# **JPA met HIBERNATE.**

**JPA** = Java Persistence API

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

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

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

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

## **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.

Je vermeldt elke entity class in persistence.xml.

## **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.

## **ENTITY ZOEKEN VIA DE PRIMARY KEY**

Je geeft aan de find method twee parameters mee:

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

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

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

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

## **TRANSACTIES**

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

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

Zoniet werpt JPA een TransactionRequiredException.

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

Er is één service class per entity class.

## **ENTITY TOEVOEGEN**

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

De method stuurt een SQL insert statement naar de database.

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

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

Als de database hierbij autonummering gebruikt (zoals hier het geval is), plaats je de parameter

strategy op IDENTITY: @GeneratedValue(strategy=GenerationType.***IDENTITY***)

## **ENTITY VERWIJDEREN**

## **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 !

**Vanuit een jsp naar dezelfde servlet gaan en percentage meegeven als parameter:**

<form method=*'post'* id=*'opslagform'*>

<label>Percentage:<span>${fouten.percentage}</span>

<input name=*'percentage'* value=*'*${param.percentage}*'* type=*'number'* min=*'0.01'* step=*'0.01'* autofocus>

</label>

<input type=*'submit'* value=*'Opslag'* id=*'submitknop'*>

</form>

**Vanuit zoeken.jsp naar een andere servlet gaan en id meegeven als parameter:**

<c:url value=*'/docenten/opslag.htm'* var=*'opslagURL'*>

<c:param name=*'id'* value=*'*${docent.id}*'* />

</c:url>

<a href=*'*${opslagURL}*'* class=*'knop'*><input type=*"button"* value=*'Opslag'*></a>

**Vanuit zoeken.jsp rechtstreeks naar de doPost-method van een andere servlet gaan en id meegeven als parameter:**

<c:url value=*'/docenten/verwijderen.htm'* var=*'verwijderURL'*>

<c:param name=*'id'* value=*'*${docent.id}*'*/>

</c:url>

<form action=*'*${verwijderURL}*'* method=*'post'*>

<input type=*'submit'* value=*'Verwijderen'*>

</form>

**Lezen, aanmaken en deleten van een entity (update van een entity gebeurt door de entity aan te passen, op het moment dat de commit wordt uitgevoerd, wordt de database aangepast, er wordt geen update-statement geschreven):**

**public class** DocentRepository **extends** AbstractRepository {

**public** Optional<Docent> read(**long** id) {

**return** Optional.*ofNullable*(getEntityManager().find(Docent.**class**, id));

}

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

getEntityManager().persist(docent);

}

**public void** delete(**long** id) {

read(id).ifPresent(docent -> getEntityManager().remove(docent));

}

}

## **JPQL**

Je leest tot nu één entity uit de database, aan de hand van de primary key,

met de EntityManager method find. Je schrijft voor andere lees operaties een query, in **JPQL** (**Java Persistence Query Language**). JPQL lijkt op SQL.

TypedQuery<Docent> query =

*getEntityManager*().createQuery("een JPQL query op docenten", Docent.**class**);

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

Ofwel in 1 statement:

List<Docent> docenten = getEntityManager().createQuery("select d from Docent d", Docent.**class**).getResultList();

## **NAMED QUERIES**

Je maakte queries tot nu met de EntityManager method createQuery. Een andere mogelijkheid zijn named queries.

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

@NamedQuery(name = "**Docent.findByWeddeBetween**",

query = "select d from Docent d where d.wedde between :van and :tot

order by d.wedde, d.id")

Oproepen van de named query **Docent.findByWeddeBetween** :

**return** getEntityManager().createNamedQuery("**Docent.findByWeddeBetween**", Docent.**class**)

.setParameter("van", van)

.setParameter("tot", tot)

.getResultList();

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

<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>

## **BULK UPDATES EN BULK DELETES blz 47**

Je voegt een query toe aan orm.xml:

<named-query name=*'Docent.algemeneOpslag'*>

<query>

update Docent d

set d.wedde = d.wedde \* :factor

</query>

</named-query>

## **INHERITANCE blz 49**

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

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

* 1 databasetable per class hierarchy.
* 1 databasetable per subclass.
* 1 databasetable per concrete class.

## **VALUE OBJECTS blz 55**

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

**Entity**: Bevat een identiteit attribuut (dat elke entity uniek voorstelt). Een entity verdwijnt niet op het moment dat een ander object verdwijnt.

**Value object**: Bevat geen identiteit attribuut (dat elk value object uniek voorstelt). Een value object verdwijnt op het moment dat een ander object verdwijnt.

**Immutable value objecten**

**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.

Value objecten hebben geen eigen service layer of repository layer. Ze worden mee uit de database

gelezen als je de entity leest die bij het value object hoort. Ze worden mee naar de database

geschreven als je de entity, die bij het value object hoort, naar de database schrijft.

**Value object classes detecteren**

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.

**Aggregate**

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

## **EEN VERZAMELING VALUE OBJECTEN MET EEN BASISTYPE (vb bijnamen) blz 62**

**Basistype kan zijn**: Byte, Short, Integer, Long, Float, Double, Boolean, Char, String, BigDecimal, Date

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

**lazy loading** = 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. De gegevens worden pas opgehaald uit de database op het moment ze echt nodig zijn.

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

## **EEN VERZAMELING VALUE OBJECTEN MET EEN EIGEN TYPE**

Het type van de value objecten in de verzameling kan een eigen geschreven class zijn.

**Voorbeeld**: een Campus entity heeft meerdere **TelefoonNr** value objecten.

De class TelefoonNr bevat geen variabele die verwijst naar het bijbehorende Campus object.

Op die manier is de class TelefoonNr herbruikbaar in andere entity classes: Klant, …

**Je detecteert met volgende tip een verzameling value objecten:**

Als in een één op veel relatie tussen de tables A en B, bij het verwijderen van een record uit de

table A de gerelateerde records uit de table B beter ook verwijderd worden, stel je de records uit

de table B voor als een verzameling value objecten die behoren tot een entity die hoort bij table A.

## **MANY-TO-ONE ASSOCIATIE blz 70**

associaties tussen entities

**Je voegt aan de class Docent een many-to-one associatie toe naar de class Campus. Dit komt erop neer dat in de class Docent een Campus-object wordt toegevoegd.**

Vb: **meerdere docenten behoren bij 1 campus**

Het is een **gerichte associatie**:

* Je weet welke campus bij een docent hoort (het campusid staat bij de docent)
* Je weet niet welke docenten bij een campus horen (in campus is er geen verwijzing naar docenten).

Het tegenovergestelde van gerichte associatie is een **bidirectionele associatie**. Daarin bevat de campus een verwijzing naar docenten ).

Een bidirectionele associatie is in gebruik handiger dan een gerichte associatie. De code van een bidirectionele associatie is echter complexer dan die van een gerichte associatie.

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

Je voegt code toe aan Docent:

@ManyToOne(optional = **false**)

@JoinColumn(name = "campusid")

**private** Campus campus;

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

**eager loading vervangen door lazy loading**:

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

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

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

## **ONE-TO-MANY ASSOCIATIE blz 73**

**Vergeet de wijzigingen die zijn gedaan voor de many-to-one-associatie.**

**Je voegt aan de class Campus een one-to-many associatie toe naar de class Docent.** **Dit komt erop neer dat in de class Campus een Set van Docent-objecten wordt toegevoegd.**

Je voegt code toe aan Campus:

@OneToMany

@JoinColumn(name = "campusid")

@OrderBy("voornaam, familienaam")

**private** Set<Docent> docenten;

**public** Set<Docent> getDocenten() {

**return** Collections.*unmodifiableSet*(docenten);

}

**public void** add(Docent docent) {

docenten.add(docent);

}

**public void** remove(Docent docent) {

docenten.remove(docent);

}

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

Vb: **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.

**JPA doet op een one-to-many associatie lazy loading.** Als je een Campus entity leest, leest JPA niet onmiddellijk de bijbehorende (Docent) entities.

## **BIDIRECTIONELE ASSOCIATIE blz 77**

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

Je voegt code toe aan Docent:

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

@JoinColumn(name = "campusid")

**private** Campus campus;

Je voegt code toe aan Campus:

@OneToMany(**mappedBy = "campus"**) 🡺 de naam van het Campus-object in de Docent-class

@OrderBy("voornaam, familienaam")

**private** Set<Docent> docenten;

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.

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

**public void** add(Docent docent) {

docenten.add(docent);

**if** (docent.getCampus() != **this**) { // als de andere kant nog niet bijgewerkt is

docent.setCampus(**this**); // werk je de andere kant bij

}

}

**public void** remove(Docent docent) {

docenten.remove(docent);

**if** (docent.getCampus() == **this**) { // als de andere kant nog niet bijgewerkt is

docent.setCampus(**null**); // werk je de andere kant bij

}

}

Je wijzigt in Docent de method setCampus:

**public void** setCampus(Campus campus) {

**if** (**this**.campus != **null** && **this**.campus.getDocenten().contains(**this**)) {

// als de andere kant nog niet bijgewerkt is

**this**.campus.remove(**this**); // werk je de andere kant bij

}

**this**.campus = campus;

**if** (campus != **null** && ! campus.getDocenten().contains(**this**)) {

// als de andere kant nog niet bijgewerkt is

campus.add(**this**); // werk je de andere kant bij

}

}

## **MANY-TO-MANY ASSOCIATIE blz 80**

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

De table verantwoordelijkheden beschrijft de verantwoordelijkheden van een docent.

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

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

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

Je drukt deze bidirectionele associatie uit

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

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

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

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

## **ONE-TO-ONE ASSOCIATIE blz 83**

Vb: Elke campus heeft 1 manager.

Je zal deze associatie beschrijven als bidirectioneel.

Je weet dan:

* welke manager bij een campus hoort.
* welke campus bij een manager hoort.

Je drukt deze associatie uit

* met een Manager variabele in de class Campus. Elk Campus object heeft zo een reference naar het bijbehorend Manager object.
* met een Campus variabele in de class Manager. Elk Manager object heeft zo een reference naar het bijbehorend Campus object.

## **ASSOCIATIES VAN VALUE OBJECTEN NAAR ENTITIES blz 85**

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

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

BELANGRIJKE BEPERKINGEN: zie blz 85

## **N + 1 PROBLEEM, JOIN FETCH QUERIES, ENTITY GRAPHS blz 86**

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

JPA stuurt één select statement naar de table docenten, daarna veel gelijkaardige select statements naar de table campussen die elk één record lezen aan de hand van de primary key. 🡺 **dit levert een performance problem op**

**OPLOSSINGEN:**

* Je kan zelf een SQL-statement schrijven 🡺 geen goeie oplossing
* Je kan JPQL een join laten uitvoeren 🡺 geen goeie oplossing
* **Entity graph** 🡺 goeie oplossing

Bij **Entity graphs** druk je in de named query zelf geen join uit. Je zal dit uitdrukken bij het oproepen van de named query.

Vb: Al je wil dat bij het lezen van een Docent ook meteen de Campus wordt meegelezen moet je onderstaande doen:

In de class Docent:

@NamedEntityGraph(name = "Docent.metCampus", attributeNodes = NamedAttributeNode("campus"))

Je vermeldt de naam van de private variabele campus in de huidige class Docent. Je drukt daarmee de behoefte uit dat bij het lezen van een Docent entity uit de database JPA direct ook de bijbehorende Campus entity moet lezen (via een join).

Je hebt deze behoefte nu als named entity graph gedefinieerd, maar JPA past die niet automatisch toe op elke named query die docenten leest. JPA zal dit pas doen als jij dit vraagt. Dit doe je met setHint in de repositoryclass.

.setHint("javax.persistence.loadgraph", getEntityManager().createEntityGraph("Docent.metCampus"))

**N+1 probleem is hiermee opgelost.**

## **CASCADE blz 89**

Gaat over de levensduur van value objecten en entities. Als je een entity verwijderd dan verdwijnen ook de bijhorende value objecten.

Een value object behoort tot een entity. Maar een entity kan ook behoren tot een andere entity.

Vb: Campus en Docent zijn beiden entities. Een campus heeft meerdere docenten. Dit wordt in een class diagram uitgedrukt met een aggregation.

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 dit wijzigen met de enum CascadeType.

Vb:

@OneToMany(cascade = CascadeType.**REMOVE**)

## **ASSOCIATIES IN JPQL CONDITIES blz 90**

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

Vb: Class Campus bevat een Manager. Je kan de Manager opnemen in de where-clause.

## **MULTI-USER EN RECORD LOCKING blz 92**

Je wijzigt één record met drie handelingen:

1. Je leest het record als een entity in het interne geheugen.
2. Je wijzigt deze entity in het interne geheugen.
3. JPA wijzigt het bijbehorende record bij een commit van de transactie.

Je hebt in een multi-user situatie het gevaar dat, tussen het lezen van het record en het wijzigen van het record, een andere gebruiker hetzelfde record wijzigde. JPA overschrijft bij de commit de wijzigingen van die andere gebruiker ! Je lost dit probleem op met **optimistic of pessimistic record locking**.

### Pessimistic record locking

Je vergrendelt hierbij het record wanneer je het leest in de database. Vanaf dan kan niemand anders het record wijzigen, verwijderen of vergrendelen. JPA ontgrendelt het record op het einde van de transactie.

Je kan met de EntityManager method find een record vergrendelen bij het lezen

met een derde parameter, die je invult met:

* PESSIMISTIC\_READ (dit wordt in SQL select … **lock in share mode**) Andere gebruikers kunnen het record lezen, maar niet wijzigen.
* of PESSIMISTIC\_WRITE (dit wordt in SQL select … **for update**) Andere gebruikers kunnen het record niet lezen met PESSIMISTIC\_READ of PESSIMISTIC\_WRITE en kunnen het record niet wijzigen.

Vb:

**public** Optional<Docent> readWithLock(**long** id) {

**return** Optional.*ofNullable*(getEntityManager().find(Docent.**class**, id, LockModeType.***PESSIMISTIC\_WRITE***));

}

### Optimistic record locking

De naam optimistic record locking is misleidend: je vergrendelt het te wijzigen record op geen enkel moment! Je doet wel volgende stappen

1. Je leest het te wijzigen record als een entity in het interne geheugen.
2. Je wijzigt deze entity in het interne geheugen.
3. JPA controleert bij de commit van de transactie of een andere gebruiker dit record wijzigde sedert het moment dat jij dit record gelezen hebt. Als dit zo is, wijzigt JPA het record niet en werpt een OptimisticLockException.

JPA kan op twee manieren controleren of een andere gebruiker het record wijzigde:

* met een versie kolom met als type een geheel getal.
* met een versie kolom met als type timestamp.

## **STAPPENPLAN blz 97**