GeekBrains

Основы веб-разработки на Spring Framework

Доступ к данным в Spring. Часть 1

Hibernate. Понятие сущности. Объектно-реляционное отображение. Отображение связей «один ко многим», «один к одному», «многие ко многим». Контекст постоянства. Менеджер сущностей. JPQL. Доступ к атрибутам. Каскадные операции. Управление транзакциями.

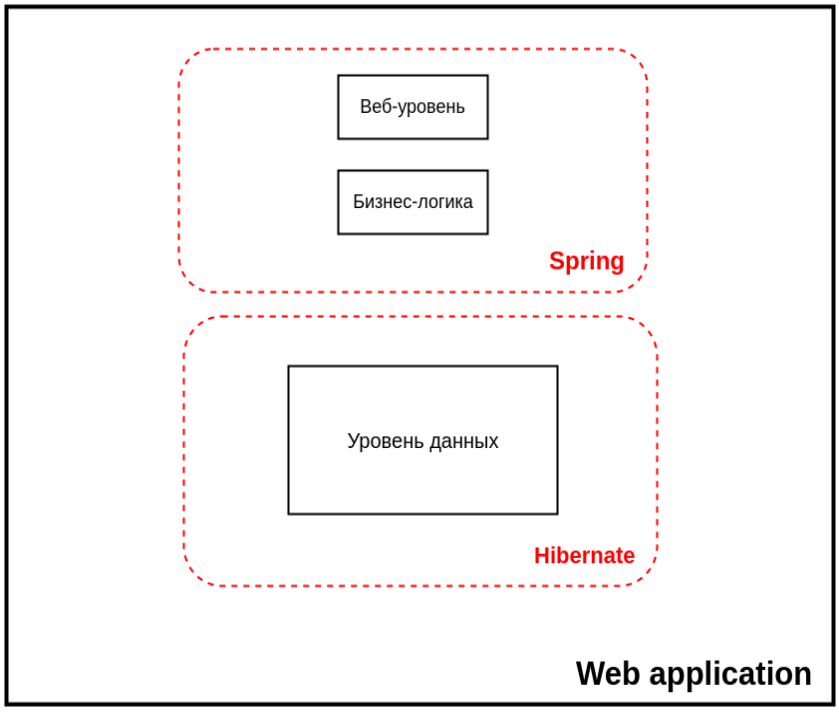
Оглавление

Введение

Spring предоставляет собственные средства для взаимодействия с СУБД. Ядром этого процесса является класс **JdbcTemplate**, но он довольно низкоуровневый и избавляет разработчика только от написания кода обработки ошибок и закрытия соединения. При создании серьезных приложений этого недостаточно, так как разработчику придется самостоятельно писать код для объектно-реляционного отображения. Для решения данной проблемы существуют ORM-библиотеки, самая популярная из которых — Hibernate.

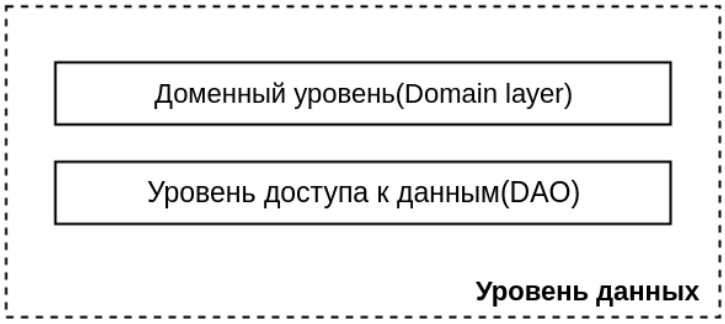
Hibernate в Spring-приложении

В предыдущих уроках мы часто упоминали компонент (бин) Spring. Теперь отойдем от темы Spring и изучим Hibernate. Это связано с тем, что практически во всех веб-приложениях, написанных с использованием Spring, для обеспечения взаимодействия с СУБД используется Hibernate, а средства фреймворка применяются для написания бизнес-логики и веб-уровня.



Уровень данных можно разделить на следующие ступени:

- **DAO** — **Data Access Object** — абстрактный объект, предоставляющий доступ к данным, хранящимся в БД. На самом деле, классы DAO-уровня — объекты Spring, но эти они используют средства Hibernate;
- **Domain** — на данном уровне хранятся модели, то есть основные объекты мэппинга. Это то, во что будут преобразовываться данные из БД (сущности).



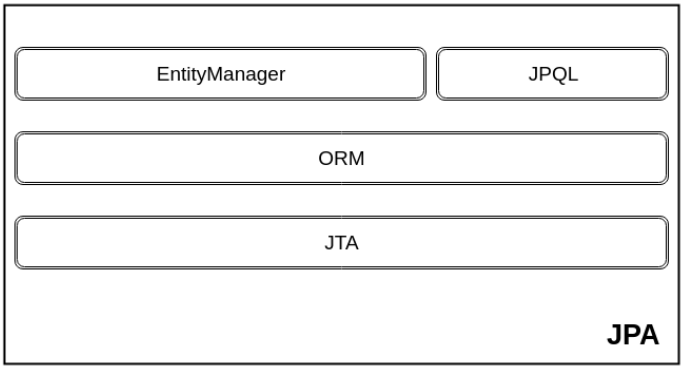
В данном уроке рассмотрим доменный уровень — способы его построения с использованием Hibernate. Работа же с базами данных с помощью Hibernate в Spring приложении будет рассмотрена на следующем занятии.

Hibernate и JPA

Изначально Hibernate развивалась как самостоятельная библиотека, не связанная со спецификацией Java EE. С третьей версии она приобрела поддержку JPA (Java Persistence API), фактически став ее реализацией: JPA определяет интерфейсы объектов для доступа к данным, а Hibernate их реализует. Поэтому в данном уроке, когда дело не касается конкретных реализаций, Hibernate и JPA будут взаимозаменяемыми понятиями.

Рассмотрим основные составляющие JPA:

- API для операций с данными, хранящимися в БД (вставки, удаления, изменения и других). Данный API описывается интерфейсом **EntityManager**;
- Объектно-реляционное отображение, которое описывает способы отображения объектов в данные, хранящиеся в БД;
- JPQL — Java Persistence Query Language — язык, позволяющий делать запросы к базе данных непосредственно из кода;
- JTA — Java Transaction API — механизмы работы с транзакциями.



При изучении доменного уровня нас интересуют способы отображения объектов в данные, хранящиеся в БД.

Для работы с PostgreSQL и подключения Hibernate к Maven проекту, необходимо добавить зависимости:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.3.Final</version>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.2.5</version>
</dependency>
```

Первая зависимость - ядро Hibernate, вторая - JDBC драйвер для работы с PostgreSQL. Далее, в ресурсы проекта добавляем файл настроек hibernate.cfg.xml.

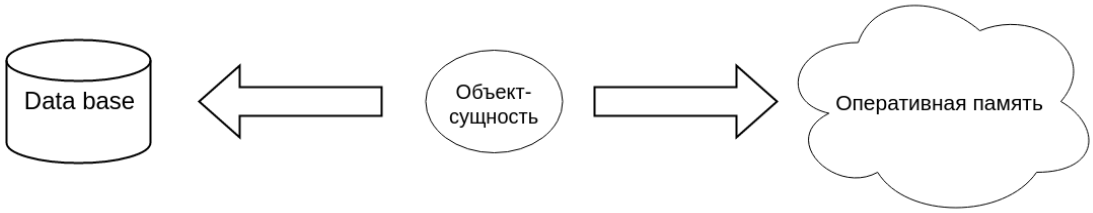
```
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property
name="connection.driver_class">org.postgresql.Driver</property>
        <property
name="connection.url">jdbc:postgresql://localhost:5432/postgres</property>
        <property name="connection.username">postgres</property>
        <property name="connection.password">admin</property>
        <property name="connection.pool_size">1</property>
        <property
name="dialect">org.hibernate.dialect.PostgreSQL94Dialect</property>
        <property name="show_sql">true</property>
        <property name="current_session_context_class">thread</property>

        <mapping class="com.geekbrains.hibernate.Person"/>
    </session-factory>
</hibernate-configuration>
```

Давайте посмотрим что означают все эти свойства в конфигурационном файле.

- **connection.driver_class** - имя класса JDBC драйвера;
- **connection_url** - url для подключения к базе данных. В данном случае сервер базы данных развернут на локальной машине, на стандартном порту 5432, и имя базы данных - postgres;
- **connection.username**, **connection.password** - логин/пароль пользователя;
- **connection.pool_size** - размер пула соединений;
- **dialect** - диалект SQL;
- **show_sql** - включение/выключение логирования выполняемых SQL запросов;
- **current_session_context_class** - указание области видимости сессии, в данном случае для каждого потока будет своя сессия;
- **mapping class** - перечисление хранимых классов.

Понятие сущности и объектно-реляционное отображение

Обычные объекты классов Java сохраняют свое состояние в оперативной памяти компьютера, но после завершения программы вся информация о них теряется. Объекты-сущности — это объекты Java, которые сохраняют свое состояние в базе данных, что обеспечивает их долгосрочное хранение. Чтобы сохранить состояние объекта, в БД должна располагаться таблица, соответствующая классу этого объекта. По аналогии со Spring, сущность — это объект, которым управляет класс **EntityManager**, относящийся к Hibernate.



Чтобы сущность могла сохранять свое состояние в базе данных, необходима возможность сохранения ее компонентов в БД. Объектами мэппинга могут быть следующие элементы:

- поля класса;
- класс;
- связи (отношения) между классами;
- и другие.

Чтобы объекты класса являлись сущностями, должны выполняться следующие условия:

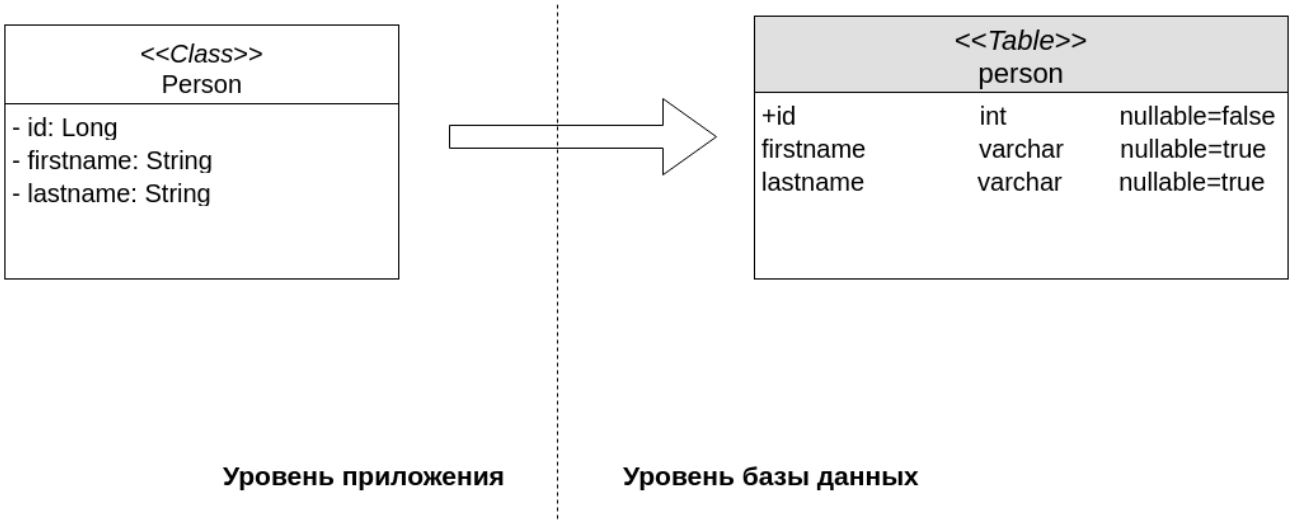
- наличие аннотации **@Entity**;
- наличие поля, в котором будет храниться уникальный идентификатор сущности. Оно должно быть снабжено аннотацией **@Id**;
- наличие конструктора без аргументов (конструктора по умолчанию);
- отсутствие модификатора **final**.

Порядок объявления сущности:

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

  private String firstname;

  private String lastname;

  // Геттеры и сеттеры
  // ...
}
```

Прежде чем вдаваться в детали конфигурирования, отметим, что **Hibernate** (как и Spring) использует подход «конфигурация в порядке исключения». Фактически это означает, что если не задано иное, то будет использоваться поведение по умолчанию.

Для объектов вышеуказанного класса порядок отображения будет таким:



Данный процесс следует правилам:

1. Имя класса отображается в имя таблицы (**Person –> person**). Если таблица, в которую необходимо отобразить сущность, имеет другое имя, то следует использовать аннотацию **@Table** с указанием имени таблицы, в которую следует отобразить класс.

2. Имена атрибутов отображаются в имена столбцов (**firstname –> firstname**). Если имя столбца отличается от имени атрибута, необходимо использовать аннотацию **@Column** с указанием имени столбца.

3. Типы атрибутов класса отображаются в типы используемой СУБД. Этот процесс интуитивно понятен (например, **Long –> integer**), но отличается в различных СУБД.

С учетом вышесказанного предыдущее объявление сущности эквивалентно следующему коду:

```
@Entity
@Table(name = "person")
public class Person {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "id")
  private Long id;

  @Column(name = "firstname")
  private String firstname;

  @Column(name = "lastname")
  private String lastname;

  // Геттеры и сеттеры
  // ...
```

```
}
```

Когда класс снабжен аннотацией **@Entity**, все его атрибуты по умолчанию будут отображаться в столбцы ассоциируемой таблицы. Но если нет необходимости в отображении какого-либо атрибута, следует применить к нему аннотацию **@Transient**.

Большинство аннотаций из данного листинга интуитивно понятны, но стоит подробнее рассмотреть **@GeneratedValue**.

Важно отметить, что объект становится объектом-сущностью только после сохранения в базе данных. Кроме того, любой объект-сущность должен иметь свой уникальный идентификатор. Но разработчику совсем не обязательно заботиться об этом. Как правило, это значение можно (и желательно) получить из самой базы данных. Большинство БД предоставляют собственные механизмы генерации значений id. Значение может инжектироваться в поле id объекта-сущности после его сохранения. Для этого необходимо использовать аннотацию **@GeneratedValue**. В зависимости от механизма генерации значений id в базе данных, атрибуту **strategy** аннотации **@GeneratedValue** присваиваются различные значения из перечисления **GenerationType**.

Атрибут **strategy** может иметь следующие значения:

- **GenerationType.SEQUENCE** — говорит о том, что значение id будет генерироваться с помощью sequence-генератора, созданного разработчиком в базе данных. При использовании данной стратегии необходимо дополнительно указывать имя генератора в атрибуте **name** аннотации **@GeneratedValue**;
- **GenerationType.IDENTITY** — указывает поставщику постоянства, что значение id необходимо получать непосредственно из столбца «id» таблицы, в которую мэппится данный объект-сущность;
- **GenerationType.AUTO** — предоставлялет Hibernate возможность самостоятельно выбрать стратегию для получения id, исходя из используемой СУБД;
- **GenerationType.TABLE** — говорит о том, что для получения значения id необходимо использовать определенную таблицу в БД, содержащую набор чисел.

Оптимальный подход — использование **GenerationType.IDENTITY**. Аннотация говорит Hibernate о том, что после сохранения объекта в базе данных необходимо получить значение из столбца, на который отображается атрибут id, и присвоить его объекту-сущности. А каким образом в этом столбце появится значение после вставки строки с информацией об объекте, остается на совести разработчика. Данный подход удобен при использовании СУБД PostgreSQL, в которой такому столбцу можно задать тип **serial** — и СУБД будет автоматически генерировать значения для данного столбца после вставки строки. В следующем уроке при рассмотрении контекста постоянства мы вернемся к данной теме.

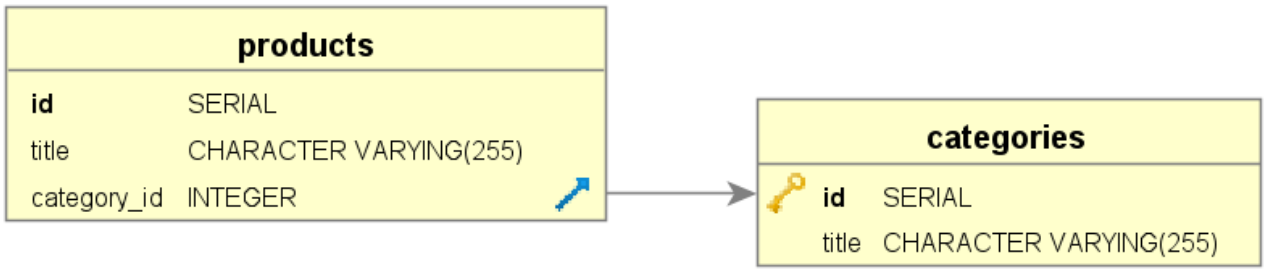
Отображение связей

На практике кроме простых атрибутов в сущностях присутствуют и связи между классами. Существует три вида связей: один к одному, один ко многим, многие ко многим. В базе данных организуются аналогичные связи между таблицами — за счет использования механизма внешних ключей. Давайте рассмотрим способы отображения связей.

Один ко многим

Связь «один ко многим» отображается с помощью аннотации **@OneToMany**, **@ManyToOne** и **@JoinColumn**. Представим, что у нас есть класс товара и категории (товаров). При этом каждый товар относится только к одной категории (*это условность в данном случае*), но одна категория может включать большое количество товаров. Данная ситуация относится к виду связи «один ко многим» (или «многие к одному»). В таблице данная связь отображается следующим образом:

```
CREATE TABLE categories (id serial, title varchar(255), PRIMARY KEY (id));

CREATE TABLE products (id serial, title varchar(255), category_id integer
REFERENCES categories (id));
```



Эта связь достигается использованием внешнего ключа **category_id**, владельцем связи является таблица **products**. А вот как это будет организовано в коде:

```
@Entity
@Table(name = "products")
public class Product {
    @Id
    @GeneratedValue
    @Column(name = "id")
    private Long id;

    @Column(name = "title")
    private String title;

    @ManyToOne
    @JoinColumn(name = "category_id")
    private Category category;

    // ...
}

@Entity
@Table(name = "categories")
public class Category {
    @Id
    @Column(name = "id")
    @GeneratedValue
    Long id;

    @Column(name = "title")
    String title;
```

```
    @OneToMany(mappedBy = "category")
    List<Product> products;

    // ...
}
```

Данное объявление подчиняется следующим правилам:

- для атрибутов обоих классов указывается аннотация **@ManyToOne** или **@OneToMany** в зависимости от стороны связи;
- для класса **Product**, который является владельцем связи, используется атрибут **name** аннотации **@JoinColumn**, которая указывает на столбец с внешним ключом в таблице (в данном случае `@JoinColumn(name = "category_id")`);
- для класса **Category** в параметре **mappedBy** указывается название ассоциируемого с ним поля в классе-владельце **Product** (`@OneToMany(mappedBy = "category")`).

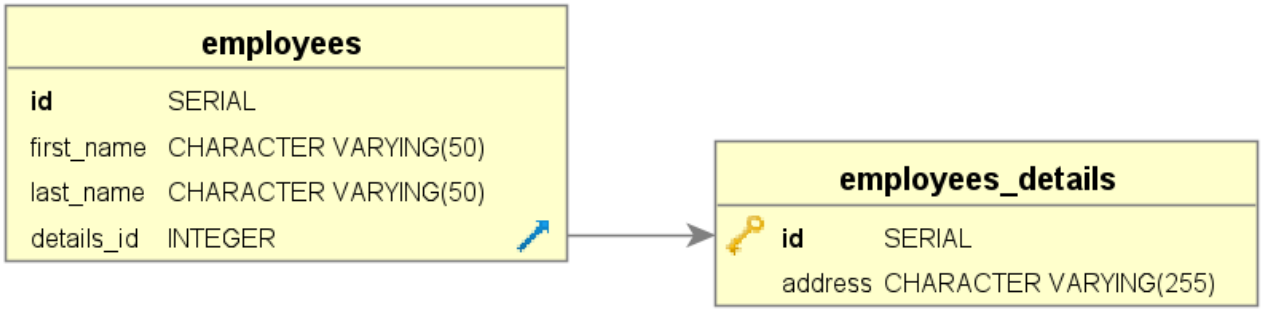
В данном случае связь между классами является двунаправленной, в отличие от таблиц, между которыми связь однонаправленная. Разработчик сам определяет, какой из вариантов использовать, но на практике оптимальна двунаправленная связь между классами. Использование двунаправленной связи избавляет нас от явного вызова кода запроса к базе данных.

Если в классе Category убрать `@OneToMany(mappedBy = "category")`, то у продукта мы сможем спросить его категорию, а вот у категории запросить список товаров уже нет (получается однонаправленная связь).

Один к одному

Представим что мы хотим хранить информацию о сотрудниках в таблице employees, но при этом для каждого сотрудника может быть заведена дополнительная “карточка” с информацией, которая будет мешать в основной таблице. Для этих дополнительных данных заведем таблицу employee_details. Запросы на создание таблиц будет выглядеть следующим образом:

```
CREATE TABLE employees_details (id serial, address varchar(255), PRIMARY KEY
(id));

CREATE TABLE employees (id serial, first_name varchar(50), last_name
varchar(50), details_id integer REFERENCES employees_details (id));
```



Связь, при которой у каждого сотрудника может быть только одна карточка с дополнительной информацией, называется “один к одному”. Для ее отображения применяется аннотация **@OneToOne**.

Объявления связей «один к одному» и «один ко многим» почти аналогичны — за исключением использования аннотации, обозначающей вид связи.

Код сущностей будет выглядеть следующим образом:

```
@Entity
@Table(name = "employees")
public class Employee {
    @Id
    @GeneratedValue
    @Column(name = "id")
    private Long id;

    @Column(name = "first_name")
    private String firstname;

    @Column(name = "last_name")
    private String lastname;

    @OneToOne
    @JoinColumn(name = "details_id")
    private EmployeeDetails details;

    // ...
}

@Entity
@Table(name = "employees_details")
public class EmployeeDetails {
    @Id
    @GeneratedValue
    @Column(name = "id")
    Long id;

    @OneToOne(mappedBy = "details")
    Employee employee;

    // ...
}
```

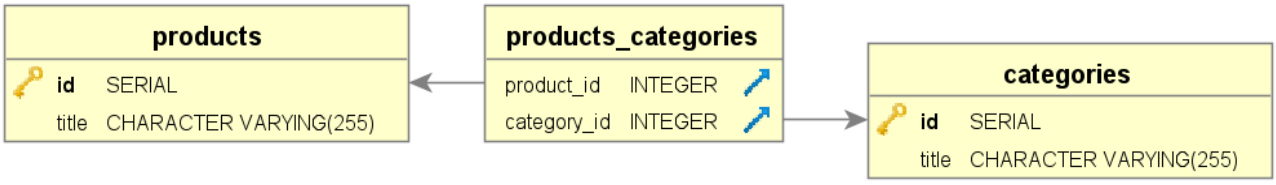
По details_id в классе Employee мы получаем ссылку на “готовый” объект типа EmployeeDetails. В таблице employee_details нет прямой ссылки на таблицу сотрудников, поэтому в классе EmployeeDetails мы используем mappedBy, чтобы получить ссылку на конкретного сотрудника.

Многие ко многим

А теперь посмотрим что изменится если мы захотим для каждому товару присваивать несколько категорий. В таком случае связь будет “многие ко многим” и нам понадобится промежуточная таблица, которая будет выступать связующим звеном между товарами и категориями.

```
DROP TABLE IF EXISTS categories CASCADE;
```

```
DROP TABLE IF EXISTS products CASCADE;

CREATE TABLE categories (id serial, title varchar(255), PRIMARY KEY (id));

CREATE TABLE products (id serial, title varchar(255), PRIMARY KEY (id));

CREATE TABLE products_categories (product_id integer REFERENCES products (id),
category_id integer REFERENCES categories (id));
```



Несмотря на кажущуюся сложность ее реализации в базе данных, в коде это выглядит довольно просто.

```
@Entity
@Table(name = "products")
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private int id;

    @Column(name = "title")
    private String title;

    @ManyToMany
    @JoinTable(
            name = "products_categories",
            joinColumns = @JoinColumn(name = "products_id"),
            inverseJoinColumns = @JoinColumn(name = "category_id")
    )
    private List<Category> categories;

    // ...
}

@Entity
@Table(name = "categories")
public class Category {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private int id;

    @Column(name = "title")
    private String title;

    @ManyToMany
    @JoinTable(
            name = "products_categories",
            joinColumns = @JoinColumn(name = "category_id"),
```

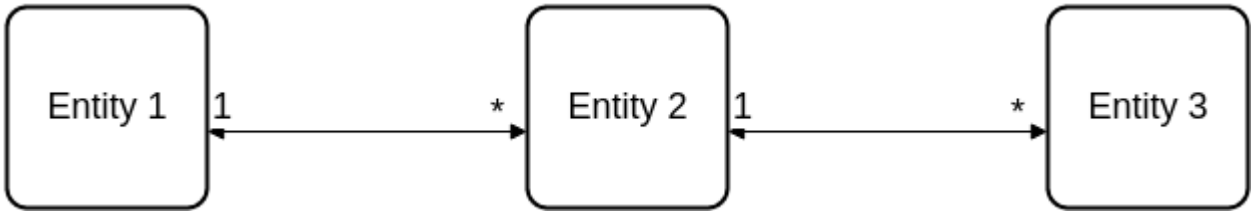
```
            inverseJoinColumns = @JoinColumn(name = "products_id")
    )
    private List<Product> products;

    // ...
}
```

Отображение связи «многие ко многим» осуществляется так:

- к атрибутам обоих классов применяется аннотация **@ManyToMany**;
- аннотация **@JoinTable** применяется для класса-владельца связи, в параметре **name** которой указывается имя таблицы соединения. В параметрах **joinColumns** и **inverseJoinColumns** указывается имя столбца, в котором отображается атрибут данного и ассоциированного класса соответственно;

Доступ к атрибутам

Представим, что есть сущность, полем которой является коллекция сущностей (связь «один ко многим»), а поле дочерней сущности — тоже коллекция сущностей и так далее. В итоге совокупность связей будет выглядеть следующим образом:



Вероятно, что при получении из БД **Entity 1** за ней тянулась бы целая цепочка коллекций других сущностей, и это сказалось бы на производительности негативно. Но JPA задает механизм для точного определения вида инициализации. Есть две стратегии выборки, которые задаются в атрибуте **fetch** аннотаций **@OneToOne**, **@OneToMany**, **@ManyToOne**, **@ManyToMany**:

- **LAZY** — «ленивая» выборка, при которой поля, являющиеся другими сущностями, не вытягиваются из базы данных вместе с основной сущностью, а запрашиваются отдельно (дополнительным запросом в БД), только при вызове геттера данного поля;
- **EAGER** — ранняя выборка, при которой поля, являющиеся сущностями, вытягиваются из базы вместе с основной сущностью с помощью одного запроса к БД.

Атрибут **fetch** аннотаций связи может иметь следующие значения:

- **FetchType.LAZY** — для ленивой выборки;
- **FetchType.EAGER** — для ранней выборки.

В коде стратегия выборки задается так:

```
@Entity
```

```java
@Table(name = "country")
public class Country {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "id")
  private Long id;

  @Column(name = "title")
  private String title;

  @OneToMany(mappedBy = "country", fetch = FetchType.EAGER)
  private List<City> cities;

  // Геттеры и сеттеры
}
```

По умолчания каждая из связей использует стратегии следующим образом:

Аннотация	Стратегия выборки по умолчанию
@OneToOne	EAGER
@ManyToOne	EAGER
@OneToMany	LAZY
@ManyToMany	LAZY

Для связи OneToOne можно применить только EAGER-стратегию.

Выбор стратегии остается на совести разработчика. EAGER-стратегия позволит загружать все данные с помощью небольшого количества запросов к БД. LAZY-стратегия не позволит заполнить всю используемую память. Вы будете контролировать, какой объект будет загружаться, но каждый раз придется осуществлять доступ к базе данных.

Задавать стратегию выборки можно и для простых атрибутов. Это можно осуществить с помощью аннотации **@Basic**.

Каскадные операции

Помимо стратегии выборки в атрибутах аннотаций связи можно указывать параметры каскадирования операций. Фактически, это означает, что при удалении сущности A из соответствующей таблицы в БД удаляется и ее дочерняя сущность B. Например, если есть сущность **Person**, которая включает в себя поле класса **Contact**, то вместе с объектом класса **Person** будут удалены строка в таблице **person** и ассоциированная с ней строка таблицы **contact**:

В коде каскадирование задается в атрибуте **cascade** аннотаций связи. Данный атрибут может принимать следующие значения перечисления **CascadeType** пакета **javax.persistence.CascadeType**:

- **CascadeType.ALL** — каскадирование будет применяться ко всем операциям;
- **CascadeType.REMOVE** — только к методу удаления;

- **CascadeType.PERSIST** — только к методу сохранения;
- **CascadeType.MERGE** — к методу обновления;
- **CascadeType.REFRESH** — к методу синхронизации с БД;
- **CascadeType.DETACH** — каскадирование применяется к методу удаления сущности из контекста постоянства (но не из БД).

В коде это выглядит так:

```
@ManyToOne(cascade = CascadeType.PERSIST)
@JoinColumn(name = "details_id")
private EmployeeDetails details;
```

Кроме того, можно задавать несколько параметров:

```
@ManyToOne(cascade = { CascadeType.REMOVE, CascadeType.PERSIST })
@JoinColumn(name = "details_id")
private EmployeeDetails details;
```

Контекст постоянства и EntityManager

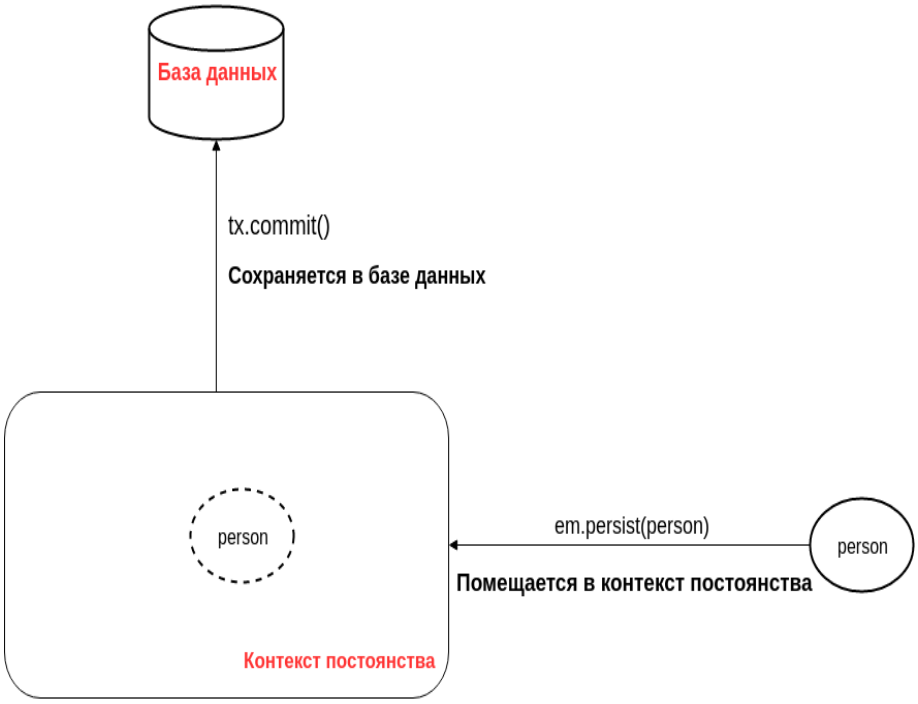
Рассмотрим порядок взаимодействия наших сущностей с базой данных. Чтобы понять этот процесс, необходимо разобраться с понятием контекста постоянства. Этот контекст представляет собой набор сущностей, которые управляются Hibernate в данный момент. Все управление сущностями возлагается на менеджер сущностей — класс **EntityManager**.

Он обладает полным набором CRUD-операций. Данные операции вызываются следующими методами:

```
public static void main(String[] args) {
    // Получаем фабрику менеджеров сущностей
    EntityManagerFactory factory = new Configuration()
            .configure("hibernate.cfg.xml")
            .buildSessionFactory();
    // Из фабрики создаем EntityManager
    EntityManager em = factory.createEntityManager();

    Person person = new Person("Ivan", "Ivanov");

    // Открываем транзакцию
    em.getTransaction().begin();
    // Create (сохраняем в базе данных, и благодаря этому сущность
    // становится управляемой Hibernate и заносится в контекст постоянства)
    em.persist(person);
    // Подтверждаем транзакцию
    em.getTransaction().commit();

    em.getTransaction().begin();
```

```
    // Read (читаем сущность из базы данных по id)
    Person anotherPerson = em.find(Person.class, 1L);
    em.getTransaction().commit();
    anotherPerson.setFirstname("Artem");

    em.getTransaction().begin();
    // Update
    em.merge(anotherPerson);
    em.getTransaction().commit();
}
```

Ниже графически представлен процесс сохранения в БД объекта-сущности **person**:



Данное изображение отражает всю суть работы контекста постоянства для любых операций. Главное, что следует запомнить, — **результат операции никак не отразится на базе данных, пока не будет произведена фиксация транзакции**.

Процесс, изображенный на данной схеме, состоит из следующих этапов:

- открытие транзакции;
- сохранение объекта в контексте постоянства с помощью метода **persist**;
- сохранение объекта в базе данных после фиксации транзакции.

Данная последовательность характерна не только для метода **persist**, но и для остальных. В случае использования Hibernate совместно со Spring не будет необходимости самостоятельно открывать и закрывать транзакции — этим будет заниматься сам Spring.

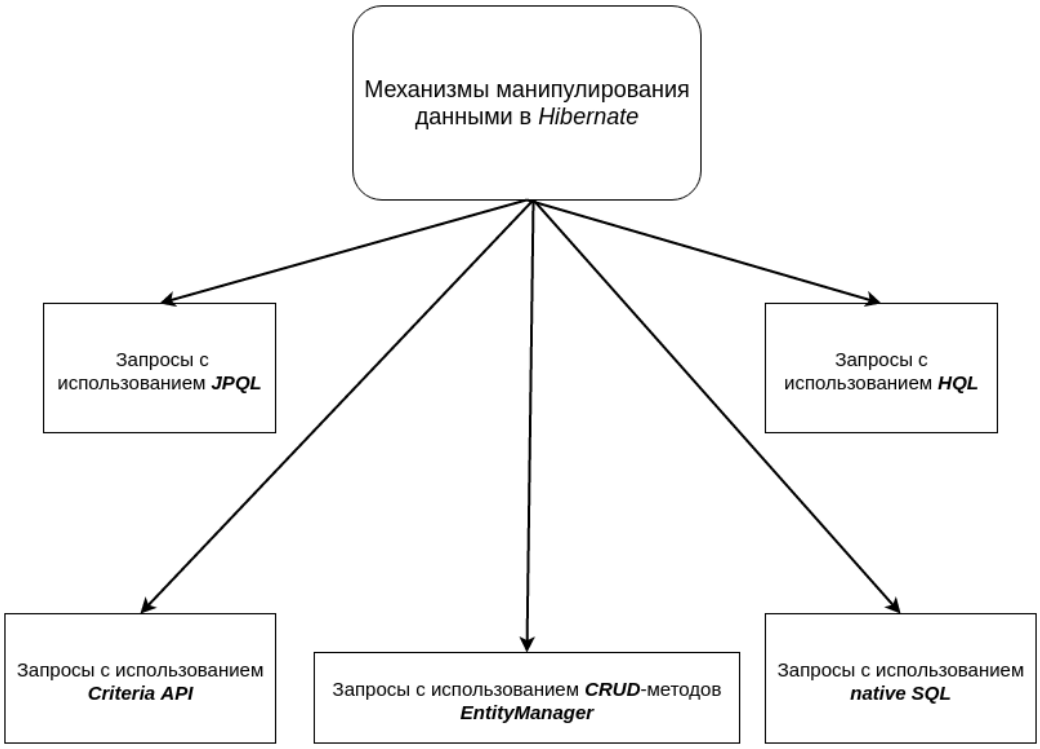
EntityManager содержит и другие методы, предназначенные для работы с сущностями:

- **detach(Object obj)** — удаляет сущность **obj** из контекста постоянства (но не из БД), и объект перестает находиться под управлением Hibernate;

- **refresh(Object obj)** — синхронизирует сущность **obj** с БД. Ее поля будут иметь те значения, которые находились в столбцах строки БД на момент применения данного метода.

JPQL

Набор методов CRUD-класса **EntityManager** ограничивает возможности по манипулированию сущностями. Например, метод **find()** позволяет искать сущность только по идентификатору. Для создания более гибких запросов нужно использовать другие механизмы:



- **Запросы с использованием JPQL (Java Persistence Query Language)** — объектно-ориентированный язык запросов, который описан в спецификации JPA. В отличие от SQL, он оперирует сущностями на уровне кода, а в дальнейшем поставщик постоянства транслирует эти запросы в SQL-запросы к БД;

- **Запросы с использованием HQL (Hibernate Query Language)** — аналогичен JPQL, но используется только в Hibernate;

- **Запросы с использованием CRUD-методов** — запросы к базе данных с помощью методов, рассмотренных в предыдущей главе;

- **Запросы с использованием Criteria API** — запросы, которые последовательно формируются с помощью объектов и методов.

Стоит отметить, что все языки запросов очень похожи на SQL.

Рассмотрим синтаксис языка запросов JPQL, который основан на HQL и позиционируется как более новая и стандартизованная версия этого языка. Запросы с использованием **Criteria API** оставим на самостоятельное изучение по дополнительным материалам.

Обзор синтаксиса

Главное отличие JPQL от SQL состоит в том, что JPQL-запросы манипулируют сущностями, то есть объектами классов. Самый простой JPQL-запрос, который делает выборку всех объектов-сущностей класса **Article** из базы данных, выглядит так:

```
SELECT a FROM Article a
```

Особенности этого запроса:

- оператор **FROM** указывает на класс (в данном случае — класс **Article**), выборку объектов которого необходимо сделать из соответствующей ему таблицы в базе данных;
- в блоке оператора **FROM** указывается псевдоним класса (в данном случае — **a**).

При использовании JPQL-запросов происходит следующее:

Если необходимо применить определенный критерий поиска, то запрос будет выглядеть так:

```
SELECT a FROM Article a WHERE a.id = 2
```

В данном случае псевдоним **a** используется для доступа к атрибутам класса.

Возвращать можно не только объекты, но и атрибуты класса. Запрос, возвращающий атрибуты объекта, может выглядеть следующим образом:

```
SELECT a.firstname, a.lastname FROM Author a
```

Если используется привязка параметров, запрос может быть таким:

```
SELECT a.firstname, a.lastname FROM Author a WHERE a.id = ?1
```

В случае именованных параметров:

```
SELECT a.firstname, a.lastname FROM Author a WHERE a.id = :id
```

Можно указать поставщику постоянства, что необходимо создать объекты из возвращаемых из БД значений. Например:

```
SELECT NEW com.geekbrains.Person(a.firstname, a.lastname) FROM Author a
```

Класс **Person** не обязан являться сущностью, но должен содержать конструктор с указанной в запросе сигнатурой.

Запросы

Динамические запросы

Изучим механизм, с помощью которого осуществляются запросы. Вся необходимая функциональность содержится в методах класса **EntityManager**.

Рассмотрим основные методы:

- **createQuery(String jpqlString)** — метод, принимающий строку JPQL-запроса и возвращающий объект класса **Query**;
- **createNamedQuery(String name)** — метод для именованных запросов, принимающий их названия и возвращающий объекты класса **Query**;
- **createNativeQuery(String sqlString)** — метод для запросов с использованием SQL, возвращает объект класса **Query** и других.

Эти методы имеют свои перегруженные аналоги, принимающие дополнительный параметр типа **Class** или **Class<T>**, которые помогают избежать лишних преобразований типов.

Но стоит отметить, что все вышеперечисленные методы не обеспечивают выполнение запроса как такового — для этого необходимо использовать:

- **getSingleResult()** — для получения одиночного объекта в качестве конечного результата запроса;
- **getResultList()** — для получения коллекции объектов как конечного результата запроса;
- и другие.

Например, получение всех авторов может выглядеть следующим образом:

```
// Осуществление запроса, возвращающего коллекцию
List<Author> authors = em.createQuery("SELECT a FROM Author a",
Author.class).getResultList();

// Осуществление запроса, возвращающего одиночный результат
Author author = em.createQuery("SELECT a FROM Author a WHERE a.id = 1",
Author.class).getSingleResult();
```

Именованные запросы

Именованные запросы более производительны, чем динамические. Это связано с тем, что преобразование JPQL-запроса в SQL происходит сразу после запуска приложения. Чтобы выполнить именованный запрос, в классе, к которому он будет осуществляться, необходимо объявить аннотацию **@NamedQuery**, содержащую следующие атрибуты:

- **name** — название именованного запроса;
- **query** — JPQL-строка запроса.

Можно объявлять несколько именованных запросов с помощью множественной аннотации **@NamedQueries**.

Объявление именованных запросов:

```
@Entity
@Table(name="author")
@NamedQueries({
  @NamedQuery(name = "Author.findAll", query = "SELECT a FROM Author a"),
  @NamedQuery(name = "Author.findById", query = "SELECT a FROM Author a WHERE
a.id = :id")
})
public class Author{
  // Fields, getter and setters
}
```

В данном листинге показан пример объявления двух именованных запросов:

- **Author.findAll** для получения всех авторов;
- **Author.findById** (аналог метода **find** класса **EntityManager**), использующий именованные параметры.

Заметьте, что формат названий именованных запросов рекомендует употреблять имя класса в качестве префикса, а само название отражает операцию и критерий.

В коде использование именованных запросов будет выглядеть так:

```
List<Author> authors = em.createNamedQuery("Author.findAll",
Author.class).getResultList();
Author author = em.createNamedQuery("Author.findById",
Author.class).setParameter("id", 1).getSingleResult();
```

Аннотации

Давайте рассмотрим список аннотаций, применяемые в Hibernate.

1. Каждый класс хранимой сущности должен иметь аннотацию **@Entity**;

2. Аннотации **@Table** позволяет задать имя таблицы, в которую будут отображаться объекты данной сущности, и настроить индексы;

```
@Table(name = "demo_annotated", indexes = {
        @Index(name = "name_idx", columnList = "name"),
        @Index(name = "id_name_idx", columnList = "id, name"),
        @Index(name = "unique_name_idx", columnList = "name", unique = true)
})
```

3. Аннотации **@OneToOne**, **@OneToMany**, **@ManyToOne**, **@ManyToMany** обозначают связи между сущностями;

4. **@Column** отвечает за настройки столбца в таблице. С помощью параметра **name** указывается имя столбца в которое будет записано значение размеченного поля. Для “ручного” формирования запроса с помощью которого будет построен столбец можно воспользоваться параметром **columnDefinition**. Здесь же можно указать ограничение NOT NULL (**nullable** = false). Чтобы запретить изменение значение какого-либо столбца параметр **updatable** переводится в false.

```
@Column(name = "manual_def_str", columnDefinition = "VARCHAR(50) NOT NULL
UNIQUE CHECK (NOT substring(lower(manual_def_str), 0, 5) = 'admin')")
String manualDefinedString;

@Column(name = "short_str", nullable = false, length = 10) // varchar(10)
String shortString;

@Column(name = "created_at", updatable = false)
LocalDateTime createdAt;
```

5. Для автогенерации времени создания объекта в базе данных и времени его обновления используются аннотации **@CreationTimestamp** и **@UpdateTimestamp**;

6. При реализации связи **@ManyToOne**, для создания внешнего ключа указывается следующая форма аннотации **@JoinColumn**

```
@ManyToOne
@JoinColumn(
        name = "product_id",
        nullable = false,
        foreignKey = @ForeignKey(name = "FK_PRODUCT_ID")
)
Product product;
```

6. Любой класс хранимой сущности обязан иметь идентифицирующий атрибут - поле, помеченное аннотацией **@Id**. Аннотация **@GeneratedValue** по умолчанию указывает что генерация будет выполнятся автоматически, и позволяет указать способ генерации.

```
public class Product {
    @Id
    @GeneratedValue
    @Column(name = "id")
    Long id;
```

8. **@Version** поле используется для версионирования, о котором речь пойдет ниже.

Оптимистическое управление параллельным доступом

Оптимистическое управление параллельным доступом подходит для случаев, когда изменения вносятся редко и в рамках одной транзакции допустимо позднее обнаружение конфликтов. Для оптимистического управления необходимо включить версионирование. При таком подходе будет “побеждать” первая подтвержденная транзакция.

Для добавления версионирования достаточно добавить в классы, помеченные аннотацией @Entity поле с аннотацией @Version.

```
@Entity
@Table(name = "items")
public class Item {
    @Id
    @GeneratedValue
    @Column(name = "id")
    Long id;

    @Column(name = "val")
    int val;

    @Column(name = "junkField")
    @OptimisticLock(excluded = true)
    int junkField;

    @Version
    long version;

    public void setVal(int val) {
        this.val = val;
    }

    public long getVersion() {
        return version;
    }

    public Item() {
    }

    public Item(int val) {
        this.val = val;
    }

    @Override
    public String toString() {
        return String.format("Item [ id = %d, val = %d, version = %d ]", id,
val, version);
    }
}
```

В таком случае, каждому экземпляру данной сущности будет присваиваться версия, которая отображается на отдельный столбец в таблицы базы данных. Для поля version можно добавить геттер, но ни в коем случае не должно быть сеттера, так как изменением этого поля занимается сам Hibernate. По сути, версия - это просто счетчик. Давайте посмотрим на следующий пример:

```
// ... тут стандартный запуск SessionFactory
session = factory.getCurrentSession();
session.beginTransaction();
Item item = session.find(Item.class, 1L);
System.out.println(item.getVersion()); // <- 1
item.setVal(20);
session.flush();
System.out.println(item.getVersion()); // <- 2
item.setVal(30);
session.flush();
System.out.println(item.getVersion()); // <- 3
session.getTransaction().rollback();

session = factory.getCurrentSession();
session.beginTransaction();
item = session.find(Item.class, 1L);
System.out.println(item.getVersion()); // <- 1
session.getTransaction().commit();
session.close();
// … а тут завершение работы
```

При внесении изменений в состояние item, Hibernate накапливает эти изменения, но не посылает запросы в базу данных. Выполнение session.flush() выталкивает контекст хранения, выполняя запросы в БД. В результате, после каждого flush() версия растет. Если выполняется rollback(), то само собой изменения версии не подтверждаются.

Для проверки работы такого варианта оптимистической блокировки, можно воспользоваться следующим кодом.

```
// ...
new Thread(() -> {
   System.out.println("Thread #1 started");
   Session session = factory.getCurrentSession();
   session.beginTransaction();
   Item item = session.get(Item.class, 1L); // <- version = 1
   item.setVal(100);
   uncheckableSleep(1000);
   session.save(item);
   session.getTransaction().commit(); // version увеличивается до 2
   System.out.println("Thread #1 committed");
   if (session != null) {
       session.close();
   }
   countDownLatch.countDown();
}).start();
```

```
new Thread(() -> {
    System.out.println("Thread #2 started");
    Session session = factory.getCurrentSession();
    session.beginTransaction();
    Item item = session.get(Item.class, 1L); // <- version = 1
    item.setVal(200);
    uncheckableSleep(3000);
    try {
        session.save(item);
        session.getTransaction().commit(); // в момент подтверждения транзакции
во втором потоке производится сравнение версии при старте транзакции (1) и
текущим значением версии (2)
        System.out.println("Thread #2 committed");
    } catch (OptimisticLockException e) {
        session.getTransaction().rollback();
        System.out.println("Thread #2 rollback");
        e.printStackTrace();
    }
    if (session != null) {
        session.close();
    }
    countDownLatch.countDown();
}).start();
try {
    countDownLatch.await();
} catch (InterruptedException e) {
    e.printStackTrace();
}
// ...
```

** unckeckableSleep(int ms) метод, выполняющий Thread.sleep(), с перехватом InterruptedException.*

Два потока параллельно пытаются изменить состояние item. При старте, в каждой транзакции выполняется “запоминание версии объекта”, которая равна 1. Транзакция может быть успешно завершена только в том случае, если версия объекта на момент начала и подтверждения транзакции совпадают.

Из кода видно, что первый поток подтвердит транзакцию раньше второго, при этом будет произведено сравнение версий, и поскольку **1 == 1**, транзакция удачно завершится и версия увеличится на 1. Через пару секунд, второй поток попытается также завершить транзакцию, но версии будут отличаться **1 != 2**, и в этом случае будет сгенерировано исключение OptimisticLockException, после перехвата которого выполняется rollback().

Если изменение какого-либо поля не должно влиять на версию объекта, то такое поле можно пометить как `@OptimisticLock(excluded = true)`, в примере кода из начала пункта, такой аннотацией было помечено поле int junkField.

Пессимистические блокировки

Давайте рассмотрим случай пессимистической блокировки.

```
// ...
session = factory.getCurrentSession();
session.beginTransaction();
int sumValue = 0;
List<Item> items = session.createQuery("SELECT i FROM Item i;", Item.class)
       .setLockMode(LockModeType.PESSIMISTIC_READ)
       .getResultList();
for (Item o : items) {
   sumValue += o.getVal();
}
session.getTransaction().commit();
// ...
```

Допустим мы хотим просуммировать значения всех item'ов в нашей таблице, и сделать это надо именно на стороне нашего приложения, а не базы данных. В таком случае, при получении списка объектов из базы данных, мы не можем давать другим транзакциям изменять значения этих элементов. Для этого может быть установлена пессимистическая блокировка с помощью метода setLockMode(). При выборе в качестве аргумента LockModeType.PESSIMISTIC_READ, полученные записи будут доступны другим транзакциям только для чтения, такой режим аналогичен блокировке PosetgreSQL "FOR SHARE". Если выбрать LockModeType.PESSIMISTIC_WRITE, то строки заблокируются как для чтения, так и для записи, что в PostgreSQL аналогично "FOR UPDATE". Блокировки будут сняты только по завершению текущей транзакции.

Практическое задание

1. В базе данных необходимо реализовать возможность хранить информацию о покупателях (id, имя) и товарах (id, название, стоимость). У каждого покупателя свой набор купленных товаров.

   Задача: написать тестовое консольное приложение, которое позволит посмотреть, какие товары покупал клиент, какие клиенты купили определенный товар, и предоставит возможность удалять из базы товары/покупателей.

2. ** Добавить детализацию по паре «покупатель — товар»: сколько стоил товар в момент покупки клиентом.

Дополнительные материалы

1. https://hibernate.org/orm/documentation/5.4/
2. **GEEKCHANGE: "Java. Изучаем Hibernate ORM для работы с базами данных"** https://www.youtube.com/watch?v=emg94BI2Jao (Введение в Hibernate)

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Крис Шефер, Кларенс Хо. Spring 4 для профессионалов (4-е издание).
2. Крейг Уоллс. Spring в действии.