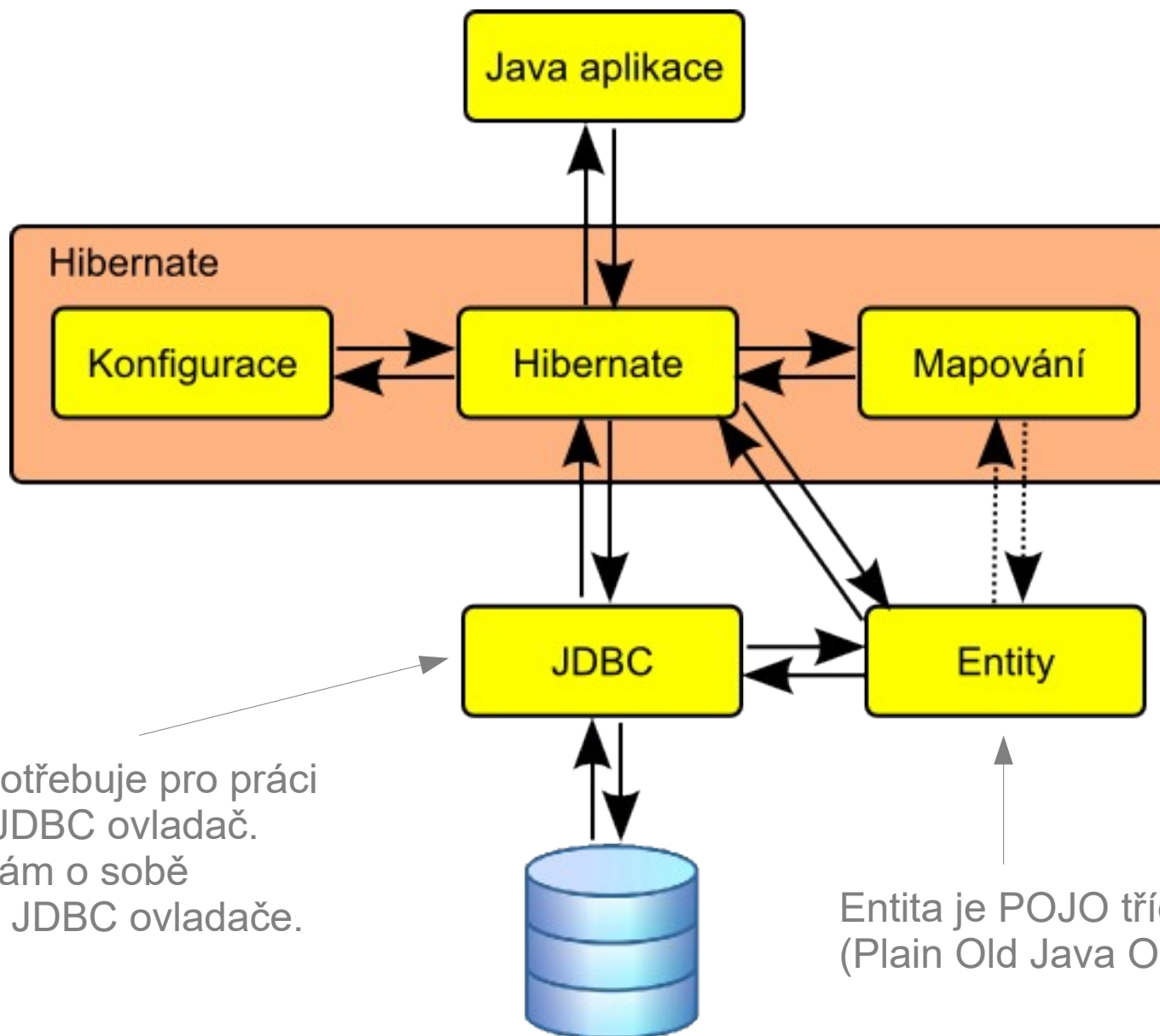


JPA, Hibernate

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 (aktuálně již JPA 2.0).
- Populární ORM frameworky, které implementují JPA 2.0:
 - JBoss Hibernate
 - EclipseLink
 - OpenJPA

Role Hibernate v Java aplikaci



Hibernate potřebuje pro práci s databází JDBC ovladač. Hibernate sám o sobě neobsahuje JDBC ovladače.

Entita je POJO třída (Plain Old Java Object)

Tvorba entit / tabulek

- V praxi existují dvě situace:
 - A) Máte již existující databázi s tabulkami a potřebujete s ní pracovat pomocí JPA – tzn. vytvořit entity. Můžete je vytvořit ručně, nebo je s pomocí nástrojů vygenerovat.
 - B) Potřebujete vytvořit projekt „na zelené louce“ - nemáte žádnou databázi. Buď ji můžete vytvořit a pak vygenerovat entity, nebo můžete vytvořit entity a podle nich vygenerovat tabulky v databázi.

Práce s JPA a Hibernate

1. Konfigurace

- a) Vytvoření souboru `hibernate.properties` nebo `hibernate.cfg.xml` (starší způsob)
- b) Vytvoření souboru `META-INF/persistence.xml` (novější způsob)

2. Vytvoření entit a jejich mapování

- a) Pomocí XML (starší způsob) ←
 - b) Pomocí anotací (novější způsob) →
- Obojí lze navíc hibernate-specifickým způsobem, nebo JPA způsobem

3. Práce s Hibernate:

- a) Pomocí instance třídy `Session` (starší způsob)
- b) Pomocí instance třídy `EntityManager` (novější způsob)

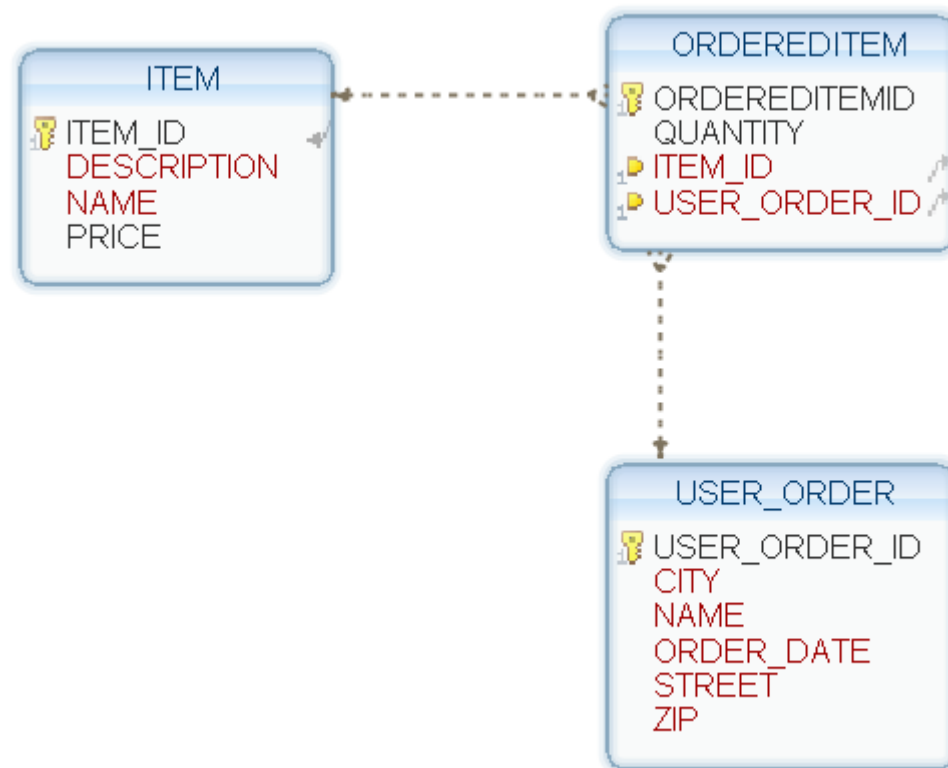
4. Získání dat z databáze:

- a) Pomocí HQL (starší způsob)
- b) Pomocí JPQL (novější způsob)

Poznámka: Všechny uvedené způsoby se dají kombinovat

Databáze eshop

- Máme databázi s následujícím ER diagramem:



Hibernate & Maven

- Je nutné přidat do pom.xml následující závislost (plus JDBC ovladač příslušné databáze):

```
<dependency>  
    <groupId>org.hibernate</groupId>  
    <artifactId>hibernate-entitymanager</artifactId>  
    <version>4.3.6.Final</version>  
</dependency>
```

- Pozn: V případě úplně nejnovější verze Hibernate je občas ještě nutné přidat JBoss repozitář:

```
<repositories>  
    <repository>  
        <id>JBoss repository</id>  
        <url>http://repository.jboss.org/nexus/content/groups/public/</url>  
    </repository>  
</repositories>
```

Entita I.

- 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`.
- Je best-practice, aby entitní třída implementovala rozhraní `Serializable`. Není to ale nutné.

Entita II.

- Poznámky:
 - Můžete mít více entit namapovaných na stejnou tabulku.
 - Entita nemusí být namapovaná jenom na tabulku, ale i na view.
 - Entita nemusí obsahovat mapování všech sloupců, ale i jejich podmnožinu.

Konfigurace

- Existuje několik způsobů jak nakonfigurovat připojení do databáze ... nejlepší způsob je pomocí Springu ... nejjednodušší způsob je pomocí souboru:

src/main/resources/META-INF/persistence.xml

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

<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

  <persistence-unit name="eshop" transaction-type="RESOURCE_LOCAL">

    <class>helloworld.entity.Item</class>

    <properties>

      <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver" />

      <property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:hsql://localhost/eshop" />

      <property name="javax.persistence.jdbc.user" value="sa" />

      <property name="javax.persistence.jdbc.password" value="" />

      <property name="hibernate.show_sql" value="true" />

    </properties>

  </persistence-unit>

</persistence>
```

Hello world JPA

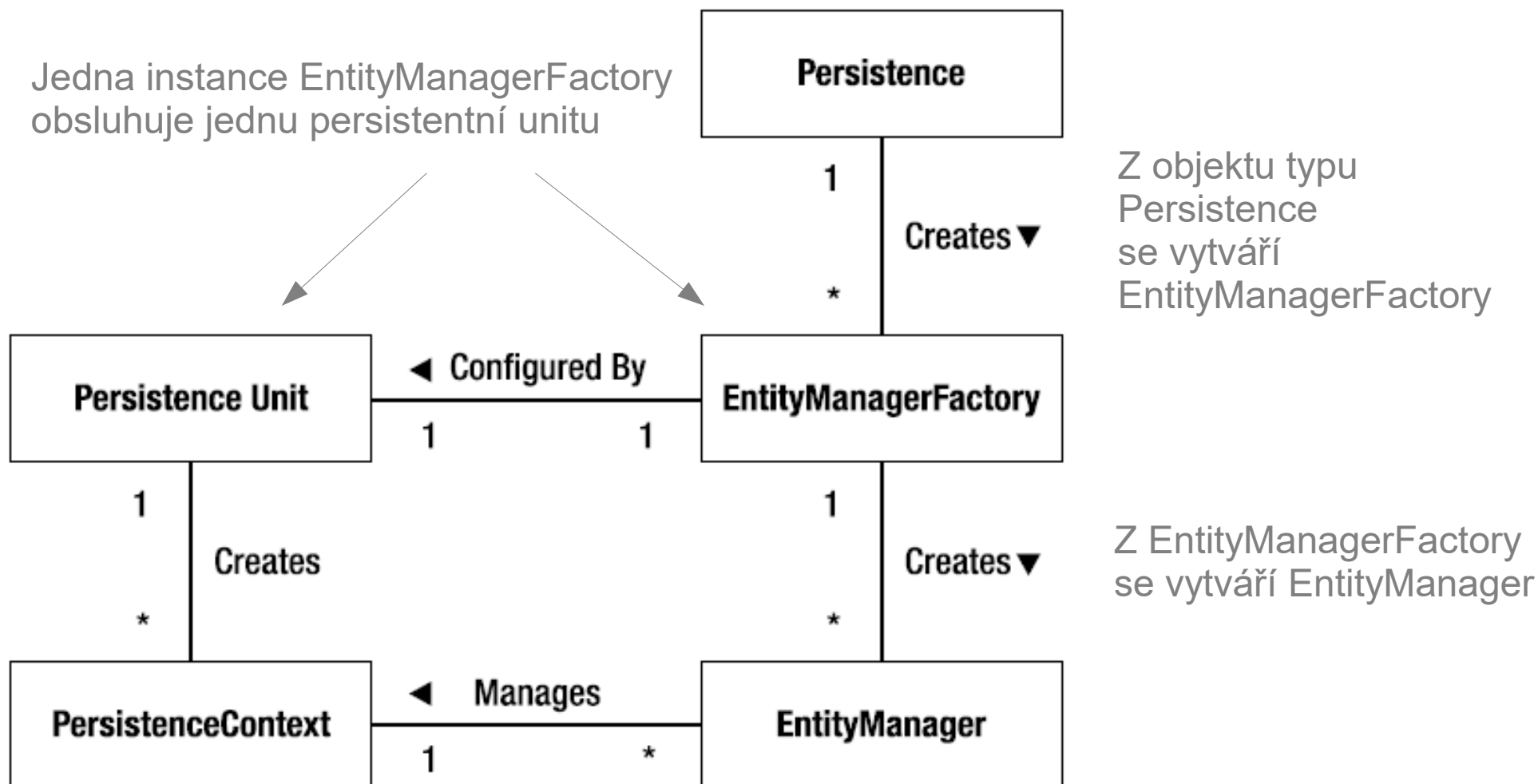
- Je nutné vytvořit `persistence.xml` soubor a příslušné entity. V classpath aplikace musí být JDBC ovladač a JPA implementace (například Hibernate). Poté udělejte třídu s metodou `main`:

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

entityManager.find(Item.class, 1);
```

- Vytvoření instance třídy `EntityManagerFactory` je časově náročná operace (řádově stovky milisekund) – obdobně jako vytvoření `DataSource` u JDBC.
- Vytvoření instance třídy `EntityManager` je časově nenáročná operace – obdobně jako vytvoření `Statement` u JDBC.

Vztahy mezi třídami



Vytvoření databáze I.

- Je možné ručně vytvořit entity a poté automaticky vygenerovat databázi následujícími způsoby:
 - Buď v Eclipse kliknout pravým tlačítkem na projekt a vybrat JPA Tools → Generate Tables from Entities ...
 - Nebo do souboru WEB-INF/persistence.xml přidat do tagu properties:

```
<property name="hibernate.hbm2ddl.auto" value="create" />
```

- Pozor! Poté se při každém startu aplikace smažou a znovu vytvoří tabulky v databázi! Po dosažení požadované struktury databáze je nutné tuto property smazat nebo zapoznámkovat ;-)
- Další zajímavou hodnotou je update, která nemaže celou databázi, ale provádí v ní incrementální změny (při přidání atributu do entity vyvolá ALTER TABLE ADD COLUMN apod.)

Poznámka: Pokud používáte konfigurační soubor hibernate.cfg.xml, pak správný tvar této property je následující:

```
<property name="hbm2ddl.auto" value="create" />
```

Vytvoření databáze II.

- Hibernate 4.3 implementuje JPA 2.1, kde je generování databáze na základě entit standardizované.
- Původně se používalo:

```
<property name="hibernate.hbm2ddl.auto" value="create" />
```

- Nyní je best practice používat:

```
<property name="javax.persistence.schema-generation.database.action"  
value="create" />
```

- Jenže neumí hodnotu „update“ :-)
- Další nové JPA 2.1 properties týkající se generování databáze:
 - https://blogs.oracle.com/arungupta/entry/jpa_2_1_schema_generation

@Entity, @Table

- To, že je třída entitou definujeme pomocí anotace `@Entity` před definicí třídy:

```
@Entity
@Table(name="USER_ORDER")
public class UserOrder implements Serializable {
    private static final long serialVersionUID = 1L;
    ...
}
```

- Pokud se entita jmenuje jinak než tabulka, použijeme anotaci `@Table`. Používá se zde přístup „convention over configuration“.
- V `persistence.xml` souboru musí být ještě informace o tom, že tato entita je součástí konkrétní persistentní unity:

```
<class>cz.skoleni.java.entities.UserOrder</class>
```

Poznámka:

Existuje několik způsobů, aby toto nebylo nutné. Nejjednodušší je při použití Spring 3.1, kde se `persistence.xml` soubor nemusí používat vůbec a v `LocalContainerEntityManagerFactoryBean` se použije atribut `packagesToScan`

@Column, @Transient

- Standardně se předpokládá, že pro všechny atributy entity existuje v tabulce sloupec se stejným názvem jako je název atributu (opět podle přístupu „convention over configuration“).
- Pokud se atribut jmenuje jinak než je název sloupce, použijeme anotaci @Column:

```
@Column(name="USER_ORDER_ID")
```

```
private int userOrderId;
```

- Pokud naopak nechceme aby byl atribut persistován do databáze, použijeme anotaci @Transient:

```
@Transient
```

```
private File tmpFile;
```

Poznámka: Je možné také použít klíčové slovo transient

@Temporal, @Lob

- Pokud je v databázi sloupec s typem DATE, je možné před atribut v entitě uvést, jestli se jedná o datum, čas, nebo jejich kombinaci pomocí anotace @Temporal:

```
@Temporal( TemporalType.DATE)
```

```
private Date orderDate;
```

Obvykle java.util.Date

Mohou zde být hodnoty:

- TemporalType.*DATE*
- TemporalType.*TIME*
- TemporalType.*TIMESTAMP*

- Pokud je v databázi sloupec s jedním ze dvou typů:
 - **BLOB (Binary Large Object)** – například obrázek
 - **CLOB (Character Large Object)** – například XML soubor
 - Použijeme anotaci @Lob před atribut typu byte [], nebo char [] (nebo String):

```
@Lob
```

```
private byte[] icon;
```

V databázi
je BLOB

```
@Lob
```

```
private String xmlFile;
```

V databázi
je CLOB

PostgreSQL DB CLOB

- V PostgreSQL databázi je nutné atributy typu TEXT (CLOBy) vytvářet následovně:

```
@Lob
```

```
@Type(type = "org.hibernate.type.TextType")
```

```
@Column(length = Integer.MAX_VALUE)
```

```
private String value;
```

@Basic

- Pro získání entity z databáze musí JPA zavolat SQL SELECT, ve kterém získá všechny data z databáze a uloží je do entity. Někdy ale nechcete získat všechna data ... typicky v případě LOB objektů. K tomu slouží anotace @Basic, u které se nastaví FetchType.LAZY (výchozí nastavení je FetchType.EAGER):

```
@Basic(fetch=FetchType.LAZY)
```

```
@Lob
```


```
private byte[] icon;
```

Identifikátor entity

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

```
@Id
@SequenceGenerator(name="ITEM_GEN", sequenceName="SEQ_ITEM")
@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="ITEM_GEN")
@Column(name="ITEM_ID")
private int itemId;
```

Entity mohou mít atributy, které jsou primitivními datovými typy. Nebo wrapper class. Ty mají navíc výhodu, že mohou nabývat hodnoty null.



- 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

Trigger pro generování primárních klíčů

- Co když používáte triggery i pro generování primárních klíčů? Použijte toto rozšíření Hibernate:

```
@Entity

public class Employee {

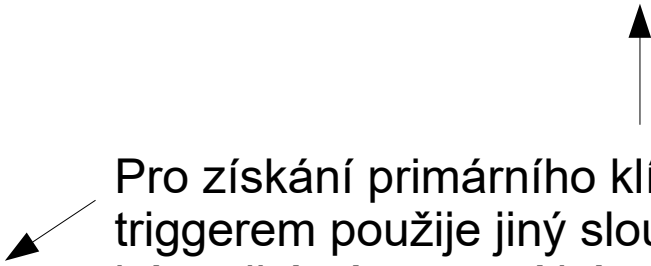
    @GeneratedValue(generator = "trigger")
    @GenericGenerator(name = "trigger",
                      strategy = "org.hibernate.id.SelectGenerator",
                      parameters = { @Parameter(name = "key", value = "email") })

    @Id
    @Column(name = "employee_id")
    private Integer id;

    @Column(unique=true, nullable=false)
    private String email;

    // TODO gettery a settery
}
```

Pro získání primárního klíče vygenerovaného triggerem použije jiný sloupec, který musí být unikátní a nesmí být null.



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).

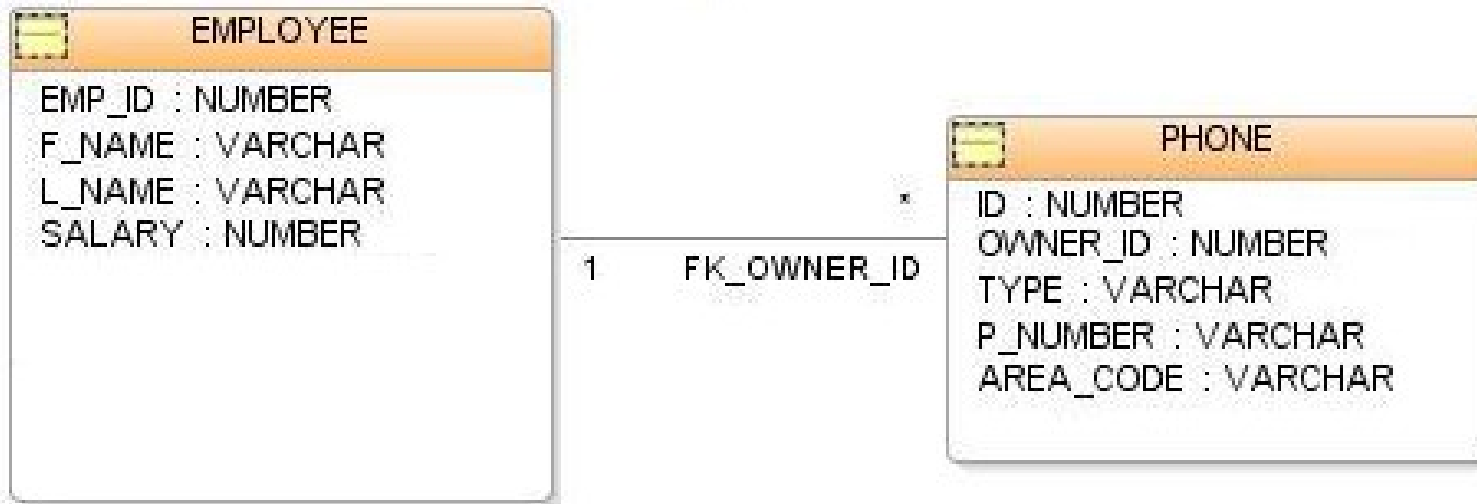
@OneToOne



```
@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)

Cascade

- Mapování vazeb má atribut cascade, pomocí kterého je možné nastavit kaskádování na operacích:
 - **PERSIST** – při zavolání metody `persist()` třídy `EntityManager` na entitě, která obsahuje odkazy na nové entity budou tyto nové entity také persistovány do databáze.
 - **REMOVE** – obdoba PERSIST. Pozor na to, že když pouze smažete objekt z kolekce s anotací `@OneToMany`, tak tím nevyvoláte operaci `remove()` na takovém objektu!
 - **MERGE** – obdoba PERSIST
 - **REFRESH** – obdoba PERSIST
 - **ALL** – kaskádování na všech výše uvedených operacích.
- Orphan removal
 - Na `@OneToMany` a `@ManyToOne` vazbách můžete nastavit atribut `orphanRemoval` na `true`, což má za následek automatické smazání příslušné entity z druhé strany vazby.

Lazy / eager fetching, @OrderBy

- Při získávání dat z databáze musíme rozhodnout, jak hodně dat se bude z databáze získávat (jestli se při požadavku na získání zákazníků načtou rovnou i informace o jejich adresách a telefonech z jiných tabulek). K tomu se používá atribut fetch, který je možné nastavit při mapování vazeb.
- Výchozí nastavení pro mapování vazeb je FetchType.LAZY kromě vazeb OneToOne a ManyToOne, kde je výchozí nastavení FetchType.EAGER.

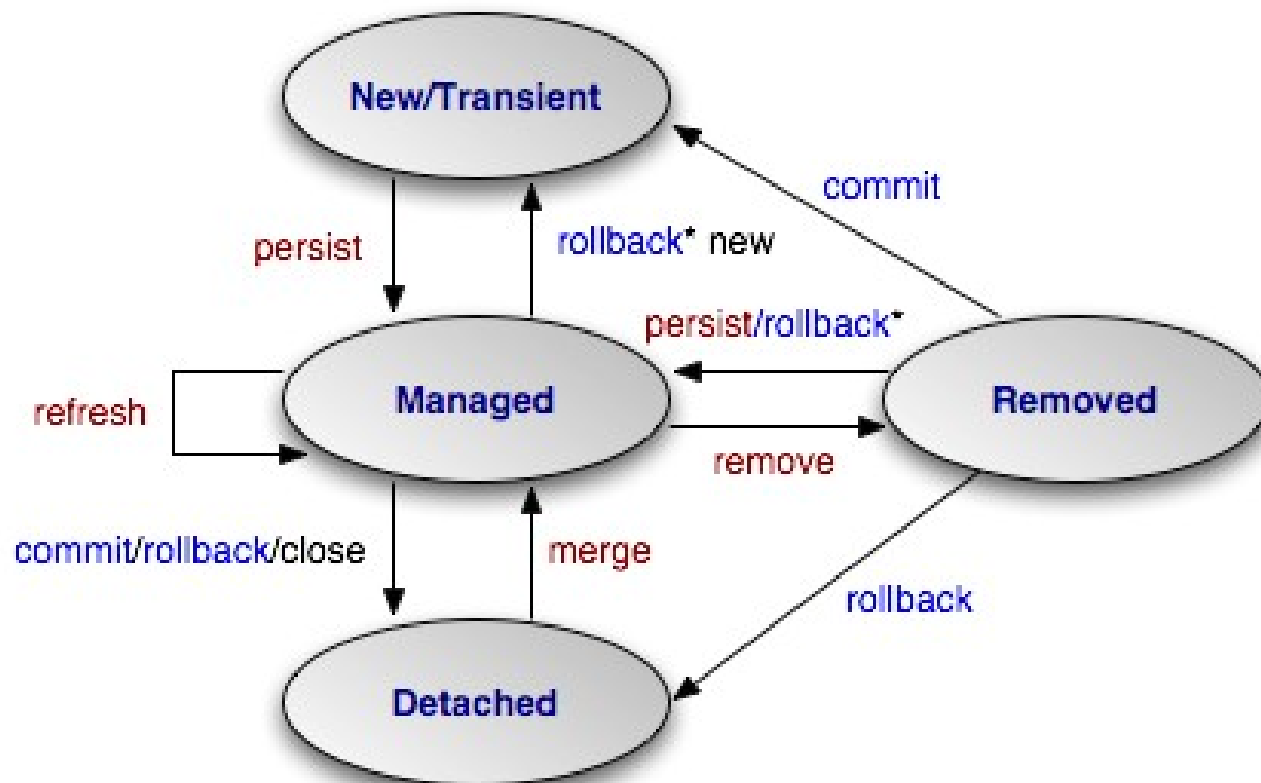
`@OneToOne(fetch=FetchType.LAZY)`

- JPA umožňuje automaticky seřadit kolekci při získávání dat z databáze pomocí anotace @OrderBy:

```
@Entity
public class UserOrder {
    @Id
    private int userOrderId;
    ...
    @OneToMany
    @OrderBy("quantity desc")
    private List<OrderedItem>
        orderedItems;
    ...
}
```

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

EntityManager vs. Session

- EntityManager je třída pro práci s JPA. Je doporučeno ji používat v maximální míře.
- Občas je zapotřebí pracovat s rozšířením Hibernate (například volání HQL – Hibernate Query Language nebo Hibernate Criteria), k tomu je zapotřebí získat objekt typu Session (pokud je ovšem Hibernate používanou implementací JPA):

```
Session session = entityManager.unwrap(Session.class);
```

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)
 - **em.refresh(class):** provede aktualizaci stavu entity a přitom přepíše všechny změny, které na ní byly provedeny (operace SELECT)
 - **em.flush():** synchronizuje persistentní kontext s databází (vynutí vykonání SQL operací INSERT, UPDATE anebo DELETE)

Trigger

- Pokud používáte Oracle databázi a triggery, které například uloží časové razítko uložení záznamu do databáze, pak ho získáte touto kombinací následujících metod:

```
entityManager.persist(employee);
```

Chcete ukládat nový záznam do databáze

```
entityManager.flush();
```

Vynutí INSERT operaci (po které se vyvolá trigger)

```
entityManager.refresh(employee);
```

Zavolá SELECT do databáze a získá data,
která byla vložena triggerem

Transakce


- Všechny operace, které mohou změnit stav databáze (vyvolat operace INSERT / UPDATE / DELETE) musí běžet v transakci. Bez použití Springu nebo EJB je nutné řídit transakce ručně:

```
public void saveItem(Item item) {  
    entityManager.getTransaction().begin();  
  
    try {  
        entityManager.persist(item);  
  
        entityManager.getTransaction().commit();  
    } catch (RuntimeException e) {  
        e.printStackTrace();  
    }  
}
```

Spuštění
transakce



Tato operace může
změnit stav databáze
→ musí běžet v transakci



Potvrzení
transakce




Spring & JPA: Spring konfigurace


- Existuje několik možností konfigurace JPA ve Springu, od nejjednodušších, které jsou vhodné pro testování po pokročilejší, které jsou vhodné na produkci.
- Změna konfigurace je pouze změnou v konfiguračním souboru Springu, není nutná žádná změna v kódu v aplikaci.
- Kromě zapojení JPA je také možné použít starší způsob zapojení Hibernate v aplikaci (bez JPA), který zde nebude probírán.

Spring & JPA: Spring konfigurace I.

- První konfigurace JPA je pomocí třídy `LocalEntityManagerFactoryBean` a je vhodná pouze pro testování nebo velice jednoduché aplikace. Výsledkem je instance třídy `EntityManagerFactory`:

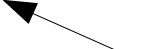
META-INF/persistence.xml:  Název persistentní unity

```
<persistence-unit name="eshop" transaction-type="RESOURCE_LOCAL">  
  <properties>  
    <property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:hsqldb://localhost/eshop" />  
    <property name="javax.persistence.jdbc.user" value="sa" />  
    <property name="javax.persistence.jdbc.password" value="" />  
    <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver" />  
  </properties>  
</persistence-unit>
```

 Konfigurace připojení je uvnitř persistence.xml

Spring konfigurace:

```
<bean id="myEmf" class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">  
  <property name="persistenceUnitName" value="eshop" />  
</bean>
```

 Název persistentní unity

Spring & JPA: Spring konfigurace II.

- Dalším ze způsobů je získání instance EntityManagerFactory z Java EE 5 serveru přes JNDI:

```
<jee:jndi-lookup id="myEmf" jndi-name="persistence/eshop" />
```



Název persistentní unity

- V tomto případě má Java EE 5 server na starosti připojení k databázi a správu entit. Spring pouze používá instanci EntityManagerFactory, injectuje ji do tříd a spravuje transakce.

Spring & JPA: Spring konfigurace III.

- Nejpoužívanějším způsobem získání instance EntityManagerFactory je pomocí LocalContainerEntityManagerFactoryBean:

```
<persistence-unit name="eshop" transaction-type="RESOURCE_LOCAL">
</persistence-unit>
```

Persistentní unita

```
<bean id="myDataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="url" value="jdbc:hsqldb:hsqldb://localhost/eshop" />
  <property name="username" value="sa" />
  <property name="password" value="" />
</bean>
```

DataSource, může být získán z Java EE serveru přes JNDI:

```
<jee:jndi-lookup id="myDataSource" jndi-name="jdbc/hsqldb" />
```

```
<bean id="myEmf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="myDataSource" />
  <property name="persistenceUnitName" value="eshop" />
  <property name="jpaVendorAdapter">
    <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
      <property name="databasePlatform" value="org.hibernate.dialect.HSQLDialect" />
    </bean>
  </property>
</bean>
```

Název persistentní unity

Spring & JPA: Spring konfigurace IV.

- Varianta BEZ persistence.xml souboru (od Spring 3.1):

```
<bean class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">  
  <property name="packagesToScan" value="com.test.entity" />  
  <property name="dataSource" ref="myDataSource" />  
  <property name="persistenceProviderClass" value="org.hibernate.ejb.HibernatePersistence" />  
  <property name="jpaVendorAdapter">  
    <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />  
  </property>  
</bean>
```

V jakém balíčku se nachází entity

Reference na DataSource

Spring & JPA: Java Config I.

```
@EnableTransactionManagement
```

```
@Configuration
```

```
public class SpringConfiguration {
```

```
    @Bean
```

```
    public BasicDataSource dataSource() {
```

```
        BasicDataSource dataSource = new BasicDataSource();
```

```
        dataSource.setUrl("jdbc:hsqldb:hsqldb://localhost/eshop");
```

```
        dataSource.setUsername("sa");
```

```
        dataSource.setPassword("");
```

```
        return dataSource;
```

```
    }
```

Spring & JPA: Java Config II.

@Bean

```
public JpaTransactionManager transactionManager(DataSource dataSource,
    EntityManagerFactory entityManagerFactory) {
    JpaTransactionManager transactionManager
        = new JpaTransactionManager(entityManagerFactory);
    transactionManager.setDataSource(dataSource);
    return transactionManager;
}
```


Spring & JPA: Java Config III.

@Bean

```
public LocalContainerEntityManagerFactoryBean emf() {  
    LocalContainerEntityManagerFactoryBean emf  
        = new LocalContainerEntityManagerFactoryBean();  
    // díky tomuto nastavení se nebude používat persistence.xml!!!  
    emf.setPackagesToScan("cz.javaskoleni.helloworld.entity");  
    emf.setDataSource(dataSource());  
    emf.setJpaVendorAdapter(new HibernateJpaVendorAdapter());  
    Properties properties = new Properties();  
    properties.setProperty("hibernate.show_sql", "true");  
    properties.setProperty("hibernate.format_sql", "true");  
    emf.setJpaProperties(properties);  
    return emf;  
}
```

Spring & JPA: Transakce

- Ještě je nutné zapnout podporu transakcí buď pomocí anotací, nebo pomocí AOP konfigurace (viz. dokumentace):

Spring konfigurace:

```
<tx:annotation-driven />
```

```
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
```

```
    <property name="entityManagerFactory" ref="myEmf" />
```

```
    <property name="dataSource" ref="myDataSource" />
```

```
</bean>
```

hibernate.cfg.xml

- Při práci s JPA si nemusíme vystačit pouze se standardem, ale můžeme použít i Hibernate rozšíření, která se vkládají do souboru obvykle pojmenovaném hibernate.cfg.xml.
- Jednoduše vytvořte soubor hibernate.cfg.xml a přidejte do souboru persistence.xml do tagu properties následující kód:

```
<property name="hibernate.ejb.cfgfile" value="hibernate.cfg.xml" />
```

Spring & JPA: Datová vrstva I.

- Data z JPA získáte následujícím způsobem:

@Repository

```
public class ItemRepository {
```

@PersistenceContext

```
private EntityManager entityManager;
```

```
public List<Item> listItems() {
```

```
    return entityManager.createQuery("select i from Item i",  
                                    Item.class).getResultList();
```

```
}
```

```
public void addItem(Item item) {
```

```
    entityManager.persist(item);
```

```
}
```

```
}
```

Tip: Je možné také injectnout instanci EntityManagerFactory:

@PersistenceUnit

```
private EntityManagerFactory entityManagerFactory;
```

Injektne instanci
třídy EntityManager

Spring & JPA: Datová vrstva II.

- Doporučuji se podívat na framework Spring Data JPA, který ještě více zjednodušuje práci s databází:
 - <http://projects.spring.io/spring-data-jpa/>
 - <http://spring.io/guides/gs/accessing-data-jpa/>

Spring & JPA: Servisní vrstva

@Service

```
public class ItemService {
```

```
    @Autowired
```

```
    private ItemRepository itemRepository;
```

```
    public List<Item> listItems() {
```

```
        return itemRepository.listItems();
```

```
    }
```

```
    @Transactional
```

```
    public void addItem(Item item) {
```

```
        itemRepository.addItem(item);
```

```
    }
```

```
}
```

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

@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)

Integrační testy

- Spring výrazně zjednodušuje tvorbu integračních testů. Nejprve je nutné mít v `pom.xml` tyto závislosti:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <version>${spring.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>${junit.version}</version>
  <scope>test</scope>
</dependency>
```

- **Poznámka:** Pokud to je možné, tak se integračním testům vyhněte (typicky mockováním):
 - <https://code.google.com/p/mockito/>
- Když budete vymýšlet jak testovat databázi, tak by Vás mohl zajímat tento článek:
 - <http://blog.kolman.cz/2012/04/jak-na-integracni-testy-s-databazi.html>

Spring & JPA: JUnit test I.

- JUnit test je při práci s JPA i Hibernate stejný:

```
@RunWith(SpringJUnit4ClassRunner.class)
```

Vytvoří při startu JUnit
test třídy Spring context

```
@ContextConfiguration(locations = "classpath:jpa-app.xml")
```

```
@Transactional
```

Testy budou běžet v transakci,
na konci každé metody se provede rollback

Konfigurace Springu

```
public class AppServiceIT {
```

```
@Autowired private AppService appService;
```

Integrační testy nemají příponu Test,
ale IT (Integration Test). Testy s příponou Test
se spouští ve fázi „test“ (čili také když dáte
mvn package). Testy s příponou IT se spouští
pomocí mvn verify

```
@Test public void testGetItems() {
```

```
    int count = appService.getItems().size();
```

```
    assertEquals(8, count);
```

```
}
```

```
@Test public void testAddEmployee() {
```

```
    int originalCount = appService.getItems().size();
```

```
    appService.addItem(new Item());
```

```
    assertEquals(originalCount + 1, appService.getItems().size());
```

```
}
```

```
}
```


Spring & JPA: JUnit test II.

- Před třídu testu je možné přidat anotaci `@TransactionConfiguration`, ve které je možné specifikovat `transactionManager` a jestli bude `rollback` výchozí operací po skončení transakce.

- Při tomto nastavení:

```
@TransactionConfiguration(defaultRollback=false)
```

- Se standardně provede po skončení transakční metody operace `commit`. `Rollback` je poté možné provést na úrovni metod anotací:

```
@Rollback(true)
```

- Metoda s anotací `@Before` běží uvnitř transakce. Pokud chcete před testem spustit kód ještě před transakcí, vytvořte metodu s anotací `@BeforeTransaction`. Anotace `@After` a `@AfterTransaction` fungují obdobným způsobem.

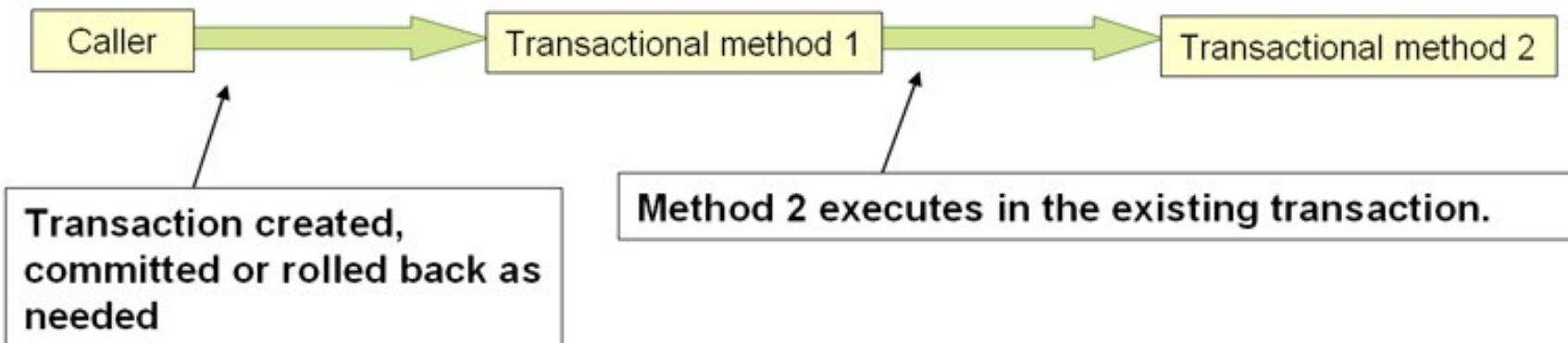
JPA + EJB

- Pomocí anotace `@TransactionManagement` je možné určit, jestli budou transakce řízeny kontejnerem (výchozí nastavení), nebo kódem v beaně.
- Pomocí anotace `@TransactionAttribute` je možné určit rozsah transakce (jako u Springu anotace `@Transactional`).
- Obě anotace je možné použít jak u třídy, tak u metody. Použitím anotace u metody se „přepíše“ případné výchozí nastavení u třídy.

Šíření transakcí (propagation) I.

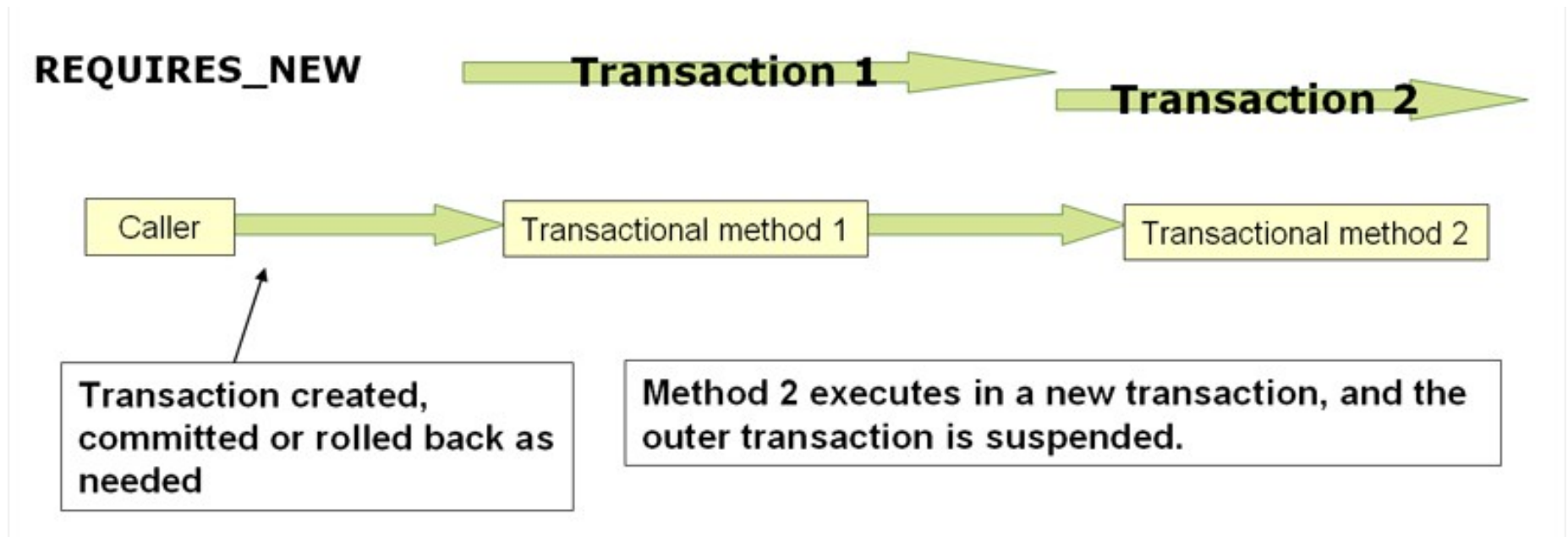
- Definuje způsob šíření transakcí, když jedna transakční metoda zavolá ve svém kódu jinou (vnořenou) transakční metodu. Pro jednotlivé zanořované metody jsou vytvářeny logické transakce.
- REQUIRED** (výchozí nastavení) – všechny metody probíhají v jediné fyzické transakci v databázi (start – commit/rollback 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



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 { }
```

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, ale mohou být i v XML souboru

JPQL příklady I.

- Získá všechny produkty z databáze:

```
public List<Item> getItems() {  
    return entityManager.createQuery("select i from Item i",  
        Item.class).getResultList();  
}
```

Taky se dá použít
plný název třídy
(v případě, že by měly
dvě entity stejný název)

Vrací List<Item>

- To samé pomocí NamedQuery:

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

Předpokládá existenci
NamedQuery s názvem
„Item.findAll“:

```
@NamedQuery(name = "Item.findAll",  
    query = "select i from Item i")
```

JPQL příklady III.

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

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

Toto je název tabulky, nikoli entity!

Vrací jeden objekt, nikoli list! Pokud se nevrátí žádný objekt, nebo se vrátí víc objektů než jeden, vyhodí se výjimka.

- Získá položky s cenou vyšší než X:

```
public List<Item> getItemsWithPriceGreaterThan(int price) {  
    return entityManager.createNamedQuery(  
        "Item.findPriceGreaterThan", Item.class)  
        .setParameter("price", price).getResultList();  
}
```

Práce s pojmenovanými parametry

```
@NamedQuery(name = "Item.findPriceGreaterThan",  
    query = "select i from Item i where i.price > :price")
```


JPQL příklady II.

- Named query nemusíte mít v kódu pomocí anotace, ale můžete je alternativně uložit do externího XML souboru (výhodné u velkých dotazů)

Dotaz se nachází v META-INF/orm.xml
(soubor může být pojmenovaný i jinak)

```
<named-query name="Item.findPriceGreaterThan">
```

```
<query>
```

```
select i.name, i.price
```

```
from Item i
```

```
where i.price > ?1
```

```
</query>
```

```
</named-query>
```

Výsledkem není entita,
ale pole objektů: Object[]

Místo pojmenovaného parametru je
možné také definovat pozici parametru

Poznámka 1: V Hibernate se ke stejnému účelu dá také použít soubor s příponou hbm.xml.

Poznámka 2: Konfigurace pomocí XML je alternativou vůči anotacím.

JPQL příklady III. - doplnění

- Předchozí select s named query by šel napsat také tímto způsobem:

```
public List<Product> getItemsWithPriceGreaterThan(int price) {  
    return entityManager.createQuery  
        ("select i from Item i where i.price > " +  
         price, Item.class).getResultList();  
}
```

- Jenže:
 - Byl by pomalejší (String se musí zparsovat a select se tvoří za běhu).
 - Nebyl by imunní vůči útoku zvaném SQL injection.
 - Při větším množství dynamických parametrů by nebyl příliš čitelný.
- Tudíž je vhodné co nejvíce používat named query (buď zapsané pomocí XML, nebo pomocí anotace) a vždy používat parametry pro dynamické hodnoty.

JPQL příklady IV.

- Vypíše všechny názvy položek v objednávce:

```
UserOrder userOrder = entityManager.find(UserOrder.class, 1);  
List<OrderedItem> ois = userOrder.getOrderedItems();  
for (OrderedItem oi : ois) {  
    System.out.println(oi.getItem().getName());  
}
```

- Bylo by také možné použít JPQL, ale je to možné udělat i tímto objektovým způsobem.
- Při použití klasického SQL by se musely spojit tři tabulky.
- Neberte tento příklad jako best-practice jak něco takového udělat! Zbytečně se vykoná spousta SELECTů!

JPQL příklady V.

- Získá počet položek v databázi:

```
Long count = entityManager.createQuery(  
    "select count(i) from Item i", Long.class)  
    .getSingleResult();
```

- Všechny agregační funkce:
 - count(), max(), min(), avg(), sum()

Native Query od JPA 2.1 I.

- Pokročilejší Native Query se od JPA 2.1 dělají tímto způsobem:
- POJO (nikoli entita):

```
@Data
```

```
public class CountriesWithRegions implements Serializable {  
  
    private String countryName;  
  
    private String regionName;  
  
    public CountriesWithRegions(String countryName, String regionName) {  
        this.countryName = countryName;  
        this.regionName = regionName;  
    }  
}
```

Native Query od JPA 2.1 II.

- Mapování (musí být na nějaké entitě!):

```
@SqlResultSetMapping(name = "CountriesWithRegionsResult", classes = {  
    @ConstructorResult(targetClass = CountriesWithRegions.class, columns = {  
        @ColumnResult(name = "countryName"),  
        @ColumnResult(name = "regionName")})})
```

- Samotný SELECT:

```
public List<CountriesWithRegions> findAllWithRegions() {  
    return entityManager.createNativeQuery(  
        "select country_name countryName, region_name regionName "  
        + "from countries join regions using (region_id)",  
        "CountriesWithRegionsResult").getResultList();  
}
```

Stored Procedure

- Od JPA 2.1 je plná podpora pro volání procedur / funkcí:
 - <http://dreamand.me/java/java-jee7-jpa-stored-procedure-example/>

Stránkování

- Ve všech aplikacích je obvykle nutné řešit stránkování. JPA k tomu má dvě metody, které obsahuje třída Query:
 - `setFirstResult(int)`: nastaví číslo řádku, od kterého se budou záznamy vracet.
 - `setMaxResults(int)`: nastaví maximální počet výsledků, které dotaz vrátí.
- Nezapomeňte Váš dotaz před jeho „ořezáním“ pomocí těchto dvou metod utřídit pomocí `order by`

Funkce

- V JPQL je celá řada funkcí:
 - `CONCAT()` – spojení dvou řetězců
 - `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP` – získání aktuálního data, času, timestamp
 - `LENGTH()` – délka řetězce
 - `LOWER()`, `UPPER()` – převod písmen v řetězci do malých, velkých písmen
 - `SIZE()` – počet prvků v kolekci
 - `TRIM()` – odstranění whitespace znaků ze začátku a konce řetězce
 - A další ...
 - Ještě více funkcí obsahuje HQL

Join I.

- JOIN operátor se používá ve složitějších SELECTech, když mezi entitami není přímá vazba. Když přímá vazba je, tak ho není nutné používat.
- Tyto selecty jsou ekvivalentní:

```
select oi from UserOrder uo join uo.orderitems oi
```

```
select uo.orderitems from UserOrder uo
```

```
select oi from UserOrder uo, Orderitem oi where oi.userOrder = uo
```

Pozor! Toto vrátí kartézský součin!:

```
select oi from UserOrder uo, Orderitem oi
```

Join II.

- Je možné vytvářet vnitřní spojení nebo vnější spojení:

Vrátí počet prvků v kolekci

```
select i.name, size(i.ordereditems) from Item i group by i
```

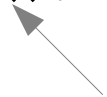
Použije se automaticky vnitřní spojení.
Jenže co když si nějaké zboží zatím
nikdo neobjednal? K tomu potřebujeme
vyvolat vnější spojení:

```
select i.name, size(ois) from Item i  
left join i.ordereditems ois group by i
```

Join Fetch I.

- Jakmile se ukončí transakce, tak není možné automaticky „donačíst“ další data, která jsou s entitou propojená pomocí @OneToMany nebo @ManyToMany anotací.
- Tento problém je možné vyřešit několika způsoby:
 1. Nastavit na příslušných anotacích atribut fetch=FetchType.EAGER
 2. Vynutit v kódu vytvoření SELECTu:

```
public UserOrder getUserOrder(int userOrderId) {  
    UserOrder userOrder= entityManager.find  
                           (UserOrder.class, userOrderId);  
    userOrder.getOrderItems().size();  
    return userOrder;  
}
```



vynucení SELECTu


3. Použít operátor JOIN FETCH – na další stránce

Join Fetch II.

- Použití operátoru JOIN FETCH:

```
public UserOrder getOne(int userOrderId) {  
    return entityManager.createQuery(  
        "select distinct o from UserOrder o "  
        + " left join fetch o.orderedItems "  
        + " where o.userOrderId = :id", UserOrder.class)  
        .setParameter("id", userOrderId).getSingleResult();  
}
```

Vyřadí
duplicity



- Pro utřídění joinované tabulky je vhodné nastavit na příslušném atributu anotaci `@OrderBy`. Příklad:

```
@OrderBy(value="quantity desc")  
private List<OrderedItem> orderedItems;
```

← List má oproti Set jednoznačně definované řazení, tudíž je vhodným typem pro atribut s anotací `@OrderBy`

Open EntityManager In View pattern I.

- Také můžete rozšířit session až na úroveň prezentační vrstvy (jedná se o Open EntityManager In View pattern).
- Přidejte do Spring konfigurace Servlet contextu:

```
<mvc:interceptors>
  <mvc:interceptor>
    <mvc:mapping path="/*" />
    <bean
      class="org.springframework.orm.jpa.support.OpenEntityManagerInViewInterceptor">
      <property name="entityManagerFactory" ref="myEmf" />
    </bean>
  </mvc:interceptor>
</mvc:interceptors>
```



id beany typu EntityManagerFactory
ve Spring kontextu

- Poznámka: Také existuje třída OpenSessionInViewInterceptor pokud nepoužíváte JPA, ale nativní Hibernate přístup.

Open EntityManager In View pattern II.

- Nebo můžete použít filtr:

```
<filter>
  <filter-name>oemInViewFilter</filter-name>
  <filter-class>org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter</filter-class>
  <init-param>
    <param-name>entityManagerFactoryBeanName</param-name>
    <param-value>myEmf</param-value>
  </init-param>
</filter>
```

id beany typu EntityManagerFactory
ve Spring kontextu

```
<filter-mapping>
  <filter-name>oemInViewFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

- Poznámka: Také existuje OpenSessionInViewFilter.

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 Dozer:
 - <http://dozer.sourceforge.net/>
- Další mapovací frameworky:
 - <http://stackoverflow.com/questions/1432764/any-tool-for-java-object-to-object-mapping>
- Velice rychlý framework je Orika:
 - <https://code.google.com/p/orika/>

JPQL – operátor NEW

- Pomocí operátoru NEW je možné přímo v JPA SELECTu vytvořit pomocí konstruktoru objekt:

```
select new UserOrderWithCount(uo, size(oi))  
  from UserOrder uo left join uo.orderedItems oi  
 group by uo
```

select

třída

```
package cz.java.skoleni.eshop;  
public class UserOrderWithCount {  
  
    private UserOrder userOrder;  
    private Integer count;  
  
    public UserOrderWithCount  
        (UserOrder userOrder, Integer count) {  
        this.userOrder = userOrder;  
        this.count = count;  
    }  
    // gettery, settery  
}
```

JPQL DELETE / UPDATE

- Pomocí JPQL můžete také volat DELETE a UPDATE dotazy:

```
entityManager.createQuery("delete from Item").executeUpdate();
```

- Zavolá DELETE FROM ITEM

```
Query query = entityManager.createQuery("update Item set price = price * 1.1");
```

```
int updateCount = query.executeUpdate();
```

- Zavolá UPDATE ITEM ...

- <http://www.objectdb.com/java/jpa/query/jpql/delete>
- <http://www.objectdb.com/java/jpa/query/jpql/update>

Criteria query I.

- Občas narazíte na situaci, kdy je nutné vytvořit select dynamicky. Například když uživatel má na výběr z několika checkboxů, radiobuttonů atd.
- Select můžete vytvořit spojováním Stringů, ale to není výkonné, protože se před vykonáním musí parsovat.
- Z toho důvodu je možné vytvořit select programově pomocí tzv. Criteria query:

```
SELECT e FROM Employee e WHERE e.name = 'John Smith'
```

↙
↘ Stejný SELECT

```
CriteriaBuilder cb = em.getCriteriaBuilder();  
CriteriaQuery<Employee> c = cb.createQuery(Employee.class);  
Root<Employee> emp = c.from(Employee.class);  
c.select(emp).where(cb.equal(emp.get("name"), "John Smith"));
```

Criteria query II.

- JPA Criteria jsou hodně „ukecaná“. Jaké jsou alternativy?
 - Hibernate Criteria – oficiálně deprecated, ale masově používaná.
 - Nevýhody: nepoužívají generics, pro jejich vytvoření je nutné použít Session.
 - Výhody: funguje u nich Hibernate anotace @Fetch, jejich zápis je kompaktnější než u JPA kritérií.
 - Querydsl – nadstavba (nejenom) nad JPA / Hibernate Criteria.
 - <http://www.querydsl.com/>

Cache

- Konfigurace cachování může být na úrovni persistentní unity, nebo jednotlivých entit.
- Nastavení vypnutí cachování v `persistence.xml`:

```
<shared-cache-mode>NONE</shared-cache-mode>
```

- Obvykle je ale zbytečné kompletně vypínat cachování (mohlo by dojít k degradaci výkonu). Můžete ho také vypnout selektivně pro určité entity (doporučovaný způsob). Tento způsob cachování je nutné v `persistence.xml` nejprve zapnout:

```
<shared-cache-mode>DISABLE_SELECTIVE</shared-cache-mode>
```

- Poté u entit, pro které chcete vypnout cachování, nastavte před definicí třídy: `@Cacheable(false)`

Doporučená literatura

- <http://www.java-skoleni.cz/literatura/hibernate.php>

