



СБЕРБАНК ТЕХНОЛОГИИ

# **ORM, Hibernate, Spring Data JPA**

- **Что такое ORM, JPA , Hibernate**
- **Объектно-реляционное отображение**
- **Операции с базой данных с помощью Hibernate**
- **Подключение Hibernate к Spring приложению**
- **Hibernate как провайдер JPA в Spring**
- **Введение в Spring Data JPA**

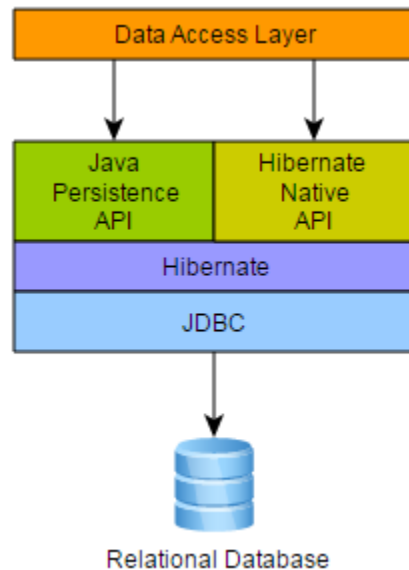
**ORM** (object relational mapping) – делегирует доступ к БД сторонним фреймворкам, которые обеспечивают объектно-ориентированное отображение реляционных данных и наоборот.

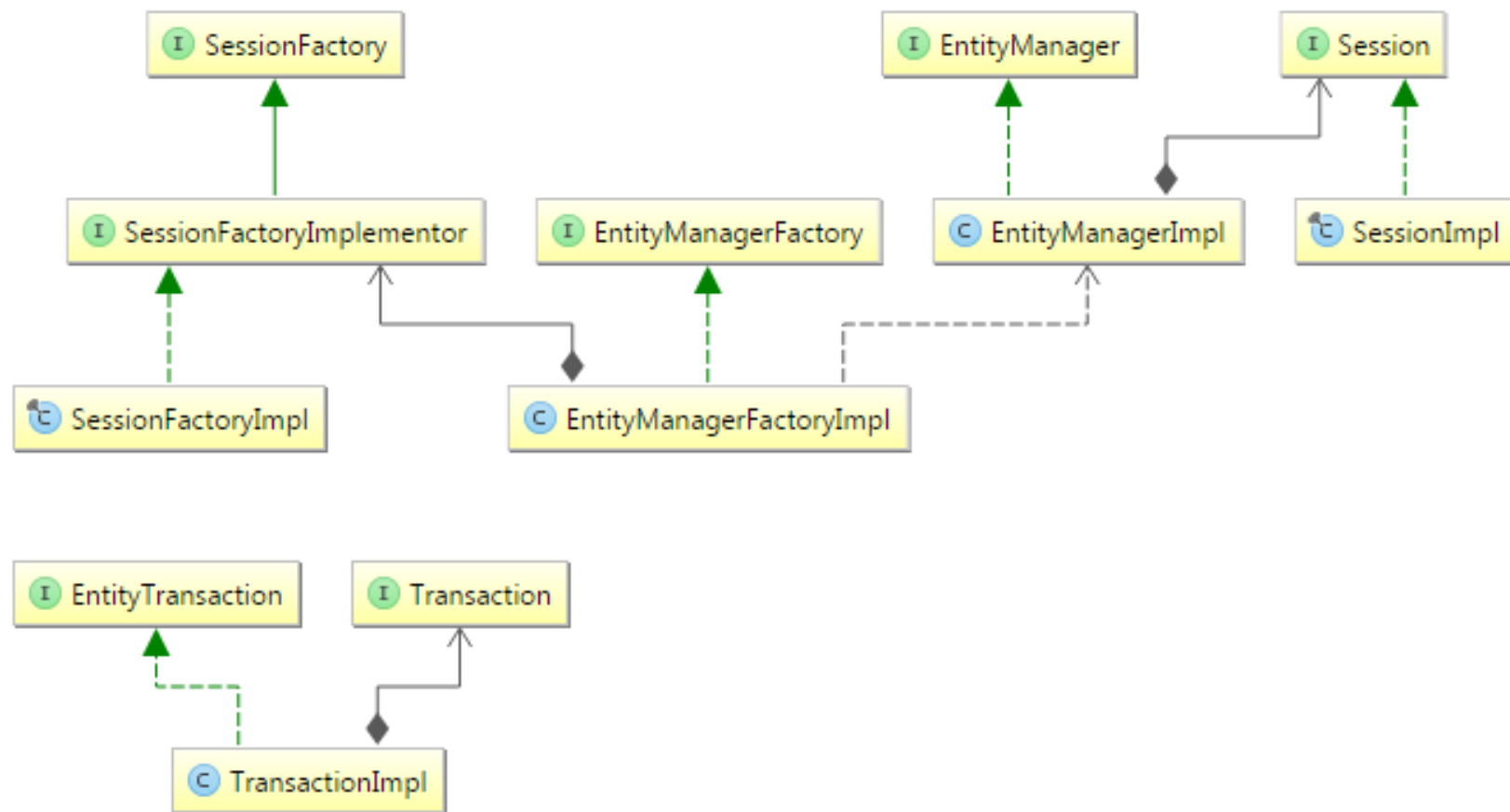
**JPA** (Java Persistence API) – спецификация описывающая API для управления ORM сущностями.

**HIBERNATE** – фреймворк реализующий спецификацию JPA

JPA позволяет решать следующие задачи:

- ORM
- `entity manager API` для выполнения CRUD операций с БД
- `Java Persistence Query Language (JPQL)` – SQL подобный язык, оперирующий объектами (не зависит от вендора БД)
- `Java Transaction API`
- Механизмы блокировок
- `Callbacks and listeners`

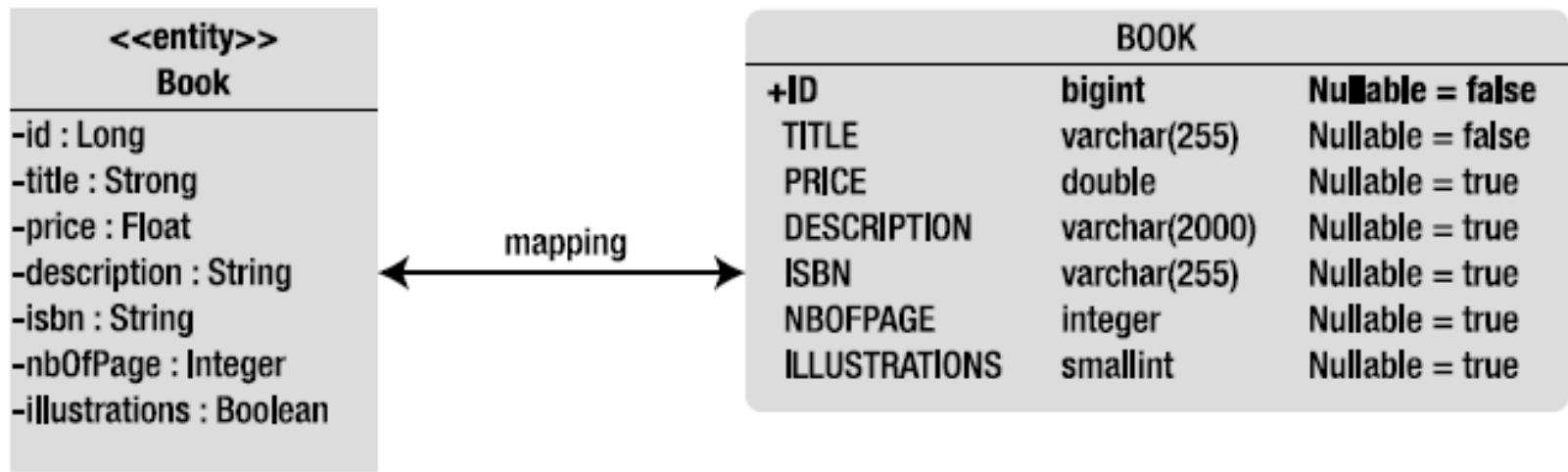




Пример класса отображаемого в базу:

```
@Entity
public class Book {
    @Id @GeneratedValue
    private Long id;
    @Column(nullable = false)
    private String title;
    private Float price;
    @Column(length = 2000)
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;
    // Constructors, getters, setters
}
```

Диаграмма мапинга:





Ограничения на отображаемый объект по спецификации JPA:

- Класс должен быть аннотирован `@Entity`
- Должен быть `public` или `protected` конструктор по-умолчанию
- Класс должен быть `top-level`
- Не могут быть `Enum` и интерфейсы
- Не может быть финальным (так же поля и методы)
- Должен имплементировать `Serializable`

Допущения Hibernate :

- Класс не обязан быть top-level
- Допускаются final классы и методы (не рекомендуется)

С помощью `@Table` можно:

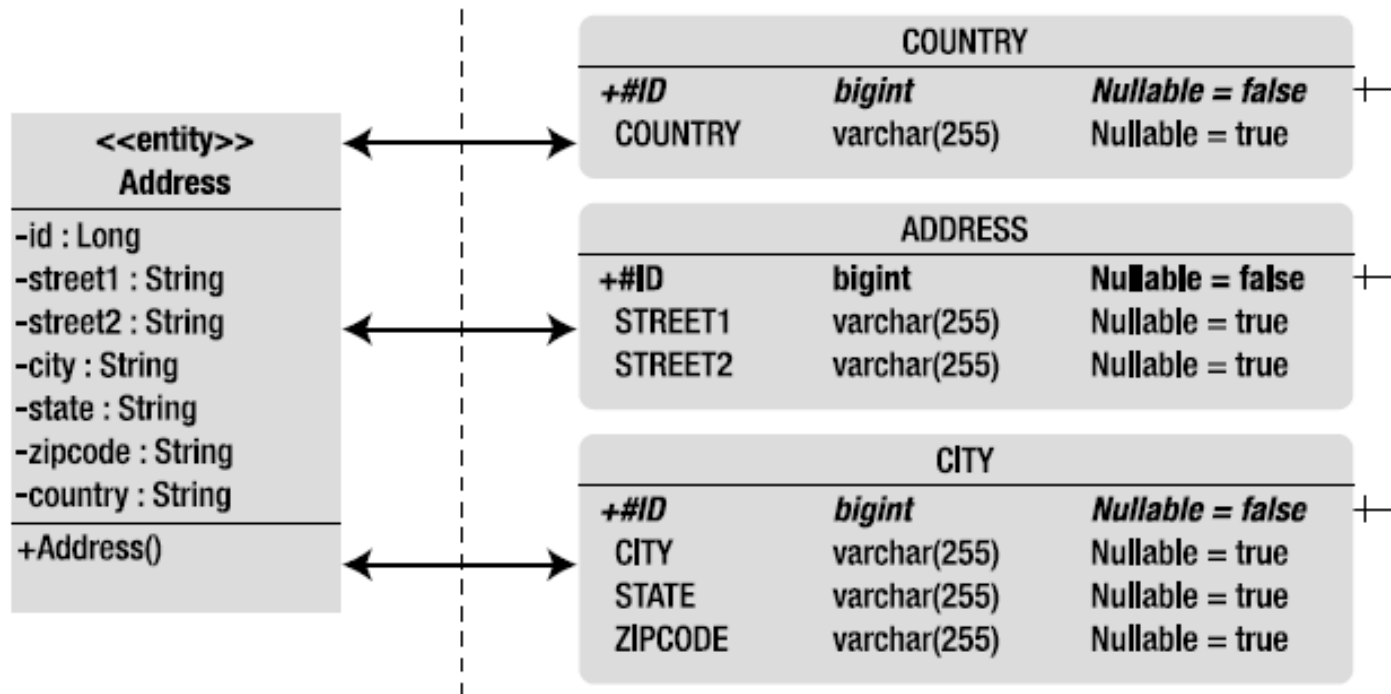
- Задать имя таблицы
- Схему

```
@Entity
@Table(name = "ITEM_TABLE")
public class Item implements Serializable{
```

С помощью `@SecondaryTable` можно добиться распределения данных в сущности между несколькими таблицами.

```
@Entity(name = "Address")
@SecondaryTables({
    @SecondaryTable(name = "City"),
    @SecondaryTable(name = "Country")
})
public class Address {
    @Id
    private Long id;
    private String street;
    @Column(table = "city")
    private String city;
    @Column(table = "country")
    private String country;
```

Модель данных для предыдущего примера:



JPA и HIBERNATE налагает ограничения на primary key:

- Каждая сущность (@Entity) должна иметь первичный ключ
- Первичный ключ может быть составным
- Первичный ключ не может измениться

Поле первичного ключа помечается @Id и может быть типом:

- Примитивный тип
- Обёртки примитивных типов
- Массивы примитивных типов
- Строки, номера, даты

Первичный ключ может быть сгенерирован автоматически на стороне приложения или HIBERNATE с помощью `@GeneratedValue`.

Поддерживаемые стратегии :

- `AUTO` – Hibernate сам выбирает подходящую стратегию
- `IDENTITY` – будут использоваться IDENTITY колонки в БД
- `SEQUENCE` – будут использоваться sequence из БД
- `TABLE` – значение будет браться из специальной таблички (для HIBERNATE - hibernate\_sequences)
- `UUID` – (только для HIBERNATE)

Составной первичный ключ можно задать и использовать так:

@Embeddable

```
public class NewsId {  
    private String title;  
    private String language;  
}
```

@Entity

```
public class News {  
    @EmbeddedId  
    private NewsId id;  
    private String content;  
}
```

```
NewsId pk = new NewsId("Richard Wright has died", "EN");  
News news = em.find(News.class, pk);
```



Или так:

```
public class NewsId {  
    private String title;  
    private String language;  
}
```

@Entity

@IdClass(NewsId.class)

```
public class News {  
    @Id private String title;  
    @Id private String language;  
    private String content;  
}
```

Сущность Entity может содержать атрибуты (поля класса).

Атрибуты могут быть:

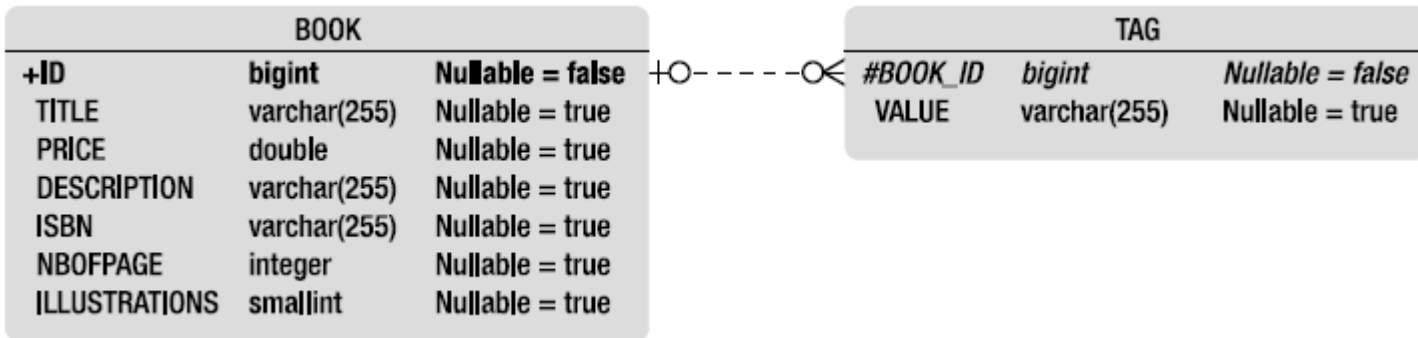
- Примитивные типы и обёртки
- Массивы байт и символов
- Строки, большие цифры, даты
- Перечисления
- Коллекции базовых и embeddable типов

Аннотация	Назначение
@Basic	Позволяет указать nullable и fetch стратегию
@Column	Позволяет указать имя колонки в БД, размер поля, nullable, updatable или insertable
@Temporal	Позволяет преобразовывать дату и время из java в формат БД и обратно (кроме java8 new Date Time API)
@Enumerated	Позволяет указать как мапить enum значения: число или строка
@Transient	Предотвращает маппинг поля

JPA и HIBERNATE позволяют хранить коллекции простых типов и встроенных типов с помощью `@ElementCollection` и `@CollectionTable`:

```
@Entity
```

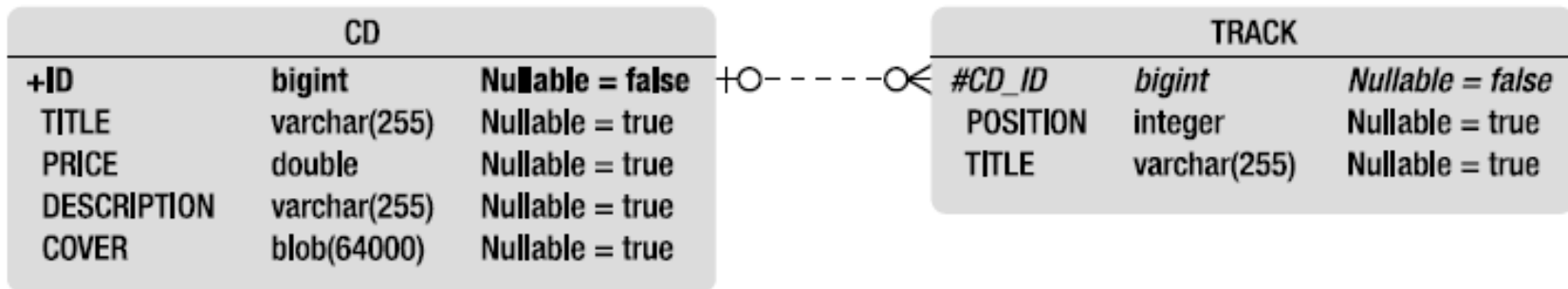
```
public class Book implements Serializable {  
    @ElementCollection(fetch = FetchType.LAZY)  
    @CollectionTable(name = "Tag")  
    @Column(name = "Value")  
    private List<String> tags = new ArrayList<String>();  
}
```



JPA и HIBERNATE так же позволяют хранить Map простых и встроенных типов:

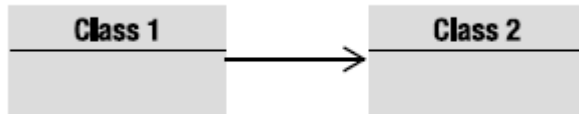
`@Entity`

```
public class CD implements Serializable {  
    @ElementCollection  
    @CollectionTable(name = "track")  
    @MapKeyColumn(name = "position")  
    @Column(name = "title")  
    private Map<Integer, String> tracks = new HashMap<>();  
}
```

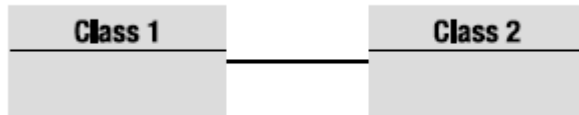


Виды:

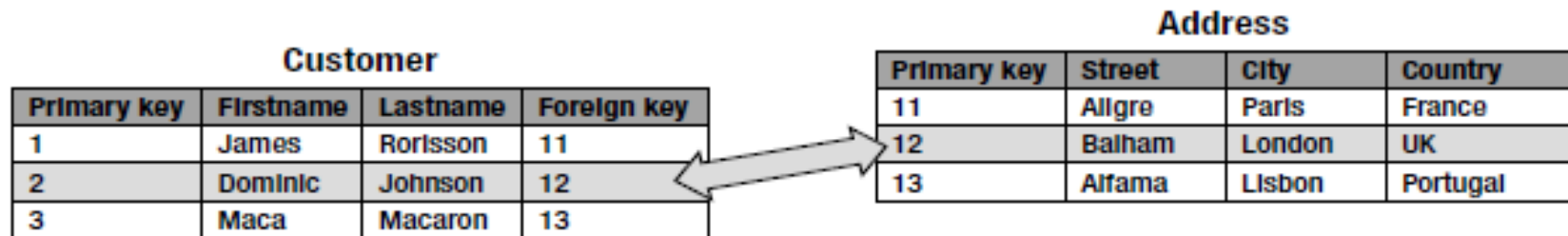
- Однонаправленный (unidirectional)



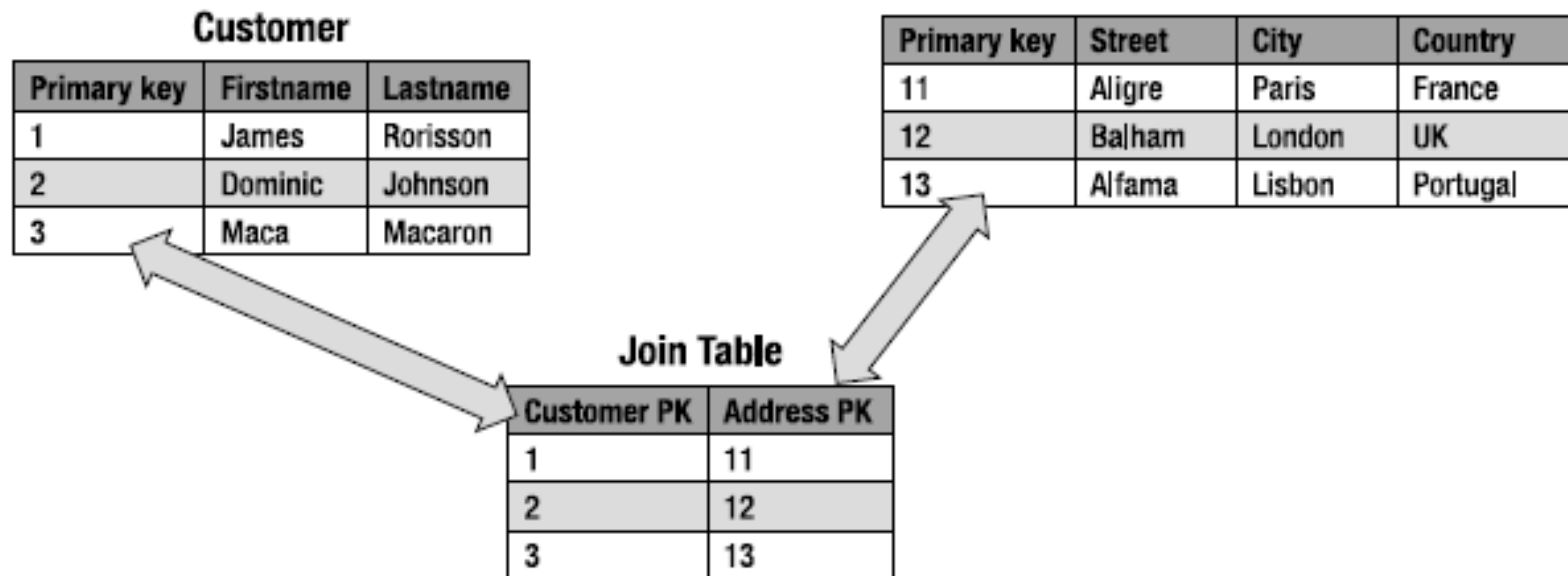
- Двухнаправленный (bidirectional)



## 1. foreign key (join column)



## 2. join table





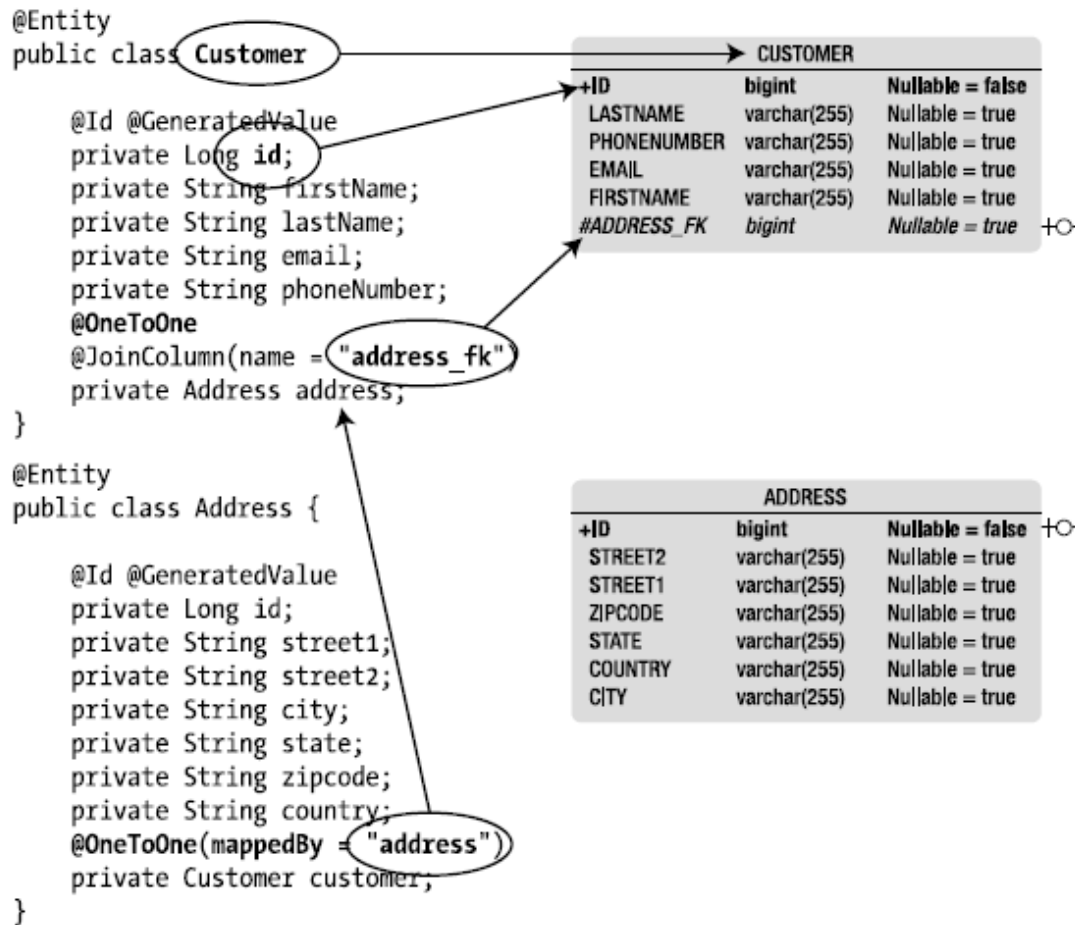
В таблице представлены возможные отношения между entities

Отношение	Направление
One-to-one	Однонаправленный
One-to-one	Двунаправленный
One-to-many	Однонаправленный
Many-to-one/one-to-many	Двунаправленный
Many-to-one	Однонаправленный
Many-to-many	Однонаправленный
Many-to-many	Двунаправленный

```
@Entity
public class Customer {
    @Id
    @GeneratedValue
    private Long id;
    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "add_fk", nullable = false)
    private Address address;
}
```

```
@Entity
public class Address {
    @Id
    @GeneratedValue
    private Long id;
}
```

# ПРИМЕР ДВУНАПРАВЛЕННОГО @ONETOONE



Рассмотрим объектную модель:

```
@Entity(name = "Order")
public class Order {
    @Id @GeneratedValue
    private Long id;
}

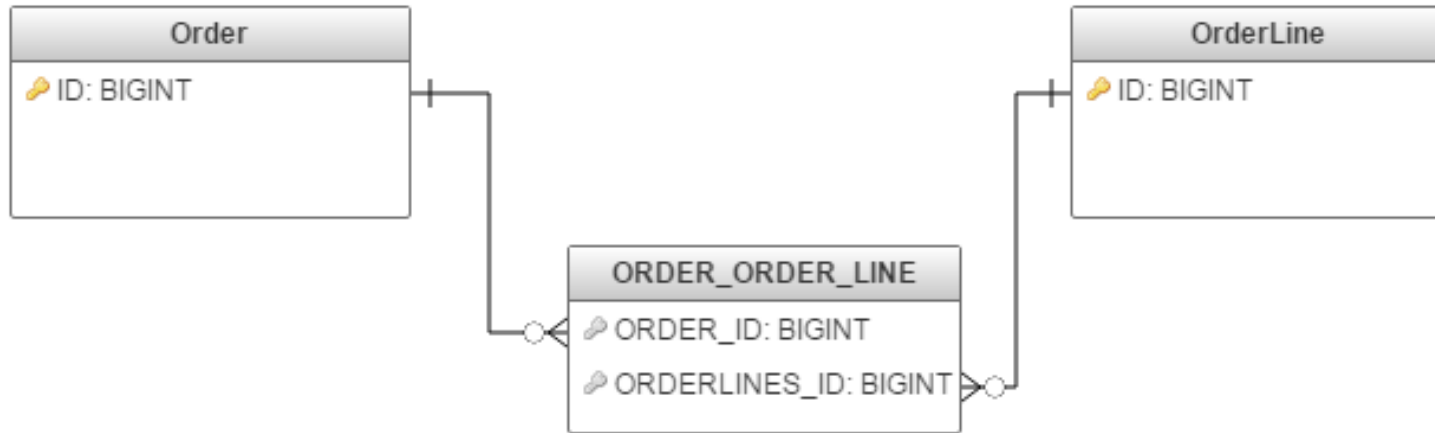
@Entity(name = "OrderLine")
public class OrderLine {
    @Id @GeneratedValue
    private Long id;
    @ManyToOne
    @JoinColumn(name = "order_id",
                foreignKey = @ForeignKey(name = "ORDER_ID_FK"))
    private Order order;
}
```

Рассмотрим объектную модель:

```
@Entity
public class Order {
    @Id @GeneratedValue
    private Long id;
    private List<OrderLine> orderLines;
}
```

```
@Entity
public class OrderLine {
    @Id @GeneratedValue
    private Long id;
}
```

По умолчанию модель данных:



Можно кастомизировать merge таблицу:

```
@OneToMany
@JoinTable(name = "jnd_ord_line",
            joinColumns = @JoinColumn(name = "order_fk"),
            inverseJoinColumns = @JoinColumn(name = "order_line_fk"))
private List<OrderLine> orderLines;
```

Можно отказаться от таблицы и использовать join column:

```
@OneToMany(fetch = FetchType.EAGER)
@JoinColumn(name = "order_fk")
private List<OrderLine> orderLines;
```

Добавим ссылку на Order из OrderLine:

```
@Entity
public class Order {
    @OneToMany(mappedBy = "order",
                cascade = CascadeType.ALL, orphanRemoval = true)
    private List<OrderLine> orderLines;
}
```

```
@Entity
public class OrderLine {
    @ManyToOne
    private Order order;
}
```

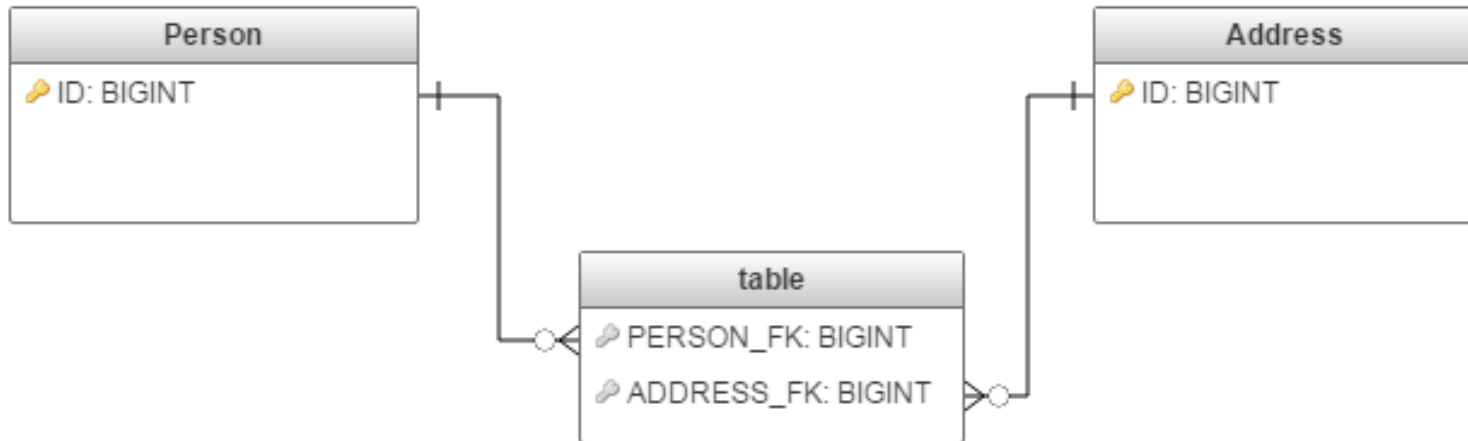


Объектная модель:

```
@Entity(name = "Person")
public class Person {
    @Id
    @GeneratedValue
    private Long id;
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    private List<Address> addresses = new ArrayList<>();
}

@Entity(name = "Address")
public class Address {
    @Id
    @GeneratedValue
    private Long id;
}
```

Модель данных:



Объектная модель:

```
@Entity(name = "Person")
public class Person {
    @Id @GeneratedValue
    private Long id;
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    private List<Address> addresses = new ArrayList<>();
}

@Entity(name = "Address")
public class Address {
    @Id @GeneratedValue
    private Long id;
    @ManyToMany(mappedBy = "addresses")
    private List<Person> owners = new ArrayList<>();
}
```

В JPA существует 2 варианта загрузки данных:

- **Eagerly** – загружаются вместе с parent объектом
- **Lazily** – загружаются при первом обращении

В hibernate можно задать extra lazy загрузку (по элементную):

`@LazyCollection(LazyCollectionOption.EXTRA)`

Отношение	Загрузка по-умолчанию
@OneToOne	EAGER
@ManyToOne	EAGER
@OneToMany	LAZY
@ManyToMany	LAZY

Для ассоциативных коллекций можно задать порядок:

```
@Entity
public class Comment {
    @Column(name = "posted_date")
    @Temporal(TemporalType.TIMESTAMP)
    private LocalDateTime postedDate;
}

@Entity
public class News {
    @OneToMany(fetch = FetchType.EAGER)
    @OrderBy("postedDate DESC")
    private List<Comment> comments;
}
```

Hibernate так же позволяет хранить коллекции ассоциаций в собственных реализациях следующих контейнеров:

- `List<>`
- `Ordered List<>` (через `@OrderBy/@OrderColumn`)
- `Set<>` (через `java equals/hashCode` контракт)
- `SortedSet<>` (через `@SortNatural/@SortComparator`)
- `Map<>`
- Массивы

JPA и Hibernate позволяют замапить наследования 3 способами:

1. **SINGLE\_TABLE** – одна таблица для каждой иерархии классов
2. **JOINED** – отдельная табличка для каждого подкласса
3. **TABLE\_PER\_CLASS** – отдельная табличка для конкретной имплементации

Настраивается через:

```
@Inheritance(strategy = InheritanceType.<стратегия>)
```

Объектная модель:

```
@Entity
public class Item implements Serializable{
    @Id @GeneratedValue
    private Long id;
    protected String title;
}

@Entity
public class Book extends Item {
    private String isbn;
}

@Entity
public class CD extends Item {
    private String musicCompany;
}
```



## Модель данных:



## Пример хранения в базе:

ID	DTYPE	TITLE	ISBN	MUSIC_COMPANY
1	Item	Pen	<i>null</i>	<i>null</i>
2	Book	Effective Java	12345	<i>null</i>
3	Cd	Soul Train	<i>null</i>	Fantastic jazz album

Можно кастомизировать поле типа:





```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE TABLE)
@DiscriminatorColumn(name = "TYPE")
@DiscriminatorValue("Book")
public class Book implements Serializable {
```




```
@Entity
@DiscriminatorValue("CompBook")
public class CompBook extends Book {
```




## Модель данных:



## Модель данных:

Book
 ID: BIGINT
 DTYPE: VARCHAR(0)
 TITLE: VARCHAR(0)
 ISBN: VARCHAR(0)

Item
 ID: BIGINT
 DTYPE: VARCHAR(0)
 TITLE: VARCHAR(0)

CD
 ID: BIGINT
 DTYPE: VARCHAR(0)
 TITLE: VARCHAR(0)
 MUSIC_COMPANY: VARCHAR(0)

Модель можно построить 2 способами:

- Проектируем объектную модель, на их основе уже модель данных (hibernate. hbm2ddl. auto)
- Сначала модель данных потом объектная модель

**Entity Manager** и **hibernate.Session** – предоставляют API для управления entity объектами и управляет их жизненным циклом.

**Persistence context** – коллекция управляемых объектов в определённое время в рамках текущей транзакции.

- **Transient** – только что созданный объект и пока не помещённый в persistence context
- **Managed (persistent)** – объект, добавленный в persistence context и имеющий идентификатор
- **Detached** – имеющий идентификатор, но отвязанный от контекста
- **Removed** – объект помещенный на удаление из БД

Метод	Назначение
save	Перевести объект в managed состояние
delete	Удалить entity
load	Ленивая загрузка entity
Find/byId().load	Полная загрузка entity
refresh	Синхронизирует entity с БД
saveOrUpdate	Снова вносит в контекст отвязанный (detach) объект
merge	Перетирает состояния объекта в БД состоянием отвязанного объекта
evict	Принудительно отвязывает объект от контекста



**JPQL** и **HQL** – SQL подобные языки, манипулирующие объектами (не типобезопасный способ)

- JPQL – входит в стандарт JPA
- HQL – расширение JPQL

Пример запроса:

```
Query<Book> query = session.createQuery(  
    "select b from Book b where b.title = :title"  
    , Book.class)  
    .setParameter("title", "Java");
```

- **Dynamic queries** – простая форма HQL/JPQL запроса
- **Named queries** – статические и не изменяемые
- **Native queries** – нативные SQL запросы
- **Criteria API** – ООП API построения запросов

Именованные запросы объявляются над Entity с помощью `@NamedQuery` и `@NamedQueries`.

```
@Entity
@NamedQueries({
    @NamedQuery(name = "findAll",
        query="select c from Customer c"),
    @NamedQuery(name = "findWithParam",
        query="select c from Customer c " +
            "                where c.firstName = :fname")
})
public class Customer {
```

Пример использования:

```
Query<Customer> query = session.createNamedQuery("findWithParam"
    , Customer.class)
    .setParameter("fname", "Java");
```

- «Почти» типобезопасный пример использования criteria api:

```
CriteriaBuilder builder = session.getCriteriaBuilder();
CriteriaQuery<Customer> query = builder.createQuery(Customer.class);
Root<Customer> c = query.from(Customer.class);
query.select(c).where(
    builder.greaterThan(c.get("age").as(Integer.class), 40));
```

- Полностью типобезопасный пример использования criteria api:

```
CriteriaBuilder builder = session.getCriteriaBuilder();
CriteriaQuery<Customer> query = builder.createQuery(Customer.class);
Root<Customer> c = query.from(Customer.class);
query.select(c).where(builder.greaterThan(c.get(Customer_.age), 40));
```

Необходимо определить:

- Data source
- Session Factory
- Transactional manager (если нам нужны транзакции)

Session Factory:

@Bean

```
public SessionFactory sessionFactory(dataSource) {  
    final LocalSessionFactoryBean factoryBean =  
        new LocalSessionFactoryBean();  
    factoryBean.setDataSource(dataSource);  
    factoryBean.setPackagesToScan("ru.sbrf.javaschool.data");  
  
    final Properties property = new Properties();  
    property.setProperty("hibernate.dialect",  
        "org.hibernate.dialect.H2Dialect");  
    property.setProperty("hibernate.show_sql", "true");  
    property.setProperty("hibernate.hbm2ddl", "validate");  
    factoryBean.setHibernateProperties(property);  
    return factoryBean.getObject();  
}
```

Transactional manager:

@Bean

```
public HibernateTransactionManager transactionManager
    (DataSource dataSource, SessionFactory sessionFactory) {
    final HibernateTransactionManager tr =
        new HibernateTransactionManager();
    tr.setDataSource(dataSource);
    tr.setSessionFactory(sessionFactory);
    return tr;
}
```

DAO уровень может выглядеть следующим образом:

```
@Transactional
@Repository("contactDao")
public class ContactDaoimpl implements ContactDao {
    private final SessionFactory sessionFactory;
    @Autowired
    public ContactDaoimpl(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
    @Transactional(readOnly=true)
    public List<Contact> findAll() {
        CriteriaBuilder builder = sessionFactory.getCurrentSession()
            .getCriteriaBuilder();
        CriteriaQuery<Contact> query = builder.createQuery(Contact.class);
        Root<Contact> c = query.from(Contact.class);
        return sessionFactory.getCurrentSession()
            .createQuery(query.select(c)).getResultList();
    }
}
```



Необходимо определить:

- Data source
- EntityManager Factory
- Transactional manager (если нам нужны транзакции)

Пример (создание data source и transactional manager – аналогично  
предыдущему примеру):

```
@Configuration
@EnableJpaRepositories
@EnableTransactionManagement
public class ApplicationConfig {
    @Bean
    public EntityManagerFactory entityManagerFactory() {
        LocalContainerEntityManagerFactoryBean factory =
            new LocalContainerEntityManagerFactoryBean();
        factory.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
        factory.setPackagesToScan("ru.sbrf.javaschool.data");
        factory.setDataSource(dataSource());
        factory.afterPropertiesSet();
        return factory.getObject();
    }
}
```

DAO уровень может выглядеть следующим образом:

```
@Transactional
@Repository("contactDao")
public class ContactDaoimpl implements ContactDao {
    private final EntityManager entityManager;
    @Autowired
    public ContactDaoimpl(EntityManager entityManager) {
        this.entityManager = entityManager;
    }
    @Transactional(readOnly=true)
    public List<Contact> findAll() {
        CriteriaBuilder builder = entityManager.getCriteriaBuilder();
        CriteriaQuery<Contact> query = builder.createQuery(Contact.class);
        Root<Contact> c = query.from(Contact.class);
        return entityManager
            .createQuery(query.select(c)).getResultList();
    }
}
```

Позволяет почти полностью отказаться от уровня DAO.

Основной маркерный интерфейс:

```
public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {
    <S extends T> S save(S entity);
    T findOne(ID primaryKey);
    Iterable<T> findAll();
    Long count();
    void delete(T entity);
    boolean exists(ID primaryKey);
    // ... more functionality omitted.
}
```

Spring может сам динамически конструировать запросы опираясь на имена методов (**find...By, read...By, query...By, count...By, and get...By**).

Пример:

```
public interface PersonRepository extends Repository<User, Long> {  
    List<Person> findByEmailAddressAndLastname(  
        EmailAddress emailAddress, String lastname);  
  
    // Enables the distinct flag for the query  
    List<Person> findDistinctPeopleByLastnameOrFirstname  
        (String lastname, String firstname);  
  
    // Enabling static ORDER BY for a query  
    List<Person> findByLastnameOrderByFirstnameAsc(String lastname);  
    List<Person> findByLastnameOrderByFirstnameDesc(String lastname);  
}
```

Предыдущий функционал подходит для ограниченного числа простых запросов.

Для более сложных можно использовать аннотацию `@Query`.

Пример:

```
public interface UserRepository extends JpaRepository<User, Long> {  
  
    @Query("select u from User u where u.emailAddress = :emailAddress")  
    User findByEmailAddress(@Param("emailAddress") String emailAddress);  
}
```

[http://docs.jboss.org/hibernate/orm/5.2/userguide/html\\_single/Hibernate User Guide.htm](http://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.htm)

<https://www.amazon.com/Beginning-GlassFish-Experts-Voice-Technology/dp/143022889X>

<http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#orm>

<http://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.at-query>