Работа со бази на податоци

Java Persistence API (JPA)

Универзитет „Св. Кирил и Методиј“ во Скопје
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ
И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Бази на податоци

- Потреба од перманентно складирање на податоци кај апликациите
  - Реискористување и по завршувањето на сесијата или терминирање на апликацијата
- Релациони бази на податоци
  - Најстари и најчесто користени кај апликации за општа намена
  - Се користи SQL јазик за манипулација со податоци
- Предизвици
  - Разлика помеѓу формата на податоците во апликациите и базите: објекти наспроти табели
  - Комплексна комуникација

Предизвици при работа со бази на податоци

- Чекори за пребарување податоци од база од страна на апликација:
  - Воспоставување на конекција
  - Испраќање на SQL команди за читање на податоци
  - Преземање на добиените резултати
  - Конверзија на добиените податоци од табеларна форма во форма погодна за користење во апликацијата
  - Затворање на конекцијата
- Справување со грешки кај секој чекор

JDBC

- Java Database Connectivity (JDBC) – стандарден Јава API за комуникација и работа со релациони бази на податоци
- Не зависи од типот на базата – содржи JDBC драјвери за секој тип
  - MySQL
  - PostgreSQL
  - Oracle
- Ја крие комплексноста на имплементацијата кај различните бази

JDBC имплементација

```
public Product findById(String id) {
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    try {
        connection = dataSource.getConnection();
        statement = connection.prepareStatement(
        "select id, name, price from Product where id=?");
        statement.setString(1, id);
        resultSet = statement.executeQuery();
        Product prod = null;
        if(resultSet.next()) {
            prod = new Product(
            resultSet.getString("id"),
                        resultSet.getString("name"),
            Decimal.valueOf(resultSet.getString("price")));
    }
    return prod;
```

```
    } catch (SQLException e) {
    // handle SQL Exception
    } finally {
        if (resultSet != null) {
            try {
            resultSet.close();
            } catch (SQLException e) {}
        }
        if (statement != null) {
            try {
            statement.close();
            } catch (SQLException e) {}
        }
        if (connection != null) {
            try {
            connection.close();
            } catch (SQLException e) {}
        }
    }
    return null;
}
```

Недостатоци на JDBC

- Комплексен код
  - Десетици линии код дури и за наједноставна операција за читање
- Не постои сепарација на Јава и SQL код
  - Проблем при промена на база
  - Тешко дебагирање
  - За секоја промена на SQL барање потребно е повторно компајлирање
- Обиди за поедноставување преку работни рамки
  - Базирани на пресликување на објекти во табели – ORM (Object-Relational Mapping)
  - Нудат механизми за размена на објекти помеѓу апликациите и базите
  - Меѓусебна некомпатибилност и нецелосна транспарентност

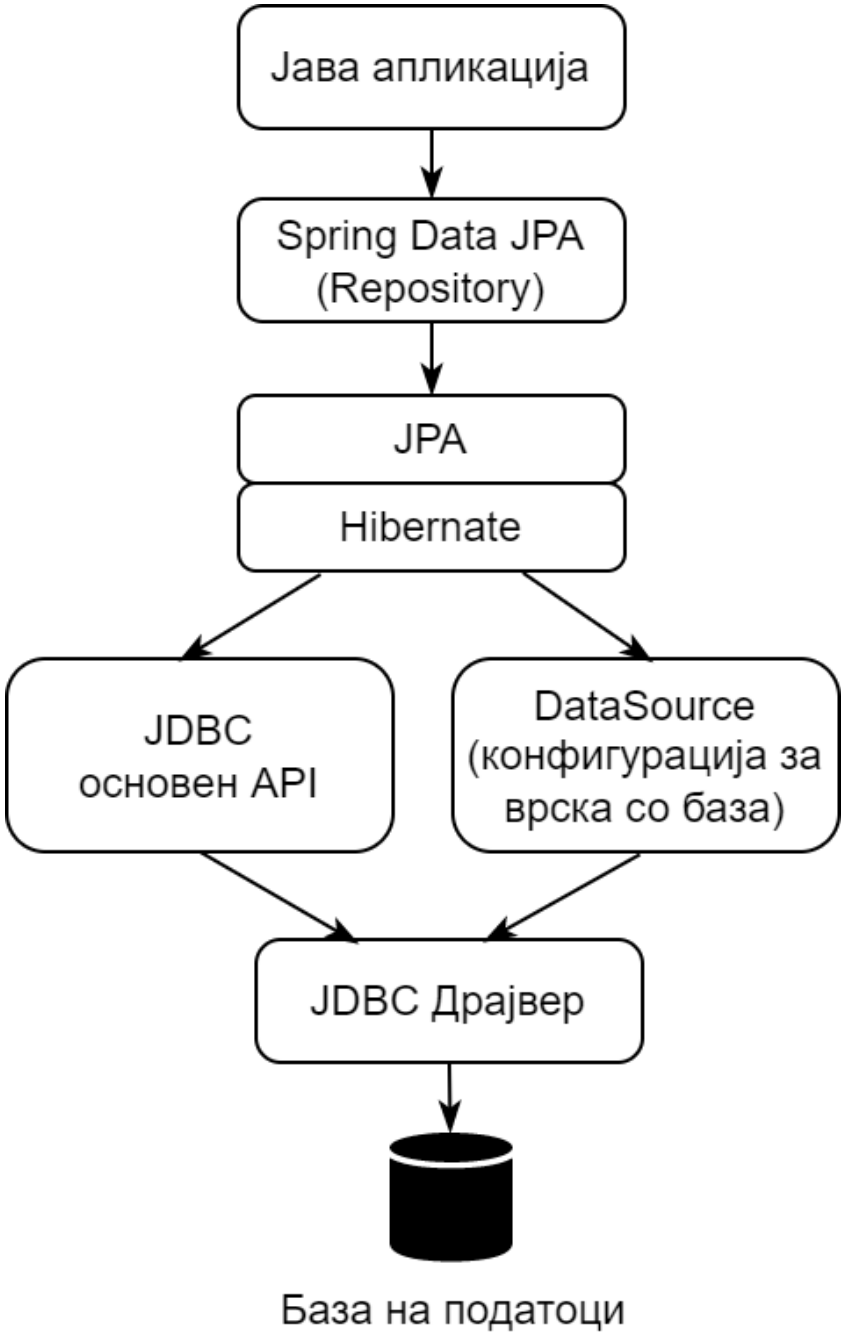
JPA

- Jakarta Persistence API (првично Java Persistence API)
  - Дел од JEE
- Спецификација од типот отворен код која дефинира само правила, но не и имплементација
- Составни делив на JPA:
  - Правила за пресликување на објекти во табели (ORM)
  - Интерфејс со методи за работа со објекти (CRUD)
  - JPQL (Java Persistence Query Language) за пишување на напредни пребарувања
- Имплементацијата ја прават JPA провајдери
  - Hibernate, EclipseLink
  - Генерираат SQL команди кои се извршуваат преку JDBC

Spring Data JPA

- JPA имплементацијата ја крие комплексноста, но за секоја мапирана класа треба да се креира репозиториум со објектно-ориентирана имплементација на CRUD операции
  - Повторување на ист код
  - Помала ефикасност на програмерите
- Spring Data JPA
  - Дел од проектот Spring Data за работа со релациони бази на податоци
  - Ново ниво на апстракција
- Креира имплементации на CRUD операции за класите користејќи JPA спецификации само преку дефиниција на **празен репозиториум**
- Креира имплементации на посложени пребарување врз основа на **имињата на методите** во репозиториумот

Преглед на апстракции



Spring Data JPA

- Конфигурација на pom.xml

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
</dependency>
<dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
</dependency>
```

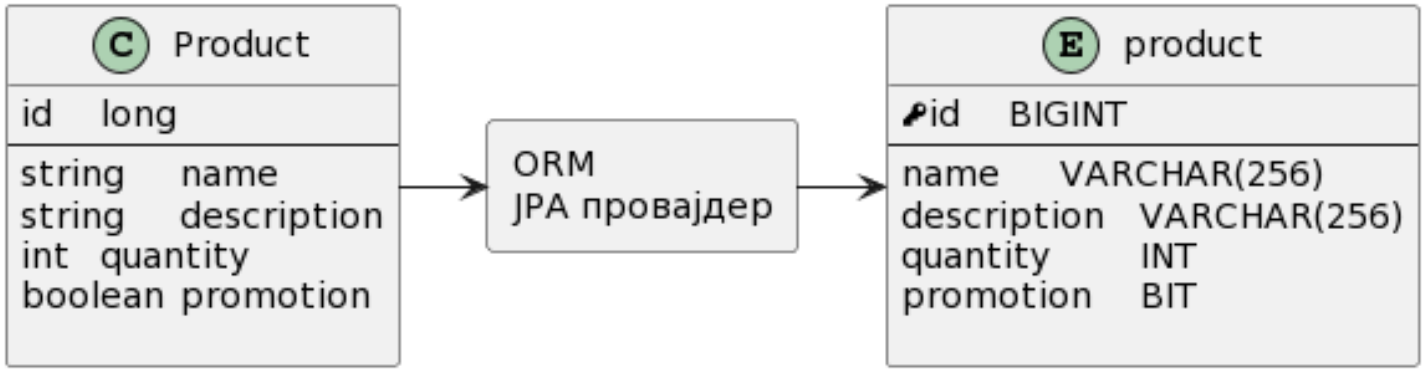
JPA

JPA ORM

- Правила за пресликување на објекти во табели (ORM)
  - Во конфигурациска .xml датотека
  - Преку анотации во самите класи

```
import java.persistance.*
```

- Ентитет - Јава класа која се пресликува во табела
  - Секоја инстанца преставува една редица во табелата
  - Својствата се колони



JPA ORM

- Својства на ентитети
  - Можност да бидат синхронизирани (persistable)
    - Во одреден момент стануваат зачувани/синхронизирани (persisted)
      - Промена на состојбата на инстанцата се синхронизира со базата
      - Креирање инстанца креира редица
      - Бришење на инстанца брише редица
    - Креирање на инстанца од ентитет не значи автоматска синхронизација
  - Уникатност
    - Поседување на **зачуван идентитет** - својство со уникатна вредност
    - Зачуваниот идентитет се пресликува во колона која е **примарен клуч**
  - Трансакционалност
    - Ако постои грешка при синхронизација на промените поради која не може процесот да продолжи, сите претходни промени се ревидираат

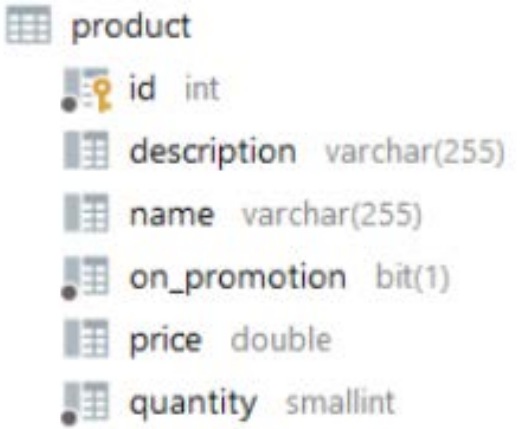
JPA Ентитети

- Услови кои треба да ги исполнува една класа за да биде ентитет:

1. Да се анотаира со **@Entity**
2. Едно од својствата да се анотира со **@Id**
3. Да содржи **public** или **protected** конструктор без аргументи

- Предефинирано именување на табели и колони: зависи од JPA имплементацијата

```
@Entity
public class Product {
    @Id
    private int id;
    private String name;
    private String description;
    private Double price;
    private short quantity;
    private boolean onPromotion;
    public Product() {}
}
```



Пресликување на табели

- Потреба од корисничко дефинирање на име на табела
  - Предефинираните имиња може да бидат клучни зборови во SQL дијалектот (user, order и сл.)
  - Се дефинираат ентитети за веќе постоечки табели во базата
  - Корисникот сака да избере различно име
- Се користи анотација `@Table` со атрибут `name`

```
@Entity
@Table(name = "eshop_user")
public class User {
    @Id
    private int id;
    private String userName;
    private String fullName;
}
```

Пресликување на колони

- Потреба од корисничко дефинирање на целокупната дефиниција на колони во базата
- Се користи анотација `@Column` со атрибути:
  - `name` - име
  - `length` – должина на знаци
  - `nullable` – дали е дозволено колоната да не содржи вредност
  - `unique` – дали вреднсота мора да биде уникатна
  - `precision` – прецизност (број на цифри)
  - `scale` – размер (број на цифри после запирката)
  - `columnDefinition` – SQL дефиниција на својствата на колоната
  - `insertable` – колоната е вклучена во INSERT
  - `updatable` – колоната е вклучена во UPDATE

Пресликување на колони

```java
@Entity
public class Product {
    @Id
    private int id;
    @Column(name = "product_name", unique = true, nullable = false)
    private String name;
    @Column(length = 2000)
    private String description;
    @Column(columnDefinition = "decimal(6,2) default 100.00")
    private Double price;
    private short quantity = 10;
    @Column(name = "promotion")
    private boolean onPromotion;
}
```

Пресликување на примарни клучеви

- Се користи анотација `@Id`
- Може дополнително да се дефинираат правила со помош на @Column
- Дефинирање на колона како примарен клуч само по себе не обезбедува генерирање уникатни вредности
- Анотација `@GeneratedValue` со аргумент `strategy` со можни вредности:
  - `AUTO` – предефинирана вредност, користи една од трите стратегии во зависност од користената база
  - `SEQUENCE` – секвентен објект во базата кој генерира вредности (глобален карактер)
  - `IDENTITY` – автоматско инкрементирање на клучевите на секоја табела
  - `TABLE` – последната генерирана вредност се чува во табела (глобален карактер)

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
    "seq_gen")
private int id;
```

Пресликување на типови на податоци

- Поддржани типови на податоци како перзистибилни:
  - Примитивните типови `boolean`, `byte`, `short`, `char`, `int`, `long`, `float` и `double`.
    - Еквивалентните обвиткани класи на примитивните типови од пакетот java.lang: `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float` и `Double`.
  - Низи од знаци и бајти: `char[]`, `byte[]`, `Char[]` и `Byte[]`
  - Нумерички вредности `java.math.BigInteger`, `java.math.BigDecimal`
  - Текстуални низи `java.lang.String`.
  - Типови за означување на време во Јава `java.util.Date`, `java.util.Calendar`
  - Типови за означување на време во JDBC `java.sql.Date, java.sql.Time` и `java.sql.Timestamp`.
  - Енумерации
  - Листи, колекции и мапи
  - Серијализибилни класи

Пресликувања кај Postgre:

- `int` во `INTEGER`
- `short` во `SMALLINT`
- `boolean` во `BIT`
- `String` во `VARCHAR(256)`

Пресликување на енумерации

- Анотација `@Enumerated`
- Во база се внесува редоследна вредност
  - Почнува од 0
- Параметар `EnumType.ORDINAL`
  - Предефинирано однесување
  - Предизвици
    - Голем број на вредности
    - Внесување нова константна во енумерациіата

```
public enum OrderStatus {INITIALIZED, PAYED, DELIVERED, FINISHED}
```

```
@Entity
@Table(name="eshop_order")
public class Order {
    @Id
    private int id;
    private Date date;
    private double amount;
    @Enumerated
    private OrderStatus status;
}
```

- Внесување на текстуална вредност во база
  - Параметар `EnumType.STRING`

```
@Enumerated(EnumType.STRING)
private OrderStatus status;
```

Транзитни полиња

- Полиња кои се игнорираат и не се пресликуваат
  - Складирање привремени вредности во рамките на апликацијата
  - Складирање пресметани вредности ...
- Анотација `@Transient`

```
public class Customer{
    private Date dateOfBirth/
    @Transient
    private short age;
    setAge(){...}
    ...
    }
```

Вградливи објекти

- Логичка поделба на табели во посебни класи
- Вградлива класа е дел од ентитет
  - Анотација `@Embeddable`
  - Не може да егзистира самостојно, туку како својство на ентитет
- Својство на една класа може да биде вградлива класа
  - Анотација `@Embedded`
- Една вградлива класа може да се искористи во дефиниција на својства на повеќе ентитети

Вградливи објекти

```java
@Entity
public class Customer {
    @Id @GeneratedValue
    private int id;
    private String firstName;
    private String lastName;
    private String street;
    private String city;
    private Integer postalCode;
    private String country;
}
```

=

```java
@Embeddable
public class Address {
    private String street;
    private String city;
    private Integer postalCode;
    private String country;
}
```

+

```java
@Entity
public class Customer {
    @Id @GeneratedValue
    private int id;
    private String firstName;
    private String lastName;
    @Embedded
    private Address address;
}
```

Релации

- Ентитетите најчесто се меѓусебно поврзани со релации
  - Својството на еден ентитет е друг ентитет или колекција од ентитети
  - Пример: Order – Customer
    - Инстанца на Order може да има референца кон инстанца од Customer
    - Инстанца на Customer може да има референца кон колекција од инстанци на Order
- Релациите имаат насока со извор и дестинација
  - Извор: Order –> Дестинација: Customer
  - Извор: Customer –> Дестинација: Order
- Поделба на релации според насока
  - Еднонасочни (изворот „знае“ за дестинацијата, но не и обратно)
  - Двонасочни релации (и двете страни „знаат“ една за друга)

Релации

- Поделба на релации според тип
  - Еден-кон-еден (анг. one-to-one)
  - Повеќе-кон-еден (анг. many-to-one)
  - Еден-кон-повеќе (анг. one-to-many) и
  - Повеќе-кон-повеќе (анг. many-to-many)
- Едноставна имплементација на релации кај објектно-ориентиран модел
- Предизвик: концептуална разлика во имплементација кај релациони бази на податоци
  - Користење на надворешни клучеви
  - Користење на дополнителни табели (повеќе-кон-повеќе)
- Преку користење на анотации, JPA провајдерот генерира соодветна имплементација во базата

Еден-кон-еден

- Една инстанца на изворниот ентитет е во релација само со една инстанца на дестинацискиот ентитет
  - Дестинацискатa инстанца не може да биде во релација со друга изворна инстанца
- Се користи кога е потребно својствата на една класа да се поделат
  - Логичка поделба на ниво на класа и
  - Физичка поделба на ниво на база на податоци
- Ентитетите од релацијата се мапираат во две различни табели
- Анотација `@OneToOne`

Еднонасочна релација еден-кон-еден

- Се користи @OneToOne само пред својството на изворниот ентитет
  - Често употребувана во пракса

```
@Entity
public class Customer {
    @Id @GeneratedValue
    private int id;
    private String firstName;
    private String lastName;
    @OneToOne
    @JoinColumn(name="address_id")
    private Address address;
}
```

```
@Entity
public class Address {
    @Id @GeneratedValue
    private int id;
    private String street;
    private String city;
    private Integer postalCode;
```

Еднонасочна релација еден-кон-еден

- Се имплементира преку надворешен клуч
- Поделба на страни во релација според позиција на надворешен клуч
  - **Сопственик на релација** (owner) – ентитетот чија табела го поседува надворешниот клуч
    - @JoinColumn се сместува кај сопственикот
    - Опционална анотација
    - Се препорачува за полесна идентификација на сопственикот
  - **Поседувана или инверзна страна** (owned/inverse side) – другата табела на која се однесува надворешниот клуч

Двонасочна релација еден-кон-еден

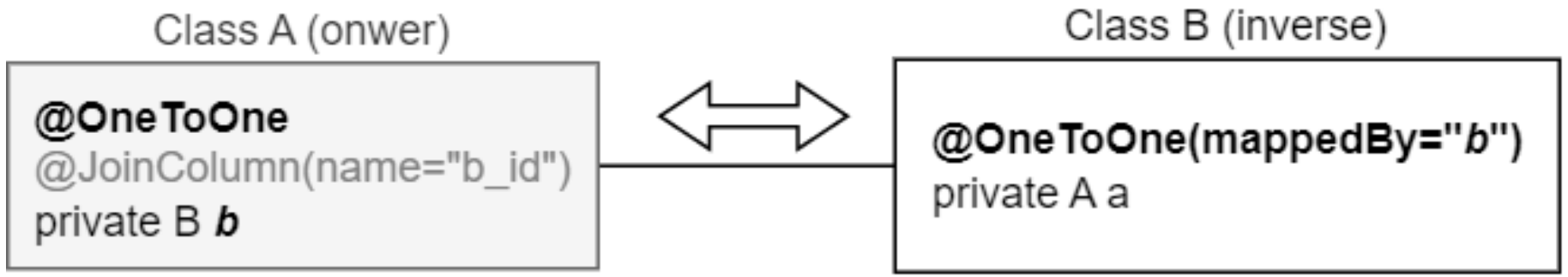
- Се користи `@OneToOne` на двете страни
  - Ентитет сопственик: `@OneToOne` и `@JoinColumn` (опционално) пред својството кое се однесува на инверзниот ентитет
  - Поседуван ентитет: `@OneToOne` со аргумент `mappedBy` и вредност име на својството во сопственикот кое се однесува на инверзниот ентитет

```
@Entity
public class Customer {
    @Id  @GeneratedValue
    private int id;
    private String firstName;
    private String lastName;
    @OneToOne
    @JoinColumn(name="address_id")
    private Address address;
}
```

```
@Entity
public class Address {
    @Id @GeneratedValue
    private int id;
    private String street;
    private String city;
    private Integer postalCode;
    private String country;
    @OneToOne(mappedBy="address")
    private Customer customer;
}
```

Еден-кон-еден

- Шематски приказ на анотации



Повеќе-кон-еден

- Повеќе инстанци од изворниот ентитет се во релација само со една инстанца на дестинацискиот ентитет
- Се користи кога е потребно да се направи асоцијација кон дестинациска инстанца **без ограничување** дали некоја друга изворна инстаца веќе има направено таква асоцијација
- Една од најчесто применуваните релации во пракса
- Анотација `@ManyToOne` кај изворен ентитет

Еднонасочна релација повеќе-кон-еден

```java
@Entity
@Table(name="eshop_order")
public class Order {
    @Id @GeneratedValue
    private int id;
    private Date date;
    private double amount;
    private Status status;
    @ManyToOne
    private Customer customer;
}
```

```java
@Entity
public class Customer {
    @Id @GeneratedValue
    private int id;
    private String firstName;
    private String lastName;
    ...
}
```

Двонасочна релација повеќе-кон-еден

- Врската во обратната насока е од типот еден-кон-повеќе и се дефинира со анотацијата `@OneToMany`
- Опис на еден-кон-повеќе во следени слајдови

Еден-кон-повеќе

- Една инстанца од изворниот ентитет е во релација со повеќе инстанци на дестинацискиот ентитет
- Имплементација со колекции
- Анотација `@OneToMany` кај изворен ентитет
- Кај еднонасочна релација, JPA провајдерот не знае каде да го вметне надворешниот клуч
  - Надворешниот клуч треба да е во табелата на дестинацискиот ентитет
  - Креира трета табела со надворешни клучеви од двата ентитета (како кај повеќе-кон-повеќе)
  - Со додавање на анотација `@JoinColumn` експлицитно се наведува каде да се стави надворешниот клуч и не се генерира дополнителна табела
- Декларација на својство од типот `List<>`, `Set<>`, `Map<>`

Двонасочна релација еден-кон-повеќе

- Врската во обратната насока е од типот повеќе-кон-еден и се дефинира со анотацијата `@ManyToOne`
  - Ентитет сопственик: `@ManyToOne`
  - Поседуван ентитет: `@OneToMany` со аргумент `mappedBy` и вредност име на својството во сопственикот кое се однесува на инверзниот ентитет

```
@Entity
public class Customer {
    @Id @GeneratedValue
    private int id;
    private String firstName;
    private String lastName;
    @OneToMany(mappedBy="customer")
    private List<Order> orders
}
```

```
@Entity
@Table(name="eshop_order")
public class Order {
    @Id @GeneratedValue
    private int id;
    private Date date;
    private double amount;
    private Status status;
    @ManyToOne
    private Customer customer;
}
```

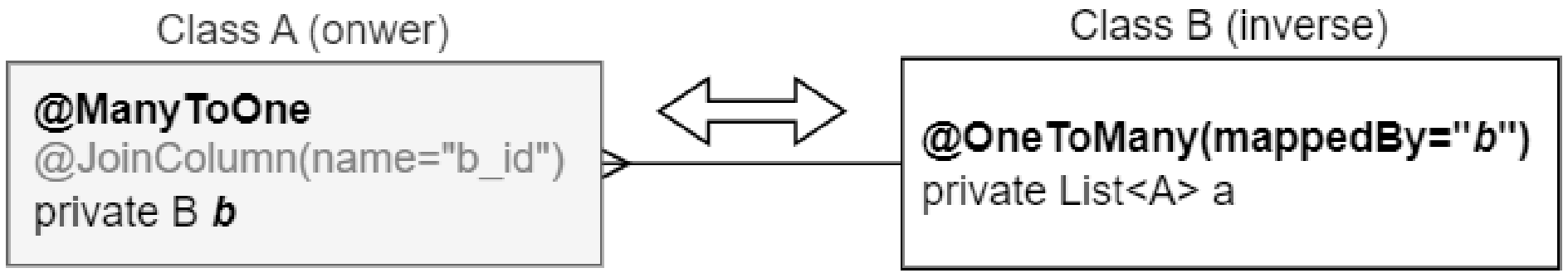
Еден-кон-повеќе

- Можност за подредување на ставките во листа (list)
- Предефинирано однесување: подредување според ID
- Анотација `@OrderBy`
  - Се посочува својство според кое се подредуваат елементите во листата
  - Се посочува насоката на подредување

```
@OneToMany(mappedBy="customer")
@OrderBy("amount DESC")
private List<Order> orders
```

Еден-кон-повеќе

- Шематски приказ на анотации



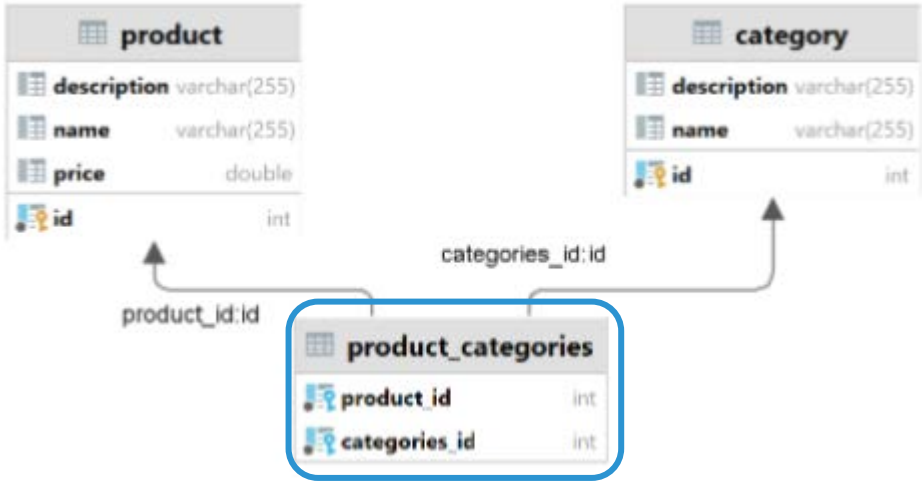
Повеќе-кон-повеќе

- Една инстанци од изворниот ентитет е во релација со повеќе инстанци на дестинацискиот ентитет
- Една инстанца од дестинацискиот ентитет е во релација со повеќе инстанци од изворниот ентитет
- Имплементација со колекции
- Анотација `@ManyToMany` кај изворен ентитет
- Се креира трета табела со надворешни клучеви од двата ентитета
- Промена на предефинирани параметри за заедничка табела и нејзините колони преку анотација `@JoinTable` и `@JoinColumn`
- Декларација на својство од типот `List<>`, `Set<>`, `Map<>`

Еднонасочна релација повеќе-кон-повеќе

```java
@Entity
public class Product {
    @Id  @GeneratedValue
    private int id;
    private String name;
    private String description;
    private Double price;
    @ManyToMany
    private List<Category> categories;
}
```

```java
@Entity
public class Category {
    @Id @GeneratedValue
    private int id;
    private String name;
    private String description;
}
```



Повеќе-кон-повеќе

- Промена на предефинирани параметри за заедничка табела и нејзините колони преку анотација `@JoinTable` и `@JoinColumn`

```
@Entity
public class Product {
    @Id  @GeneratedValue
    private String name;
    private String description;
    private Double price;
    @ManyToMany
    @JoinTable(name = "product_category",
        joinColumns = @JoinColumn(name = "prod_id"),
        inverseJoinColumns = @JoinColumn(name = "cat_id"))
    private List<Category> categories;
}
```

Двонасочна релација повеќе-кон-повеќе

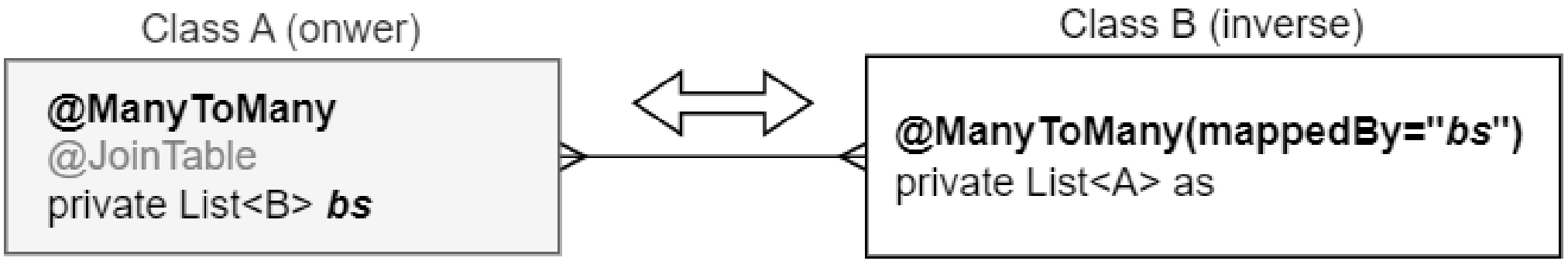
- Врската во обратната насока е исто така од типот повеќе-кон-повеќе
  - Било која страна од релацијата може да биде сопственик
  - Сопственикот го одредува програмерот по свој избор
  - Ентитет сопственик: `@ManyToMany`
  - Поседуван ентитет: `@ManyToMany` со аргумент `mappedBy` и вредност име на својството во сопственикот кое се однесува на инверзниот ентитет

```java
@Entity
public class Product {
    @Id @GeneratedValue
    private int id;
    private String name;
    private String description;
    private Double price;
    @ManyToMany
    private List<Category> categories;
}
```

```java
@Entity
public class Category {
    @Id @GeneratedValue
    private int id;
    private String name;
    private String description;
    @ManyToMany(mappedBy="categories")
    private List<Product> products
}
```

Повеќе-кон-повеќе

- Шематски приказ на анотации



Архитектура на JPA

- Креирање на инстанца од ентитет само по себе не значи и негова синхронизација со база
- За да може да е управува со ентитетите, потребно е тие да станат перзистенти (зачувани)
- EntityManager – JPA интерфејс кој дефинира методи за управување со ентитети
  - Имплементација од страна на JPA провајдер
  - Методи за CRUD операции со ентитетите
  - Управување со Query објект кој се користи за дефиниција на напредни методи за пребарување ентитети преку JPQL (JPA Query Language)

JPA архитектура

- Перзистентен контекст (persistence context) – мемориска единица која ги содржи перзистентните објекти
  - Кеш меморија за управувани објекти
  - Било која промена на објектите во него се синхронизира со база
  - Ако еден објект се извади од контекстот, не се синхронизира со база
- EntityManager овозможува
  - Внесување на инстанци во перзистентен контекст
  - Отстранување од перзистентен контекст
  - Бележење за бришење
  - Синхронизација

JPA архитектура

- Методи на EntityManager
  - `persist()` - додавање на ново-креирани ентитети во перзистентниот контекст преку методот
  - `merge()` – додавање на постоечки ентитети во перзистентниот контекст
  - `remove()` –бришење на ентитети од перзистентниот контекст
  - `find()` - вчитување на ентитети во перзистентниот контекст
  - `flush()` - зачувување на сите промени направени во ентитетите во перзистентниот контекст во базата на податоци
  - `clear()` - отстранување на сите ентитети од перзистентниот контекст
  - `createQuery()` - извршување на уникатни пребарувања во перзистентниот контекст

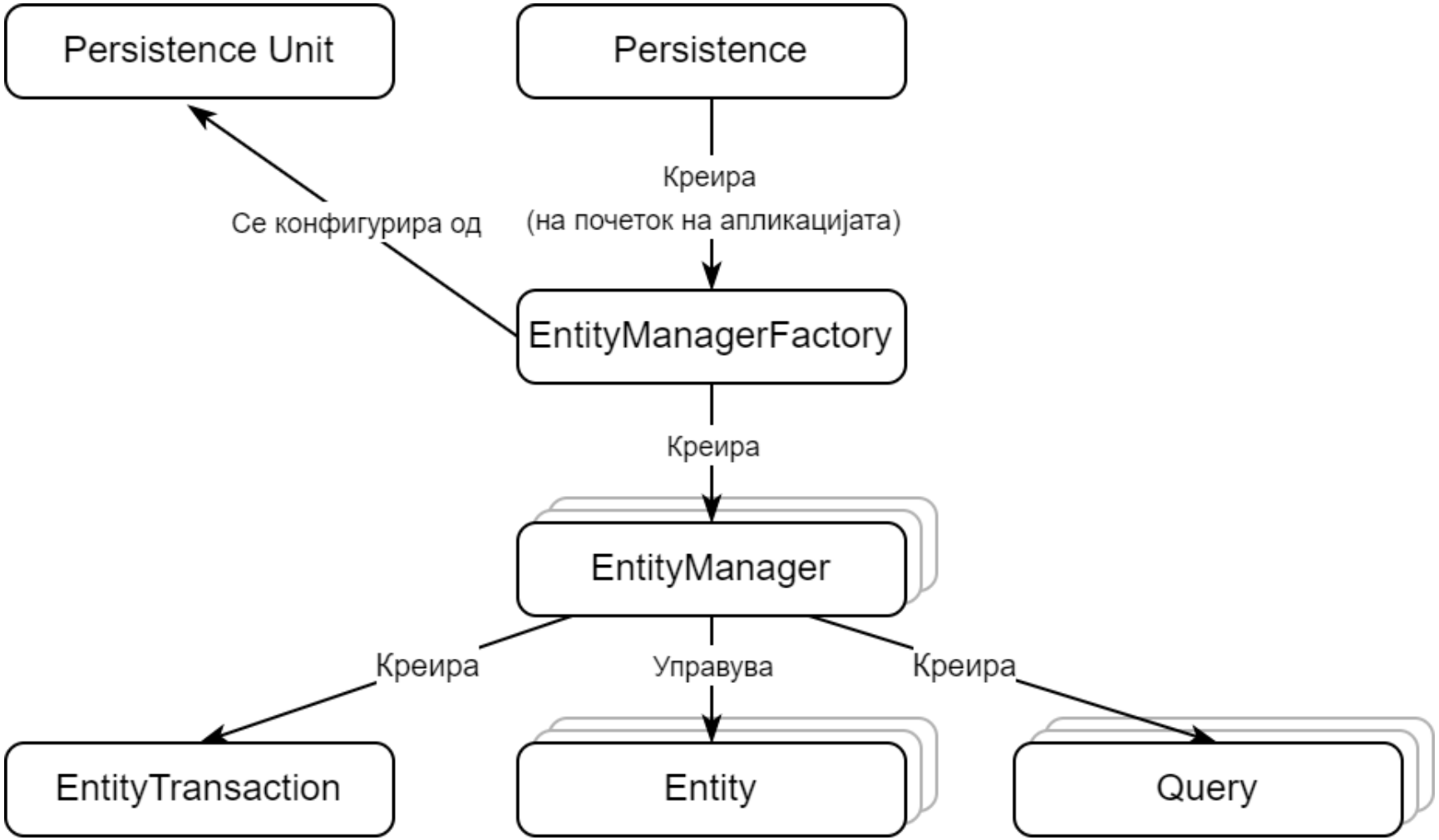
JPA архитектура

- EntityTransaction – објект преку кој се извршуваат трансакциите (SQL наредбите за работа со ентитети) спрема базата
  - Потребно е да се отвори/затвори
  - Сите објекти кои се дел од една трансакција се извршуваат атомично
  - Креиран и управуван од страна на EntityManager
- EntityManagerFactory – објект кој креира инстанци од EntityManager кога е потребно
  - Се креира при иницијализација на апликацијата врз основа на конфигурациска датотека persistence.xml (од страна на објект Persistence)
  - Креира и управува со склад од JDBC конекции кои ги доделува на креираните инстанци од EntityManager

JPA архитектура

- Persistence Unit – перзистентна единица која содржи
  - Информации специфични за базата на податоци (име, локација, корисничко име и лозинка)
  - Информации специфични за JPA (провајдер, пр. Hibernate)
  - Информации за множеството на ентитети со кои управува EntityManager
  - Мета-податоци за пресликување на ентитите во табели
    - Преземени од анотации или посебна orm.xml датотека
- Persistence.xml - датотека во која се поставува конфигурацијата за перзистентната единица
- Може да постојат повеќе перзистентни единици во една апликација (за работа со различни бази истовремено)

JPA архитектура



JPA архитектура

- EntityManager управуван од апликацијата (application managed)
  - Програмерот треба да креира инстанца на EntityMangerFactory, па од неа де креира EntityManager и потоа да управува со EntityTransaction
  - Нуди голема контрола
  - Покомплексна работа

```
@PersistanceUnit
EntityManagerFactory emf;
```

- EntityManager управуван од контејнер (container managed)
  - За целокупното управување се грижи контејнерот (пр. Spring)
  - Најчесто користен пристап
  - Програмерот треба само да добие референца кон инстанца од EntityManager со помош на анотација @PersistanceContext и да управува со ентитетите

```
@PersistanceContext
EntityManager em;
```

JPA архитектура

- Ако некој метод треба да работи со ентитети преку EntityManager, се означува со анотација `@Transactional`
- Преку концептот на AOP (Aspect Oriented Programing) контејнерот креира код кој се извршува пред и по повикот на методот
  - Пред да се влезе во методот, ако претходно не бил креиран, контејнерот креира трансакциски објект и го отвора за работа
  - Откако ќе се излезе од методот, ако повеќе нема други @Transactional методи, контејнерот ги поднесува барањата до базата (commit) и ја затвора трансакцијата

```
@Transactional
public Address create(String street, String city, int postalCode){
    Address addr = new Address();
    addr.setStreet(street);
    addr.setCity(city);
    addr.setPostalCode(postalCode);
    return em.persist(addr)
}
```

JPA архитектура

- Кај веб апликациите, за секое барање се креира нова инстанца на EntityManager
- Откако одговорот ќе биде генериран, таа инстанца се затвора.