

JPA, Hibernate

# 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
- 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

# Vazby mezi entitami

- Entity mohou mít mezi sebou čtyři typy vazeb:
  - **One to one** – moc často se tento typ vazby nevyskytuje.
    - **Příklad:** Manžel ↔ Manželka, Zaměstnanec ↔ Adresa
  - **One to many, Many to one** – nejčastější typ vazeb.
    - **Příklad:** Kategorie eshopu má více položek, přičemž jedna položka je svázaná s jednou konkrétní kategorií, Zaměstnanec může mít více telefonů, přičemž každý telefon patří právě jednomu zaměstnanci.
  - **Many to many** – speciální případ kombinace One to many a Many to one vazby, kdy v asociační (vazební) tabulce se nachází pouze dva cizí klíče.
    - **Příklad:** Kategorie eshopu má více položek, přičemž jedna položka se může vyskytovat ve více kategoriích.

**Poznámka:** Vazby mohou být jednosměrné, nebo obousměrné (obvyklé nastavení).

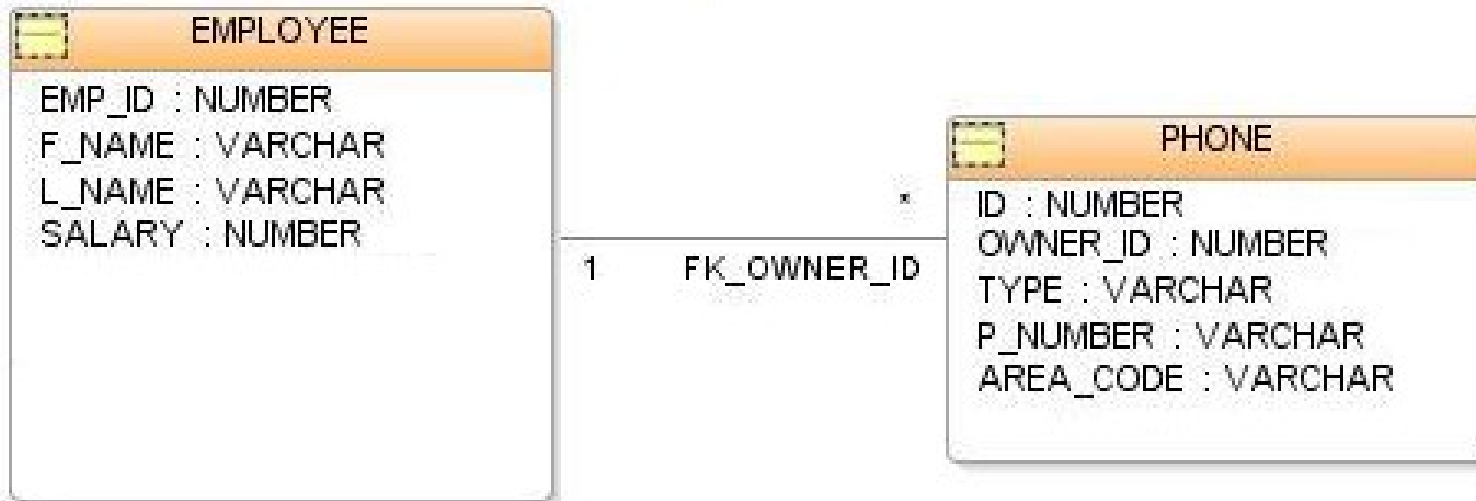
# @OneToOne



```
@Entity
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private Integer id;
    ...
    @OneToOne
    @JoinColumn(name="ADDRESS_ID")
    private Address address;
    ...
}
```

```
@Entity
public class Address {
    @Id
    @Column(name="ADDRESS_ID")
    private Integer id;
    ...
    @OneToOne(mappedBy="address")
    private Employee owner;
    ...
}
```

# @OneToMany, @ManyToOne



```
@Entity
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private Integer id;
    ...
    @OneToMany(mappedBy="owner")
    private List<Phone> phones;
    ...
}
```

```
@Entity
public class Phone {
    @Id
    private Integer id;
    ...
    @ManyToOne
    @JoinColumn(name="OWNER_ID")
    private Employee owner;
    ...
}
```

# @ManyToMany

```
@Entity
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private long id;
    ...
    @ManyToMany
    @JoinTable(
        name="EMP_PROJ",
        joinColumns={@JoinColumn(name="EMP_ID",
            referencedColumnName="EMP_ID")},
        inverseJoinColumns={@JoinColumn(name="PROJ_ID",
            referencedColumnName="PROJ_ID")})
    private List<Project> projects;
    ...
}
```

```
@Entity
public class Project {
    @Id
    @Column(name="PROJ_ID")
    private long id;
    ...
    @ManyToMany(mappedBy="projects")
    private List<Employee> employees;
    ...
}
```

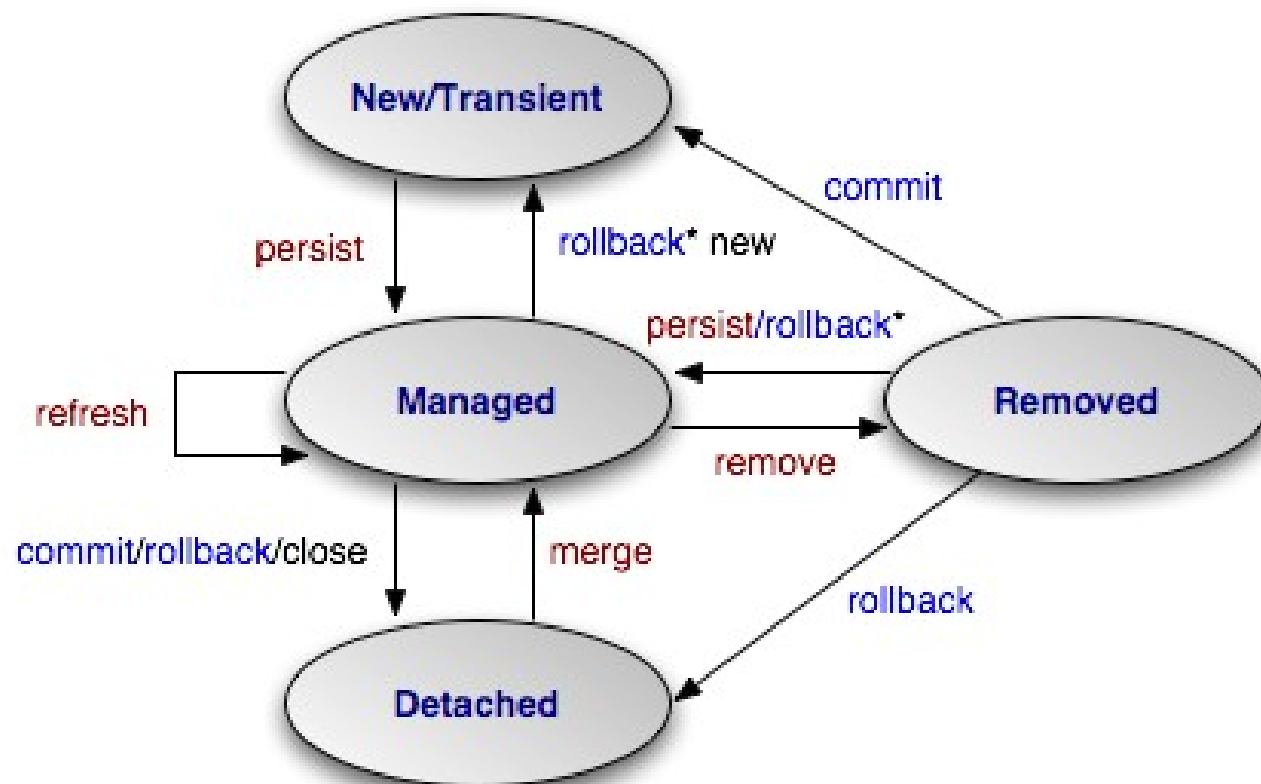


# List vs. Set vs. ...

- Nejčastěji používané typy kolekcí ve vazbách jsou:
  - `java.util.List` – utříděná kolekce, ve které jsou prvky přístupné přes index.
  - `java.util.Set` – neutříděná kolekce, ve které jsou unikátní objekty.
  - `java.util.SortedSet` – utříděná kolekce s unikátními objekty.
  - a další ... (viz. dokumentace)

# 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 definovanou 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);
```

# JPA + Spring

- Data z JPA získáte následujícím způsobem:

@Service

```
public class AppService {
```

Injektne instanci  
třídy EntityManager

```
@PersistenceContext
```

```
private EntityManager entityManager;
```

```
public List<Customer> listCustomers() {
```

```
    return entityManager.createQuery("select c from Customer c",  
                                     Customer.class).getResultList();
```

```
}
```

```
@Transactional
```

Obalí tuto metodu transakcí - před spuštěním metody se zahájí transakce a po ukončení metody se provede commit. Pokud se v metodě vyhodí neošetřená výjimka typu RuntimeException, pak se zavolá rollback

```
public void addCustomer(Customer customer) {
```

```
    entityManager.persist(customer);
```

```
}
```

```
}
```

@Transactional je také možné uvést před názvem třídy, pak mají podporu transakcí všechny metody. V tom případě je vhodné u metod, které nemění stav databáze přidat: @Transactional(readonly = true)

# Spring Data JPA + EntityManager

- Správný způsob jak pracovat s EntityManager u Spring Data JPA:
  - <https://dzone.com/articles/accessing-the-entitymanager-from-spring-data-jpa>
- Více datasourců:
  - <https://www.baeldung.com/spring-data-jpa-multiple-databases>

# 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

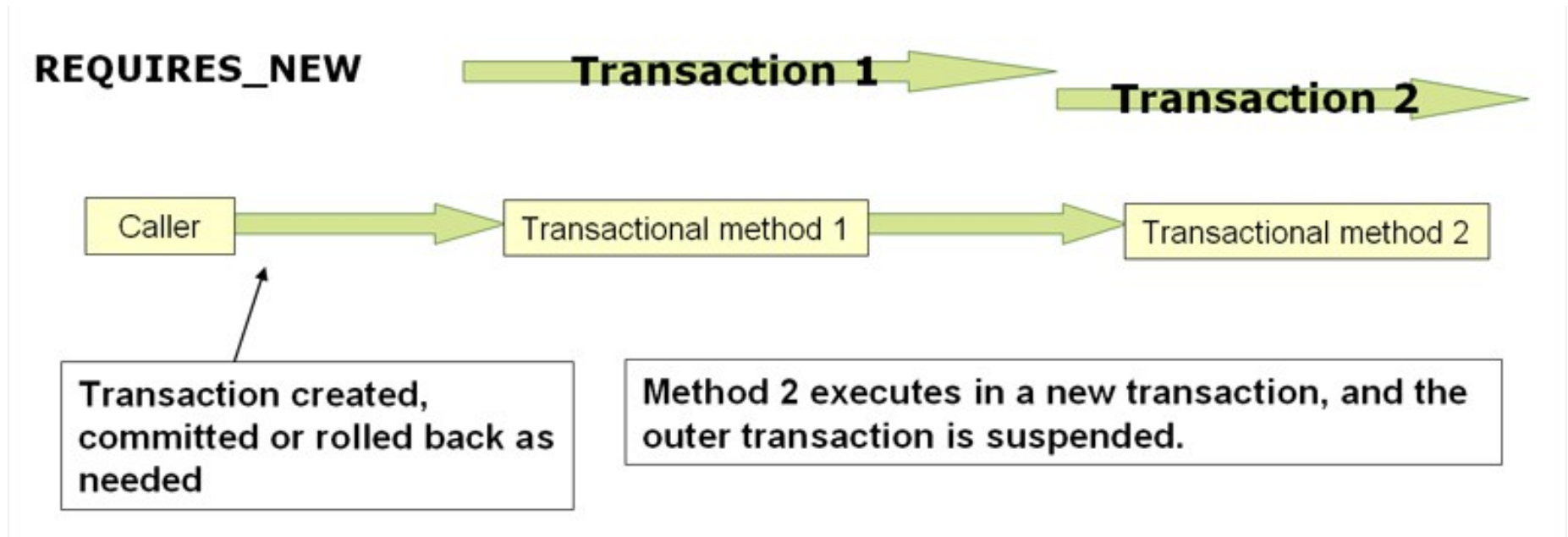


Transaction created,  
committed or rolled back as  
needed

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).





# Vlastní @Transactional anotace

- Velice často se vytvářejí vlastní @Transactional anotace, které se používají místo výchozí @Transactional anotace:

```
@Transactional(rollbackFor = Exception.class, readOnly = true)
```

```
@Target({ ElementType.METHOD, ElementType.TYPE })
```

```
@Retention(RetentionPolicy.RUNTIME)
```

```
@Inherited @Documented
```

```
public @interface TransactionalRO { }
```

```
@Transactional(rollbackFor = Exception.class)
```

```
@Target({ ElementType.METHOD, ElementType.TYPE })
```

```
@Retention(RetentionPolicy.RUNTIME)
```

```
@Inherited @Documented
```

```
public @interface TransactionalRW { }
```

# DTO (Data Transfer Object)

- Další způsob řešení problémů s donáčením záznamů v prezentační vrstvě je, že se v prezentační vrstvě nebudou používat entity, ale DTO objekty.
  - V servisní vrstvě, kde máte otevřenou transakci načtete data z databáze, provedete jejich transformaci na DTO objekty a s nimi pracujete v prezentační vrstvě.
- Je několik způsobů jak vytvářet DTO objekty. Pro jejich jednoduchou tvorbu se velice často používá mapovací framework MapStruct:
  - <https://mapstruct.org/>

# Doporučená literatura

