Урок 3

# Доступ к данным в Spring. Часть 1

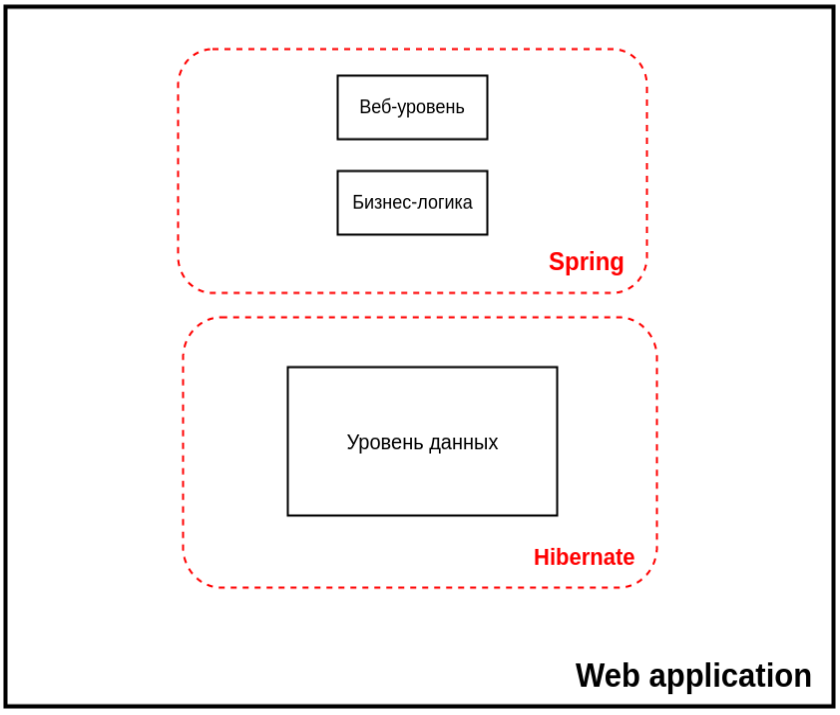
Использование Hibernate в Spring-приложениях. Понятие сущности. Объектно-реляционное отображение. Отображение связей «один ко многим», «один к одному», «многие ко многим».

# Введение

Spring предоставляет собственные средства для взаимодействия с СУБД. Основным ядром этого взаимодействия 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.

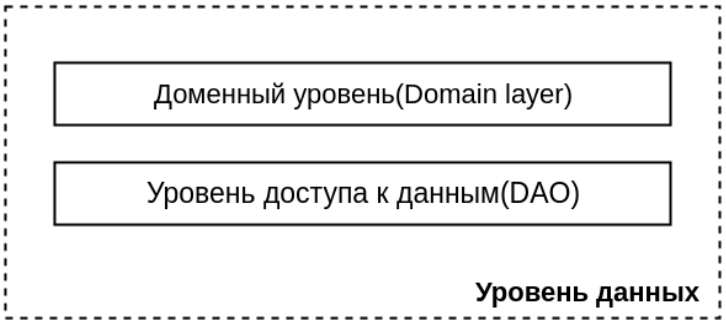
# Использование Hibernate в Spring-приложениях

В предыдущих уроках мы часто упоминали выражение «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 Spring применяются для написания бизнес-логики и веб-уровня.




Уровень данных можно разделить на следующие уровни:

- 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, т.е. основные объекты мэппинга. Это то, во что будут преобразовываться данные из БД (сущности).



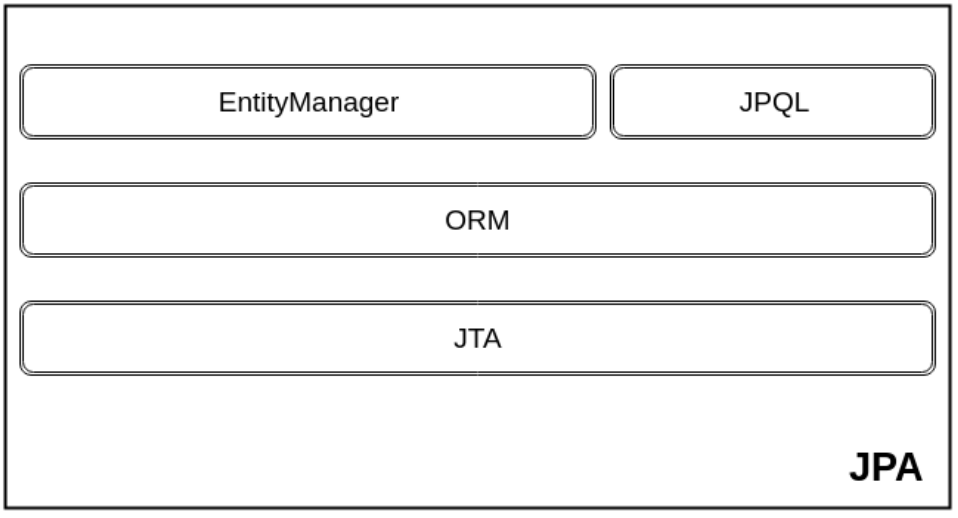

В данном уроке рассмотрим доменный уровень – способы его построения с использованием Hibernate.

# Hibernate и JPA

Изначально Hibernate развивалась как самостоятельная библиотека, никак 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:

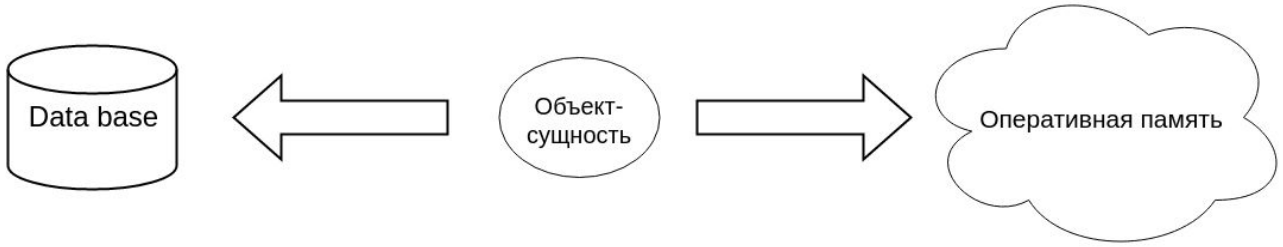
- API для различных операций с данными, хранящимися в БД (вставка, удаление, изменение и другое. Данный API описывается интерфейсом **EntityManager**;
- Объектно-реляционное отображение, которое описывает способы отображения объектов в данные, хранящиеся в БД;
- JPQL – Java Persistence Query Language – 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.

# 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. Для сохранения состояния объекта в базе данных должна располагаться таблица, соответствующая классу этого объекта. По аналогии со Spring, сущность – это объект, управляемый классом **EntityManager**, относящимся 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 др.

Чтобы объекты класса  являлись сущностями, необходимо выполнение следующих условий:

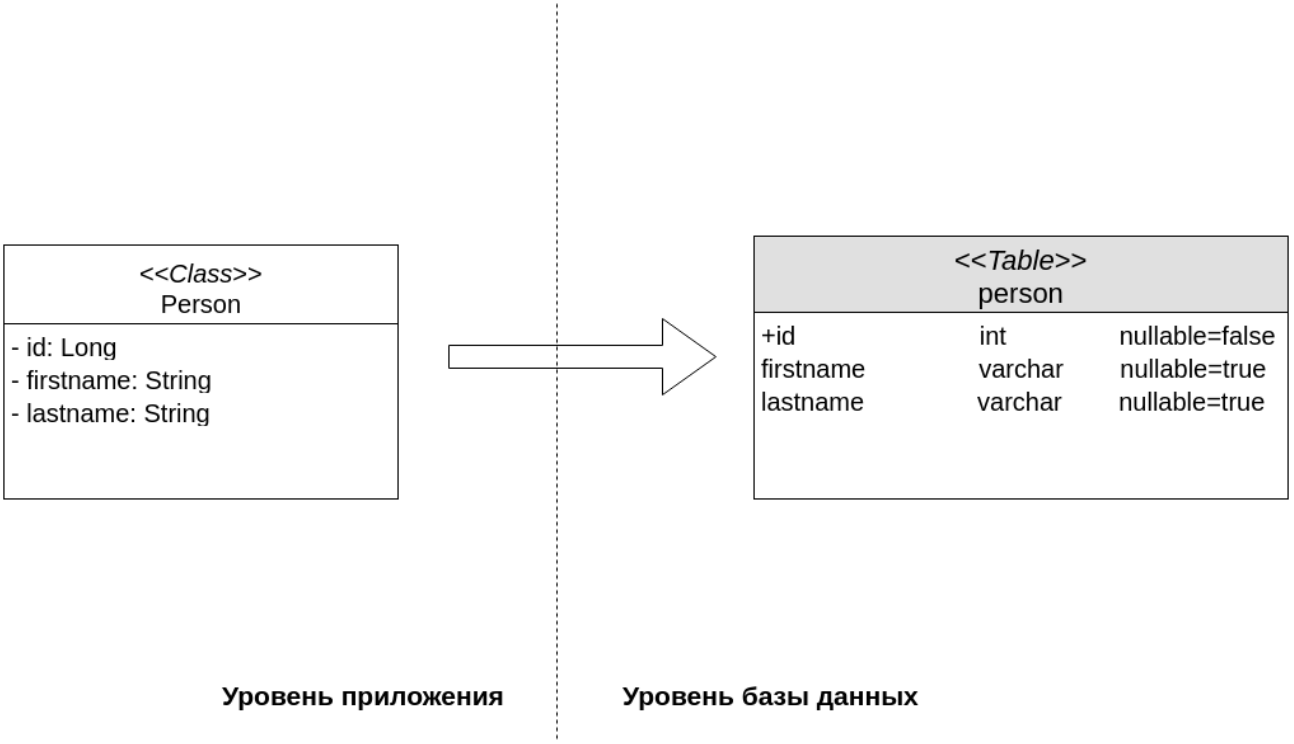
- наличие аннотации **@Entity**;
- наличие поля, в котором будет храниться уникальный идентификатор сущности. Оно должно быть снабжено аннотацией **@Id**;
- наличие конструктора без аргументов (конструктор по умолчанию);
- отсутствие модификатора **final**.

Порядок объявления сущности имеет следующий вид:

```
@Entity
public class Person {

      @Id
       @GeneratedValue
      private Long id;
       private String firstname;
       private String lastname;
       // Геттеры и сеттеры
       ...
}
```

Прежде чем вдаваться в детали конфигурирования, необходимо отметить, 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. Например, если для внедрения зависимости используется только аннотация **@Autowired**, то внедрение производится по типу (т.к. это является поведением по умолчанию). Но если добавить аннотацию **@Qualifer("name")**, то поведение по умолчанию изменится и будет происходить внедрение по имени.

Для объектов вышеуказанного класса порядок отображения будет иметь следующий вид:




Данный процесс следует следующим правилам:

- имя класса отображается в имя таблицы (Person->person). Если таблица, в которую необходимо отобразить сущность, имеет другое имя, то необходимо использовать аннотацию **@Table** с указанием имени таблицы, в которую необходимо отобразить класс;
- имена атрибутов отображаются в имена столбцов (firstmol -> firstmol). Если имя столбца отлично от имени атрибута, необходимо использовать аннотацию **@Column** с указанием имени столбца;
- типы атрибутов класса отображаются в типы используемой СУБД. Этот процесс интуитивно понятен (например, Long -> integer), но отличается для 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 необходимость в отображении какого-либо атрибута отсутствует, необходимо 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 баз данных предоставляют собственные механизмы генерации значений id. Значение может инжектироваться в поле id объекта-сущности после его сохранения этого объекта, 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** – предоставялет 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.

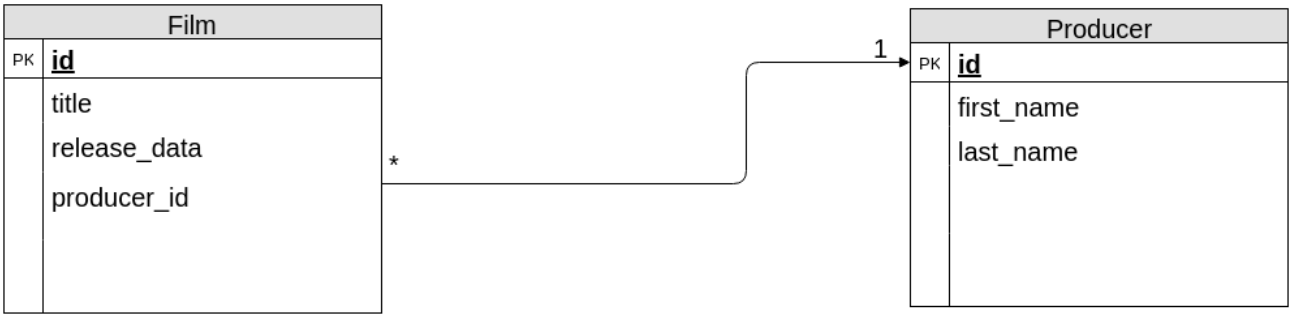
Оптимальный подход – использование **GenerationType.IDENTITY**. Аннотация говорит Hibernate о том, что после сохранения объекта в базе данных необходимо получить значение из столбца, на который отображается атрибут id, и присвоить его объекту-сущности. А каким образом в этом столбце появится значение 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 еще вернемся к данной теме.

# Отображение связей

В предыдущей главе мы рассмотрели случай, когда атрибуты сущности имели простые типы. Но на практике кроме простых атрибутов 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 использованию механизма внешних ключей. В данной главе мы подробно рассмотрим способы отображения связей.

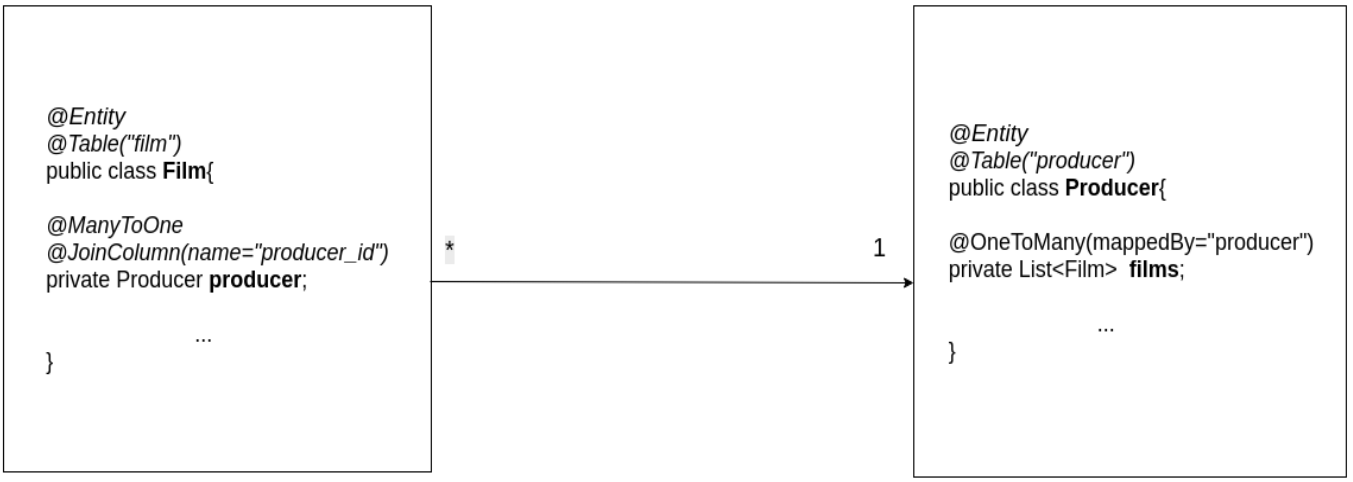
## Один ко многим

Связь один ко многим отображается с помощью аннотации **@OneToMany**, **@ManyToOne** и **@JoinColumn**. Представим, что у нас есть класс фильма и продюсера. При этом у каждого фильма есть только один продюсер, но каждому продюсеру может принадлежать большое количество фильмов. Данная ситуация относится к виду связи «один ко многим» (или «многие к одному»). В таблице данная связь отображается следующим образом:




Эта связь достигается использованием внешнего ключа **producer_id**, владельцем связи является таблица **Film**.

В таком случае отображение достигается так:




Данное объявление подчиняется следующим правилам:

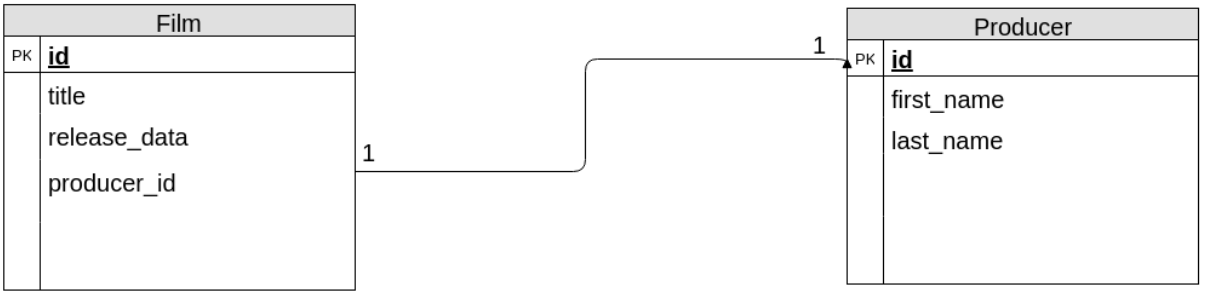
- для атрибутов обоих классов указывается аннотация **@ManyToOne** или **@OneToMany** в зависимости от стороны связи;
- для класса **Film**, который является владельцем связи, используется атрибут **name** аннотации **@JoinColumn**, которая указывает на столбец с внешним ключом в таблице;
- для класса **Producer** в параметре **mappedBy** указывается название ассоциируемого с ним атрибута в классе-владельце **Film**.

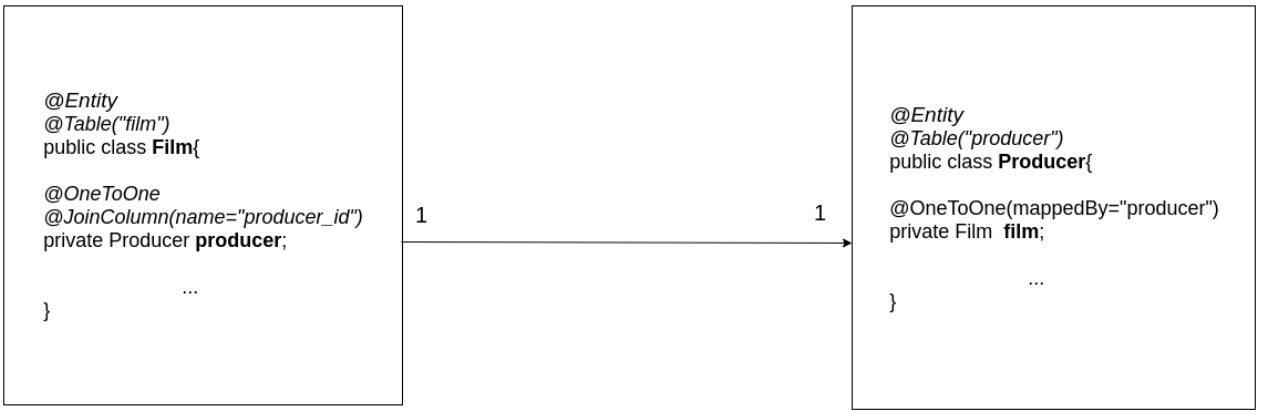
В данном случае связь между классами является двунаправленной, в отличие от таблиц, между которыми связь однонаправленная. Разработчик сам определяет, какой из вариантов использовать, но на практике оптимальна двунаправленная связь между классами. Например, в данном случае очень вероятна ситуация, когда необходимо получить все фильмы какого-либо продюсера. Использование двунаправленной связи избавляет нас от явного вызова кода запроса к базе данных. Получение всех фильмов определенного продюсера достигается за счет вызова метода **getFilms()**.

## Один к одному

Теперь представим ситуацию, где для каждого фильма существует свой продюсер, и у каждого продюсера есть только один фильм. Тогда данная связь называлась бы «один к одному». Для ее отображения применяется аннотация **@OneToOne**. Объявление связи «один к одному» почти полностью аналогично связи «один ко многим», за исключением использования аннотации, обозначающей вид связи.

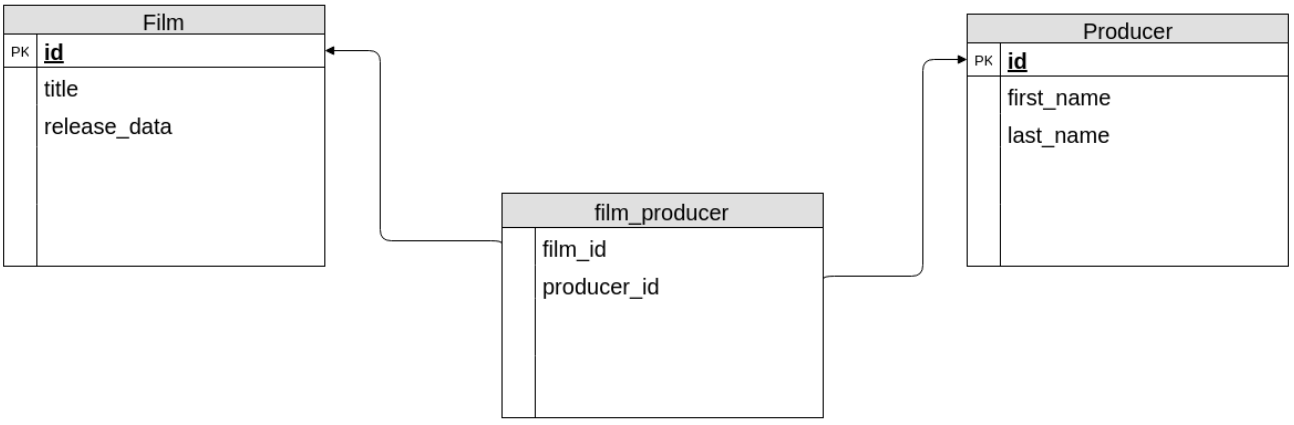
Отображение данной связи осуществляется следующим способом:





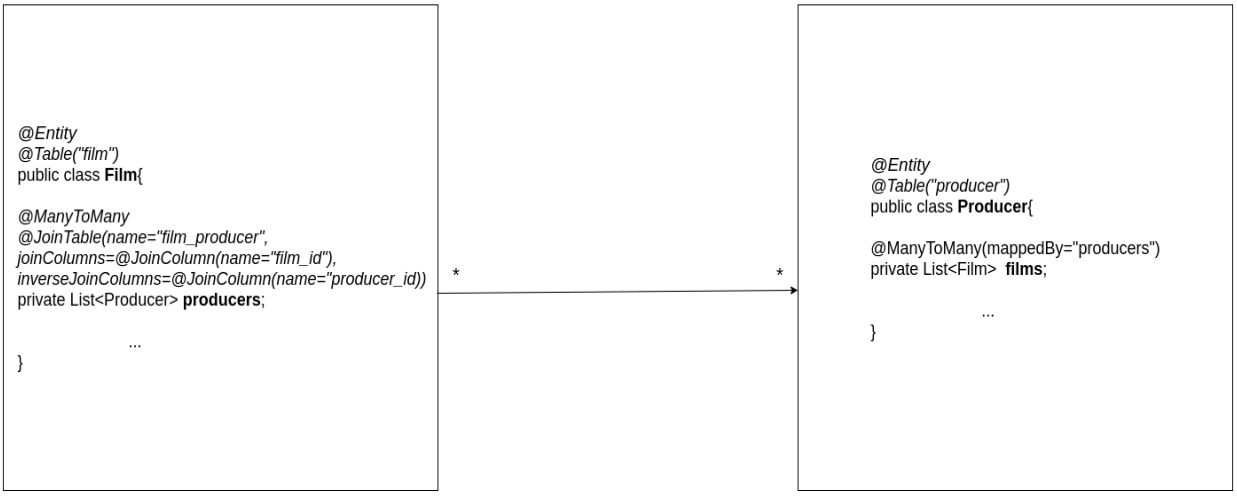

## Многие ко многим

Если у каждого фильма может быть несколько продюсеров, а каждый продюсер мог работать над несколькими фильмами, то такая связь называлась бы «многие ко многим». Ее характерная особенность – использование дополнительной таблицы. В базе данных эта связь выглядит следующим образом:




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

Отображение данной связи будет иметь следующий вид:




Отображение данной связи 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;
- в классе, ассоциированном с классом-владельцем, указывается значение параметра **mappedBy**, которое обозначает имя атрибута в классе-владельце.

# Практика

С этого урока забудем о примере с фотоаппаратом и начнем разрабатывать реальный проект.

**Задание на практику:** разработать сайт публикации статей IT-тематики с возможностью добавления и удаления. Все статьи будут принадлежать определенным авторам и отображаться на главной странице сайта. Каждая из статей должна относиться к определенной категории. Проект будет разрабатываться поэтапно. На этом уроке создадим базу данных и доменный слой.

Порядок разработки проекта в данном уроке:

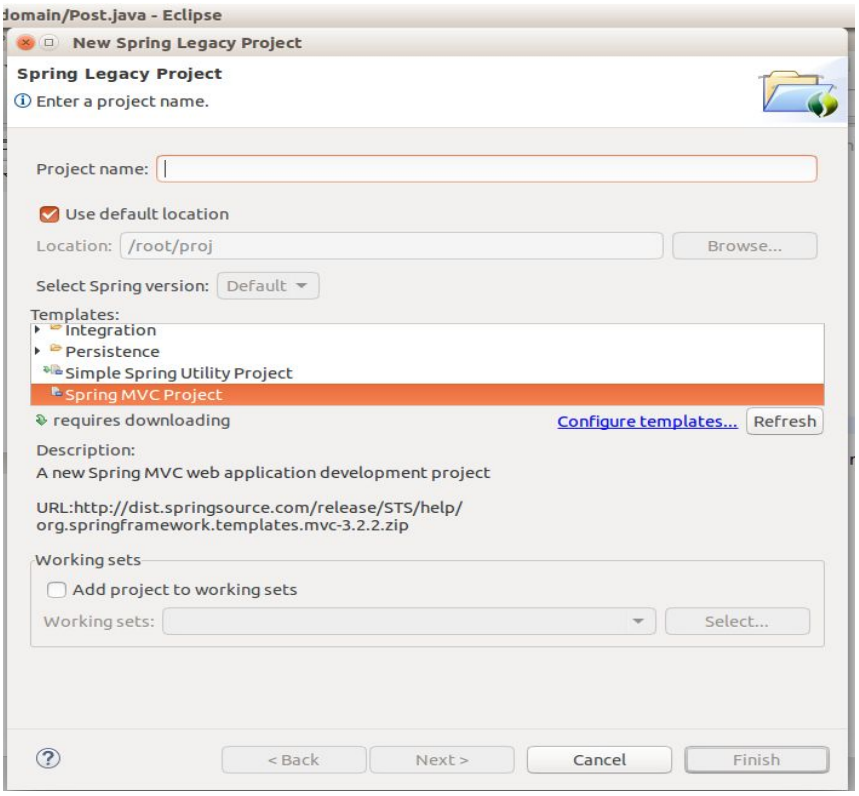
- определение основных сущностей веб-приложения и связей между ними;
- создание базы данных проекта;
- разработка классов-сущностей.

Код, разработанный на этом уроке, мы будем изменять в дальнейшем, изучая новый материал. Этот проект будет подспорьем при выполнении домашнего задания. Но перед тем как начать разработку, нам необходимо правильно сконфигурировать проект и добавить необходимые зависимости.

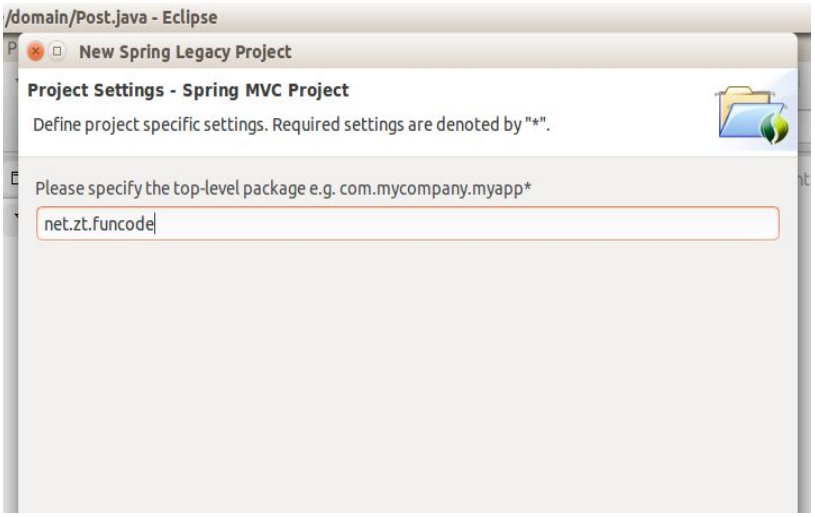
# Создание проекта

Чтобы создать проект, выполним следующие действия:

- перейдем в **File -> Spring Legacy Project**;
- увидим окно:




- в поле **Project name** зададим имя нашему проекту, а в окне **Templates** выберем шаблон проекта **Spring**. Шаблоны избавляют нас от действий по созданию необходимой структуры веб-приложения Spring;
- введем название корневого проекта в соответствующем поле:



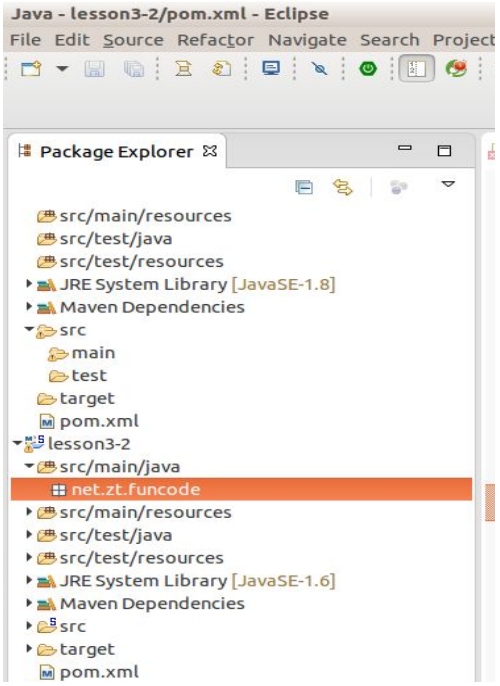

Именно от корневого проекта мы будем создавать дочерние пакеты, каждый из которых будет ассоциирован с определенным уровнем веб-приложения. Например, пакет уровня, разработанного

нами в данном уроке, будет иметь полное название net.zt.funcode.domain, что соответствует доменному уровню приложения.

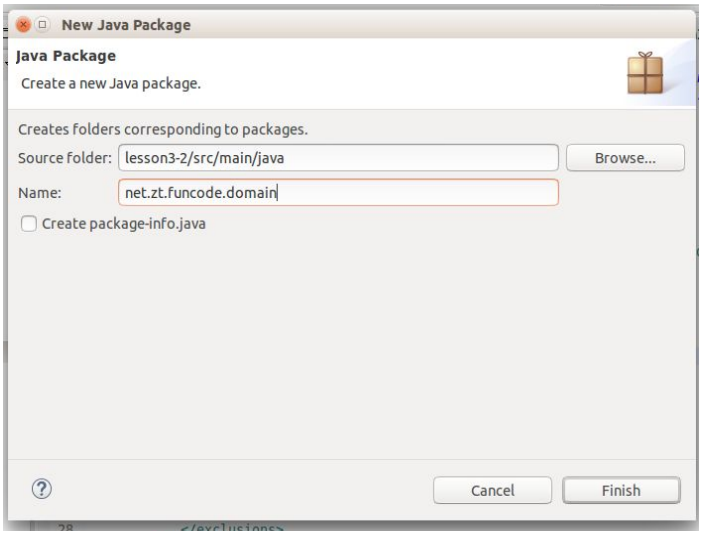
В появившейся структуре проекта в корневом пакете **net.zt.funcode** присутствует класс **HomeController**. Этот класс необходимо удалить.

Нужно создать пакет для разрабатываемого доменного уровня:

- нажать правой кнопкой мыши на корневой пакет:




- выбрать **New->Package**;
- в появившемся окне в поле имени пакета уже будет располагаться название корневого пакета. Надо через точку дописать название уровня:



## Подключение зависимостей

После создания проекта по заданному шаблону мы имеем файл **pom.xml**. Необходимо удалить определенное содержимое данного файла, чтобы он имел следующий вид:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
      <modelVersion>4.0.0</modelVersion>
      <groupId>net.zt</groupId>
      <artifactId>funcode</artifactId>
      <name>lesson3</name>
      <packaging>war</packaging>
      <version>1.0.0-BUILD-SNAPSHOT</version>
      <properties>
            <java-version>1.7</java-version>
      </properties>
      <dependencies>
      </dependencies>
    <build>
        <plugins>
            <plugin>
                <artifactId>maven-eclipse-plugin</artifactId>
                <version>2.9</version>
                <configuration>
                    <additionalProjectnatures>
<projectnature>org.springframework.ide.eclipse.core.springnature</projectnature>
                    </additionalProjectnatures>
                    <additionalBuildcommands>

<buildcommand>org.springframework.ide.eclipse.core.springbuilder</buildcommand>
                    </additionalBuildcommands>
                    <downloadSources>true</downloadSources>
                    <downloadJavadocs>true</downloadJavadocs>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.5.1</version>
                <configuration>
                    <source>1.6</source>
                    <target>1.6</target>
                    <compilerArgument>-Xlint:all</compilerArgument>
                    <showWarnings>true</showWarnings>
                    <showDeprecation>true</showDeprecation>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>1.2.1</version>
                <configuration>
                    <mainClass>org.test.int1.Main</mainClass>
                </configuration>
            </plugin>
```

```
        </plugins>
    </build>
</project>
```

В данном уроке не ставится цель обьяснить работу с Maven, но для тех, кому не понятен приведенный выше листинг, рекомендую повторить материалы предшествующего курса.

Как видите, данный листинг не содержит ни одной зависимости. Для начала нам необходимо добавить зависимости Spring:

```
<dependencies>
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>4.3.2.RELEASE</version>
</dependency>
         <!-- https://mvnrepository.com/artifact/org.springframework/spring-core
-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>4.3.3.RELEASE</version>
</dependency>
  <!-- https://mvnrepository.com/artifact/org.springframework/spring-beans -->
 <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-beans</artifactId>
    <version>4.3.3.RELEASE</version>
 </dependency>
 <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>4.3.3.RELEASE</version>
 </dependency>
</dependencies>
```

С большинством зависимостей, приведенных в данном листинге, мы уже познакомились. Но для обеспечения взаимодействия *Hibernate* и *Spring* была добавлена ***spring-orm***. Что дает данная зависимость, покажем далее.

В проект необходимо добавить зависимости для работы *Hibernate*:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.2.4.Final</version>
</dependency>
<dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>9.4.1208.jre7</version>
</dependency>
```

В этом листинге подключен драйвер для работы с базой данных – в нашем случае это PostgreSQL.

Когда мы настроили проект, можно приступать к определению основных сущностей приложения.

## Определение сущностей

Исходными данными для нас является задание, согласно которому необходимо спроектировать веб-приложение с возможностью публикации, редактирования и удаления статей. Основными сущностями будут:

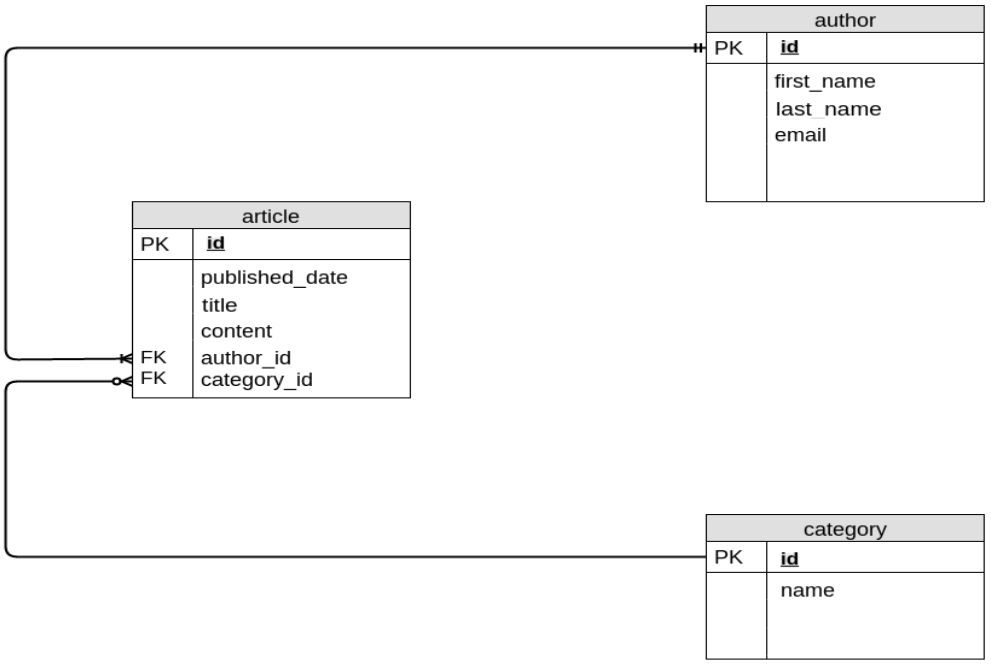
- **статья**, состоящая из заголовка, основного контента и даты публикации. Статья должна относиться к определенной категории IT-тематики;
- **автор**, публикующий статьи. У него будет **имя** и **фамилия**, а также контактный **email**;
- **категория**, которая будет состоять из **названия**.

Данные сущности определялись без учета возможности авторизации и регистрации. В дальнейших уроках (по мере изучения нового материала) мы будем добавлять новые сущности и изменять имеющиеся.

## Проектирование базы данных

Исходя из предыдущих пунктов, необходимо создать три таблицы: **article**, **author**, **category**.

Схема базы данных должна выглядеть следующим образом:




Чтобы разработать базу данных согласно приведенной схеме, необходимо выполнить следующие шаги:

- создать базу данных для учебного проекта:

```
CREATE DATABASE lesson3;
```

- создать таблицы:

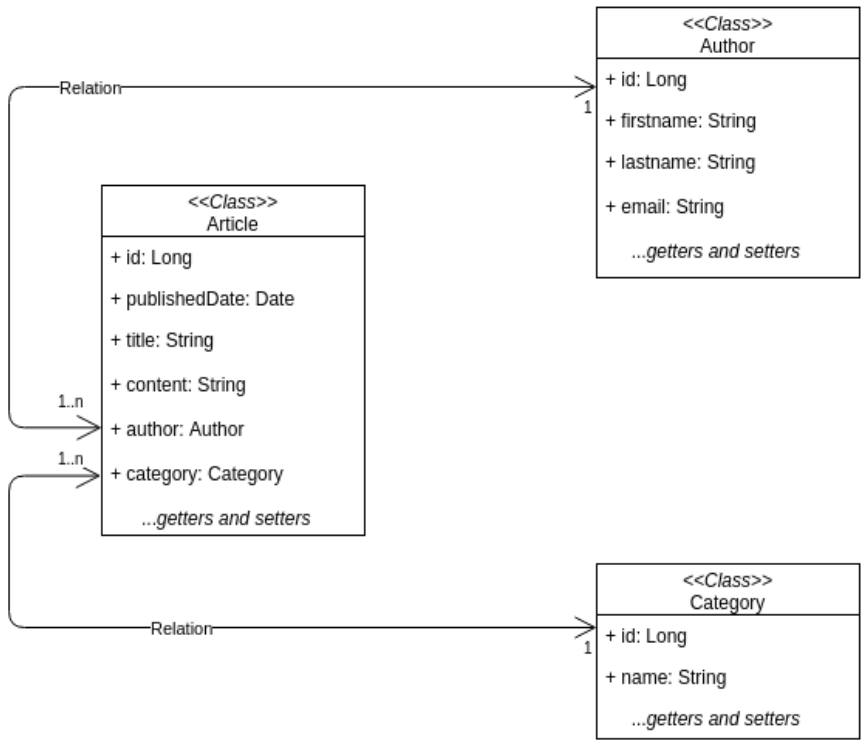
```
DROP TABLE IF EXISTS author;
CREATE TABLE author
(
  id serial,
  firstname varchar(30) NOT NULL,
  lastname varchar(30) NOT NULL,
  email varchar(30) NOT NULL,
  PRIMARY KEY(id)
  );

DROP TABLE IF EXISTS category;
CREATE TABLE category
(
  id serial,
  name varchar(30),
  PRIMARY KEY(id)

);

DROP TABLE IF EXISTS article;
CREATE TABLE article
(
  id serial,
  published_date TIMESTAMP DEFAULT current_timestamp,
  title text,
  content text,
  author_id int NOT NULL,
  category_id int NOT NULL,
  PRIMARY KEY(id),
  FOREIGN KEY(author_id) REFERENCES  author(id),
  FOREIGN KEY(category_id) REFERENCES category(id)
);
```

Поясним пару моментов в вышеприведенной DDL-инструкции:

- для генерации id задается внутренний генератор **serial**;
- атрибут **published_date** по умолчанию имеет значение текущего времени, возвращаемое функцией **current_timestamp**.

# Создание доменного уровня

Аналогично разработанной схеме в БД определяем UML-диаграмму классов:




Данная диаграмма является аналогом схемы в БД – за исключением использования двунаправленных связей между классами-сущностями.

После определения UML-диаграммы необходимо создать классы сущностей, которые будут располагаться в пакете **net.zt.funcode.domain**.

Код классов-сущностей будет иметь следующий вид:

- Класс **Author**:

```
@Entity
@Table(name="author")
public class Author {

	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	@Column(name="id")
	private Long id;

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

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

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

	@OneToMany(mappedBy="author")
	private List<Article> articles;
```

```
	public List<Article> getArticles() {
		return articles;
	}

	public void setArticles(List<Article> articles) {
		this.articles = articles;
	}

	public Long getId() {
		return id;
	}

	public void setId(Long id) {
		this.id = id;
	}

	public String getFirstname() {
		return firstname;
	}

	public void setFirstname(String firstname) {
		this.firstname = firstname;
	}

	public String getLastname() {
		return lastname;
	}

	public void setLastname(String lastname) {
		this.lastname = lastname;
	}

	public String getEmail() {
		return email;
	}

	public void setEmail(String email) {
		this.email = email;
	}

}
```

Все необходимые аннотации для объявления сущности берутся из пакета **javax.persistence**. В данном листинге применяется аннотация **@Temporal** – она нужна для указания типа данных SQL, в который будет отображаться дата из java-кода.

- Класс **Category**:

```
@Entity
@Table(name="category")
public class Category {

	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	private Long id;

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

	@OneToMany(mappedBy="category")
	private List<Article> articles;


	public List<Article> getArticles() {
		return articles;
	}

	public void setArticles(List<Article> articles) {
		this.articles = articles;
	}

	public Long getId() {
		return id;
	}

	public void setId(Long id) {
		this.id = id;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

}
```

В данном листинге применяются те же аннотации.

- Класс Article:

```
@Entity
@Table(name="article")
public class Article {

	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	@Column(name="id")
	private Long id;

	@Column(name="published_date")
	@Temporal(TemporalType.TIMESTAMP)
	private Date publishedDate;
```

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


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

    @ManyToOne
    @JoinColumn(name="author_id")
    private Author author;

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

    public Long getId() {
          return id;
    }

    public void setId(Long id) {
          this.id = id;
    }

    public Date getPublishedDate() {
          return publishedDate;
    }

    public void setPublishedDate(Date publishedDate) {
          this.publishedDate = publishedDate;
    }

    public String getTitle() {
          return title;
    }

    public void setTitle(String title) {
          this.title = title;
    }

    public String getContent() {
          return content;
    }

    public void setContent(String content) {
          this.content = content;
    }

    public Author getAuthor() {
          return author;
    }

    public void setAuthor(Author author) {
          this.author = author;
    }

    public Category getCategory() {
          return category;
    }
```

```
	public void setCategory(Category category) {
		this.category = category;
	}
}
```

## Создание конфигурации

Для класса-конфигурации необходимо создать отдельный пакет, который будет называться **net.zt.funcode.config**.

В этом пакете необходимо создать класс **AppConfig**:

```java
@Configuration
public class AppConfig {


	@Bean(name="dataSource")
	public DataSource getDataSource(){
		// Создания источника данных
		DriverManagerDataSource dataSource = new DriverManagerDataSource();
		// Задание параметров подключения к базе данных

dataSource.setUrl("jdbc:postgresql://localhost:5432/geekbrains-lesson3");
		dataSource.setUsername("postgres");
		dataSource.setDriverClassName("org.postgresql.Driver");
		dataSource.setPassword("123");
		return dataSource;
	}

	@Bean(name="entityManagerFactory")
	public LocalContainerEntityManagerFactoryBean getEntityManager(){
		// Создание класса фабрики, реализующей интерфейс
FactoryBean<EntityManagerFactory>
		LocalContainerEntityManagerFactoryBean factory = new
LocalContainerEntityManagerFactoryBean();
		// Задание источника подключения
		factory.setDataSource(getDataSource());
		// Задание адаптера для конкретной реализации JPA
		//  указывает,  какая  именно  библиотека  будет  использоваться  в
качестве поставщика постоянства
		factory.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
		// Указание пакетов, в которых будут находиться классы-сущности
		factory.setPackagesToScan("net.zt.funcode.domain");

		// Создание свойств для настройки Hibernate
		Properties jpaProperties = new Properties();
		//  Указание  диалекта  конкретной  базы  данных  -  необходимо  для
генерации запросов Hibernate к БД
		jpaProperties.put("hibernate.dialect",
"org.hibernate.dialect.PostgreSQLDialect");
		//  Указание  максимальной  глубины  связи  (будет  рассмотрено  в
следующем уроке)
		jpaProperties.put("hibernate.max_fetch_depth", 3);
		// Определение максимального количества строк, возвращаемых за один
запрос из БД
		jpaProperties.put("hibernate.jdbc.fetch_size", 50);
		// Определение максимального количества запросов при использовании
пакетных операций
		jpaProperties.put("hibernate.jdbc.batch_size", 10);

		// Включает логирование
		jpaProperties.put("hibernate.show_sql", true);

		factory.setJpaProperties(jpaProperties);

		return factory;

	}
}
```

Мы определили классы-сущности, но для манипулирования ими необходим механизм. Спецификация JPA 2+ определяет этот механизм как менеджер сущностей **EntityManager** (этот интерфейс будет рассмотрен подробно в следующем уроке). Hibernate реализует этот интерфейс. Для дальнейшего использования (взаимодействия с базой данных) необходимо получить из контекста Spring объект **EntityManagerFactory**, который создает объект **EntityManager**. Но применить к этому классу аннотацию **@Component** не представляется возможным. Тогда на помощь приходят **FactoryBean<T>**. Классы, реализующие данный интерфейс, используются как фабрики бинов для создания объектов класса T, которые будут управляться контейнером Spring. Это становится возможным, так как контейнер Spring знает, как обращаться с подобными классами.

# Код учебного проекта

Конечный вариант кода программы, разработанной в ходе данного урока: https://github.com/Firstmol/Geekbrains-lesson3

Заметьте, что целью практической части было лишь проектирование доменного уровня. Класса, который запускал бы данную программу, вы не найдете.

# Практическое задание

Перед выполнением данного задания настоятельно рекомендуется изучить дополнительный материал №1.

Необходимо  разработать сайт рекламных объявлений компаний. Задание по данному уроку:

1. Создать проект, подключить необходимые зависимости и настроить конфигурацию аналогично выполненной в уроке 3.
2. Спроектировать схему БД, состоящую из следующих таблиц:
    - Company (Компания);
    - Category (Категория);
    - Ad (объявление).

Основными атрибутами данных таблиц будут:

- для компании – идентификатор, название компании, описание, адрес;
- для категории – идентификатор, название категории;
- для объявления – идентификатор, категория, название,  содержание, номер телефона.

Все вышеописанные атрибуты являются простыми, а связи между таблицами необходимо продумать самостоятельно.

3. Создать классы-сущности.
4. Выложить сделанный проект в свой репозиторий на github для проверки преподавателем.

# Дополнительные материалы

1. Крис Шефер, Кларенс Хо. Spring 4 для профессионалов (4-е издание). – стр. 351-359;
2. Аннотации Hibernate: https://www.tutorialspoint.com/hibernate/hibernate_annotations.htm.

# 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.