Java для молодых специалистов Банка ВТБ

Урок 11

Hibernate. Часть 1

Java Persistence API. Основы использования Hibernate. Сущности. Каскадные операции. @OneToOne, @OneToMany, @ManyToOne, @ManyToMany.

Hibernate и Java Persistence API (JPA)

Понятие сущности и объектно-реляционного отображения

Отображения связей

- Один ко многим
- Один к одному
- Многие ко многим

Практическое задание

Дополнительные материалы

Используемая литература

Hibernate и Java Persistence API (JPA)

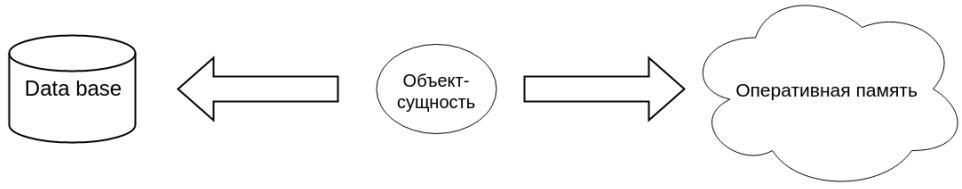
Изначально Hibernate развивалась как самостоятельная библиотека, не связанная со спецификацией Java EE. С третьей версии она приобрела поддержку JPA (Java Persistence API), фактически став ее реализацией: JPA определяет интерфейсы объектов для доступа к данным, а Hibernate их реализует. Поэтому в данном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

Понятие сущности и объектно-реляционного отображения

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. 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:

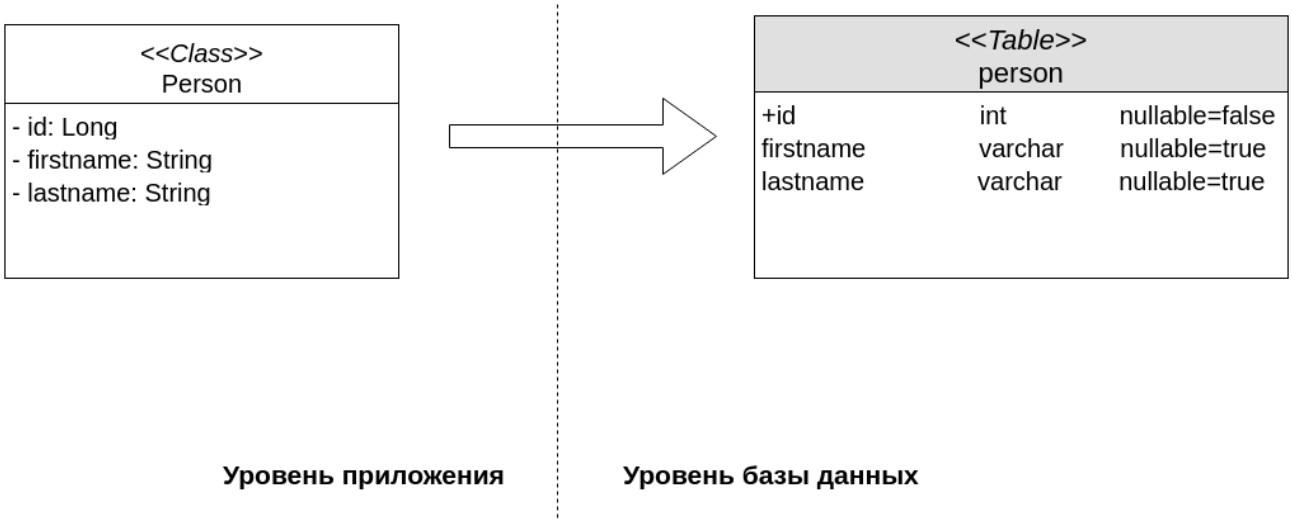
- наличие аннотации **@Entity**;
- наличие поля, в котором будет храниться уникальный идентификатор сущности. Оно должно быть снабжено аннотацией **@Id**;
- наличие конструктора без аргументов (конструктора по умолчанию);
- отсутствие модификатора **final**.

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** 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 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.

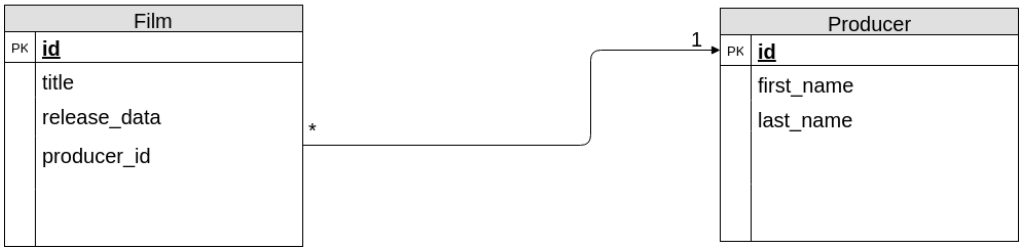
Оптимальный подход — использование **GenerationType.IDENTITY**. Аннотация говорит Hibernate о том, что после сохранения объекта в базе данных необходимо получить значение из столбца, на который отображается атрибут id, и присвоить его объекту-сущности. А каким образом в этом столбце появится значение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

Отображения 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 использования механизма внешних ключей. В данной главе мы подробно рассмотрим способы отображения связей.

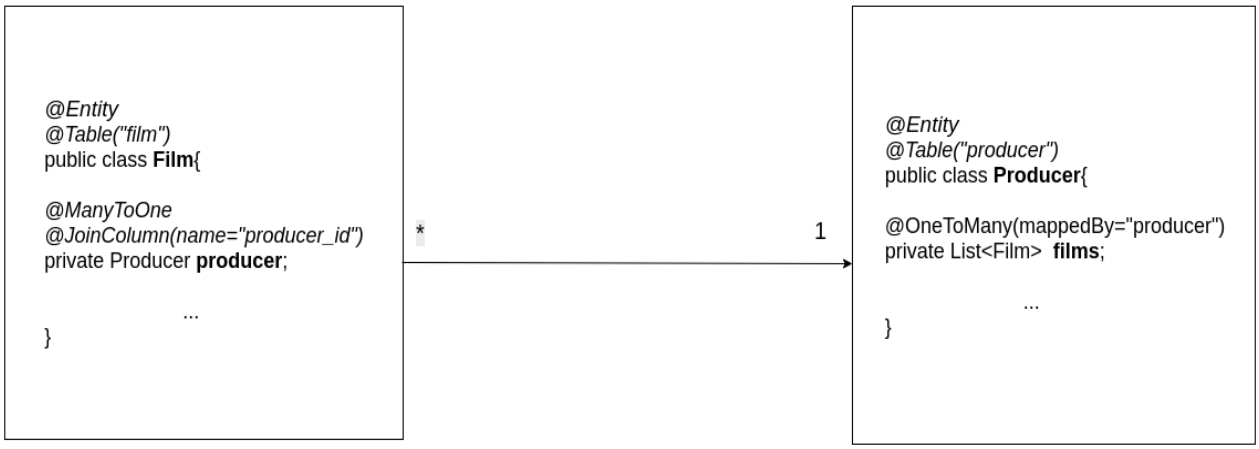
Один ко многим

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:

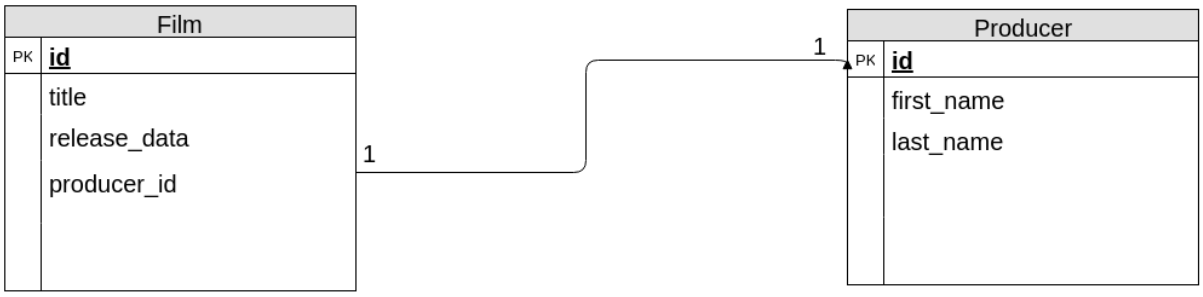
- для атрибутов обоих классов указывается аннотация **@ManyToOne** или **@OneToMany** в зависимости от стороны связи;
- для класса **Film**, который является владельцем связи, используется атрибут **name** аннотации **@JoinColumn**, которая указывает на столбец с внешним ключом в таблице;
- для класса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

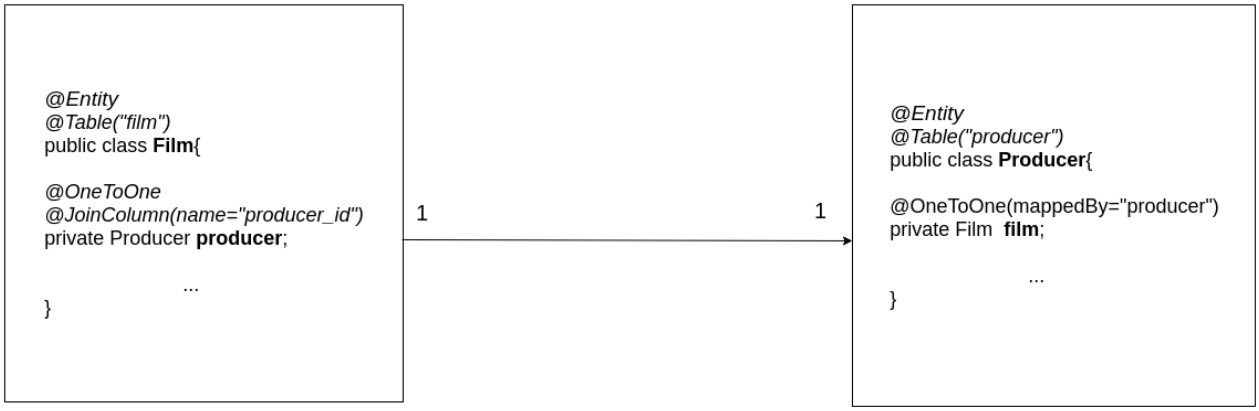
В данном случае связь между классами является двунаправленной, в отличие от таблиц, между которыми связь однонаправленная. Разработчик сам определяет, какой из вариантов использовать, но на практике оптимальна двунаправленная связь между классами. Например, в данном случае очень вероятна ситуация, когда необходимо получить все фильмы 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 у каждого фильма один продюсер, и у каждого продюсера есть только один фильм. Тогда данная связь называлась бы «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.

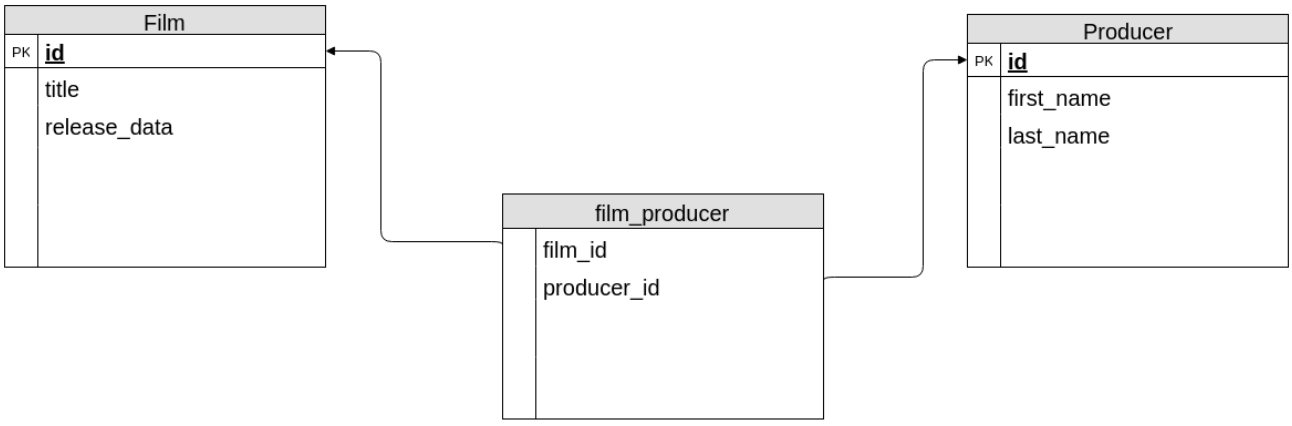
Отображение связи «один к одному»:





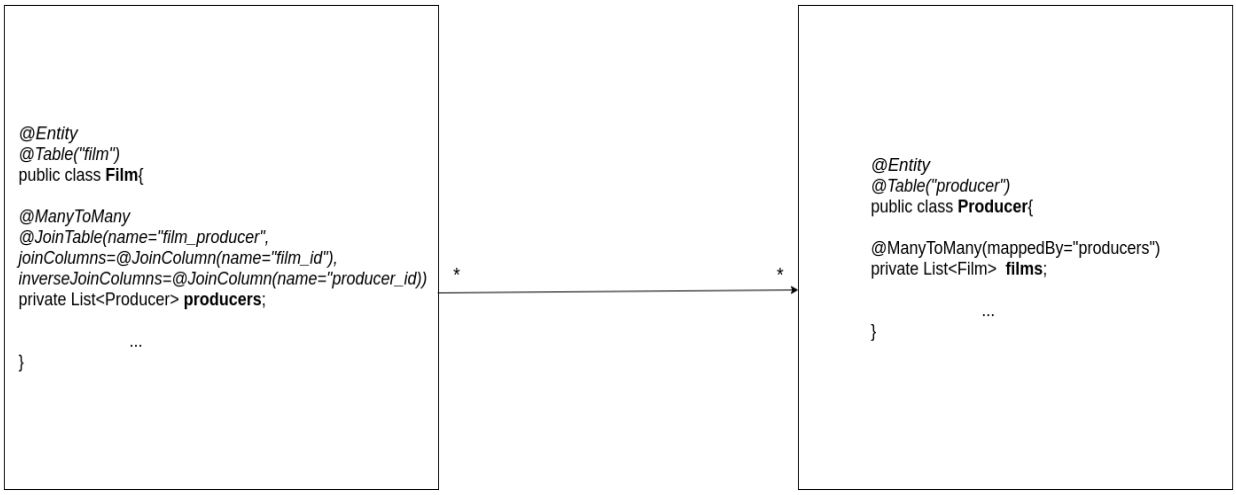
Многие ко многим

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 особенность — использование дополнительной таблицы. В базе данных эта связь выглядит следующим образом:



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

Отображение данной связи будет иметь следующий вид:



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

Практическое задание

1. В базе данных необходимо иметь возможность хранить информацию о покупателях (id, имя) и товарах (id, название, стоимость);

2. У каждого покупателя свой набор купленных товаров, одна покупка одного товара это отдельная запись в таблице (группировать не надо);

3. Написать тестовое консольное приложение (просто Scanner и System.out.println()), которое позволит выполнять команды:
/showProductsByPerson имя_покупателя - посмотреть какие товары покупал клиент;
/findPersonsByProductTitle название_товара - какие клиенты купили определенный товар;
/removePerson(removeProduct) имя_элемента - предоставить возможность удалять из базы товары/покупателей,
/buy имя_покупателя название_товара - организовать возможность “покупки товара”.

4. * Добавить детализацию по паре покупатель-товар: сколько стоил товар, в момент покупки клиентом;

Заметка: желательно все таблицы создать “вручную” и приложить скрипты создания.

Дополнительные материалы

1. Кей С. Хорстманн, Гари Корнелл Java. Библиотека профессионала. Том 1. Основы // Пер. с англ. — М.: Вильямс, 2014. — 864 с.
2. Брюс Эккель. Философия Java // 4-е изд.: Пер. с англ. — СПб.: Питер, 2016. — 1 168 с.
3. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.
4. Г. Шилдт. Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 720 с.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.