
}
```

Je voert de tests uit. Ze lukken.

18.2 Entity met verzameling value objects: Campus

Je drukt de composition uit met extra code in Campus:

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

①
②
③
④

- (1) @ElementCollection staat bij een variabele met een verzameling value objects.
- (2) @CollectionTable duidt de naam van de table aan die de value objects bevat.
- (3) @JoinColumn duidt een kolom in deze table aan.
Het is de foreign key kolom die verwijst naar primary kolom in de table (campussen) die hoort bij de huidige entity class (Campus).
Je vult met @JoinColumn de parameter joinColumns van @CollectionTable.
- (4) @OrderBy definieert de volgorde waarmee JPA de value objects leest 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 opdracht toe aan de geparametriseerde constructor:

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

Je maakt een method:

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

Je zou natuurlijk ook een method `add(TelefoonNr telefoonNr)` en een method `remove(TelefoonNr telefoonNr)` kunnen toevoegen.

Je zou ook tests schrijven voor deze methods.

Deze zijn gelijkaardig aan de tests die je maakte in `DocentTest` in verband met de bijnamen.

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 `Campus`.

18.3 Verzameling value objects lezen uit de database

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

Je voegt een opdracht toe aan `insertCampus.sql`:

```
insert into campussentelefoonnr(campusid,nummer,fax,opmerking)
values((select id from campussen where naam='test'),'1',false,'test');
```

Je maakt een test in `JpaCampusRepositoryTest`:

```
@Test
public void telefoonNrsLezen() {
    Campus campus = repository.read(idVanTestCampus()).get();
    assertEquals(1, campus.getTelefoonNrs().size());
    assertTrue(campus.getTelefoonNrs().contains(
        new TelefoonNr("1", false, "test")));
}
```

18.4 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 `campussentelefoonnr`.

18.5 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 `campussentelefoonnr`.

18.6 Entity verwijderen

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

- Als je een docent verwijdert, verwijdert JPA de bijbehorende records in `docentenbijnamen`.
- Als je een campus verwijdert, verwijdert JPA de bijbehorende records in `campussentelefoonnr`.

18.7 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 objects:

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 objects die behoren tot een entity die hoort bij table A.



Je commit de sources. Je publiceert op GitHub.



Kortingen: zie takenbundel

19 MANY-TO-ONE ASSOCIATIE 🧑 → 🧑

Je leert vanaf dit hoofdstuk werken met associaties tussen entiteiten.

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

Je maakt in Docent een many-to-one associatie naar 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 wordt de associatie bidirectioneel, 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.

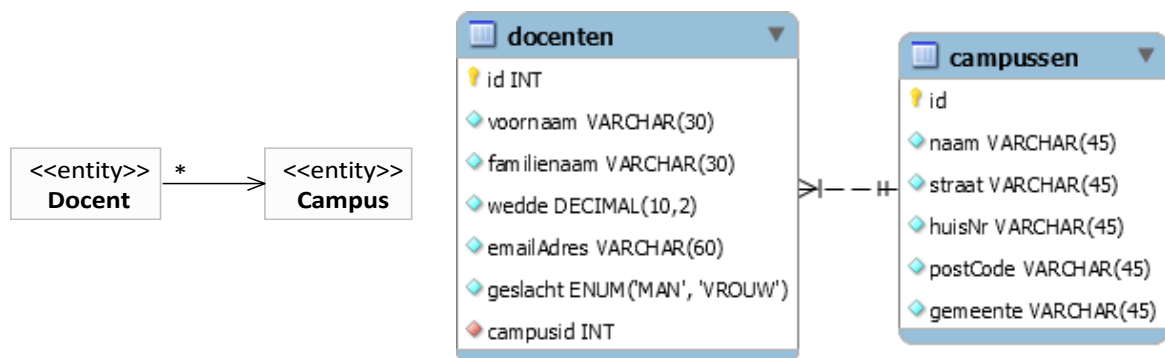
19.1 Database

Je voert het script `CampusBijDocent.sql` uit.

Dit maakt in de table docenten een kolom `campusid`.

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 tables:

1. Je klappt in de Data Source Explorer database niveau's open tot je de map Tables ziet.
2. Je klikt met de rechtermuisknop op Tables en je kiest Refresh.

19.2 Java

Je definieert de associatie in 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 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;
// een getter en een setter voor campus
// vooraan in de setter: if (campus==null) throw new NullPointerException();
// een extra Campus parameter bij de geparametriseerde constructor
// je voegt setCampus(campus) toe aan de code in deze constructor
```

①
②

- (1) @ManyToOne staat 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.
@JoinColumn duidt 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 Docent en Campus.

Je maakt een variabele in DocentTest:

```
private Campus campus1;
```

Je wijzigt de method before:

```
@Before
```

```
public void before() {
    campus1 = new Campus("test", new Adres("test", "test", "test", "test"));
    docent1 = new Docent("test", "test", ORIGINELE_WEDDE, "test@fietsacademy.be",
        Geslacht.MAN, campus1);
}
```

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

1. Je leest het Campus object dat je met het Docent object wil associëren.
2. Je maakt een nieuw Docent object en je geeft het Campus object mee als parameter aan de geparametriseerde constructor.
3. Of je leest een Docent object uit de database en 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 maakt insertCampus.sql:

```
insert into campussen(naam,straat,huisNr,postCode,gemeente)
values('test','test','test','test','test');
```

Je wijzigt insertDocent.sql:

```
insert into docenten(voornaam,familienaam,wedde,emailadres,geslacht,campusid)
values ('testM','testM',1000,'testM@fietsacademy.be','MAN',
(select id from campussen where naam='test'));
insert into docenten(voornaam,familienaam,wedde,emailadres,geslacht,campusid)
values ('testV','testV',1000,'testV@fietsacademy.be','VROUW',
(select id from campussen where naam='test'));
insert into docentenbijnamen(docentid,bijnaam)
values ((select id from docenten where voornaam='testM'),'test');
```

Je voegt een regel toe aan JpaDocentRepositoryTest, voor @Sql("/insertDocent.sql"):

```
@Sql("/insertCampus.sql")
```

Je maakt een variabele in JpaDocentRepositoryTest:

```
private Campus campus;
```

Je wijzigt de method before:

```
@Before
```

```
public void before() {
    campus = new Campus("test", new Adres("test", "test", "test", "test"));
    docent = new Docent("test", "test", BigDecimal.TEN, "test@fietsacademy.be",
        Geslacht.MAN, campus);
}
```

Je voegt vooraan in de method create een opdracht toe:

```
manager.persist(campus);
```

Je voegt achteraan in de method create een opdracht toe:

```
assertEquals(campus.getId(), super.jdbcTemplate.queryForObject(
    "select campusid from docenten where id=?", Long.class, docent.getId())
    .longValue());
```

Je voegt vooraan in de method bijnaamToevoegen een opdracht toe:

```
manager.persist(campus);
```

Je kan de test uitvoeren.

Je vervangt de eerste opdracht in de method before in DefaultDocentServiceTest:

```
Campus campus = new Campus("test", new Adres("test", "test", "test", "test"));
docent = new Docent("test", "test", BigDecimal.valueOf(100),
    "test@fietsacademy.be", Geslacht.MAN, campus);
```

Je voegt een regel toe aan DefaultDocentServiceIntegrationTest,
voor @Sql("/insertDocent.sql"):

```
@Sql("/insertCampus.sql")
```

19.3 Eager loading

JPA gebruikt op een many-to-one associatie default eager loading:

als JPA een Docent entity leest, leest JPA tegelijk de bijbehorende Campus entity.

Je ziet dit als je enkel de method read uitvoert van JpaDocentRepositoryTest.

Je ziet in het venster Console dat JPA in zijn select statement

een record uit de table docenten én het bijbehorend record uit de table campussen leest:

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

Je vraagt enkel de docent te lezen, maar via de inner join leest JPA ook de bijbehorende campus.

Het is niet zo dat je in je applicatie de method read uitvoert van JpaDocentRepository
en altijd de campus data van die docent nodig hebt. Toch wordt die altijd gelezen.

Dit benadeelt de performantie.

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

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

Je ziet dit als je enkel de method findBijWeddeBetween uitvoert van JpaDocentRepositoryTest.

Je ziet in het venster Console de 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.

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

19.4 Lazy loading

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

Dit betekent: als JPA een Docent leest, leest JPA niet onmiddellijk de bijbehorende Campus.

Pas als je in Java code of in een JSP de Campus aanspreekt, leest JPA die Campus 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)
```


Als je nu de methods `read` of `findByWeddeBetween` van `JpaDocentRepositoryTest` uitvoert, stuurt JPA enkel een `select` statement naar de table `docenten`.

Je maakt een test in `JpaDocentRepositoryTest` die aantoont dat lazy loading werkt:

```
@Test
public void campusLazyLoaded() {
    Docent docent = repository.read(idVanTestMan()).get();
    assertEquals("test", docent.getCampus().getNaam());
}
```

①
②

- (1) JPA leest enkel een record uit de table `docenten`.
- (2) Je spreekt de campus aan die bij de docent hoort.
JPA leest nu het bijbehorende record uit de table `campussen`.



Je commit de sources. Je publiceert op GitHub.



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.

20 ONE-TO-MANY ASSOCIATIE →

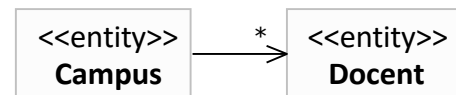
20.1 Database

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.

20.2 Java

Je maakt in Campus een one-to-many associatie naar 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 Docent in commentaar:

- Je plaatst de variabele campus en bijbehorende annotations in commentaar.
- Je plaatst de methods getCampus en setCampus in commentaar.
- Je plaatst in de geparametriseerde constructor de parameter campus in commentaar.
- Je plaatst in die constructor de opdracht setCampus(campus) in commentaar.

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

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 boolean add(Docent docent) {
    if (docent == null) {
        throw new NullPointerException();
    }
    return docenten.add(docent);
}
  
```

❶
❷
❸
❹

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

Je voegt een opdracht toe aan de geparametriseerde constructor:

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

20.3 Testen corrigeren

Je roept in DocentTest de geparametriseerde Docent constructor op.

Je plaatst de parameterwaarde campus1 in commentaar.

Je roept in de JpaDocentRepositoryTest method before de geparametriseerde Docent constructor op. Je plaatst de parameterwaarde campus in commentaar.

Je voegt in de method before achteraan een opdracht toe:

```
campus.add(docent);
```

Je plaatst de method campusLazyLoaded in commentaar.

Je voegt in de method create een opdracht toe na repository.create(docent);:

```
manager.flush();
```

❶

- (1) Bij een gerichte one-to-many associatie vult JPA de kolom campusid in het nieuw record in de table docenten in door een update statement naar deze table te sturen. JPA verstuurt dit statement normaal op het einde van de transactie. Met deze regel verstuurt JPA dit statement onmiddellijk.

Je roept in DefaultDocentServiceTest de geparametriseerde Docent constructor op.

Je plaatst de parameterwaarde campus in commentaar.

20.4 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 op een object meerdere keren hashCode uitvoert, moet die telkens hetzelfde getal teruggeven. In de Java documentatie over hashCode: *whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer*



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.

1. Je maakt een nieuwe docent. Zijn id bevat 0.
2. Je associeert de docent met een campus: campus.add(docent). Je voegt daarbij een reference naar die docent toe aan de Set<Docent> in de campus. Deze Set roept hashCode op bij het toevoegen van de docent aan de Set. Deze hashCode geeft 0 terug.
3. Je slaat de docent op in de database. JPA vult zijn id met de autonumber in het nieuw record. Als je de docent nu terug opzoekt in de Set, roept die Set de method hashCode op van die docent. Deze method geeft nu een getal terug dat verschilt van de oorspronkelijke waarde 0 (bij het toevoegen van de docent). De Set vindt de docent niet meer terug in zichzelf.

Je toont dit aan met een unit test.

Je maakt eerst in Docent de methods equals en hashCode, verkeerdelijk gebaseerd op id:

```
@Override
```

```
public boolean equals(Object object) {
    if (!(object instanceof Docent)) {
        return false;
    }
    Docent docent = (Docent) object;
    return this.id == docent.id;
}
```

```
@Override
public int hashCode() {
    return (int) id;
}
```

Je voegt een opdracht toe als laatste in de method `create` van `JpaDocentRepositoryTest`:

```
assertTrue(campus.getDocenten().contains(docent));
```

Je voert de test uit. Hij mislukt.

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 toont dit aan met een unit test.

Je maakt een variabele in `DocentTest`:

```
private Docent docent2;
```

Je voegt een opdracht toe als laatste in de method `before`:

```
docent2 = new Docent("test2", "test2", ORIGINELE_WEDDE, "test2@fietsacademy.be",
    Geslacht.MAN);
```

Je maakt een test:

```
@Test
public void meerdereDocentenKunnenTotDezelfdeCampusBehoren() {
    assertTrue(campus1.add(docent1));
    assertTrue(campus1.add(docent2));
}
```

Je voert de test uit. Hij mislukt op de 2^e `assertTrue`: er is al eenzelfde docent aanwezig in de `Set` in `Campus`. Java beslist dit op basis van de `equals` method in `Docent`.

Die geeft aan dat `docent2` gelijk is aan `docent1`: ze hebben beiden `0` als `id`.

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 `emailAdres` is ideaal: elke docent heeft een uniek email adres.

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

Je corrigeert in `Docent` de methods `equals` en `hashCode`:

```
@Override
public boolean equals(Object obj) {
    if ( ! (obj instanceof Docent)) {
        return false;
    }
    if (emailAdres == null) {
        return false;
    }
    return emailAdres.equalsIgnoreCase(((Docent) obj).emailAdres);
}
@Override
public int hashCode() {
    return emailAdres == null ? 0 : emailAdres.toLowerCase().hashCode();
}
```

(1) Je maakt bij de vergelijking geen onderscheid tussen kleine en hoofdletters.

(2) Je maakt geen onderscheid tussen kleine en hoofdletters.

Je voert in JpaDocentRepositoryTest de method create opnieuw uit. Hij lukt.

Je voert in DocentTest meerdereDocentenKunnenTotDezelfdeCampusBehoren opnieuw uit. Hij lukt.

Je voegt code toe aan DocentTest, om equals en hashCode ten gronde te testen.

Je voegt een variabele toe:

```
private Docent nogEensDocent1;
```

Je voegt een opdracht toe aan before:

```
nogEensDocent1 = new Docent("test", "test", ORIGINELE_WEDDE,
    "test@fietsacademy.be", Geslacht.MAN);
```

Je maakt tests:

```
@Test
public void docentenZijnGelijkAlsHunEmailAdressenGelijkZijn() {
    assertEquals(docent1, nogEensDocent1);
}

@Test
public void docentenZijnVerschillendAlsHunEmailAdressenVerschillen() {
    assertNotEquals(docent1, docent2);
}

@Test
public void eenDocentVerschiltVanNull() {
    assertNotEquals(docent1, null);
}

@Test
public void eenDocentVerschiltVanEenAnderTypeObject() {
    assertNotEquals(docent1, "");
}

@Test
public void gelijkeDocentenGevenDezelfdeHashCode() {
    assertEquals(docent1.hashCode(), nogEensDocent1.hashCode());
}
```

Je voert de testen uit. Ze lukken.

20.5 Equals en hashCode laten genereren

De methods equals en hashCode zelf uitschrijven vraagt tijd, zeker als deze methods gebaseerd zijn op *meerdere* private variabelen. Dit is het geval in Adres: Adres objecten verwijzen naar hetzelfde adres als hun straat, huisnummer, postcode én gemeente gelijk zijn.

Je kan tijd sparen door deze methods met 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 variabelen aangevinkt waarop de methods equals en hashCode 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 toevoegt.
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 JPA intern soms met afgeleide classes van entities en value objects werkt, is dit aanvinken belangrijk.
6. Je klikt op OK.



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

Je kan soms de code, die Eclipse genereert, verfijnen.

Je maakt in Campus een equals en hashCode gebaseerd op naam.

De gegeneerde code maakt een verschil tussen een naam in kleine letters en een naam in hoofdletters. In de werkelijkheid wordt echter aanzien dat een Campus object met de naam test gelijk is aan een Campus object met de naam Test.

Je wijzigt daarom in de equals method equals naar equalsIgnoreCase.

Je tikt in de hashCode method toUpperCase(). voor hashCode.

20.6 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.

Je toont lazy loading aan met een test in JpaCampusRepositoryTest.

Je voegt een regel toe na @Sql("/insertCampus.sql"):

@Sql("/insertDocent.sql")

Je voegt een test toe:

@Test

```
public void docentenLazyLoaded() {
    Campus campus = repository.read(idVanTestCampus()).get();
    assertEquals(1, campus.getDocenten().size());
    assertEquals("test",
        campus.getDocenten().stream().findFirst().get().getVoornaam());
}
```

Je voert de test uit. Je ziet in het venster Console dat JPA bij ❶ eerst enkel een record uit de table campussen leest. JPA leest pas bij ❷ de records uit de table docenten die bij de campus horen.

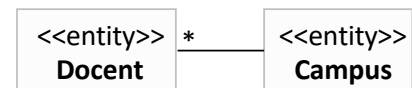


Je commit de sources. Je publiceert op GitHub.

21 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



21.1 Docent (many kant)

Je haalt in Docent de private variabele `campus`, de bijbehorende JPA annotations, de bijbehorende getter en de bijbehorende setter uit commentaar. Je haalt in de constructor de parameter `campus` en de opdracht `setCampus(campus)` 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;
  
```

21.2 One kant: Campus

De JPA annotations wijzigen in Campus:

```

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

①
②

- (1) `@OneToMany` bevat nu de parameter `mappedBy`.
Deze bevat 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 `Docent` (de entity class aan de many-kant van de associatie).

21.3 Testen corrigeren

Je roept in `DocentTest` de geparametriseerde `Docent` constructor op.

Je haalt de parameterwaarde `campus1` uit commentaar.

Je voegt de parameterwaarde `campus1` toe aan de andere constructor oproepen.

Je corrigeert de method `meerdereDocentenKunnenTotDezelfdeCampusBehoren`:

```

@Test
public void meerdereDocentenKunnenTotDezelfdeCampusBehoren() {
    assertTrue(campus1.getDocenten().contains(docent1));
    assertTrue(campus1.getDocenten().contains(docent2));
}
  
```

Je roept in `JpaDocentRepositoryTest` de geparametriseerde `Docent` constructor op.

Je haalt de parameterwaarde `campus` uit commentaar.

Je verwijdert in de method `before` de opdracht `campus.add(docent)`;

Je haalt de method `campusLazyLoaded` uit commentaar.

Je roept in `DefaultDocentServiceTest` de geparametriseerde `Docent` constructor op.

Je haalt de parameterwaarde `campus` uit commentaar.

Je moet nu de testen nog niet uitvoeren, er is extra code nodig (volgende pagina's).

21.4 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.

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

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

Je kan op twee manieren een Docent object met een Campus object associëren:

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

Je wijzigt in Docent de method `setCampus`:

```
public void setCampus(Campus campus) {
    if (campus == null) {
        throw new NullPointerException();
    }
    if (!campus.getDocenten().contains(this)) {
        campus.add(this);
    }
    this.campus = campus;
}
```

Je wijzigt in Campus de method `add(Docent docent)`:

```
public boolean add(Docent docent) {
    if (docent == null) {
        throw new NullPointerException();
    }
    boolean toegevoegd = docenten.add(docent);
    Campus oudeCampus = docent.getCampus();
    if (oudeCampus != null && oudeCampus != this) {
        oudeCampus.docenten.remove(docent);
    }
    if (this != oudeCampus) {
        docent.setCampus(this);
    }
    return toegevoegd;
}
```

Je test dit in `DocentTest` uit vanuit het standpunt van `Docent`.

Je voegt een variabele toe :

```
private Campus campus2;
```

Je voegt een regel toe aan `before`:

```
campus2 = new Campus("test2", new Adres("test2", "test2", "test2", "test2"));
```

Je maakt tests:

```
@Test
public void docent1KomtVoorInCampus1() {
    assertEquals(docent1.getCampus(), campus1);
    assertEquals(2, campus1.getDocenten().size());
    assertTrue(campus1.getDocenten().contains(docent1));
}

@Test
public void docent1VerhuistNaarCampus2() {
    docent1.setCampus(campus2);
    assertEquals(docent1.getCampus(), campus2);
}
```



```

    assertEquals(1, campus1.getDocenten().size());
    assertEquals(1, campus2.getDocenten().size());
    assertTrue(campus2.getDocenten().contains(docent1));
}

```

Je voert de tests uit. Ze lukken.

Je test de functionaliteit in een nieuwe class CampusTest uit vanuit het standpunt van Campus.

```

package be.vdab.fietsacademy.entities;
// enkele imports
public class CampusTest {
    private Docent docent1;
    private Campus campus1;
    private Campus campus2;

    @Before
    public void before() {
        campus1 = new Campus("test", new Adres("test", "test", "test", "test"));
        campus2 = new Campus("test2", new Adres("test2", "test2", "test2", "test2"));
        docent1 = new Docent("test", "test", BigDecimal.TEN, "test@fietsacademy.be",
            Geslacht.MAN, campus1);
    }

    @Test
    public void campus1IsDeCampusVanDocent1() {
        assertEquals(campus1, docent1.getCampus());
        assertEquals(1, campus1.getDocenten().size());
        assertTrue(campus1.getDocenten().contains(docent1));
    }

    @Test
    public void docent1VerhuistNaarCampus2() {
        assertTrue(campus2.add(docent1));
        assertTrue(campus1.getDocenten().isEmpty());
        assertEquals(1, campus2.getDocenten().size());
        assertTrue(campus2.getDocenten().contains(docent1));
        assertEquals(campus2, docent1.getCampus());
    }
}

```

Je voert de tests uit. Ze lukken.

Bestaande testen tonen aan dat de associatie *bidirectioneel* is bij het lezen uit de database:

- campusLazyLoaded uit JpaDocentRepositoryTest toont aan dat je een docent kan lezen uit de database en daarna de bijbehorende campus kan lezen.
- docentenLazyLoaded uit JpaCampusRepositoryTest toont aan dat je een campus kan lezen uit de database en daarna de bijbehorende docenten kan lezen.



Je commit de sources. Je publiceert op GitHub.



Artikelgroepen: zie takenbundel

22 MANY-TO-MANY ASSOCIATIE 🧑 - 🧑

22.1 Database

In een database kunnen twee tables geen directe veel-op-veel relatie hebben.

De 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



22.2 Java

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



Je drukt deze bidirectionele associatie uit

- met een `Set<Verantwoordelijkheid>` variabele in `Docent`.
Elk `Docent` object heeft zo een verzameling references naar de bijbehorende `Verantwoordelijkheid` objecten.
- met een `Set<Docent>` variabele in `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.fietsacademy.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;
    private String naam;
    // een geparametriseerde constructor met enkel een naam parameter
    // een protected default constructor
    // getters voor id en naam
    // equals en hashCode gebaseerd op naam
```

```

    public boolean add(Docent docent) {
        throw new UnsupportedOperationException();
    }
    public boolean remove(Docent docent) {
        throw new UnsupportedOperationException();
    }
    public Set<Docent> getDocenten() {
        throw new UnsupportedOperationException();
    }
}

```

Je voegt code toe aan Docent:

```

    public boolean add(Verantwoordelijkheid verantwoordelijkheid) {
        throw new UnsupportedOperationException();
    }
    public boolean remove(Verantwoordelijkheid verantwoordelijkheid) {
        throw new UnsupportedOperationException();
    }
    public Set<Verantwoordelijkheid> getVerantwoordelijkheden() {
        throw new UnsupportedOperationException();
    }
}

```

Je maakt een variabele in DocentTest:

```
private Verantwoordelijkheid verantwoordelijkheid1;
```

Je voegt een opdracht toe aan before:

```
verantwoordelijkheid1 = new Verantwoordelijkheid("EHBO");
```

Je maakt tests:

```

@Test
public void verantwoordelijkheidToevoegen() {
    assertTrue(docent1.getVerantwoordelijkheden().isEmpty());
    assertTrue(docent1.add(verantwoordelijkheid1));
    assertEquals(1, docent1.getVerantwoordelijkheden().size());
    assertTrue(
        docent1.getVerantwoordelijkheden().contains(verantwoordelijkheid1));
    assertEquals(1, verantwoordelijkheid1.getDocenten().size());
    assertTrue(verantwoordelijkheid1.getDocenten().contains(docent1));
}

@Test
public void verantwoordelijkheidVerwijderen() {
    assertTrue(docent1.add(verantwoordelijkheid1));
    assertTrue(docent1.remove(verantwoordelijkheid1));
    assertTrue(docent1.getVerantwoordelijkheden().isEmpty());
    assertTrue(verantwoordelijkheid1.getDocenten().isEmpty());
}

```

Je voert de tests uit. Ze mislukken.

Je maakt een class VerantwoordelijkheidTest:

```

package be.vdab.fietsacademy.entities;
// enkele imports
public class VerantwoordelijkheidTest {
    private Verantwoordelijkheid verantwoordelijkheid1;
    private Docent docent1;
    private Campus campus1;

    @Before
    public void before() {
        verantwoordelijkheid1 = new Verantwoordelijkheid("EHBO");
        campus1 = new Campus("test", new Adres("test", "test", "test", "test"));
    }
}

```

```

        docent1 = new Docent("test", "test", BigDecimal.TEN, "test@fietsacademy.be",
            Geslacht.MAN, campus1);
    }

    @Test
    public void docentToevoegen() {
        assertTrue(verantwoordelijkheid1.getDocenten().isEmpty());
        assertTrue(verantwoordelijkheid1.add(docent1));
        assertEquals(1, verantwoordelijkheid1.getDocenten().size());
        assertTrue(verantwoordelijkheid1.getDocenten().contains(docent1));
        assertEquals(1, docent1.getVerantwoordelijkheden().size());
        assertTrue(
            docent1.getVerantwoordelijkheden().contains(verantwoordelijkheid1));
    }

    @Test
    public void docentVerwijderen() {
        assertTrue(verantwoordelijkheid1.add(docent1));
        assertTrue(verantwoordelijkheid1.remove(docent1));
        assertTrue(verantwoordelijkheid1.getDocenten().isEmpty());
        assertTrue(docent1.getVerantwoordelijkheden().isEmpty());
    }
}

```

Je implementeert de class Verantwoordelijkheid. Je voegt code toe en corrigeert methods:

```

@ManyToMany
@JoinTable(
    name = "docentenverantwoordelijkheden",
    joinColumns = @JoinColumn(name = "verantwoordelijkheidid"),
    inverseJoinColumns = @JoinColumn(name = "docentid"))
private Set<Docent> docenten = new LinkedHashSet<>();

public boolean add(Docent docent) {
    boolean toegevoegd = docenten.add(docent);
    if ( ! docent.getVerantwoordelijkheden().contains(this)) {
        docent.add(this);
    }
    return toegevoegd;
}

public boolean remove(Docent docent) {
    boolean verwijderd = docenten.remove(docent);
    if (docent.getVerantwoordelijkheden().contains(this)) {
        docent.remove(this);
    }
    return verwijderd;
}

public Set<Docent> getDocenten() {
    return Collections.unmodifiableSet(docenten);
}

```

①
②
③
④
⑤

- (1) @ManyToMany staat bij een variabele die een many-to-many associatie voorstelt.
- (2) @JoinTable duidt de tussentable aan die hoort bij associatie.
- (3) name bevat de naam van de tussentable.
- (4) joinColumns bevat 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.
- (5) inverseJoinColumns bevat 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 implementeert de class Docent. Je voegt code toe en corrigeert methods:

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

1
2

- (1) @ManyToMany staat bij een variabele die een many-to-many associatie voorstelt.
- (2) mappedBy bevat 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 Verantwoordelijkheid en zijn associatie met Docent.

Je voert de tests uit. Ze lukken.

22.3 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 tikt.

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")
```

22.4 Repository testen

Je maakt insertVerantwoordelijkheid.sql:

```
insert into verantwoordelijkheden(naam) values ('test');
```

Je maakt insertDocentVerantwoordelijkheid.sql:

```
insert into docentenverantwoordelijkheden(docentid,verantwoordelijkheidid)
values ((select id from docenten where voornaam='testM'),
(select id from verantwoordelijkheden where naam='test'));
```

Je slaat de bestanden op.

Je voegt een regel toe in JpaDocentRepositoryTest, na @Sql("/insertCampus.sql"):

```
@Sql("/insertVerantwoordelijkheid.sql")
```

Je voegt een regel toe na `@Sql("/insertDocent.sql")`:

`@Sql("/insertDocentVerantwoordelijkheid.sql")`

Je maakt tests:

`@Test`

```
public void verantwoordelijkhedenLezen() {
    Docent docent = repository.read(idVanTestMan()).get();
    assertEquals(1, docent.getVerantwoordelijkheden().size());
    assertTrue(docent.getVerantwoordelijkheden().contains(
        new Verantwoordelijkheid("test")));
}
```

`@Test`

```
public void verantwoordelijkheidToevoegen() {
    Verantwoordelijkheid verantwoordelijkheid = new Verantwoordelijkheid("test2");
    manager.persist(verantwoordelijkheid);
    manager.persist(campus);
    repository.create(docent);
    docent.add(verantwoordelijkheid);
    manager.flush();
    assertEquals(verantwoordelijkheid.getId(),
        super.jdbcTemplate.queryForObject(
            "select verantwoordelijkheidid from docentenverantwoordelijkheden " +
            "where docentid=?", Long.class, docent.getId()).longValue());
}
```

Je voert de tests uit. Ze lukken.

Normaal maak je nog een interface `VerantwoordelijkheidRepository` met de methods `read` en `create`.

Je implementeert deze interface in een class `JpaVerantwoordelijkheidRepository`.

Je test deze methods in een class `JpaVerantwoordelijkheidRepositoryTest`.

Deze class bevat ook test methods `docentenLezen` en `docentenToevoegen`.

Je zou dit kunnen maken met de kennis die je al opgedaan hebt.



Je commit de sources. Je publiceert op GitHub.

23 ASSOCIATIES VAN VALUE OBJECTS NAAR ENTITIES →

Je hebt geleerd hoe je associaties definieert van entities naar entities.

Je kan ook gerichte associaties definiëren van value objects naar entities.

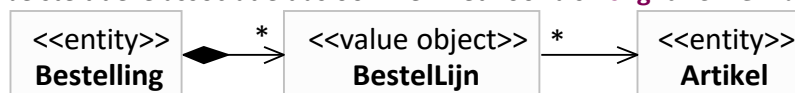
Je gebruikt bij de variabelen in de value object class, die verwijzen naar de entities, de JPA annotation `@ManyToOne` en `@ManyToOne`.

In het volgende voorbeeld bevat `BestelLijn` de regels



```
@ManyToOne(...)
```

```
private Artikel artikel;
```

Je stelt deze associatie dus ook hier niet voor als `long artikelid`;



Belangrijke beperkingen in JPA

- De associatie tussen een entity en zijn value objects is een gerichte associatie () , geen bidirectionele associatie.
Je kan dus in `BestelLijn` geen private variabele `Bestelling` toevoegen en er JPA annotations voor schrijven.
- De associatie tussen een value object en een andere entity is een gerichte associatie () , geen bidirectionele associatie.
Je kan dus in `Artikel` geen private variabele `List<BestelLijn>` 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.

24 N + 1 PROBLEEM, JOIN FETCH QUERIES, ENTITY GRAPHS

24.1 N + 1 probleem



Het N + 1 probleem is een veel voorkomend performantieprobleem in JPA applicaties.

Voorbeeld om het probleem te leren kennen: je toont de docenten van-tot wedde.

Je roept daarbij de method `findByWeddeBetween` op van `JpaDocenRepository`.

Je beslist in de JSP naast de naam en wedde van de docent ook zijn campusnaam te tonen.

Je simuleert dit met extra code in de method `findByWeddeBetween` in `JpaDocentRepositoryTest` na de opdracht `assertTrue(docent.getWedde().compareTo(tweeduizend) <= 0);`:

```
System.out.println(docent.getFamiliennaam() + ':' + docent.getWedde() + ' ' +
    docent.getCampus().getNaam());
```

Je voert de test uit.

Je ziet in het 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.

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 statement uit telkens je voor het eerst een bepaalde campus aanspreekt.

Je ziet wel het probleem, maar JUnit vindt de test OK: de balk in het venster JUnit is groen.

Het is nochtans de bedoeling dat de balk rood wordt bij een probleem.

Je voegt daarom volgende opdracht toe in de test, na `List<Docent> docenten = repository.findByWeddeBetween (duizend, tweeduizend);`

```
manager.clear();
```

❶

- (1) De `clear` opdracht koppelt de gelezen `Docent` entities los van de `EntityManager` waarmee ze gelezen zijn. Als je daarna `Campus` entities aanspreekt die geassocieerd zijn met deze `Docent` entities, kan de `EntityManager` deze niet meer uit de database lezen en werpt een exception.

Je voert de test uit. Hij mislukt.

24.2 Join fetch

Als je zelf het SQL statement tikt 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 1° lijn in de named query `Docent.findByWeddeBetween`

```
select d from Docent d join fetch d.campus
```

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 test opnieuw uit. Hij lukt.

JPA vertaalt de named query naar een select statement. Je ziet in het venster Console 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.

24.3 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 kan dat je in een andere use case ook de docenten toont met een wedde tussen twee grenzen, maar zonder de bijbehorende campus data. Als je dan de named query `Docent.findByWeddeBetween` gebruikt krijg je een trage use case: die query leest ook campussen. Je maakt daarom voor die use case 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 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 die Campus entities geassocieerde Manager entities zouden hebben, mag je dit volgens de JPA standaard niet uitdrukken als
`select d from Docent d join fetch d.campus c join fetch c.manager ...`

Je drukt bij een Entity graph de join niet uit in de named query zelf.

Je zal dit uitdrukken bij de oproep van de named query.

Je mag in `orm.xml` de 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`

24.3.1 @NamedEntityGraph

Je tikt `@NamedEntityGraph` voor de entity class die de behoefte heeft om niet enkel die entity te lezen, maar ook een geassocieerde entity.

Je tikt 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.
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 zo 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.

JPA past die behoefte niet automatisch toe op elke query die docenten leest.

JPA zal dit pas doen als jij dit vraagt, bij het oproepen van een named query.

24.3.2 De oproep van de named query

Je tikt in de `JpaDocentRepository` method `findByWeddeBetween` voor `.getResultList()`:

```
.setHint("javax.persistence.loadgraph",           ❶
    manager.createEntityGraph("Docent.metCampus")) ❷
```

- (1) Je geeft een hint aan JPA. JPA past deze hint toe bij het uitvoeren van de query. Elke hint heeft een naam. De hint om bij het uitvoeren van een query rekening te houden met de behoefte beschreven met `@NamedEntityGraph` is `javax.persistence.loadgraph`.
- (2) Je specificeert de named entity graph die je (met `@NamedEntityGraph`) definieerde onder de naam `Docent.metCampus`. JPA ziet 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 test uit en je ziet in het venster Console dat het N + 1 probleem vermeden is.

24.3.3 @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(...)
})
```

24.3.4 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 entiteiten 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 entiteiten te lezen:

```
@NamedEntityGraph(name = "Docent.metCampusEnVerantwoordelijkheden",
    attributeNodes = {
        @NamedAttributeNode("campus"), @NamedAttributeNode("verantwoordelijkheden")})
```

Je vermeldt bij attributeNodes met { } één array van @NamedAttributeNode annotations.

24.3.5 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",
        subgraph = "metManager"),
    subgraphs = @NamedSubgraph(name = "metManager",
        attributeNodes = @NamedAttributeNode("manager")))
```

❶
❷
❸
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) @NamedSubgraph duidt de behoefte van de geassocieerde Campus entity aan. 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.

24.3.6 Refactoring

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 maakt een constante in Docent:
`public static final String MET_CAMPUS = "Docent.metCampus";`
- Je gebruikt deze constante: `@NamedEntityGraph(name = Docent.MET_CAMPUS, ...)`
- Je vervangt in JpaDocentRepository "Docent.metCampus" door Docent.MET_CAMPUS



Je commit de sources. Je publiceert op GitHub.




Artikellijst: zie takenbundel

25 CASCADE

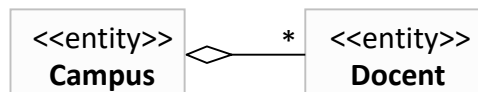
De levensduur van een value object is beperkt tot de levensduur van de bijbehorende entity.

- Als je een entity opslaat, slaat JPA de bijbehorende value objects op.
- Als je een entity verwijdert, verwijdert JPA de bijbehorende value objects.
- Als je een value object toevoegt aan de verzameling value objects van een entity, voegt JPA een record toe dat hoort bij dit value object.
- Als je een value object verwijdert uit de verzameling value objects 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 uit zoals een gewone associatie:

- Campus bevat een `Set<Docent>` variabele docenten
- 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 tikt, 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.

26 ASSOCIATIES IN JPQL CONDITIES 🧐

Je kan in condities van JPQL queries verwijzen naar de geassocieerde entities van een entity.

26.1 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 is null
```

Het aantal geassocieerde entities 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
```

27MULTI-USER EN RECORD LOCKING → → ← ←

JPA wijzigt één record met drie handelingen:

1. JPA leest het record als een entity in het interne geheugen.
2. Jij 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 !

27.1 Pessimistic record locking

JPA 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 vergrendelt met de EntityManager method `find` een record bij het lezen met een 3^o 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 maakt een method declaratie in `DocentRepository`:

```
Optional<Docent> readWithLock(long id);
```

Je implementeert de method in `JpaDocentRepository`:

```
public Optional<Docent> readWithLock(long id) {
    return Optional.ofNullable(
        manager.find(Docent.class, id, LockModeType.PESSIMISTIC_WRITE));
}
```

Je vervangt in de `DefaultDocentService` class in de method opslag:

```
docentRepository.read(id)... door
docentRepository.readWithLock(id)...
```

Terwijl jij de opslag geeft, kan geen andere gebruiker hetzelfde record wijzigen.

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 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

27.2 Optimistic record locking

De naam optimistic record locking is misleidend:

JPA vergrendelt het te wijzigen record op geen enkel moment! JPA doet wel volgende stappen

1. JPA leest het te wijzigen record als een entity in het interne geheugen.
2. Jij 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`.

27.2.1 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:

- ☹ Alle applicaties moeten bij een recordwijziging het getal in de kolom verhogen.

Je voert het script `IntVersie.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 klappt in de Data Source Explorer database niveau's open tot je de map Tables ziet.
2. Je klikt met de rechtermuisknop op Tables en je kiest Refresh.

Je maakt een private variabele in `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.fietsacademy.exceptions;
public class RecordAangepastException extends RuntimeException {
    private static final long serialVersionUID = 1L;
}
```

Je roept in de `DefaultDocentService` method `opslag` terug de `read` method op in plaats van de `readWithLock` method.

27.2.2 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.

Nadeel

- Een timestamp 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.

Nadelen als de applicaties zelf de kolom moeten bijwerken bij een recordwijziging:

- Alle applicaties moeten bij een recordwijziging de systeemtijd invullen in de kolom.
- Als de applicaties draaien op verschillende servers, moet de systeemtijd van die servers exact gelijk lopen, of je krijgt problemen.

Voordeel 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 `TimeStampVersie.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.

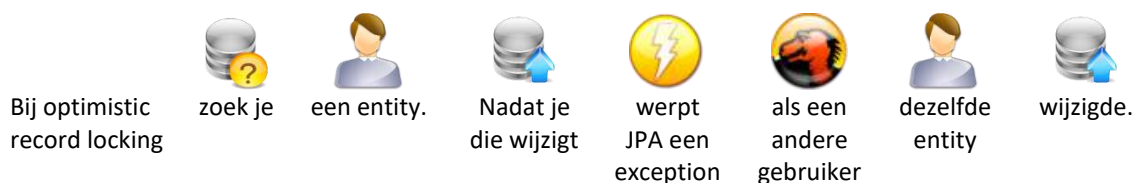
 versie TIMESTAMP

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

1. Je klappt in de Data Source Explorer database niveau's open tot je de map Tables ziet.
2. Je klikt met de rechtermuisknop op Tables en je kiest Refresh.

Je wijzigt in Docent het type van de private variabele versie naar Timestamp (uit de package `java.sql`):

`private Timestamp versie;`



27.3 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.
- Je weet bij pessimistic record locking vroeg in een use case dat de recordwijziging mislukt. Je weet dit bij optimistic record locking maar laat in de use case. Misschien moet je op dat moment andere handelingen van die use case ongedaan maken...

Als je bij een use cases twijfelt tussen pessimistic en optimistic locking, is er 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.



JPA maakt het gemakkelijker om de database aan te spreken.

Het blijft echter jouw verantwoordelijkheid om de database optimaal aan te spreken (zie cursus JDBC) en alles in goede banen te leiden als meerdere gebruikers tegelijk dezelfde data willen wijzigen.



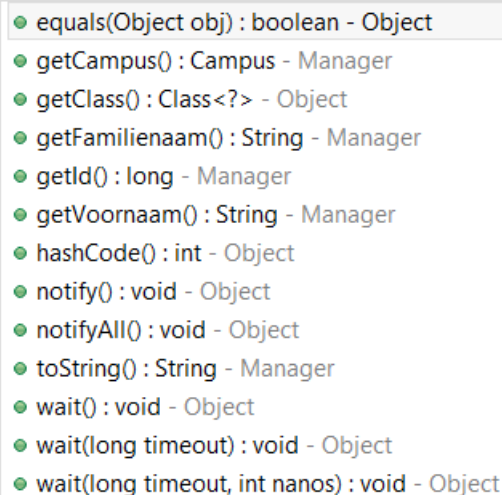
Je commit de sources. Je publiceert op GitHub.

28 STAPPENPLAN

Je kan volgend stappenplan volgen als je een web applicatie maakt met JPA en Spring



1. Converteer het project naar een JPA project.
2. Maak Entities en Value objects. Als je enkel Entities vindt en geen Value objects, heb je de werkelijkheid waarschijnlijk niet goed bestudeerd.
 - a. Maak geen Entities of Value objects voor database tables die jij niet gebruikt.
 - b. Maak de Value object classes immutable (geen setters).
 - c. Maak een Entity immutable als je die Entity enkel *leest* uit de database. 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 tikt 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.
```



Press 'Ctrl+Space' to show Template Proposals

- d. Maak geen setters voor 'alleen-lezen' eigenschappen (bvb. autonumber kolom).
- e. Maak in deze classes de methods equals en hashCode als je objecten van deze classes opneemt in een Set. Baseer equals en hashCode niet op de private variabele die bij een autonumber 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).
- f. 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>

- g. Plaats een berekening die met een Entity of Value object te maken heeft in de class van die Entity of Value object, niet in controllers, services of repositories

Eerste voordeel: je kan een berekening in een Entity of Value object 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 {
    ...
    @NumberFormat(pattern = "0.00")
    private BigDecimal prijsExclusief;
    @NumberFormat(pattern = "0.00")
    private BigDecimal btwPercentage;
    @NumberFormat(pattern = "0.00")
    public BigDecimal getPrijsInclusief() {
        return prijsExclusief.multiply(BigDecimal.ONE
            .add(btwPercentage.divide(BigDecimal.valueOf(100), 2,
                RoundingMode.HALF_UP)));
    }
}
```

3. Doe volgende stappen per nieuwe pagina van de web applicatie
 - a. Voeg de Repository interfaces en classes toe die je in die pagina nodig hebt.
Plaats hierin enkel de code die je nodig hebt. Als je een entity enkel *leest* uit de database, moet je Repository geen create, update of delete method bevatten.
 - b. Voeg de Service interfaces en classes toe die je in die pagina nodig hebt.
Als een method een record leest én daarna 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 de wijziging in de database voor jou.
 - c. Voeg een Controller toe of een method aan een bestaande controller toe.
Als je session data 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

29 HERHALINGSOEFENINGEN



Muziek: zie takenbundel

30 COLOFON

Domeinexpertisemanager:	Jean Smits
Moduleverantwoordelijke:	Hans Desmet
Medewerkers:	Hans Desmet
Versie:	13/9/2018
Nummer dotatielijst:	