ORM

Petr Adámek, petr.adamek@ibacz.eu

Osnova I.

Úvod

- Co je ORM
- Proč relační databáze, proč objektový model
- Alternativy ORM

Základní principy ORM

- Základní pojmy
- Standardy & přístupy
- Definice mapování

JPA

- Úvod
- Entity v JPA
- Operace s entitami
- Konfigurace
- Životní cyklus entity
- Kvíz
- Hands-on lab: Databáze kontaktů

Osnova II.

JPA

- Kolekce
- vazby mezi entitami
- Hands-on lab: Databáze kontaktů a více telefonních čísel
- Demonstrační příklad: Podpora pro kategorie

Dotazování a vyhledávání

- JPQL
- Criteria API
- Hands-on lab: Vyhledávání v databázi kontaktů

Závěrečný test

Úvod

Co je to ORM

Objektově-relační mapování (ORM)

- Zajišťuje konverzi dat mezi objektovým a relačním datovým modelem.
- Umožňuje pracovat s daty jako s objekty, ale ukládat je do klasické relační databáze.

INSERT INTO people (id, name) VALUES (1, "Pepa");

```
Person p = new Person(1, "Pepa");
em.persist(p);
em.getTransaction().commit();
```

UPDATE people SET name = "Honza" WHERE id = 2;

```
Person p = em.find(Person.class,2);
p.setName("Honza");
em.getTransaction().commit();
```

K čemu je to dobré?

Proč relační databáze?

Data lze ukládat různým způsobem

- Relační databáze
- Objektové databáze
- XML databáze
- DMS
- Post-relační databáze (Caché)
- Jiný informační systém (CRM, ERP, apod.)

Přesto se ve většině případů používá relační databáze

- Relační model dat je velmi jednoduchý a přitom dostatečně mocný pro většinu běžných aplikací.
- Jednoduchost => Vysoký výkon (např. optimalizace).
- Osvědčená a etablovaná technologie (40 let vývoje, nástroje, standardy, spolehlivost, vysoká penetrace, dostatek administrátorů, apod.)
- Data jsou oddělena od aplikace a mohou být snadno sdílena mezi různými aplikacemi.
- Nezávislost na programovacím jazyku nebo platformě.

Proč objektový model?

Když používáme relační databázi, proč chceme objektový model?

- V objektově orientovaném jazyce se s objektovým modelem pracuje lépe (je přirozenější).
- Viz následující příklad.

Příklad: Zjištění jména osoby využitím JDBC

```
public String getPersonName(long personId)
    throws SQLException {
  PreparedStatement st = null;
  try {
    st = connection.prepareStatement(
      "SELECT name FROM people WHERE id = ?");
    st.setLong(1, personId);
    ResultSet rs = st.executeQuery();
    if (rs.next()) {
      String result = rs.getString("name");
      assert !rs.next();
      return result;
    } else {
      return null;
    }
  } finally {
    if (st != null) { st.close(); }
  }
}
```

Příklad: Zjištění jména osoby využitím ORM

```
public String getPersonName(long personId) {
    Person p = em.find(Person.class,personId);
    return p.getName();
}
```

Co získáme a ztratíme použitím ORM

Přínosy ORM

- Možnost pracovat s objektovým modelem, který je v objektovém jazyce přirozenější.
- Přenositelnost aplikací mezi různými DB systémy.
- Typová kontrola v době překladu aplikace.
- Eliminuje potenciální chyby v SQL, které se projeví až za běhu aplikace.
- Usnadňuje testování.
- Často zjednodušuje a zpřehledňuje implementaci.
- Efektivnější vývoj (automatické doplňování názvů, snazší přístup k dokumentaci JavaDoc, apod.)

Nevýhody ORM

- Potenciálně menší výkon (ORM má jistou režii).
- Nemožnost využití výhod relačního modelu a všech možností relačních databází na aplikační úrovni (např. uložené procedury).

ORM není jediné řešení

Embedded SQL

- SQL výrazy píšeme přímo do zdrojového kódu.
- Tento kód se před vlastním překladem zpracuje speciálním preprocesorem.
- Preprocesor zpracuje SQL výrazy, zkontroluje jejich správnost, provede typovou kontrolu a přeloží je do výrazů daného programovacího jazyka.
- Preprocesor při překladu potřebuje připojení k příslušné databázi.

```
public String getPersonName(long personId) {
    String name;
    #sql {
        SELECT name INTO :name
        FROM people WHERE id = :personId
    };
    return name;
}
```

ORM není jediné řešení

Spring JDBC

- Jedna z knihoven implementujících návrhový vzor *Template Method*.
- Zpřehledňuje kód, zrychluje vývoj, usnadňuje údržbu.
- Narozdíl od ORM nebo Embedded SQL neřeší problém s případnými chybami v SQL, které se projeví až v době běhu aplikace.

```
public String getPersonName(long personId) {
  return jdbc.queryForObject(
    "SELECT name FROM people WHERE id = ?",
    String.class,personId);
}
```

Apache Commons DbUtils

- Další knihovna implementujících návrhový vzor *Template Method*.

Základní principy ORM

Základní pojmy

Pojmy, které byste měli znát

- **JDBC, SQL, Transakce,**

Pojmy, které si definujeme

- **Entita** – doménový objekt, který reprezentuje data ukládaná do databáze (např. osoba, faktura, předmět).
- **DTO** (*Data Transfer Object*) – objekt, který slouží pro zapouzdření dat a jejich transport mezi komponentami.
- **POJO** (*Plain Old Java Object*) – jednoduchá třída, na kterou nejsou kladeny žádné zvláštní požadavky. Nemusí implementovat žádné rozhraní, nemusí rozšiřovat žádnou jinou třídu, ani není závislá na žádné jiné třídě, balíku nebo frameworku. Jediné podmínky, které na ni mohou být kladeny, je přítomnost bezparametrického konstruktoru, a dodržení konvencí pro pojmenování get/set metod.

Příklad: Entita a DTO

```
public class CustomerEntity {
    Long id;
    String name;
    String address;
    String notes;
    Collection<Contract> contracts;
}

public class CustomerDTO {
    Long id;
    String name;
    int contractsCount;
    BigDecimal ContractsPriceSum;
}
```

Obě třídy jsou zároveň POJO.

Standardy & přístupy

Entity EJB (EJB 2.1/JSR 153; J2EE 1.4)

- Vyžaduje aplikační server s EJB kontejnerem.
- Entita je heavyweight komponenta, jejíž instance se nachází v EJB kontejneru a přístup k ní probíhá prostřednictvím vzdáleného volání metod.
- Problém s latencemi (řeší se pomocí DTO, příp. DAO).
- CMP a BMP
- Od verze EJB 3.0 (JSR 220) je preferováno JPA.

JDO (JDO 3.0/JSR 243)

- Obecný standard pro perzistenci v Javě.
- Není omezeno na relační databáze, objekty mohou být ukládány do úložiště libovolného typu

JPA (JPA 2.0/JSR 317; Java EE 6)

- Java EE standard pro ORM inspirovaný Hibernate.
- Entita je lightweight POJO, který může být libovolně předáván mezi komponentami, lokálně i vzdáleně.

Příklad: Entity EJB (CMP)

```
public abstract class PersonBean implements EntityBean {
    private EntityContext context;

    public abstract Long getId();
    public abstract void setId(Long id);
    public abstract String getName();
    public abstract void setName(String name);

    public Long ejbCreate (Long id, String name) throws CreateException {
       setId(id); setName(name); return id;
    }
    public void ejbPostCreate (Long id, String name) throws CreateException {}

    public void setEntityContext(EntityContext ctx)  { context = ctx; }
    public void unsetEntityContext() { context = null; }

    public void ejbRemove() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void ejbPassivate() {}
    public void ejbActivate() {}

    public void PersonDTO getPersonDTO() {
        return new PersonDTO(getId(),getName());
    }
}
```

Příklad: POJO Entita

```
public class Person {

    private Long id;
    private String name;

    public Long getId()                 { return id; }
    public void setId(Long id)          { this.id = id; }
    public String getName()             { return name; }
    public void setName(String name)    { this.name = name; }

    public boolean equals(Object o) {
        if (this == o) { return true; }
        if (getId() == null) { return false; }
        if (o instanceof Person) {
            return getId().equals(((Person) o).getId());
        } else {
            return false;
        }
    }

    public int hashCode() { return id==null?0:id.hashCode(); }
}
```

Atributy

get/set metody

equals() & hashCode()

Definice mapování

Prostřednictvím anotací

- Definice objektového modelu i jeho mapování je na jednom místě.
- Větší přehlednost, jednodušší vývoj, snazší údržba.

Prostřednictvím externího souboru (obvykle XML)

- nezávislost kódu entit na konkrétní technologii zajišťující ORM;
- možnost měnit mapování bez nutnosti modifikovat kód.

Pomocí speciálních JavaDoc komentářů

- Z doby, kdy Java nepodporovala anotace.
- Viz XDoclet.

Generování schématu databáze nebo mapování

Generování schématu databáze na základě definice mapování

- Máme vytvořené entity a definici mapování a chceme si ušetřit práci s vytvářením schématu databáze.
- Je možné automaticky vytvářet tabulky při prvním spuštění aplikace.
- Výhodné zejména při vývoji, kdy dochází ke změnám datového modelu.
- Vhodné, pokud je datový model zcela pod kontrolou naší aplikace.
- Problém, pokud se mění datový model a již máme v databázi existující data.

Generování entit a definice mapování na základě schématu databáze

- Máme vytvořené schéma databáze a chceme si ušetřit práci s vytvářením entit a definicí mapování (např. vyvíjíme aplikaci pro přístup k již existujícím datům).
- Obvykle je nutné vygenerované soubory ručně opravit.

Java Persistence API

Úvod do JPA

Java Persistence API

- POJO Entity, inspirované ORM nástrojem Hibernate
- Jedná se o rozhraní, které implementují různé ORM nástroje od různých dodavatelů.
- Obsahuje základní funkce, jednotlivé implementace mohou prostřednictvím svého proprietárního rozhraní poskytovat řadu dalších služeb a možností.

Verze a specifikace

- **JPA 1.0** – součást Java EE 5; vzniklo jako součást EJB 3.0 (JSR 220), lze jej ale použít zcela nezávisle.
- **JPA 2.0** – součást Java EE 6; JSR 317.

ORM nástroje implementující JPA

- Hibernate
- TopLink, TopLink Essentials
- Eclipse Link (JPA 2.0)
- Open JPA

Entity v JPA

Entity

- Reprezentují jednotlivé doménové objekty.
- Jsou klasické POJO, tj. jednoduché a obyčejné objekty.
- Entita má atributy, které reprezentují vlastnosti doménového objektu.
- Atributy jsou přístupné pomocí get/set metod.
- Entita musí mít bezparamentrický konstruktor a pokud má být používána k přenosu dat prostřednictvím RMI, musí být serializovatelná.

Definice mapování

- Způsob uložení entity do relační databáze je definován pomocí anotací, nebo pomocí XML souboru.
- Důsledně se uplatňuje princip *convention-over-configuration.*

Příklad: Entita v JPA

Další konfigurace mapování není třeba díky principu *convention-over-configuration*

```
@Entity
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;

    public Long getId()                { return id; }
    public void setId(Long id)         { this.id = id; }
    public String getName()            { return name; }
    public void setName(String name)   { this.name = name; }

    public boolean equals(Object o) {
        if (this == o) { return true; }
        if (getId() == null) { return false; }
        if (o instanceof Person) {
            return getId().equals(((Person) o).getId());
        } else {
            return false;
        }
    }

    public int hashCode() { return id==null?0:id.hashCode(); }
}
```

Konfigurace

Konfigurace

- Uložena v souboru persistence.xml
- Může obsahovat více tzv. Persistence Unit.

Persistence Unit

- Seznam tříd, které daná PU spravuje
- Konfigurace připojení k databázi
  - JNDI název DataSource
  - JDBC url, jméno, heslo
- Způsob řízení transakcí
- Způsob generování tabulek při spuštění aplikace

Konfigurační parametry

- U JPA 1.0 nejsou názvy konfiguračních parametrů standardizovány, u JPA 2.0 již ano.
- Parametry mohou být nastaveny také při vytváření instancí EntityManagerFactory a EntityManager.

Příklad: persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
    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_1_0.xsd">

  <persistence-unit name="MyPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <property name="hibernate.connection.username" value="app"/>
      <property name="hibernate.connection.password" value="app"/>
      <property name="hibernate.hbm2ddl.auto"        value="update"/>
      <property name="hibernate.connection.driver_class"
        value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="hibernate.connection.url"
        value="jdbc:derby://localhost:1527/sample"/>
      <property name="hibernate.cache.provider_class"
        value="org.hibernate.cache.NoCacheProvider"/>
    </properties>
  </persistence-unit>

</persistence>
```

Manipulace s Entitami

EntityManager

- Umožňuje provádět CRUD operace s entitami.
- Instanci získáme pomocí EntityManagerFactory nebo prostřednictvím *dependency injection.*
- Vytvoření instance je levná operace.
- Není vláknově bezpečná.
- Instance vytváříme podle potřeby (často se pro každou operaci vytváří nová instance).

EntityManagerFactory

- Slouží jako tovární třída pro EntityManager.
- Načítá konfiguraci z persistence.xml.
- Instanci získáme pomocí metody `Persistence.createEntityManagerFactory()` nebo pomocí *dependency injection.*
- Vytvoření instance je drahá operace.
- Je vláknově bezpečná.
- Obvykle vytváříme pouze jednu instanci.

Příklad: Práce s JPA Entitou

```
public class PersonExample {

  public static void main(String ... args) {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("MyPU");
    EntityManager em = emf.createEntityManager();

    // Vytvoření entity
    em.getTransaction().begin();
    Person sheldon = new Person();
    sheldon.setName("Sheldon Cooper");
    em.persist(sheldon);
    em.getTransaction().commit();

    // Dotazování + změna entity
    em.getTransaction().begin();
    Query q = em.createQuery(
        "SELECT p FROM Person p WHERE name = 'Bernadette Rostenkowski'");
    Person bernadette = (Person) q.getSingleResult();
    bernadette.setName("Bernadette Wolowitz");
    em.getTransaction().commit();

  }
}
```

Název Persistence Unit, která definuje konfiguraci

Vytvoření a provedení dotazu

Změna entity, která byla vrácena jako výsledek dotazu.

Tato změna se automaticky uloží do databáze v okamžiku commitu transakce.

Persistence Context

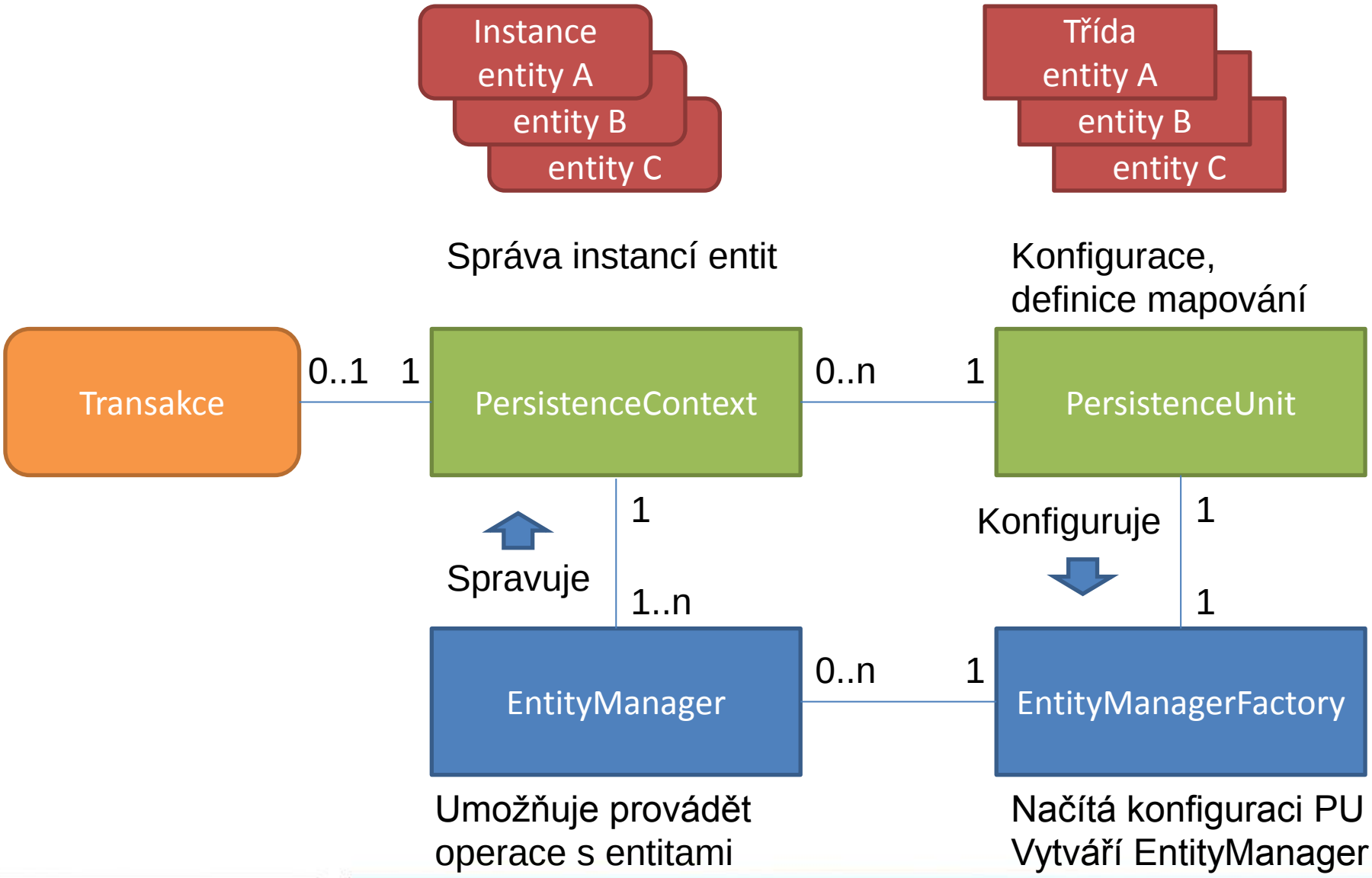
EntityManager

- ….
- …

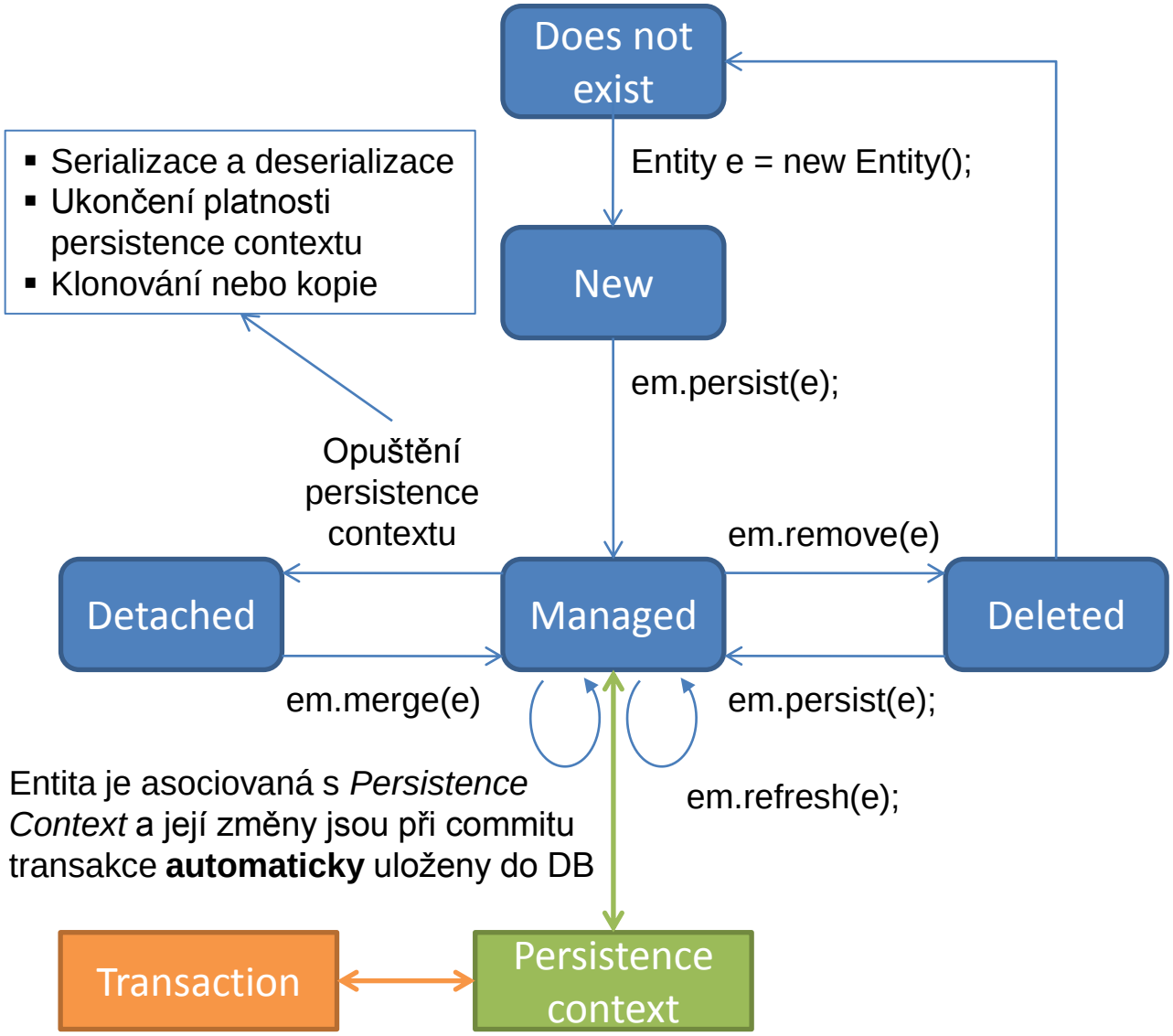
EntityManagerFactory

- …
- …

Rekapitulace vztahů mezi základními pojmy



Životní cyklus entity



Kvíz
Životní cyklus entity

Pokročilé mapování

Vztahy mezi entitami

Persistence Context

EntityManager

- ....
- ...

EntityManagerFactory

- Typ vazby
- Kardinalita
- Obousměrná/jednosměrná
- Lazy fetching
- Kaskádování operací
- Vazby vytvářejí závislosti

Problematické závislosti

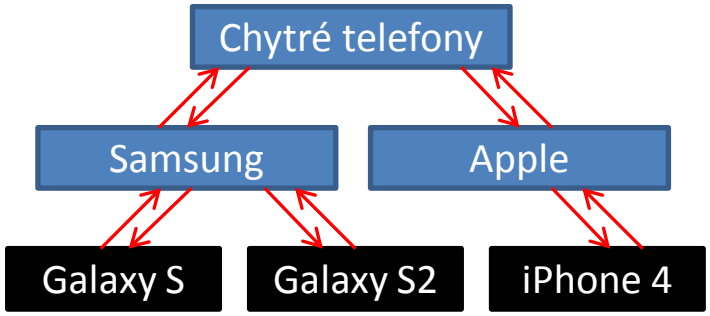
Problém s vazbami

- Vazby vytvářejí závislosti
- ...

Řešení

- Typ vazby
- Kardinalita
- Obousměrná/jednosměrná
- Lazy fetching
- Kaskádování operací
- Vazby vytvářejí závislosti

Příklad: Katalog produktů



Problém: Obousměrné vazby vytváří tranzitivní závislost všeho na všem

Důsledek: Při načtení jakéhokoliv produktu dojde k načtení všeho

Řešení

- Lazy fetching (řeší důsledek místo příčiny, nelze použít vždy).
- Jednosměrné vazby (řeší problém pouze částečně).
- Úplné odstranění vazeb, obsah kategorie získávat pomocí nových metod.

Product
id : Long name : String ~~parent : Category~~

Category
id : Long name : String ~~parent : Category~~ ~~products : List<Product>~~ ~~categories : List<Category>~~

ProductDAO
create(p : Product) : void get(id : Long) : Product update(p : Product) : void delete(p : Product) : void

CategoryDAO
create(g : Group) : void get(id : Long) : Group update(g : Group) : void delete(g : Group) : void

CategoryContentDAO
getProducts(c : Category) : List<Product> getCategories(c : Category) : List<Group> addCategory(parent : Category, c : Category) : void addProduct(parent : Category, p : Product) : void removeCategory(parent : Category, c : Category) : void removeProduct(parent : Category, p : Product) : void

Dědičnost

EntityManager

- ….
- …

EntityManagerFactory

- …
- …

Dotazování a vyhledávání

Dotazování v JPA

JPQL

- Standardní dotazovací jazyk pro JPA
- Syntaxe vychází z SQL
- Je nezávislý na použitém databázovém systému
- Umožňuje získávat kolekce entit,

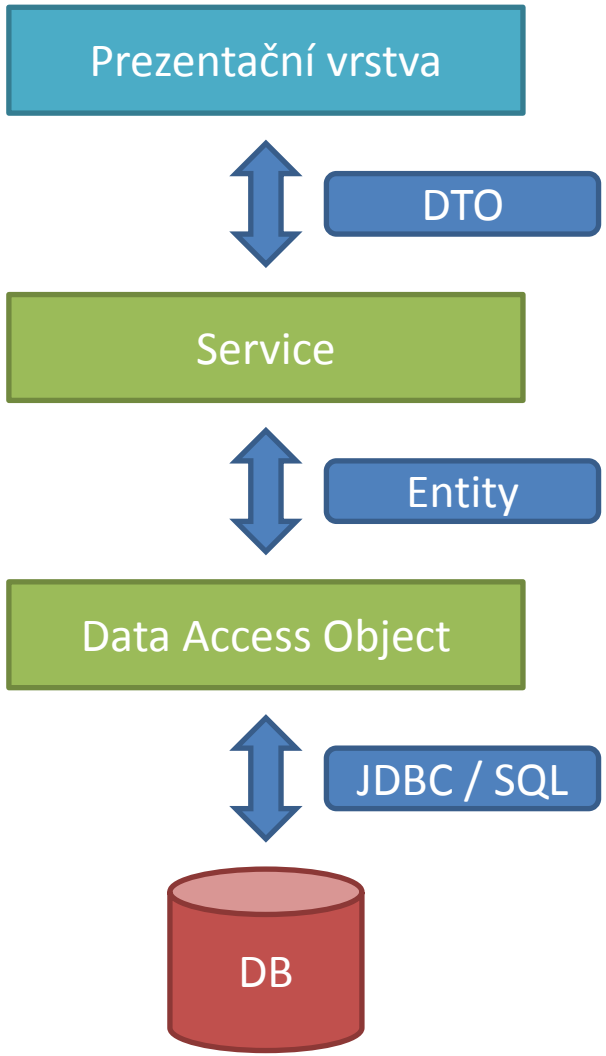
Criteria API

- ...
- ...

Transakce

Best Practices

Typická architektura databázových aplikací



Kontakty

IBA CZ

Petr Adámek
University Relations
petr.adamek@ibacz.eu

IBA CZ, s.r.o.
Petržílkova 2565/23
158 00 Praha 5

IBA CZ Development Center
Křenová 72
602 00 Brno

Tel.: (+420) 543 426 800
http://www.ibacz.eu/