Спецификация JPA Entity

- Репозиторий
- Использование в коде
- Связь с базой данных

Для работы с базой данных Spring Boot использует Spring Data JPA (Jakarta Persistence API). Об этом механизме мы и поговорим в этом уроке.

Цель Spring Data JPA — автоматизировать типовые операции по работе с базой данных, то есть создание, изменение, удаление и извлечение. В итоге вам приходится писать значительно меньше шаблонного кода при ручном взаимодействии с базой данных. В большинстве случаев весь необходимый SQL-код создается автоматически и не виден программисту.

Работа этого механизма завязана на два элемента:

- Модель – класс, который соотносится с таблицей в базе данных (например, `User`)
- Репозиторий – класс, отвечающий за CRUD-операции над сущностью и ее коллекциями

Так этот механизм работает на практике:

```
var user = new User(/* данные */);
userRepository.save(user); // Создание

user.setName("Another name")
userRepository.save(user); // Обновление

userRepository.deleteById(user.getId()); // Удаление

var user2 = userRepository.findById(/* идентификатор */);
```

Spring Data JPA автоматически генерирует репозитории для модели на основе того, как модель проаннотирована. В свою очередь, аннотирование модели опирается на структуру таблицы в базе данных.

Предположим, что в нашей базе данных есть табличка *users* с такой структурой:

Имя	Тип
id	Идентификатор
first_name	Имя
email	Почта
last_name	Фамилия
created_at	Дата создания

Создадим для нее модель и проаннотируем ее:

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

import static jakarta.persistence.GenerationType.IDENTITY;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.EqualsAndHashCode;
import lombok.Getter;
import lombok.Setter;

@Entity
@Table(name = "users")
@Getter
@Setter
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
public class User {
    @Id
    @GeneratedValue(strategy = IDENTITY)
    @EqualsAndHashCode.Include
    private Long id;

    @Column(unique = true)
    private String email;

    private String firstName;
```

```
    private String lastName;
}
```

Обратите внимание, что аннотации импортируются из пакета Jakarta. Это происходит из-за неочевидного момента в терминах:

- JPA (Jakarta Persistence API) — это спецификация, то есть набор интерфейсов и аннотаций
- Spring Data JPA — это конкретная реализация этой спецификации

Именно поэтому большая часть аннотаций берется из Jakarta. Аннотации, которые импортируются из других пакетов, не входят в стандарт JPA — это специфика самого Spring Boot или каких-то других пакетов.

Сама по себе модель — это обычный POJO-класс с полями, геттерами, сеттерами и конструкторами, если нужно. Добавление аннотаций никак не влияет на сам класс, но зато оно помогает Spring Data JPA сгенерировать код репозитория.

Аннотирование модели начинается с двух аннотаций на уровне класса:

- Аннотация `@Entity` указывает, что эта модель считается сущностью, связанной с таблицей в базе данных
- Аннотация `@Table` задает имя таблицы

С полями аннотирование работает чуть хитрее. По умолчанию любое добавленное поле в модель отображается на базу данных при условии сопоставления имен. Работает это так:

- Spring Data JPA выполняет запрос в базу и извлекает структуру таблицы
- Поля транслируются из *snake_case* в *camelCase* и сравниваются
- Если поле найдено, оно будет отображаться на базу данных во время работы с репозиторием

Но иногда этого недостаточно. Для примера представим, что нам нужно создать уникальный индекс. В этом случае используется аннотация `@Column` с указанием на то, что нам нужна уникальность. Кроме уникальности, эта аннотация позволяет задать:

- Максимальную длину `length`
- Отсутствие *null*-значений `nullable`
- Имя колонки, если оно отличает от стандартного `name`

Рассмотрим такой пример:

```
@Column(name = "email", unique = true, nullable = false, length = 200)
private String email;
```

Особняком стоит история с идентификатором. Идентификатор играет особую роль не только с точки зрения базы данных, но и для JPA. Это важно, потому что идентификатор используется в SQL для поиска записей в базе данных. Поэтому для него есть свои аннотации:

- Аннотация `@Id`, которая указывает, что именно это и есть идентификатор
- Аннотация `@GeneratedValue(strategy = IDENTITY)`, которая определяет стратегию создания идентификатора. Внутри аннотации `IDENTITY` указывает, что идентификатор генерируется автоматически — его не нужно создавать

Репозиторий

Мы уже можем использовать созданную сущность, но пока не получится сохранить ее в базу. Чтобы решить эту проблему, нам нужен репозиторий. Посмотрим пример кода:

```
package io.hexlet.spring.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import io.hexlet.spring.model.User;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
}
```

Репозиторий в исходных файлах – это интерфейс, помеченный аннотацией `@Repository`. Spring Data JPA берет этот интерфейс и создает из него класс, который автоматически подставляется в приложение. Об этом мы поговорим чуть позже.

Набор методов для генерации определяется интерфейсом `JpaRepository`. Содержимое этих методов генерируется на основе того, как проаннотирована модель. Это позволяет держать репозитории пустыми, не используя дополнительные методы.

Это будет работать до тех пор, пока приложение не станет большим и сложным — тогда придется дописывать свои методы. Изучим пример встроенных методов:

```
userRepository.save(user);
Optional<User> user = userRepository.findById(id)
List<User> users = userRepository.findAll();
userRepository.deleteById(id);
```

Использование в коде

Использование репозитория в коде приложения базируется на инъекции зависимостей, которую мы изучим в одном из ближайших уроков. Пока просто запомним, как выполняется эта задача.

Предположим, что мы хотим использовать репозиторий в контроллере. Для этого нужно добавить соответствующее поле и пометить его аннотацией `@Autowired`. Во время работы программы Spring автоматически подставит объект репозитория, которым мы можем свободно пользоваться:

```
import org.springframework.beans.factory.annotation.Autowired;
import hexlet.code.repository.UserRepository;
// Остальные импорты

public class UsersController {

    @Autowired
    private UserRepository userRepository;

    @PostMapping("/users")
    @ResponseStatus(HttpStatus.CREATED)
    public User create(@RequestBody User user) {
        userRepository.save(user);
        return user;
    }

    @GetMapping("/users/{id}")
    @ResponseStatus(HttpStatus.OK)
    public User show(@PathVariable Long id) {
        var user = userRepository.findById(id).get();
        return user;
    }
}
```

Связь с базой данных

В реальных приложениях используют базы данных, подобные PostgreSQL. Для обучения или даже иногда для разработки можно пойти более простым путем — использовать базу данных H2, аналог SQLite. Эти базы данных работают внутри запущенного приложения. Другими словами, они не ставятся как отдельная программа, поэтому их не нужно настраивать и обслуживать. Данные в таких базах хранятся в файлах или в памяти, что еще проще. Мы пойдем именно по такому пути.

Для подключения этой базы данных нужно установить зависимость:

```
dependencies {
    runtimeOnly("com.h2database:h2")
}
```

На этом все. Spring Boot автоматически создает базу данных в памяти и подключается к ней.

Для работы нам понадобится кое-что еще. Мы создали модель или даже несколько, но каким образом будут создаваться таблицы? В реальных приложениях для этого используется Liquibase и другие механизмы миграции. Мы же воспользуемся встроенным механизмом, который позволяет создавать схему базы данных и обновлять ее автоматически:

```
spring:
  jpa:
    show-sql: true // Заодно включает показ sql-запросов
    hibernate:
      ddl-auto: update
```

За реализацию этой возможности отвечает Hibernate, который используется под капотом Spring Data JPA.

Далее →