HoGent

BEDRIJF
EN
ORGANISATIE

Java Persistence API

Introductie

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, ...)

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 **Java EE specificatie, maar** kan ook gebruikt worden binnen Java SE.

HoGent

Hoe JPA gebruiken in Java SE ?

- Plaats de nodige configuratie in persistence.xml.
- 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

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).

HoGent

Hoe JPA gebruiken in Java SE ?

- 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.
- Alle bewerkingen gebeuren via deze entity manager:
  - persist
  - merge
  - createQuery
  - ...

HoGent

Hoe JPA gebruiken in Java SE ?

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

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

HoGent

Hoe JPA gebruiken in Java SE ?

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

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

HoGent

HoGent

BEDRIJF
EN
ORGANISATIE

Java Persistence API

Object-relationele mapping

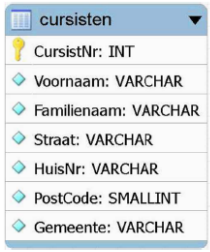
Object-relational mismatch

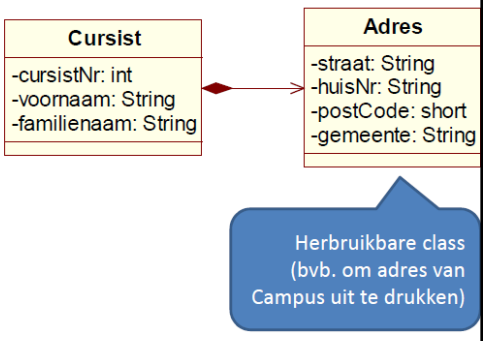
- Je stelt gegevens uit de werkelijkheid op een verschillende manier voor
  - als objecten in het interne geheugen
  - als records in een RDBMS
- Belangrijkste verschillen
  - Granularity
  - Inheritance
  - Associaties
- Een ORM library helpt je de mismatch aan te pakken:

objecten <-> ORM library <-> records

Granularity

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



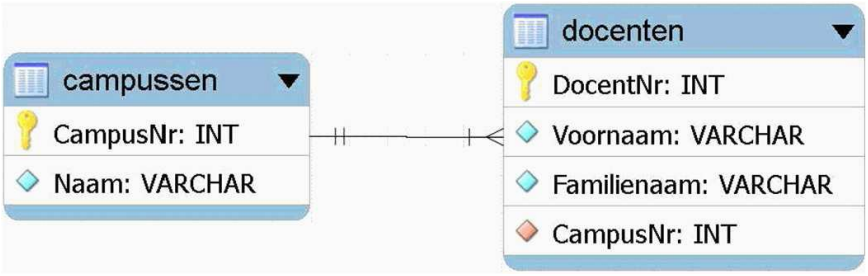


Inheritance

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

Associaties

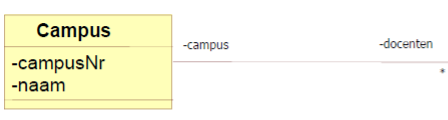
- RDBMS: uitgedrukt met foreign key-primary key



HoGent

Associaties

- OOP: uitgedrukt met reference variabelen



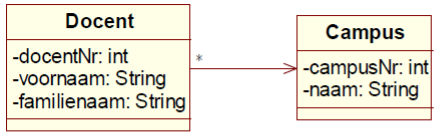

```
public class Campus {
  private Set<Docent> docenten;
}
```

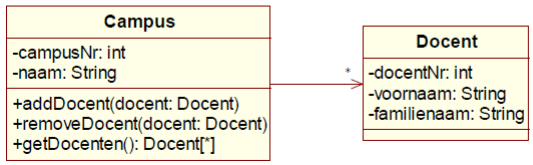
```
public class Docent {
  private Campus campus;
}
```

HoGent

Associaties

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

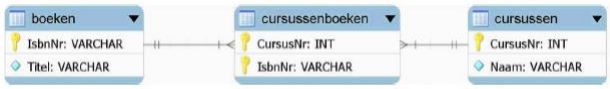




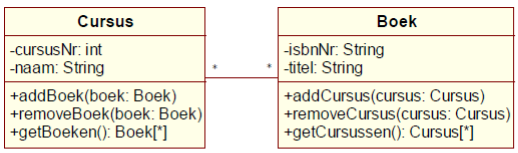
HoGent

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;
}
```

HoGent

JPA Entity

- Entity:
  Java object met bijbehorend record in database

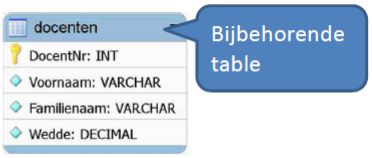
- Entity class: Java class die entity beschrijft
  - moet public of protected default constructor hebben
  - mag geen final class zijn
  - mag geen nested class zijn

HoGent

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

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/orm.xml
- XML overschrijft @annotations

HoGent

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

HoGent

Mapping met @annotations

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

Mapping met @annotations

- @Temporal(TemporalType.DATE)
De kolom die bij een Date variabele hoort is van het type Date (enkel datum)

- @Temporal(TemporalType.TIME)
De kolom die bij een Date variabele hoort is van het type Time (enkel tijd)

- @Temporal(TemporalType.TIMESTAMP)
De kolom die bij een Date variabele hoort is van het type DateTime (datum én tijd)

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

HoGent

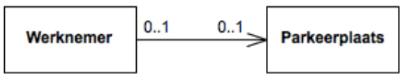
de rest van de Docent class

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

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

public void opslag(BigDecimal bedrag) {
    wedde = wedde.add(bedrag);
}
@Override
public String toString() {
    return id + " " + voornaam + " " + familienaam + " " + wedde;
}
```

HoGent

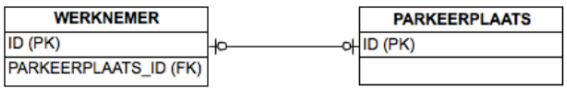
Relaties

One-to-one - Unidirectioneel




```
class Werknemer {
  @Id int id;

  @OneToOne
  Parkeerplaats p;
}
```

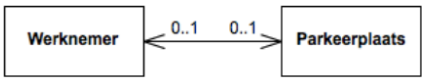
```
class Parkeerplaats {
  @Id int id;
}
```



HoGent

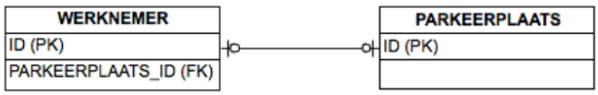
Relaties

One-to-one - Bidirectioneel




```
class Werknemer {
  @Id int id;

  @OneToOne
  Parkeerplaats p;
}
```

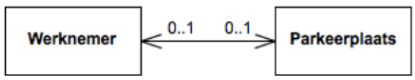
```
class Parkeerplaats {
  @Id int id;

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



HoGent

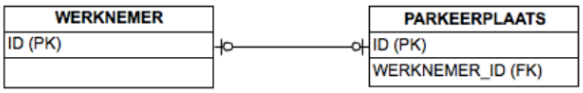
Relaties

One-to-one - Bidirectioneel (2)




```
class Werknemer {
  @Id int id;

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

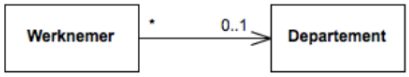
```
class Parkeerplaats {
  @Id int id;

  @OneToOne
  Werknemer w;
}
```



HoGent

Relaties

Many-to-one - Unidirectioneel



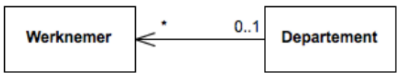

```
class Werknemer {
  @Id int id;

  @ManyToOne
  Departement d;
}
```

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

HoGent

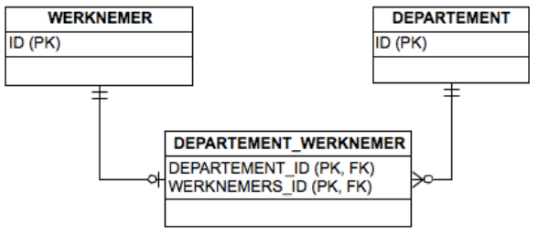
Relaties

One-to-many - Unidirectioneel




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

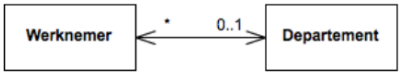
```
class Departement {
  @Id int id;

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



HoGent

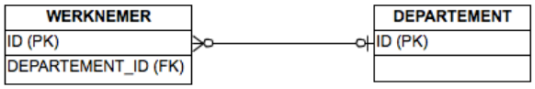
Relaties

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




```
class Werknemer {
  @Id int id;

  @ManyToOne
  Departement d;
}
```

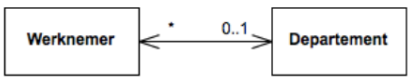
```
class Departement {
  @Id int id;

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



HoGent

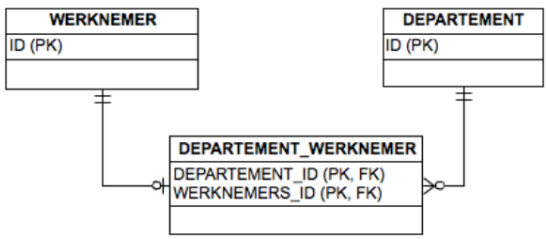
Relaties

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




```
class Werknemer {
  @Id int id;

  @ManyToOne
  @JoinTable
  Departement d;
}
```

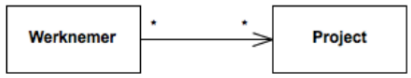
```
class Departement {
  @Id int id;

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



HoGent

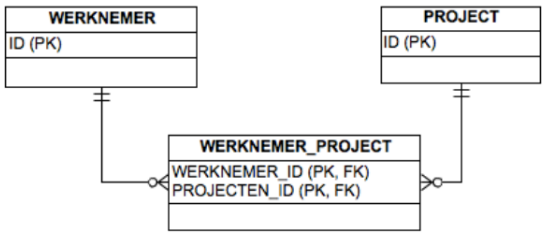
Relaties

Many-to-many - Unidirectioneel




```
class Werknemer {
  @Id int id;

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

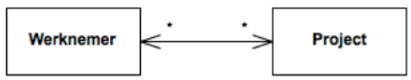
```
class Project {
  @Id int id;
}
```



HoGent

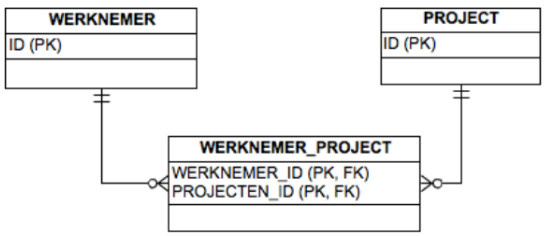
Relaties

Many-to-many - Bidirectioneel




```
class Werknemer {
  @Id int id;

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

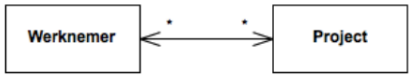
```
class Project {
  @Id int id;

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



HoGent

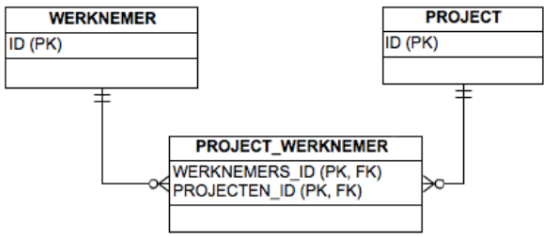
Relaties

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

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

Relaties
Lazy loading

- Meerwaardige relaties gebruiken lazy loading.
- Eager loading is mogelijk met bijvoorbeeld:

  @OneToMany(fetch=FetchType.EAGER)

HoGent

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

Collections

- Collections van een entiteittype zijn reeds behandeld: dit zijn de meerwaardige relaties.
- Andere collections kunnen opgeslagen worden met basic mapping (serialisatie) of als extra tabel.
- Het gebruik van een extra tabel is flexibeler en geef je aan met @ElementCollection:
  - Van toepassing op Collection, List, Set en Map.
  - Vervangt basic mapping.
  - De extra tabel bevat de inhoud van de collection en heeft een join column naar de entiteit die eigenaar is van de collection.
- Collections van een embeddable type zijn ook mogelijk.

HoGent

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

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.

HoGent

Overerving
Single table

- De volledige hiërarchie komt terecht in één tabel.
- Deze tabel bevat een kolom die het subtype aangeeft.
- Positief:
  - Meest performante oplossing voor queries (vereist geen joins).
- Negatief:
  - Veel wijzigingen aan dezelfde tabel.
  - Veel lege velden.
  - Attributen in subklassen kunnen niet verplicht zijn.

HoGent

Overerving
Joined

- Aparte tabel per klasse in de hiërarchie.
- Positief:
  - Sluit goed aan bij het objectgeoriënteerd ontwerp.
  - Geen plaatsverspilling zoals bij single table.
  - Genormaliseerd.
- Negatief:
  - De performantie van queries is afhankelijk van het aantal klassen in de hiërarchie (vereist joins).

HoGent

Overerving
Table per class

- Aparte tabel per concrete klasse in de hiërarchie.
- Deze tabel bevat ook alle overgeërfde attributen.
- Positief:
  - Performant voor queries over één subklasse (vereist geen joins).
- Negatief:
  - Veel denormalisatie.
  - Queries over alle subklassen zijn minder performant.

HoGent

Overerving
Conclusie

- Gebruik bij voorkeur de joined werkwijze.
- Voor grote overervingshiërarchiën is de single table werkwijze performanter maar minder flexibel.

HoGent

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

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

Mapping in META-INF/orm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
  http://java.sun.com/xml/ns/persistence/orm/orm_2_0.xsd"
  version="2.0">
  <persistence-unit-metadata>
    <persistence-unit-defaults>
      <access>FIELD</access>
    </persistence-unit-defaults>
  </persistence-unit-metadata>
  <package>be.vdab.entities</package>
  <entity class="Docent">
    <table name="docenten"/>
    <attributes>
      <id name="id">
        <column name="DocentNr"/>
        <generated-value/>
      </id>
    </attributes>
  </entity>
</entity-mappings>
```

HoGent

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 javax.persistence
  - properties eigen aan één JPA implementatie
    Namen van implementatie properties beginnen niet met javax.persistence
  - lijst van entity classes
- Geef iedere persistence unit een unieke naam

HoGent

META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="vdab1">
    <class>be.vdab.entities.Docent</class>
    <properties>
      <property name="javax.persistence.jdbc.driver"
        value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.jdbc.password" value="lok1109"/>
      <property name="javax.persistence.jdbc.url"
        value="jdbc:mysql://localhost/vdab1"/>
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.format_sql" value="true"/>
      <property name="hibernate.use_sql_comments" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

HoGent

HoGent
BEDRIJF
EN
ORGANISATIE

Java Persistence API

Entity manager

EntityManagerFactory

- Leest volgende informatie binnen in het interne geheugen:
  - de configuratie van één persistence unit
  - de meta informatie van de classes van die persistence unit
- Je maakt een instance met de static method
  Persistence.createEntityManagerFactory(...)
- Een EntityManagerFactory instance is thread safe.
- 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.

EntityManagerFactory

```
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class JPAUtil
 {
    private final static EntityManagerFactory entityManagerFactory =
     Persistence.createEntityManagerFactory("vdab1");
          // vdab1 = naam persistence unit
    public static EntityManagerFactory getEntityManagerFactory() {
     return entityManagerFactory;
    }

    private JPAUtil() {
    }
}
```

HoGent

Entity Manager

- Een entity manager vormt het toegangspunt tot de persistentielaag:
  - Hij beheert de entiteiten.
  - Hij ondersteunt CRUD operaties op entiteiten.
  - Hij kan JPQL-queries uitvoeren.
- Aangemaakt door EntityManagerFactory.createEntityManager()

HoGent

Persistence context

- Elke entity manager is verbonden aan een zogenaamde *persistence context.*
- Dit is de verzameling van enteiten (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

Persistence context
In Java SE

- Het default gedrag van een entity manager in Java SE verbindt de levensduur van de context aan die van de entity manager.
- De context is leeg bij het aanmaken van de entity manager.
- Entiteiten die deel uitmaken van de input naar of output van de entity manager zijn, belanden in de context.
- De context blijft dezelfde tot de entity manager wordt afgesloten.

HoGent

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

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

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

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

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

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

Cascade

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

HoGent

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

HoGent
BEDRIJF
EN
ORGANISATIE

Java Persistence API

Java Persistence Query Language (JPQL)

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, ...

JPQL
Structuur van een query

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

HoGent

JPQL
Eenvoudige voorbeelden

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

HoGent

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

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.sex = Sex.FEMALE

HoGent

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

SELECT
Voorbeeld constructor

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

HoGent

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

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

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

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

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

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

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

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

Queries aanmaken

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

HoGent

Resultaten opvragen

- Om een query uit te voeren en de resultaten op te vragen, gebruik je een van de volgende methoden:
  - getSingleResult()
  - getResultList()
- De eerste methode geeft een Object (of T) terug.
- De tweede methode geeft een List (of List<T>) terug.
- Uiteraard is het nodig eerst de parameters in te vullen.
- Dit kan met de verschillende setParameter() methoden.

HoGent

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

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

Voorbeelden

- @NamedQuery(name="Employee.findAll",
  query="SELECT e FROM Employee e")

```
...
TypedQuery<Employee> q =
  em.createNamedQuery("Employee.findAll", Employee.class);
List<Employee> l = q.getResultList();
...
```

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

```
...
Query q = em.createNamedQuery("Employee.shortInfo");
List l = q.getResultList();
for (Object o : l) {
  Object[] result = (Object[])o;
  int id = (Integer)result[0];
  String name = (String)result[1];
...
}
```

HoGent