Связь «Один ко многим»

- Разновидности «Один ко многим»
  - Связь «Многие к одному» (ManyToOne)
  - Связь «Один ко многим» (OneToMany)

В Spring Boot сущности в приложениях существуют не сами по себе — они так или иначе всегда связаны друг с другом. Например, у каждого поста в блоге есть автор, у постов есть комментарии, у комментариев есть авторы и лайки. В этом уроке мы рассмотрим самую распространенную связь между сущностями — **«Один ко многим»** (*OneToMany* или *O2M*). Именно так называют ситуации, когда у одной сущности может быть множество зависимых.

На уровне базы данных связь «Один ко многим» проявляется как внешний ключ. В случае постов, таким внешним ключом будет `author_id`, указывающий запись в таблице пользователей:

```
CREATE TABLE POSTS (
  id bigint generated by default as identity,
  author_id bigint not null,
  slug varchar(255) not null unique,
  name varchar(255),
  body TEXT,
  updated_at timestamp(6),
  created_at timestamp(6),
  primary key (id)
)

-- Добавление ключа
ALTER TABLE IF EXISTS POSTS
  ADD CONSTRAINT FK6xvn0811tkyo3nfjk2xvqx6ns foreign key (author_id) references users
```

Spring Boot создает и выполняет эти запросы автоматически, если стоит соответствующая опция:

```
# application.yml

spring:
```

```
  jpa:
    generate-ddl: true
```

Разновидности «Один ко многим»

Связь «Один ко многим» в Spring Data JPA существует в двух вариантах:

- «Один ко многим» *(OneToMany)* — устанавливается со стороны сущности, которая представлена в единственном числе. В случае постов и авторов со стороны авторов.
- «Многие к одному» *(ManyToOne)* — устанавливается со стороны сущности, которая представлена во множественном числе. В случае постов и авторов со стороны постов.

Для создания связи *OneToMany* нужно как минимум сделать связь *ManyToOne* для сущности `Post`. В таком случае мы сможем обращаться к автору поста. Если мы захотим работать с постами одного автора, для этого придется добавить связь *OneToMany* в сущность `User`. С практической точки зрения есть смысл сразу устанавливать связь с обеих сторон.

Связь «Многие к одному» (ManyToOne)

Начнем с постов. Для создания связи нам понадобится аннотация `@ManyToOne`. Ее наличие создает внешний ключ в базе. При этом сама колонка будет называться по имени поля в классе с добавлением *_id* в конце. В нашем случае имя поля — это `author`, поэтому поле в базе будет называться `author_id`:

```
package io.hexlet.spring.model;

import static jakarta.persistence.GenerationType.IDENTITY;

import java.time.LocalDate;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;
import jakarta.validation.constraints.NotBlank;
```

```java
import jakarta.validation.constraints.NotNull;
import lombok.Getter;
import lombok.Setter;

@Entity
@Getter
@Setter
@Table(name = "posts")
public class Post {
    @Id
    @GeneratedValue(strategy = IDENTITY)
    private Long id;

    // Связь ManyToOne
    @ManyToOne
    @NotNull
    private User author;

    @Column(unique = true)
    @NotNull
    private String slug;

    @NotBlank
    private String name;

    @NotBlank
    @Column(columnDefinition = "TEXT")
    private String body;

    @LastModifiedDate
    private LocalDate updatedAt;

    @CreatedDate
    private LocalDate createdAt;
}
```

Почему мы выбрали имя поля `author`, а не `user` или что-то подобное? На это есть две причины. Во-первых, имя `user` не очевидно. Это может быть как создатель поста, так и человек, который опубликовал этот пост. Во-вторых, со временем связей с пользователями даже в рамках одной модели становится больше. Сегодня у нас есть только автор, а завтра может появиться соавтор или модератор.

На практике работа со связью выглядит так же, как и с любым другим полем. Зависимость устанавливается через сеттер и извлекается через геттер:

```
post.setAuthor(user);
postRepository.save(post);

post.getAuthor().getName();
```

Но есть и отличия. По умолчанию загрузка связанной сущности происходит в ленивом режиме. Это значит, что при выборке поста из базы автора внутри не будет. Такая стратегия помогает экономить ресурсы. Без нее при большом количестве связей создавалось бы множество ненужных запросов. Реальный запрос в базу происходит тогда, когда мы обращаемся к автору:

```
// Автор не извлекается
var post = postRepository.findById(id);
// Если автор есть, то на первое обращение выполняется запрос в базу
post.getAuthor();
// Здесь запроса уже нет, потому что автор извлечен
post.getAuthor();
```

Связь «Один ко многим» (OneToMany)

Обратная связь создается с помощью аннотации `@OneToMany` с параметром `mappedBy`, значением которого должно быть имя поля в зависимой сущности. В нашей ситуации примере — это `author`:

```
// Остальные импорты
import jakarta.persistence.OneToMany;

@Entity
// Остальные аннотации
public class User {

    // Остальные поля

    @OneToMany(mappedBy = "author")
    // По умолчанию пустой список помогает не проверять на null
    // Выборка данных выполнится при обращении к getPosts()
```

```
    private List<Post> posts = new ArrayList<>();
}
```

С одной стороны, появление двунаправленной связи упрощает выборки, а с другой — создает дополнительную сложность в синхронизации данных в базе и в коде. Представьте, что у нас есть такой код:

```
var post = new Post(/* параметры */);
post.setAuthor(user);
postRepository.save(post);

// Дальше по коду мы хотим работать с постами пользователя
user.getPosts();
```

Сработает ли этот код? Да, но неправильно. Вновь созданный пост не появится внутри списка постов пользователей. Нам придется добавить его туда вручную:

```
var post = new Post(/* параметры */);
post.setAuthor(user);
postRepository.save(post);
user.getPosts().add(post);

// Теперь сработает
user.getPosts();
```

Такой код допустим. Но нам все еще надо помнить про связь, поэтому есть риск забыть его дописать, что приведет к неочевидным ошибкам. То же самое произойдет и при удалении поста. Ситуацию можно улучшить, если сохранять пост не отдельно, а в рамках пользователя:

```
import jakarta.persistence.CascadeType;

@Entity
// Остальные аннотации
public class User {

    // Остальные поля

    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL, orphanRemoval = true
    private List<Post> posts = new ArrayList<>();
```

```
    public void addPost(Post post) {
        posts.add(post);
        post.setAuthor(this);
    }

    public void removePost(Post post) {
        posts.remove(post);
        post.setAuthor(null);
    }
}
```

В этом коде мы видим два изменения:

- Появились два метода для добавления и удаления поста, которые синхронизируют действия с зависимой сущностью. В одном случае автор добавляется, в другом — удаляется.
- В аннотации появились два параметра:
  - `cascade = CascadeType.ALL` отвечает за каскадное применение всех операций к зависимым сущностям. Без него сохранение пользователя не приведет к сохранению поста.
  - `orphanRemoval = true` отвечает за удаление зависимых сущностей. Если мы удалим пост из списка постов пользователя, он удалится из базы во время сохранения пользователя.

```
user.addPost(post);
// Сохраняем post с добавленным author
userRepository.save(user);
```

В этом уроке мы изучили два способа сохранять данные. Чтобы подвести итоги, обсудим их применение на практике:

- Сохранение через зависимую сущность (`Post`) стоит использовать всегда, когда есть такая возможность — в большинстве случаев работа с сущностью напрямую проще и эффективнее
- Сохранение через базовую сущность (`User`) не такое универсальное. Советуем использовать его только тогда, когда мы знаем, что после сохранения зависимой сущности есть код, работающий с базовой сущностью.

Дополнительные материалы

1. N+1 Problem

Далее →