Java Persistence API

Introductie

1

Wat is JPA?

- JPA = **Java Persistence API:**
  De standaard API voor persistentie in Java.
- Deze heeft ondersteuning voor:
  - Object-relationele mapping.
  - Objectgeoriënteerde queries (JPQL).
  - Schema generatie.
  - ...
- JPA is gebouwd bovenop JDBC en gebruikt JDBC om de databank aan te spreken.
- De API maakt deel uit van de **Jakarta EE specificatie, maar** kan ook gebruikt worden binnen Java SE.

HoGent

2

JPA

- een ORM (Object-Relational Mapping) library
- een officiële Java specificatie
- geïnspireerd door andere ORM libraries (Hibernate, TopLink, JDO)
- JPA heeft meerdere implementaties:
  - Hibernate
  - **EclipseLink**
  - Apache OpenJPA
  - ...
- Deze implementaties ondersteunen populaire databases (Oracle, DB2, SQL Server, MySQL, ...)

HoGent

3

Hoe JPA gebruiken in Java SE ?

- Plaats de nodige configuratie in **persistence.xml** bestand.

- Deze configuratie heet een **persistence unit:**
  - Een naam voor de persistence unit.
  - De JDBC URL waarop de databank te vinden is.
  - Het type schema generatie dat gebruikt moet worden:
    - **create**: de nodige tabellen aanmaken.
    - **drop-and-create**: de bestaande tabellen wissen en opnieuw aanmaken.
    - **none**: enkel de bestaande tabellen gebruiken.
  - Een opsomming van de entiteitklassen.

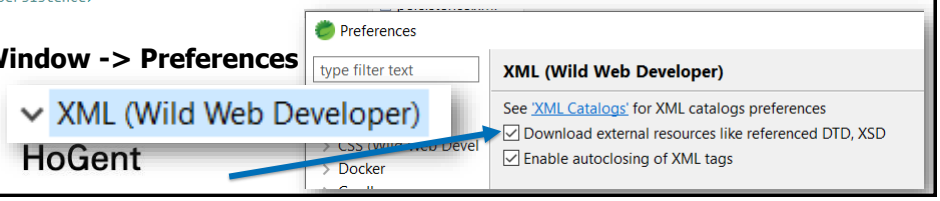
HoGent

4

META-INF/persistence.xml configuratie persistence unit

```xml
<?xml version="1.0" encoding="UTF-8"?>

<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
             https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
             version="3.0">

  <persistence-unit name="school" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>domein.Docent</class>
    <properties>
      <property name="jakarta.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/schooldb?serverTimezone=UTC"/>
      <property name="jakarta.persistence.jdbc.user" value="root"/>
      <property name="jakarta.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
      <property name="jakarta.persistence.jdbc.password" value="root"/>
      <property name="jakarta.persistence.schema-generation.database.action" value="create"/>
    </properties>
  </persistence-unit>
</persistence>
```

Window -> Preferences



HoGent

5

META-INF/persistence.xml

- Configuratie van één of meerdere persistence units.
- Een persistence unit bevat informatie over
  - aan te spreken database (via standaard JPA properties)
    Namen van standaard JPA properties beginnen met jakarta.persistence
  - lijst van entity classes
- Geef iedere persistence unit een unieke naam

HoGent

6

Entity Manager

- Een entity manager vormt het toegangspunt tot de persistentielaag:
  - Hij beheert de entiteiten.
  - Hij ondersteunt CRUD operaties op entiteiten
    - **persist**
    - **remove**
    - **….**
  - Hij kan JPQL-queries uitvoeren.
    - **createQuery**
    - **…**
  - Alle bewerkingen gebeuren via deze entity manager:

- Een EntityManagerFactory instance is thread safe.
- Aangemaakt door
  EntityManagerFactory.createEntityManager()

HoGent

7

Persistence context

- Elke entity manager is verbonden aan een zogenaamde *persistence context.*
- **Dit is de verzameling van entiteiten (objecten) die op dat moment gekend zijn bij de entity manager.**

- **Wijzigingen aan entiteiten** uit de context worden vanzelf doorgegeven naar de databank (weliswaar **via transacties**, maar **zonder** dat een **persist** of **merge** noodzakelijk is).

- Entiteiten die behoren tot de context worden *managed entities genoemd.*
- Entiteiten die niet behoren tot de context worden *detached entities genoemd.*

HoGent

8

Hoe JPA gebruiken in Java SE ?

- Voeg de vereiste bibliotheken toe aan je project:
  - De JDBC driver voor het type databank dat je gebruikt.
  - Een implementatie van JPA (EclipseLink of Hibernate).
- Maak een **entity manager aan via de factory:**

EntityManagerFactory emf =

Persistence.createEntityManagerFactory(“unitName");
EntityManager em = emf.createEntityManager();

De ”unitName” vervang je uiteraard door de naam van je persistence unit.

HoGent

9

Hoe JPA gebruiken in Java SE ? EntityManager éénmalig instantiëren

Een EntityManagerFactory instance maken vraagt veel tijd.
- Je maakt zo'n instance één keer per applicatie.
- Je houdt hem best bij via een singleton utility class.

```
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;
import lombok.AccessLevel;
import lombok.Getter;
import lombok.NoArgsConstructor;

@NoArgsConstructor(access = AccessLevel.PRIVATE)
public class JPAUtil
{
    @Getter private final static EntityManagerFactory entityManagerFactory =
            Persistence.createEntityManagerFactory(“school");
                                        // school= naam persistence unit
}
```

HoGent

10

Hoe JPA gebruiken in Java SE ?

- Bewerkingen die de databank aanpassen, verpak je steeds in een **transactie:**

```
em.getTransaction().begin();
...
em.getTransaction().commit();
```

- Vergeet tenslotte niet zowel de entity manager als de factory af te sluiten:

```
em.close();
emf.close();
```

HoGent

11

Java Persistence API

Object-relationele mapping

12

Object-relational mismatch

- Je stelt gegevens uit de werkelijkheid op een verschillende manier voor
  - als objecten in het interne geheugen
  - als records in een RDBMS
- Belangrijkste verschillen
  - Granularity
  - Inheritance
  - Associaties
- Een ORM library helpt je de mismatch aan te pakken:

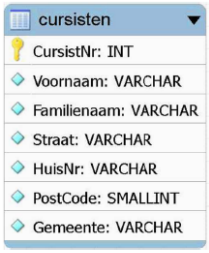
  objecten <-> ORM library <-> records

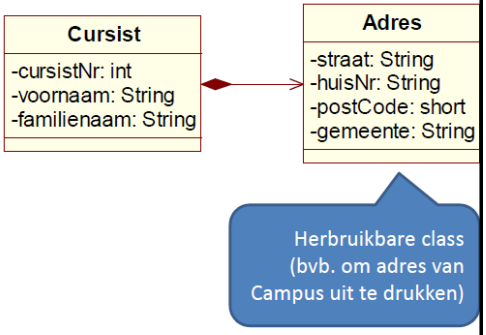
HoGent

13

Granularity

- In welke mate splits je een gegeven op in onderdelen
- Granularity RDBMS is kleiner dan granularity OOP





HoGent

14

Inheritance

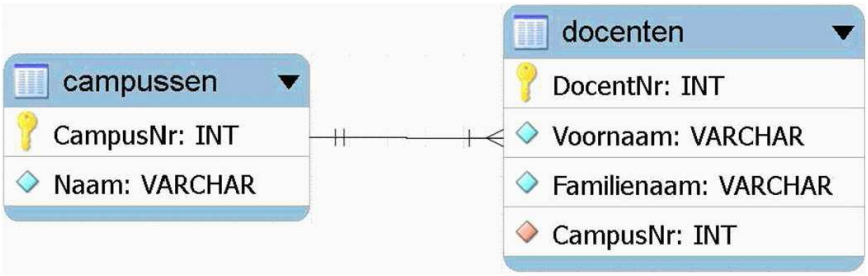
- Essentieel onderdeel OOP
- Onbestaand in RDBMS, enkel na te bootsen (zie verder)

HoGent

15

Associaties

- RDBMS: uitgedrukt met foreign key-primary key

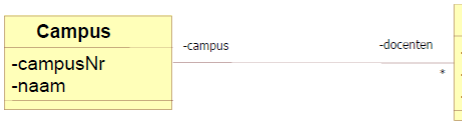


HoGent

16

Associaties

- OOP: uitgedrukt met reference variabelen




```
public class Campus {
  private Set<Docent> docenten;
}
```

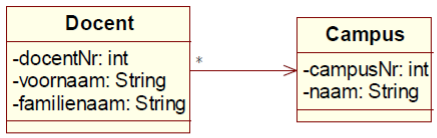
```
public class Docent {
  private Campus campus;
}
```

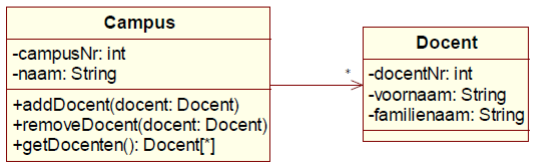
HoGent

17

Associaties

- RDBMS: associatie is altijd bidirectioneel
- OOP: associatie kan bidirectioneel of gericht zijn



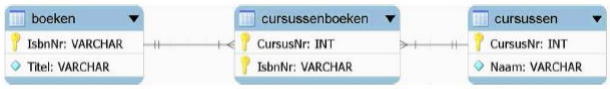


HoGent

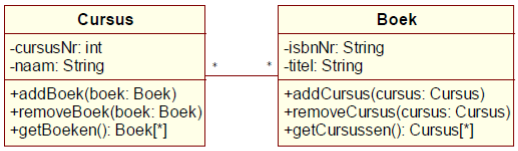
18

Veel-op-veel associaties

- RDBMS: altijd tussentabel nodig



- OOP: 'tussentabel' niet altijd nodig




```
public class Cursus {
  private Set<Boek> boeken;
}
```

```
public class Boek {
  private Set<Cursus> cursussen;
}
```


JPA Entity

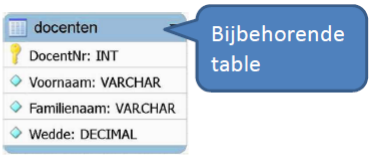
- **Entity**:
  Java object met bijbehorend record in database

- Entity class: Java class die entity beschrijft
  - moet **public** of **protected** default constructor hebben
  - mag geen final class zijn
  - mag geen nested class zijn
  - **Serializable**: aangeraden voor sessiebeheer en netwerkcommunicatie. EclipseLink vereist het niet, maar sommige frameworks en andere JPA-implementaties wel.

Instance variabele van Entity class

- Bevat waarde die opgeslagen is in het record dat bij de entity hoort.

```
public class Docent {
  private int id;
  private String voornaam;
  private String familienaam;
  private BigDecimal wedde;
}
```



HoGent

21

Mapping informatie

- Mapping informatie definieert:
  - welke klasse bij welke tabel hoort
  - welke instance variabele bij welke kolom hoort
  - ...
- Kan je schrijven
  - met @annotations in de entity class
  - in XML: META-INF/persistence.xml
- XML overschrijft @annotations

HoGent

22

Mapping met @annotations

• @Entity
Verplicht bij iedere entity class

• @Table(name="NaamVanDeTableDieEntitiesBevat")
Verplicht als de naam van de table ≠ naam van de entity class

• @Id
Verplicht bij instance variabele die hoort bij primary key

• @Column(name="NaamVanDeBijbehorendeKolom")
Verplicht als kolomnaam ≠ instance variabele naam

• @Transient
De private variabele heeft geen bijbehorende kolom, wordt dus niet in de database opgeslagen.

HoGent

23

Mapping met @annotations

```
@Entity
@Table(name = "docenten")
@NoArgsConstructor(access = AccessLevel.PROTECTED) //nodig voor JPA-ORM tool
public class Docent {
  @Id
  @GeneratedValue(            // primary key is door database ingevuld
    strategy = GenerationType.IDENTITY) // en is autonumber
  @Column(name = "DocentNr")  // DocentNr is kolom die hoort bij var. id
  private int id;
  private String voornaam;    // hoort automatisch bij kolom voornaam
  private String familienaam; // hoort automatisch bij kolom familienaam
  private BigDecimal wedde;   // hoort automatisch bij kolom wedde
}
```

Je kan de JPA @annotation schrijven:
• Vóór een instance variabele
JPA leest en schrijft dan de instance variabele direct
• Vóór een JavaBean getter method (getVoornaam, ...)
JPA gebruikt dan set en get methods

HoGent

24

Vervolg Docent class

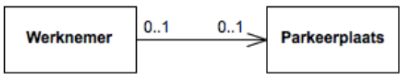
```
public Docent(String voornaam, String familienaam, BigDecimal wedde)
{
    this.voornaam = voornaam;
    this.familienaam = familienaam;
    this.wedde = wedde;
}

public void opslag(BigDecimal bedrag) {
    wedde = wedde.add(bedrag);
}

@Override
 public String toString() {
      return "%s %s %s".formatted(voornaam, familienaam, wedde);
 }
```

HoGent

25

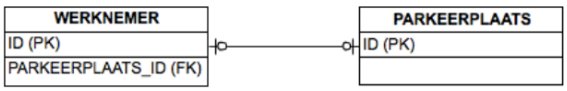
Relaties

One-to-one - Unidirectioneel




```
class Werknemer {
  @Id int id;

  @OneToOne
  Parkeerplaats p;
}
```

```
class Parkeerplaats {
  @Id int id;
}
```

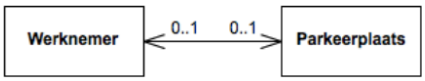


HoGent

26

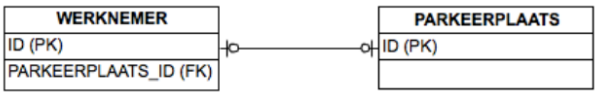
Relaties

One-to-one - Bidirectioneel




```
class Werknemer {
  @Id int id;

  @OneToOne
  Parkeerplaats p;
}
```

```
class Parkeerplaats {
  @Id int id;

  @OneToOne(mappedBy="p")
  Werknemer w;
}
```

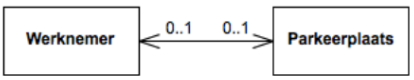


HoGent

27

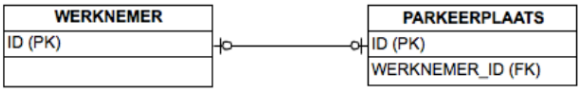
Relaties

One-to-one - Bidirectioneel (2)




```
class Werknemer {
  @Id int id;

  @OneToOne(mappedBy="w")
  Parkeerplaats p;
}
```

```
class Parkeerplaats {
  @Id int id;

  @OneToOne
  Werknemer w;
}
```

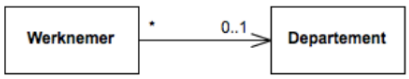


HoGent

28

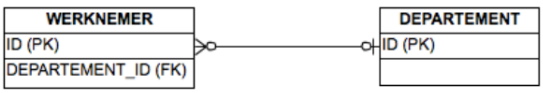
Relaties

Many-to-one - Unidirectioneel




```
class Werknemer {
  @Id int id;

  @ManyToOne
  Departement d;
}
```

```
class Departement {
  @Id int id;
}
```

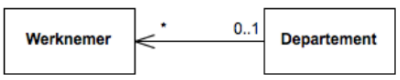


HoGent

29

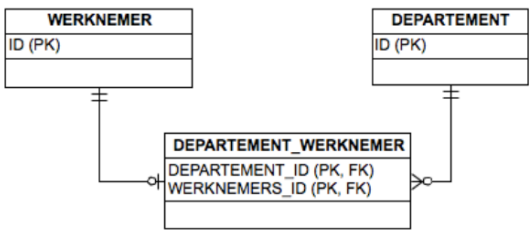
Relaties

One-to-many - Unidirectioneel




```
class Werknemer {
  @Id int id;
}
```

```
class Departement {
  @Id int id;

  @OneToMany
  List<Werknemer> w;
}
```

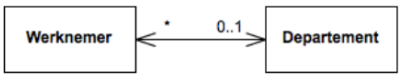


HoGent

30

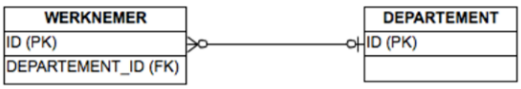
Relaties

One-to-many / Many-to-one - Bidirectioneel




```
class Werknemer {
  @Id int id;

  @ManyToOne
  Departement d;
}
```

```
class Departement {
  @Id int id;

  @OneToMany(mappedBy="d")
  List<Werknemer> w;
}
```



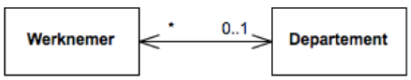
Een **@ManyToOne** is **altijd de owning-side** omdat deze één referentie naar het andere object bevat en de foreign key bewaart in de database. M.a.w. deze bevat **nooit mappedBy**, want hij beheert de relatie.

HoGent

31

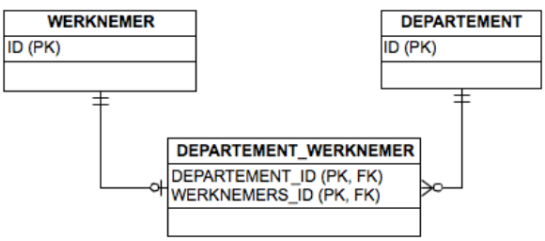
Relaties

One-to-many / Many-to-one - Bidirectioneel (2)




```
class Werknemer {
  @Id int id;

  @ManyToOne
  @JoinTable
  Departement d;
}
```

```
class Departement {
  @Id int id;

  @OneToMany(mappedBy="d")
  List<Werknemer> w;
}
```

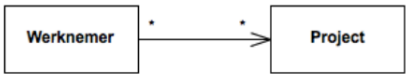


HoGent

32

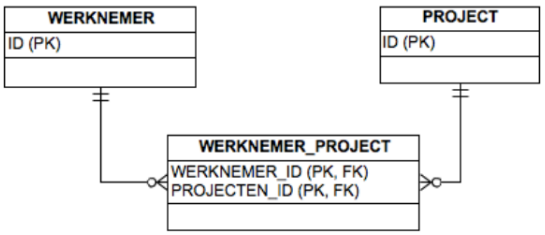
Relaties

Many-to-many - Unidirectioneel




```
class Werknemer {
  @Id int id;

  @ManyToMany
  List<Project> p;
}
```

```
class Project {
  @Id int id;
}
```

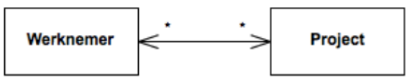


HoGent

33

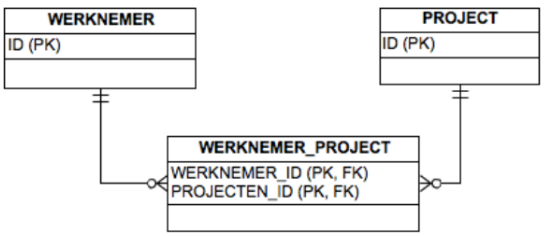
Relaties

Many-to-many - Bidirectioneel




```
class Werknemer {
  @Id int id;

  @ManyToMany
  List<Project> p;
}
```

```
class Project {
  @Id int id;

  @ManyToMany(mappedBy="p")
  List<Werknemer> w;
}
```

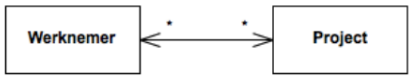


HoGent

34

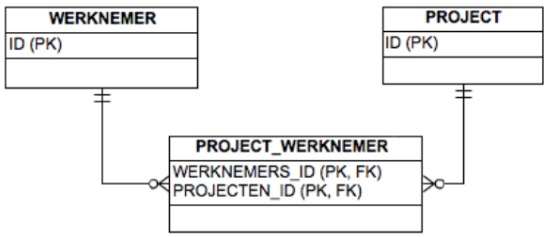
Relaties

Many-to-many - Bidirectioneel (2)




```
class Werknemer {
  @Id int id;

  @ManyToMany(mappedBy="w")
  List<Project> p;
}
```

```
class Project {
  @Id int id;

  @ManyToMany
  List<Werknemer> w;
}
```



HoGent

35

Relaties

Bidirectionele relaties

- We maken onderscheid tussen de owning side en de inverse side.
- De inverse side geeft de owning side aan met **mappedBy**.
- **Enkel wijzigingen aan de owning side hebben gevolgen voor de databank.**
- De applicatie moet zelf zorgen dat de inverse side consistent blijft met de owning side.
- Geef duidelijk aan dat het gaat om een bidirectionelerelatie (met mappedBy), zoniet ontstaan er twee relaties.

HoGent

36

Relaties

Lazy en eager loading

- Meerwaardige relaties gebruiken **lazy loading**.

- **Eager loading** kan op twee manieren worden bereikt:
  - Door EAGER toe te voegen:

    ... public class Werknemer …

    @OneToMany(fetch = FetchType.EAGER)

    private Set<Project> projecten;

    projecten worden altijd eager geladen bij het ophalen van een werknemer
  - Door JOIN FETCH te gebruiken in een query

    SELECT w FROM Werknemer w JOIN FETCH w.projecten

    projecten worden bij deze query eager geladen

HoGent

37

Embeddables

- Klassen die geen entiteitklasse worden, kunnen als embeddable gebruikt worden.
- Embeddable klassen geef je aan met **@Embeddable**.
- Voor attributen van een embeddable type is er de optionele annotatie **@Embedded**.
- Een embeddable object heeft geen eigen identiteit en krijgt geen eigen tabel.
- De attributen van een embeddable komen terecht in de tabel van de entiteit die eigenaar is van het embedded attribuut.
- Deze relatie is vergelijkbaar met compositie in UML.

HoGent

38

Embeddables: voorbeeld

```
import jakarta.persistence.Embeddable; ...
@Embeddable
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class Adres implements Serializable{
    private String straat;
    private String huisnummer;
    ...


@Entity
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class Klant implements Serializable{
    …

    @Embedded
    private Adres adres;
```

De velden van Adres worden opgeslagen in de tabel Klant

HoGent

39

Collections volgorde

- De volgorde van de elementen in een List gaat verloren in de databank.
- Een volgorde kan toegewezen worden tijdens het inlezen.
- Hiervoor gebruik je @OrderBy().
- Deze annotatie resulteert in een ORDER BY clausule in SQL.
- De default volgorde is oplopend volgens primaire sleutel.
- Een persistente ordening is ook mogelijk, maar is weinig performant.

HoGent

40

Overerving
Tussen entiteitklassen

- De hoogste entiteitklasse kiest een implementatiewijze:
  - @**Inheritance**(strategy=InheritanceType.SINGLE_TABLE)
    → één tabel voor de volledige hiërarchie.
  - @**Inheritance**(strategy=InheritanceType.JOINED)
    → één tabel per klasse in de hiërarchie.
  - @**Inheritance**(strategy=InheritanceType.TABLE_PER_CLASS)
    → één tabel per concrete klasse in de hiërarchie.
- Elke entiteitklasse **gebruikt dezelfde primaire** sleutel.
- Deze kan overgeërfd worden.
- *Zie "Slides JPA Overerving voorbeelden"*

HoGent

41

Overerving
Abstracte klassen

- Op vlak van persistentie is er geen verschil tussen abstracte klassen en concrete klassen !
- Ook abstracte klassen kunnen dus entiteitklassen zijn.

HoGent

42

Overerving
Niet-entiteitklassen

- Klassen die geen entiteitklasse worden, nemen niet deel aan overerving.
- **Wanneer een entiteitklasse overerft van een niet entiteitklasse, worden de overgeërfde attributen niet persistent gemaakt**.

- **Een uitzondering zijn klassen voorzien van @MappedSuperclass**.
- Een mapped superclass gedraagt zich als een embeddable:
  - De klasse is geen entiteitklasse en krijgt dus geen eigen tabel.
  - Alle attributen en relaties (incl. annotaties) komen terecht in de subklassen.

HoGent

43

Voorbeeld @MappedSuperclass

```
@MappedSuperclass
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public abstract class Info implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private LocalDate createdDate; …

@Entity
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class Product extends Info {
    private String naam;
```

- Geen aparte tabel voor Info.
- De tabellen van de subklassen (Product, …) bevatten wel de velden van de superklasse (zoals id en createdDate).

HoGent

44

Java Persistence API

Entity manager bewerkingen

45

Bewerkingen
Find

- Zoekt een entiteit op in de databank op basis van een opgegeven sleutel.
- Geeft null terug indien er geen entiteit met deze sleutel gevonden werd.
- Het gevonden object belandt in de context.

HoGent

46

Bewerkingen
Persist

- Voegt een nieuwe entiteit toe aan de context.
- Deze bewerking is enkel bedoeld voor nieuwe entiteiten.
- De controle gebeurt op basis van de primaire sleutel.
- Om bestaande entiteiten aan te passen met nieuwe gegevens, gebruik je een merge.

HoGent

47

Bewerkingen
Merge

- Voegt een bestaande entiteit opnieuw toe aan de context.
- Deze bewerking gebruik je wanneer entiteiten gewijzigd zijn buiten de context en deze wijzigingen ook in de databank terecht moeten komen.
- Opgelet: merge smelt de oude en nieuwe toestand samen tot een nieuw object. Het oorspronkelijke object belandt dus niet in de context. Wanneer je dit object nog wenst aan te passen, gebruik je het object dat merge teruggeeft:

```
object = em.merge(object);
```

HoGent

48

Bewerkingen
Remove

- Verwijdert een entiteit uit de context en uit de databank.
- Enkel entiteiten uit de context kunnen verwijderd worden.
- Bij het verwijderen van entiteiten is het belangrijk na te denken over de relaties.
- Vaak is het nodig eerst de relaties met de entiteit te verwijderen, alvorens deze entiteit zelf te verwijderen.

HoGent

49

Bewerkingen
Detach

- Haalt de opgegeven entiteit uit de context.
- Verdere wijzigingen aan deze entiteit komen niet meer terecht in de databank, tenzij na een merge.
- In Java EE gebeurt dit op het einde van elke transactie, voor alle entiteiten in de context.

HoGent

50

Cascade

- Bewerkingen kunnen doorgegeven worden tussen entiteiten op basis van hun relaties.
- Een bewerking op de ene entiteit resulteert dan in eenzelfde bewerking op de andere entiteit(en).
- Mogelijkheden zijn:
  - CascadeType.PERSIST
  - CascadeType.MERGE
  - CascadeType.DETACH
  - CascadeType.REMOVE
  - CascadeType.REFRESH
  - CascadeType.ALL

HoGent

51

Cascade

- Het cascadetype geef je aan in de relatieannotaties.
- Bijvoorbeeld:
  - @OneToMany(cascade=CascadeType.ALL)
  - @ManyToOne(cascade={CascadeType.
    PERSIST , CascadeType.MERGE})

HoGent

52

Cascade
Orphan removal

- @OneToOne en @OneToMany ondersteunen ook *orphan removal, bv:*

  @OneToOne(orphanRemoval=true)
- Dit zorgt ervoor dat het object dat het ‘kind’ is van de relatie automatisch verwijderd wordt wanneer de relatie wordt verbroken.
- Dit resulteert automatisch ook in een remove cascade.

HoGent

53

Java Persistence API

Java Persistence Query Language (JPQL)

54

JPQL

- Objectgeoriënteerde querytaal.
- Gebaseerd op SQL.
- Werkt op basis van entiteiten en attributen, niet op basis van tabellen en kolommen.
- Ondersteunt parameters:
  - Positioneel: ?1, ?2, ...
  - Met naam: :name, :project, ...

HoGent

55

JPQL
Structuur van een query

- SELECT ...
  FROM ...
  WHERE ...
  GROUP BY ...
  HAVING ...
  ORDER BY ...

HoGent

56

JPQL
Eenvoudige voorbeelden

- SELECT e
  FROM Employee e
- SELECT e.name, e.salary
  FROM Employee e
- SELECT DISTINCT e.department
  FROM Employee e

HoGent

57

FROM

- Geeft aan welke entiteiten gebruikt worden als bron.
- Kent verplicht een alias toe aan elke entiteit.
- Ondersteunt joins op basis van relaties tussen entiteiten.

HoGent

58

FROM
Voorbeelden inner join

- SELECT p
  FROM Employee e JOIN e.phones p
  WHERE e.id = :id

- SELECT COUNT(e)
  FROM Project p JOIN p.employees e
  WHERE p.name = ‘Top Secret Project‘
  AND e.gender = Gender.FEMALE

HoGent

59

SELECT

- Geeft aan welke entiteiten, attributen of waarden het antwoord van de query vormen.
- Ondersteunt property-notatie (.) voor het opvragen van attributen of navigeren van relaties.
- Ondersteunt aggregatiefuncties.
- Ondersteunt polymorfie en overerving.
- Kan losse waarden bundelen tot een nieuw object met een constructor.
- Attributen van een Collection-type zijn niet toegelaten.
  - Gebruik hiervoor een inner join, zoals op de vorige slide.

HoGent

60

SELECT
Voorbeeld constructor

- SELECT NEW EmployeeInfo(e.name, e.salary)
  FROM Employee e
  WHERE SIZE(e.projects) > 5

HoGent

61

WHERE

- Ondersteunt volgende operatoren:
  - +, -, *, /
  - =, <>, <, >, <=, >=
  - AND, OR, NOT
  - [NOT] BETWEEN, [NOT] LIKE
  - [NOT] IN, [NOT] MEMBER OF
  - IS [NOT] NULL, IS [NOT] EMPTY
  - EXISTS, ANY, ALL, SOME
- Ondersteunt subqueries.

HoGent

62

WHERE
Voorbeelden operatoren

- SELECT e
  FROM Employee e
  WHERE e.hireDate BETWEEN
  {d '2012-01-01'} AND CURRENT_DATE
- SELECT p
  FROM Project p
  WHERE p.name NOT LIKE 'QoS%'
- SELECT e
  FROM Employee e
  WHERE e.department IN (:d1, :d2)

HoGent

63

WHERE
Voorbeelden operatoren

- SELECT e
  FROM Employee e
  WHERE :project MEMBER OF e.projects
- SELECT p
  FROM Project p
  WHERE p.employees IS EMPTY
- SELECT e
  FROM Employee e
  WHERE e.salary <
  ANY (SELECT d.salary FROM e.directs d)

HoGent

64

GROUP BY en HAVING

- Deze clausules werken net zoals in SQL:
  - GROUP BY bundelt de resultaten op basis van een of meerdere entiteiten of attributen.
  - HAVING filtert de gegroepeerde resultaten.
    - WHERE filtert de resultaten vóór de groepering gebeurt!
- JPQL ondersteunt volgende aggregatiefuncties:
  - AVG
  - COUNT
  - MIN
  - MAX
  - SUM

HoGent

65

GROUP BY en HAVING
Voorbeelden

- SELECT d.name, AVG(e.salary)
  FROM Department d JOIN d.employees e
  GROUP BY d.name
  HAVING AVG(e.salary) > 2000

- SELECT e, COUNT(p)
  FROM Employee e JOIN e.projects p
  GROUP BY e
  HAVING COUNT(p) >= 2

HoGent

66

ORDER BY

- Deze clausule werkt net zoals in SQL.
- Enkel entiteiten of attributen die voorkomen in de SELECT-clausule kunnen gebruikt worden om te sorteren.

HoGent

67

ORDER BY
Voorbeelden

- SELECT e
  FROM Employee e
  ORDER BY e.name ASC

- SELECT e.name, e.salary * 0.05 AS bonus
  FROM Employee e
  ORDER BY bonus DESC

HoGent

68

Resultaat van een query

- Het resultaat van een query wordt bepaald door de entiteiten, attributen of waarden in de SELECT-clausule.
- Wanneer de SELECT-clausule uit meerdere delen bestaat, worden deze delen gebundeld in een Object[].
- Wanneer de query meerdere resultaten heeft, worden deze resultaten gebundeld in een List.

HoGent

69

Queries aanmaken

- Een query wordt aangemaakt door een entity manager.
- Hiervoor gebruik je een van de volgende methoden:
  - createQuery(String)
    - geeft een Query terug en zal zijn resultaten teruggeven als Object of List.
  - createQuery(String, Class<T>)
    - geeft een TypedQuery<T> terug en zal zijn resultaten teruggeven als T of List<T>.

HoGent

70

Parameters doorgeven

- **setParameter(String naam, Object waarde)**
  - Gebruik de naam van de parameter zoals gedefinieerd in de query, bvb:

```
SELECT e
FROM Employee e
WHERE :project MEMBER OF e.projects
```

entityManger.createQuery(…).setParameter(**"project"**, aProject) …

HoGent

71

Resultaten opvragen

- Om een query uit te voeren en de resultaten op te vragen, gebruik je een van de volgende methoden:
  - getSingleResult()
    - geeft een Object (of T) terug als er exact één record is.
    - gooit een NoResultException als er geen resultaat is.
    - gooit een NonUniqueResultException als er meer dan één resultaat is.
  - getResultList()
    - geeft een List (of List<T>) terug met de gevonden resultaten.
    - geeft een lege lijst terug als er geen resultaten zijn.
  - getResultStream()
    - idem als getResultList, behalve geeft een Stream in plaats van een List terug.

HoGent

72

Named queries

- Het uitvoeren van queries kan veel sneller wanneer de structuur van de query op voorhand gekend is.
- De annotaties @NamedQuery en @NamedQueries hebben precies deze bedoeling.
- Ze worden geplaatst bij een entiteitklasse.
- Een named query kan voorgecompileerd worden en is veel efficiënter dan een gewone query.
- De naam van een query moet uniek zijn binnen de persistence unit.

HoGent

73

Named queries

- Named queries worden op dezelfde manier gebruikt als gewone queries, maar worden aangemaakt met:
  - createNamedQuery(String)
  - createNamedQuery(String, Class<T>)
- De String-parameter is niet langer de query zelf, maar de naam van de query.

HoGent

74

Voorbeelden

- ```
  @NamedQuery(name="Employee.findAll",
                query="SELECT e FROM Employee e")
  ...
  List<Employee> l =
      em.createNamedQuery("Employee.findAll", Employee.class).
      getResultList();
  ...
  ```

- ```
  @NamedQuery(name="Employee.shortInfo",
                query =  """
                         SELECT e.id, e.name
                         FROM Employee e
                        """)
  ...
  ```

HoGent

75