

JPA (Hibernate, Eclipselink)

JPA (Java Persistence API)

- JPA je specifikace, která je standardem pro objektově relační mapování (ORM) v jazyce Java. Je pouze závislá na Java SE, ale nejčastěji se využívá v Java EE.
- JPA vznikla standardizací ORM, jehož průkopníkem jsou Hibernate a JDO. V posledních verzích Hibernate implementuje JPA specifikaci (aktuálně již JPA 2.0).
- Populární ORM frameworky, které implementují JPA 2.0:
 - JBoss Hibernate
 - EclipseLink
 - OpenJPA

Entita

- Entita je objekt, který reprezentuje data v databázi. Typicky entitní třída reprezentuje tabulku v relační databázi a každá instance této třídy pak koresponduje s jednou řádkou tabulky.
- Entitní třída musí splňovat následující vlastnosti:
 - Musí být oannotována anotací `@Entity`.
 - Musí mít public nebo protected konstruktor bez parametrů.
 - Nesmí být deklarována jako final (to samé platí i pro metody).
 - Atributy musí být deklarovány jako private, protected nebo s package viditelností a přístup k nim musí být pomocí getterů / setterů.
 - Musí obsahovat jeden atribut, který je oannotován anotací `@Id`.

Identifikátor entity

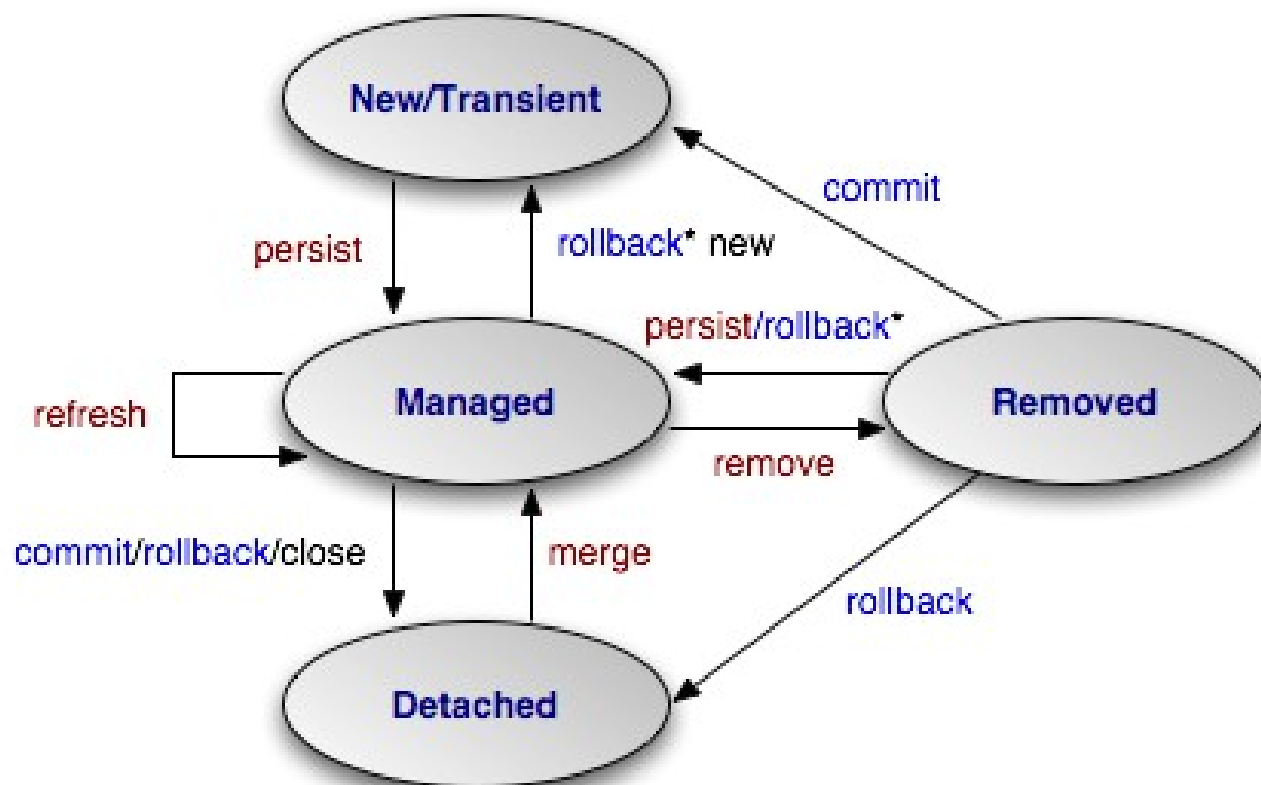
- U identifikátoru entity je nutné definovat informaci, odkud se příslušný identifikátor vygeneruje. Například:

```
@Id
@SequenceGenerator(name="CUST_GEN", sequenceName="SEQ_ID")
@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="CUST_GEN")
@Column(name="CUSTOMER_ID")
private int customerId;
```

- JPA umožňuje generování primárních klíčů několika způsoby:
 - **AUTO**: generování se nechá na implementaci JPA
 - **IDENTITY**: generování se nechá na databázi
 - **SEQUENCE**: vygeneruje primární klíč ze sekvence
 - **TABLE**: vygeneruje primární klíč z tabulky

Životní cyklus entity

- Každá entita má svůj určitý stav, ve kterém se nachází. Stav entity se mění pomocí metod instance třídy EntityManager:



* = Extended persistence context

Základní metody třídy EntityManager

- Předpokládejme, že máme definovanu třídu typu EntityManager s názvem em a entitu s názvem entita. Na instanci třídy EntityManager je možné volat následující metody:
 - `em.persist(entita)`: uloží objekt entita do databáze (operace INSERT)
 - `em.remove(entita)`: smaže objekt entita z databáze (operace DELETE)
 - `em.merge(entita)`: entita byla persistována, ale následně byla změněna. Po operaci merge se tyto změny projeví v databázi (operace UPDATE).
 - `em.find(class, id)`: vrátí objekt v tabulce, která koresponduje s class a má primární klíč id (operace SELECT)

Hello world JPA

- Je nutné vytvořit `persistence.xml` soubor a příslušné entity. V classpath aplikace musí být JDBC ovladač a JPA implementace (například Hibernate). Poté udělejte třídu s metodou `main`:

```
//ziskani entity manazera
EntityManager entityManager = Persistence
    .createEntityManagerFactory("nazev persistentni unity")
    .createEntityManager();

entityManager.find(Customer.class, 2);
```

Transakce


- Všechny operace, které mohou změnit stav databáze (vyvolat operace INSERT / UPDATE / DELETE) musí běžet v transakci:

```
public void saveLektor(Lektor lektor) {  
    entityManager.getTransaction().begin();  
    try {  
        entityManager.persist(lektor);  
        entityManager.getTransaction().commit();  
    } catch (RuntimeException e) {  
        e.printStackTrace();  
    }  
}
```

Spuštění
transakce



Tato operace může
změnit stav databáze
→ musí běžet v transakci



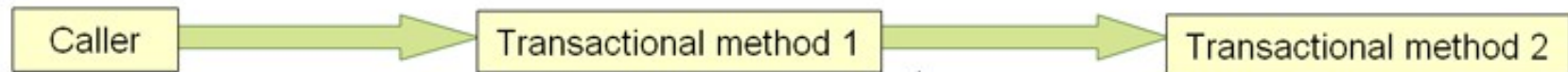
Potvrzení
transakce



Šíření transakcí (propagation) I.

- Definuje způsob šíření transakcí, když jedna transakční metoda zavolá ve svém kódu jinou (vnořenou) transakční metodu. Pro jednotlivé zanořované metody jsou ve Springu vytvářeny logické transakce.
- REQUIRED** (výchozí nastavení) – všechny metody probíhají v jediné fyzické transakci v databázi (start – commit/rollback v databázi). Když logická transakce pro vnitřní metodu nastaví příznak rollbacku, vyvolá se výjimka `UnexpectedRollbackException`, která zabrání vnější transakční metodě, aby dál pokračovala v provádění svého kódu (celá transakce bude zrušena).

REQUIRED

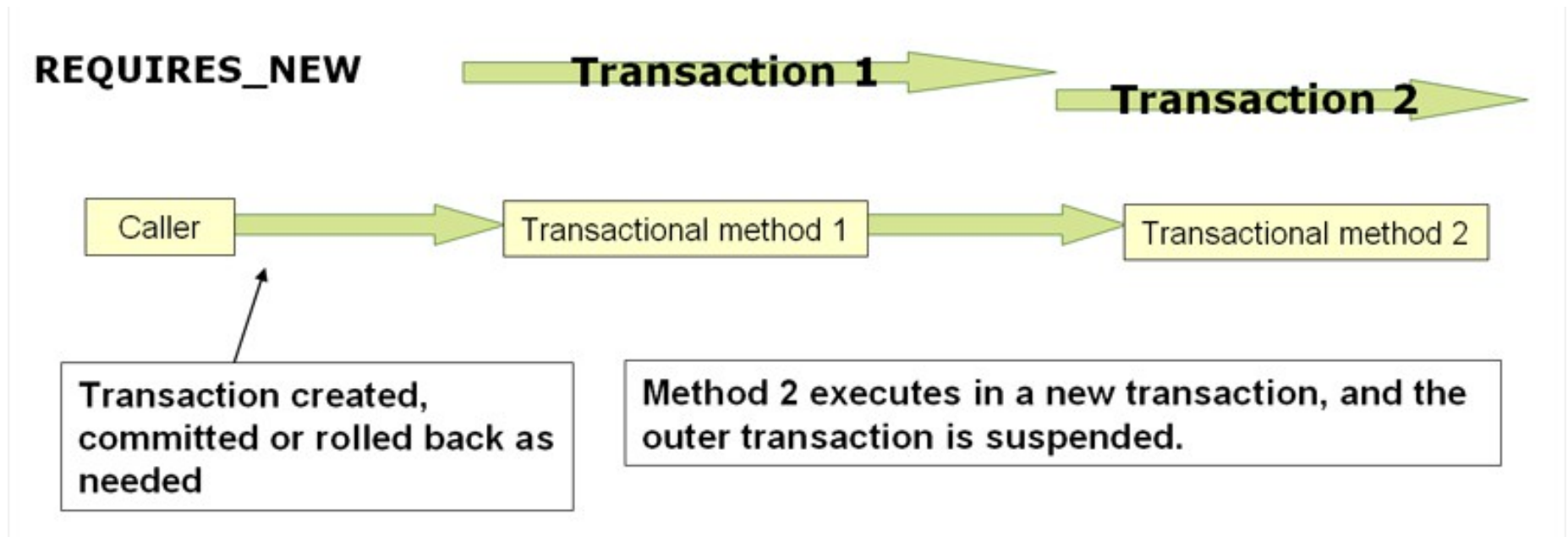


Transaction created,
committed or rolled back as
needed

Method 2 executes in the existing transaction.

Šíření transakcí (propagation) II.

- **REQUIRES_NEW** – pro každou transakční metodu je vytvořena samostatná fyzická transakce v databázi. Vnější logická (a zároveň fyzická) transakce může provést commit nebo rollback nezávisle na způsobu ukončení vnitřní transakce. Toto nastavení umožňuje vnější metodě pokračovat v transakci (se šancí na commit), i když logická (a fyzická) transakce pro vnitřní metodu skončila rollbackem (vnější metoda běží v jiné fyzické transakci).



Metody třídy EntityManager pro získání dat z databáze

- Další metody pro volání JPQL a native query (SQL):
 - `createQuery(jpqlQuery)`: vytvoří dotaz do databáze pomocí JPQL.
 - `createNativeQuery(sqlQuery)`: vytvoří dotaz do databáze pomocí SQL.
 - `createNamedQuery(jpqlQuery nebo sqlQuery)`: vytvoří dotaz do databáze pomocí předuložené šablony JPQL nebo SQL dotazu.



Obvykle se nacházejí definované pomocí anotací u entit

JPQL příklady I.

- Získá všechny zákazníky z databáze:

```
public List<Customer> getCustomers() {  
    return entityManager.createQuery("select c from Customer c",  
        Customer.class).getResultList();  
}
```

- Získá všechny produkty z databáze:

```
public List<Product> getProducts() {  
    return entityManager.createNamedQuery("Product.findAll",  
        Product.class).getResultList();  
}
```

Předpokládá existenci
NamedQuery s názvem
„Product.findAll“:

```
@NamedQuery(name = "Product.findAll",  
    query = "select c from Product c order by c.productId")
```

JPQL příklady II.

- Získá počet produktů v databázi (šlo by také udělat pomocí JPQL):

```
public int getProductsCount() {  
    return (Integer) entityManager.createNativeQuery(  
        "select count(*) from product").getSingleResult();  
}
```

- Získá produkty s cenou vyšší než X:

```
public List<Product> getProductWithCostGreaterThan(int purchaseCost) {  
    return entityManager.createNamedQuery(  
        "Product.findPurchaseCostGreaterThan", Product.class)  
        .setParameter("purchaseCost",  
purchaseCost).getResultList();  
}
```

NamedQuery:

```
@NamedQuery(name = "Product.findPurchaseCostGreaterThan",  
query = "select c from Product c where c.purchaseCost > :purchaseCost")
```