Съхранение на данните

… чрез JPA технологията

Петьо Димитров

01 Април 2013

Съдържание на лекциите

Въведение в Java EE технологиите

Презентация на данните

Бизнес логика

Съхранение на данните

Съдържание

Какво са JPA?

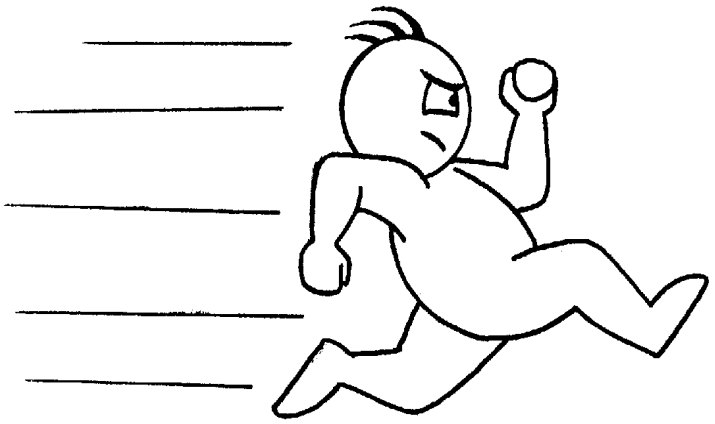
Object-Relational Mapping

JPA концепции и особености

JPA за напреднали

Какво са JPA?

- ORM е Object-Relational Mapping
- JPA е ORM спецификация
- Hibernate и EclipseLink
  са ORM библиотеки и JPA
  имплементации

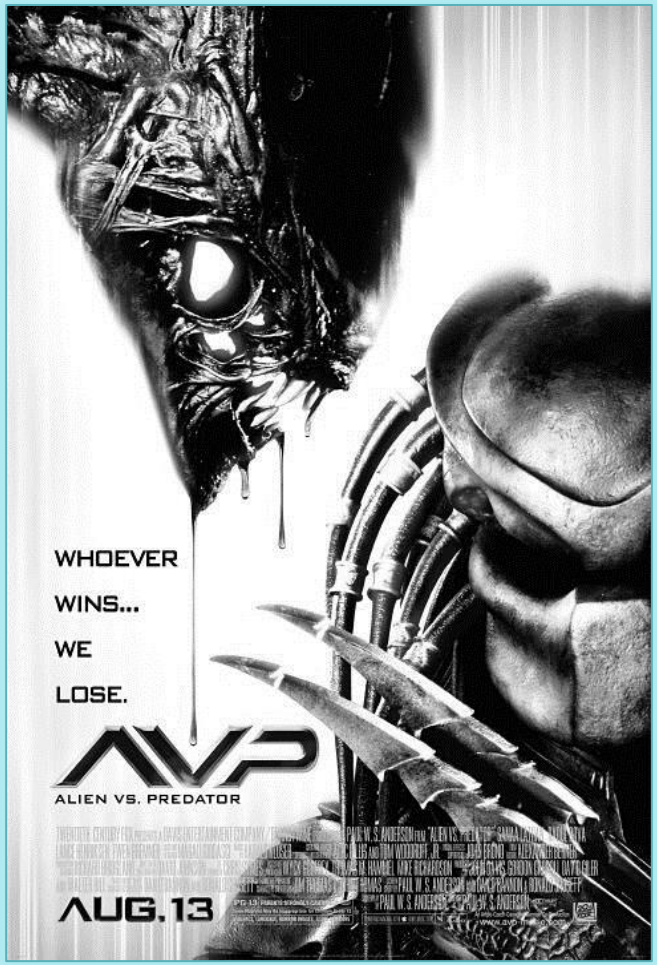


Съхранение на данни

- всяко приложение складира данни

Конфликт между бази данни и ООП

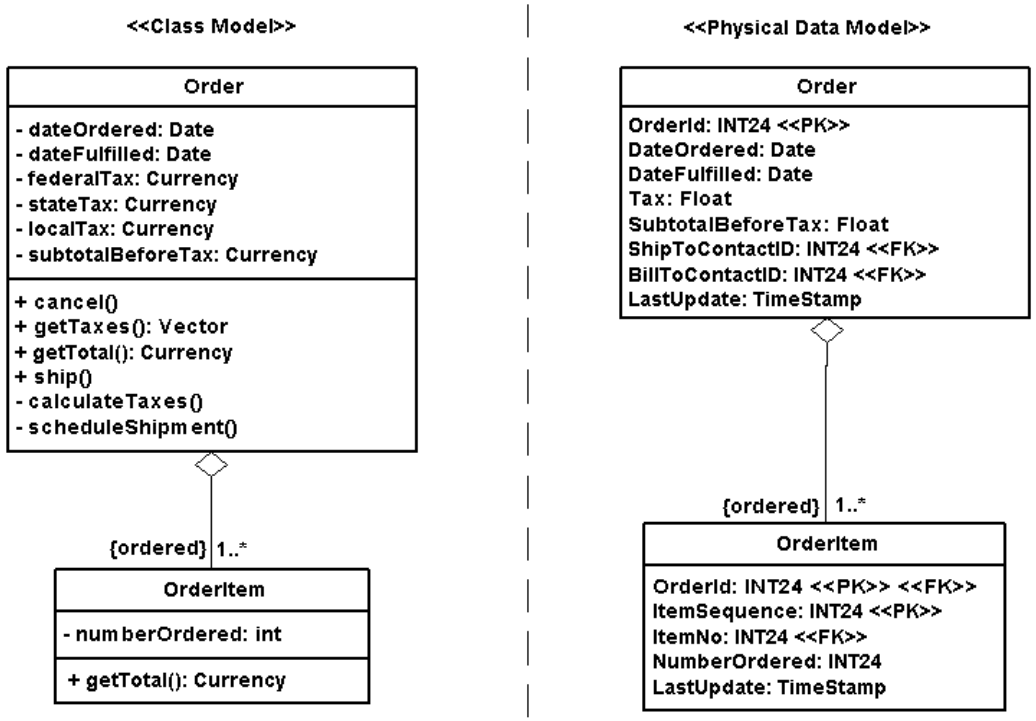
- таблици
- редове
- колони
- обединение
- сечение



- идентичност
- състояние
- поведение
- капсулация
- наследяване
- полиморфизъм

Object-Relational Mapping (ORM)

- преобразуване на обекти във формат за релационни бази данни и обратно



Лесно решение?

- клас ↔ таблица
- свойство ↔ колона
- асоциация ↔ foreign key
- int ↔ INTEGER, String ↔ VARCHAR

Проблеми (ORM impedance mismatch)

- идентичност ①
- наследяване ②
- еднопосочни асоциации ③
- комплексни типове (гранулярност) ④
- извличане на данни (QbE, QbA, QbL) ⑤
- зареждане на данни и кеширане ⑥
- *две схеми на данните*
- *капсулация и др.*

История/Алтернативи

- отделни фирмени решения (proprietary): TopLink 1992, Hibernate 2001
  - никаква стандартизация
  - нужда от специално обучение
- Java Database Connectivity (JDBC, 1997)
  - трудоемък и еднообразен код
  - смесва код и заявки (injection)
  - DAO pattern (Data Access Object)

История/Алтернативи (продължение)

- Enterprise JavaBeans (entity beans, ~1999)
  - over-engineered
  - сложни за създаване (interfaces, RMI)
  - много конфигурация (XML)
- Java Data Objects (2002)
  - POJO + обогатяване на байт-кода
  - слабо използване в RDBMS средите

Настояще

- Java Persistent API (2007)
  - наследник на всяко от изброените решения
  - разработчиците на библиотеки участват в „зачеването“
  - носи големи надежди за стандартизация

Java Persistence API

- POJO модел (Plain Old Java Object)
- разделение между API и обекти за съхранение
- лесна конфигурация (стойности по подразбиране)
- заявки върху обекти

Терминология

- `Entity`
- Entity релации
- `Embeddable`
- `PersistenceUnit`
- Persistence context
- Persistence provider
- `EntityManager`
- Metadata
- JPQL и Criteria API

Entity

- клас чиито инстанции ще съхраняваме
- инстанциите са обикновени Java обекти

- изисквания
  - *public* конструктор (от компилатора)
  - не може да е *final* (заради проксито)
  - трябва да има *primary key* ①
  - може да е конкретен или абстрактен клас
  - да е маркира (с анотация или XML)

Entity пример

```
@Entity
public class Prospect {
   ...
   @Id
   Long id; ①
   ...
   public Prospect() {
      ...
   }

}
```

Стратегии за `@Id`

- автоматично
  - лесен начин
- чрез sequence
  - performance
  - зависи от базата данни
- чрез таблица
  - id-та за няколко таблици
  - не зависи от базата данни
- чрез identity
  - зависи от поддръжката на базата данни

Достъп до данните

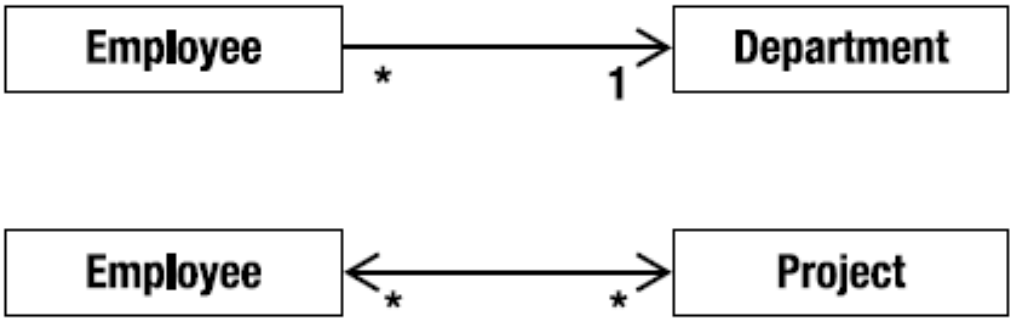
- дефинира се с анотация `@Access`
- чрез поле (`AccessType.FIELD_ACCESS`)
  - по подразбиране се използват полетата
- чрез аксесор (`AccessType.PROPERTY_ACCESS`)
- смесен

Релации между Entity-та

- One-To-One
- Many-To-One
- One-To-Many
- Many-To-Many

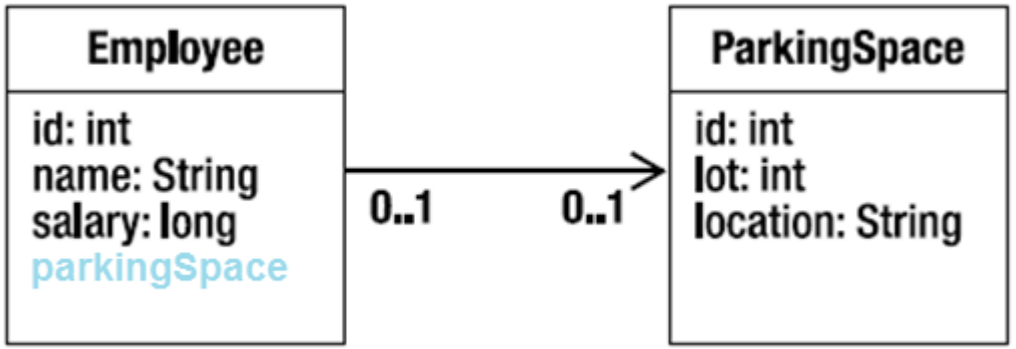
Характеристики на релацията

- роли:
  - източник (собственик)
  - дестинация
- кардиналност:
  - едно
  - много
- посока:
  - еднопосочна
  - двупосочна



OneToOne

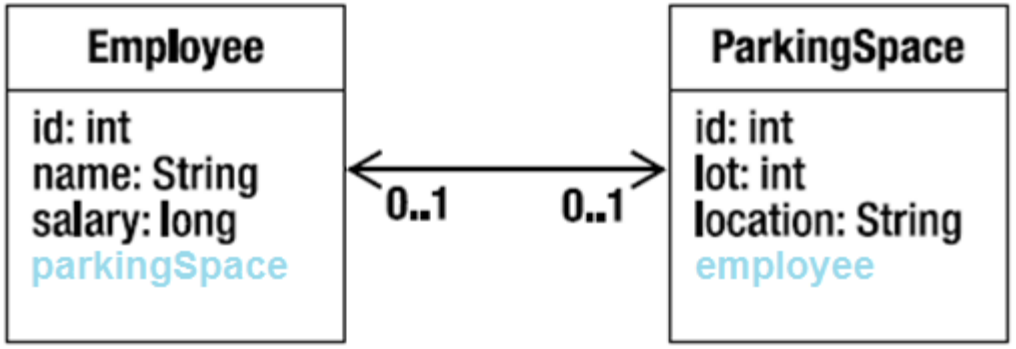
еднопосочна ③




```
@Entity
public class Employee {
    @Id private int id;
    ...
    @OneToOne
    @JoinColumn(name="PSPACE_ID")
    private ParkingSpace parkingSpace;
    ...
}
```

OneToOne

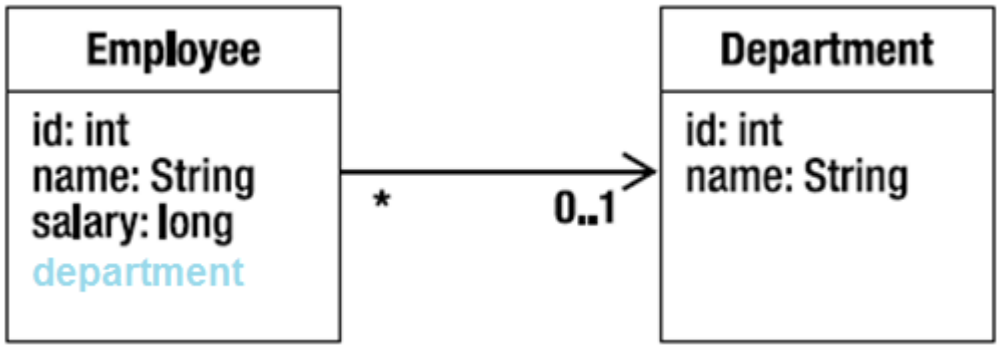
двупосочна




```
@Entity
public class ParkingSpace {
    @Id private int id;
    ...
    @OneToOne(mappedBy="parkingSpace")
    private Employee employee;
    ...
}
```

ManyToOne

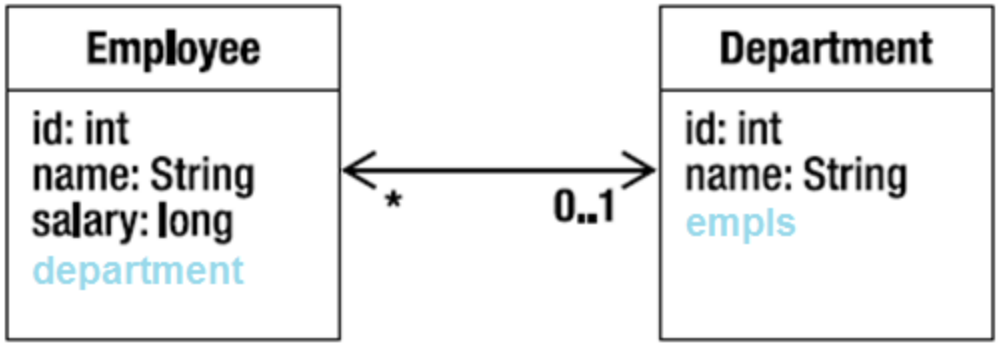
еднопосочна ③




```
@Entity
public class Employee {
    ...
    @ManyToOne
    private Department department;
    ...
}
```

OneToMany

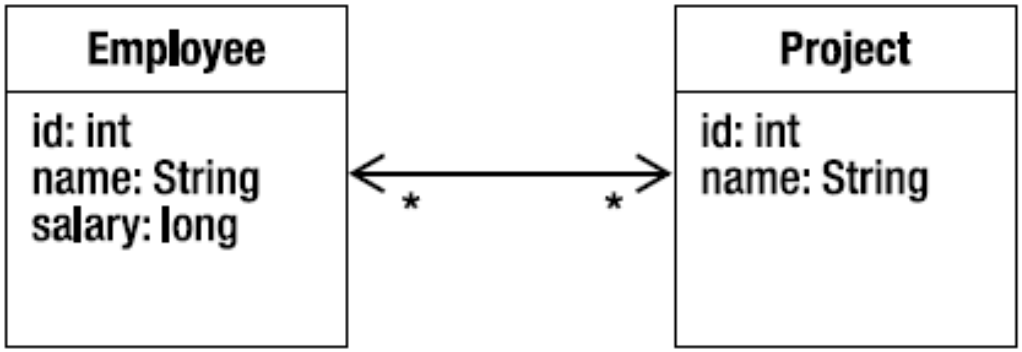
двупосочна




```
@Entity
public class Department {
    @Id private int id;
    ...
    @OneToMany(mappedBy="department")
    private Collection<Employee> empls;
    ...
}
```

ManyToMany

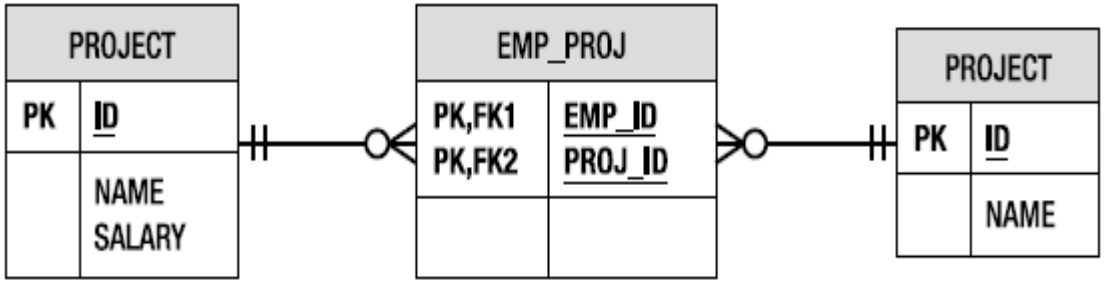
двупосочна




```
@Entity
public class Project {
    @Id private int id;
    ...
    @ManyToMany(mappedBy="prjs")
    private Collection<Employee> empls;
    ...
}
```

ManyToMany

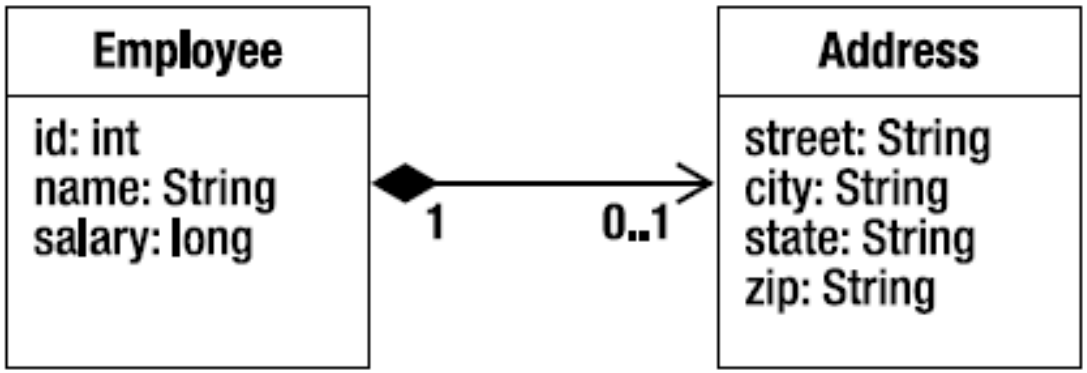
двупосочна




```
@Entity
public class Employee {
    @Id private int id;
    @ManyToMany
    @JoinTable(name="EMP_PROJ",
            joinColumns=@JoinColumn(name="EMP_ID"),
            inverseJoinColumns=@JoinColumn(name="PROJ_ID"))
    private Collection<Project> prjs;
    ...
}
```

Embeddable ④

- позволяват да групираме свойства без да създаваме ново *entity*
- могат да се използват повторно
- стойностите им могат да се предефинират



Embeddable

```
@Embeddable
public class Address {
    private String street;
    private String city;
    private String state;
    @Column(name="ZIP_CODE")
    private String zip;
}
```

```
@Entity public class Employee {
    ...
    @Embedded private Address address;
}
```

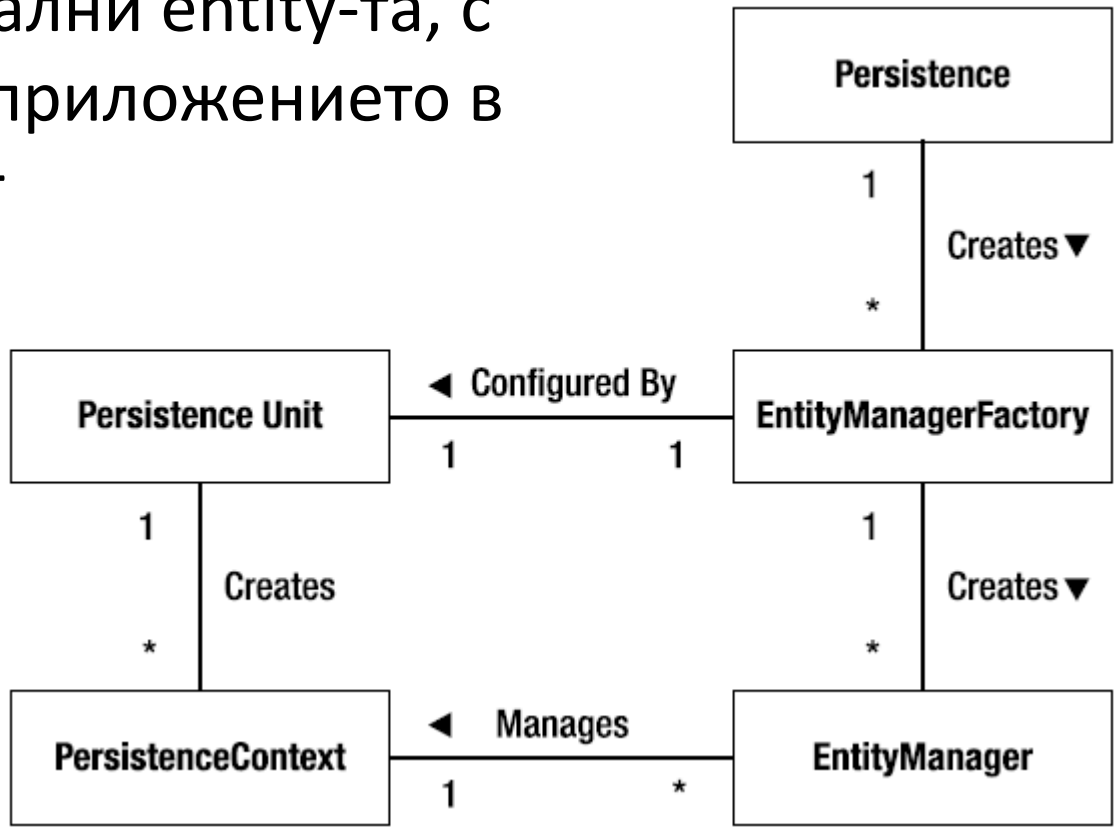
PersistenceUnit

- набор от всички класове използвани от приложението в дадена база данни
- дефинира се в `META-INF/persistence.xml`
- може да са много в едно приложение

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence …>
  <persistence-unit name="opa" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <properties>
        ...
    </properties>
  </persistence-unit>
</persistence>
```

Persistence контекст

- набор от уникални entity-та, с които работи приложението в даден момент



EntityManager

- JPA интерфейс за работа с entity-та
- еквивалентен на Hibernate Session интерфейса
- базови методи:
  - `persist`
  - `merge`
  - `find`
  - `flush`
  - `lock`
  - `remove`
  - `detach`
  - `refresh`
  - `clear`
  - `close`

Видове EntityManager

- транзакционален (transactional)
  - нуждае се от JTA
  - запазва промените в края на транзакцията и изчиства контекста
- разширен (extended)
  - запазва контекста между заявките (stateful/stateless)
- самостоятелен (application)
  - създава се от `EntityManagerFactory`
  - за отделни приложения

Създаване на EntityManager

- чрез `EntityManagerFactory`

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("emp");
EntityManager em = emf.createEntityManager();
...
em.close();
emf.close();
```

Създаване на EntityManager

- чрез Dependency Injection (DI)

```
@PersistenceUnit(unitName="EmployeeService")
EntityManagerFactory emf;

@PersistenceContext(unitName="EmployeeService",
type=PersistenceContextType.EXTENDED)
EntityManager em;

@PersistenceContext(unitName="EmployeeService")
EntityManager em;
```

Използване на EntityManager

```
Person p = new Person ();
EntityManager em = ...;
em.persist(p);

Long id = ...;
Person p = em.find(Person.class, id);
em.remove(p);

em.flush();
```

Наследяване ②

- три стратегии
  - обща таблица за клас йерархия (SINGLE_TABLE)
  - join на наследници (JOINED)
  - една таблица за конкретен (TABLE_PER_CLASS)

- `@MappedSuperclass` (съхранение на данните от родителите)

Обща таблица за клас йерархия

```
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(...)
public abstract class Institution {...}

@DiscriminatorValue("SECONDARY")
public class SecondarySchool extends Institution {
    ...
}

@DiscriminatorValue("POST_SECONDARY")
public class PostSecondarySchool extends Institution {
    ...
}
```

Извличане на данни ⑤

- Query by Example:
  - `EntityManager.find(class, id)`
- Query by API:
  - JPA Criteria API
- Query by Language:
  - JPQL = Java Persistence Query Language

Java Persistence Query Language

- вдъхновен от SQL и HQL
- *entity*-та вместо таблици
- OOP модел за обхождане на релации (`'.'`)

```
SELECT e
FROM Employee e
WHERE e.department.name = 'NA42'
     AND e.address.state IN ('NY','CA')
```

Java Persistence Query Language

- единичен и множество резултати
- функции за агрегиране (count, max, etc.)
- прост синтаксис за join операции
- update и delete заявки за bulk промени на данни
- под-заявки и параметризация на заявките

JPQL - пример

```
private static final String QUERY = "SELECT  e.salary
FROM Employee e WHERE e.dept.name = :deptName"


public long queryEmpSalary(String deptName) {
    return em.createQuery(QUERY, Long.class)
        .setParameter("deptName", deptName)
        .getSingleResult();
}
```

Criteria API

```
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"));
```

- Meta-model API

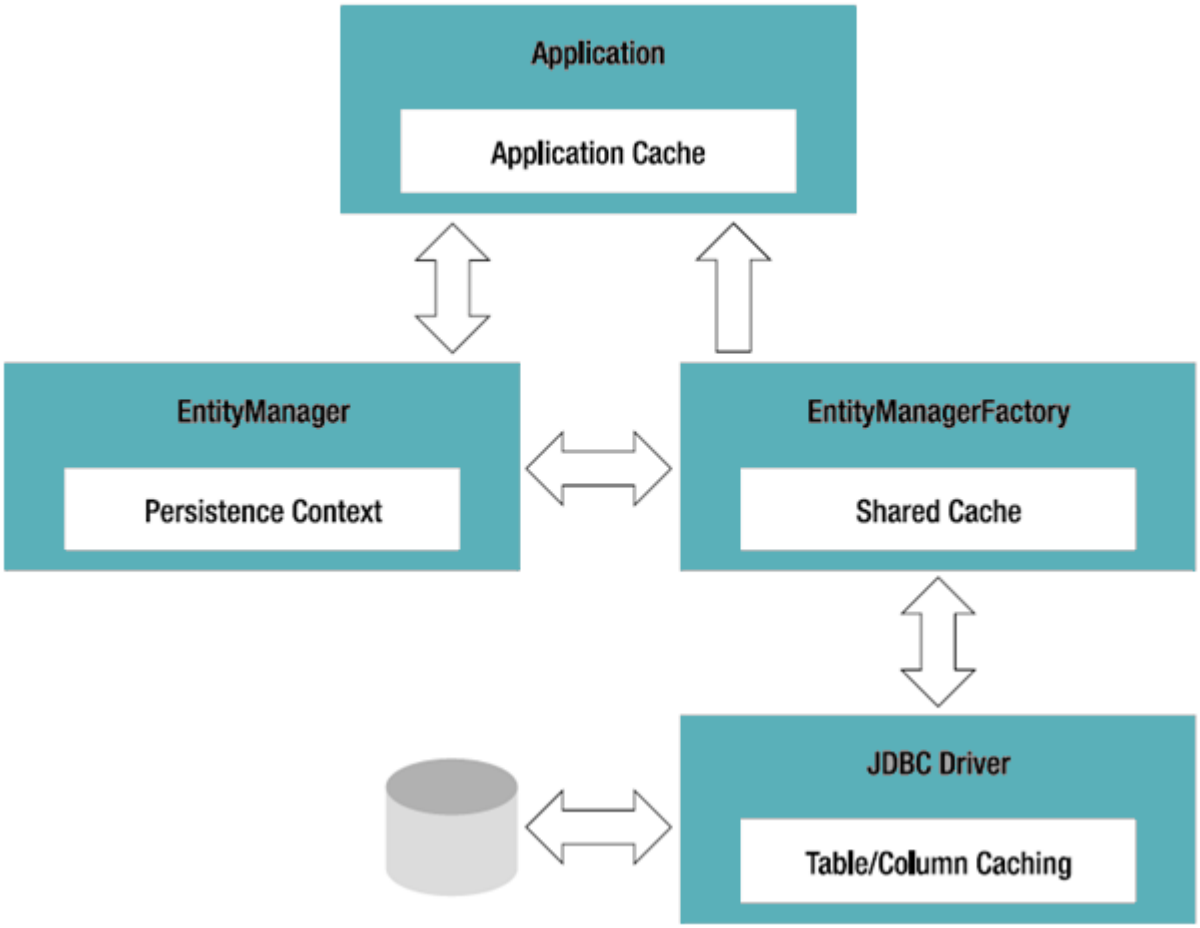
Зареждане на данни ⑥

- proxy за POJO обекта
- EAGER (свойства, ManyToOne, OneToOne)
- LAZY (бинарни свойства, OneToMany, ManyToMany)
- FETCH join (при JPQL и Criteria API)

Кеширане ⑥

- `EntityManagerFactory.getCache()`
- операции
  - `contains()`
  - `evict()`
  - `evictAll()`

Кеширане


Application
Application Cache
EntityManager
Persistence Context
EntityManagerFactory
Shared Cache
JDBC Driver
Table/Column Caching

Optimistic Locking

- добавя се версия към всяко *entity* - `@Version`
- не се взима *lock* върху записа, докато наистина не трябва да се пише
- само се проверява дали версията е същата като в началото на транзакцията
- `OptimisticLockException`

JPA Callbacks

- инжектиране на код в жизнения цикъл на обекта:
  - `@PostLoad`
  - `@PrePersist/@PostPersist`
  - `@PreUpdate/@PostUpdate`
  - `@PreRemove/@PostRemove`
- за прости модификации (конвертиране, Y = yes, N = no)
- алтернатива им са `@EntityListeners`

Анотации и XML

- анотации
  - декларативни
  - прости и лесни за добавяне
  - изискват повторно компилиране при промяна
- XML конфигурации (orm.xml)
  - модификация без повторно компилиране
  - позната технология
  - имат по-висок приоритет

Демонстрация

Out of scope

- XML синтаксис
- сложни колекции (map)
- сложни ключове
- native SQL и именувани заявки
- pessimistic locking
- transactions (JTA)
- генериране на схема
- unit & integration тестване

Въпроси

Благодаря за вниманието