Поиск

- JPA Specifications
  - Обновление репозитория
  - Создание DTO
  - Создание спецификации
  - Использование спецификации в контроллере

Выборка списка сущностей в API почти всегда подразумевает какую-то фильтрацию данных. Например, список постов конкретного автора, за какой-то срок или только опубликованных. А если данных много даже после фильтрации, то они отдаются постранично. Реализовать необходимую логику можно с помощью:

- Автоматической генерации методов в JPA Repository в простых случаях
- JPA Specifications в более сложных случаях
- QueryDSL или других сторонних библиотек

В этом уроке мы поговорим о JPA Specifications — механизме, который позволяет динамически собирать сложные запросы в рамках одного метода без необходимости создавать новый метод под каждое условие выборки.

JPA Specifications

Для реализации этого механизма нужно выполнить следующие шаги:

- Добавить интерфейс `JpaSpecificationExecutor` в репозиторий
- Создать DTO для параметров запроса, который будет использоваться для фильтрации
- Описать спецификацию для конкретной сущности
- Внедрить использование спецификации в контроллере

Все это мы будем добавлять для сущности `Post`. Ее код выглядит так:

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

import static jakarta.persistence.GenerationType.IDENTITY;

import java.time.LocalDate;
```

```java
import org.springframework.data.annotation.CreatedDate;
import org.springframework.data.annotation.LastModifiedDate;
import org.springframework.data.jpa.domain.support.AuditingEntityListener;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.EntityListeners;
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;
import jakarta.validation.constraints.NotNull;
import lombok.EqualsAndHashCode;
import lombok.Getter;
import lombok.Setter;
import lombok.ToString;

@Entity
@Getter
@Setter
@EntityListeners(AuditingEntityListener.class)
@ToString(includeFieldNames = true, onlyExplicitlyIncluded = true)
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@Table(name = "posts")
public class Post {
    @Id
    @GeneratedValue(strategy = IDENTITY)
    @ToString.Include
    @EqualsAndHashCode.Include
    private Long id;

    @ManyToOne(optional = false)
    private User author;

    @Column(unique = true)
    @ToString.Include
    @NotNull
    private String slug;
```

```
    @NotBlank
    @ToString.Include
    private String name;

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

    @LastModifiedDate
    private LocalDate updatedAt;

    @CreatedDate
    private LocalDate createdAt;
}
```

Обновление репозитория

Для работы динамического фильтра на базе спецификации нужно добавить интерфейс `JpaSpecificationExecutor`. В нем описаны методы для работы с данными на основе спецификации:

```
@Repository
public interface PostRepository extends JpaRepository<Post, Long>, JpaSpecificationExecuto

    // Объявлять метод не нужно
    // Он уже есть в JpaSpecificationExecutor
    // Page<Post> findAll(Specification<Post> spec, Pageable pageable);
}
```

Метод `findAll` интерфейса `JpaSpecificationExecutor` возвращает страницу с постами на основе переданной спецификации. Вторым параметром метод принимает `Pageable`, который определяет смещение и количество данных в части `LIMIT`. Это хорошая практика, потому что возвращение всех данных почти всегда приводит к проблемам с производительностью:

```
var posts = repository.findAll(/* спецификация */, PageRequest.of(/* текущая страница */,
```

Создание DTO

Обычно фильтры состоят больше, чем из одного параметра. В этом случае неудобно получать каждый параметр по отдельности. Гораздо проще создать для них DTO, который будет создан при вызове метода контроллера. Spring Boot автоматически сопоставляет параметры запроса со свойствами объекта и заполняет их, если они переданы:

```
// Имя DTO содержит часть Params
// Так становится понятно, для чего нужен этот DTO
Page<PostDTO> index(PostParamsDTO params) {
```

Сам DTO включает те параметры, по которым мы хотим фильтровать. В нашем случае это будет:

- Параметр `authorId` выбирает посты по автору
- Параметр `nameCont` выбирает посты по вхождению в название поста (здесь *Cont* обозначает *contain* — «содержать»)
- Параметр `createdAtGt` выбирает посты, появившиеся позже указанной даты (здесь *gt* обозначает *greater than* — «более чем»)
- Параметр `createdAtLt` он выбирает посты, появившиеся раньше указанной даты (здесь *lt* обозначает *lesser than* — «менее чем»)

Добавлять суффиксы *Cont*, *Gt* и *Lt* в название полей не обязательно. С другой стороны, это очень удобно, потому что позволяет использовать одно и то же поле несколько раз так, что сразу понятно, для чего нужен этот параметр и как он примерно работает. Ниже код соответствующего DTO:

```
package io.hexlet.spring.dto;

import java.time.LocalDate;

import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
public class PostParamsDTO {
    private String nameCont;
    private Long authorId;
    private LocalDate createdAtGt;
```

```
    private LocalDate createdAtLt;
}
```

Создание спецификации

Спецификация работает как билдер, которому передаются различные условия фильтрации. На базе этой спецификации Spring Boot JPA выполняет генерацию SQL. Ниже один из примеров описания спецификации:

```
// src/main/java/spring/specification/PostSpecification.java
package io.hexlet.spring.specification;

import java.time.LocalDate;

import org.springframework.data.jpa.domain.Specification;
import org.springframework.stereotype.Component;

import io.hexlet.spring.dto.PostParamsDTO;
import io.hexlet.spring.model.Post;

@Component // Для возможности автоматической инъекции
public class PostSpecification {
    // Генерация спецификации на основе параметров внутри DTO
    // Для удобства каждый фильтр вынесен в свой метод
    public Specification<Post> build(PostParamsDTO params) {
        return withAuthorId(params.getAuthorId())
                .and(withCreatedAtGt(params.getCreatedAtGt()));
    }

    private Specification<Post> withAuthorId(Long authorId) {
        return (root, query, cb) -> authorId == null ? cb.conjunction() : cb.equal(root.ge
    }

    private Specification<Post> withCreatedAtGt(LocalDate date) {
        return (root, query, cb) -> date == null ? cb.conjunction() : cb.greaterThan(root.
    }

    // Остальные методы
}
```

В методе `build` происходит сборка спецификации на основе переданных параметров. Каждый параметр формирует свое условие фильтрации данных. Обработка каждого параметра вынесена в свой метод для удобства. Внутри этих методов есть общая логика, связанная с проверкой наличия параметра. Если он отсутствует, то возвращается `cb.conjunction()`, который ни на что не влияет, но нужен для работы цепочки методов.

Спецификация представляет собой лямбда-функцию с тремя параметрами:

- Объект `root` (`Root<T>`), который считается представлением сущности. С помощью него мы указываем, по какому свойству нужно выполнять фильтрацию, включая обращение к свойствам зависимых сущностей
- Объект `cb` (`CriteriaBuilder`), который предоставляет методы для создания фильтров — `equal()`, `like()` и `greaterThan()`
- Объект `query` (`CriteriaQuery<T>`), который отвечает за формирование правильной структуры запроса. Еще с его помощью можно указывать используемые колонки, таблицы, условия фильтрации и сортировки данных

Использование спецификации в контроллере

```
package io.hexlet.blog.controller.api;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import io.hexlet.spring.dto.PostDTO;
import io.hexlet.spring.dto.PostParamsDTO;
import io.hexlet.spring.mapper.PostMapper;
import io.hexlet.spring.repository.PostRepository;
import io.hexlet.spring.specification.PostSpecification;

@RestController
@RequestMapping("/api")
public class PostsController {
```

```
    @Autowired
    private PostRepository repository;

    @Autowired
    private PostSpecification specBuilder;

    @Autowired
    private PostMapper postMapper;

    @GetMapping("/posts")
    @ResponseStatus(HttpStatus.OK)
    public Page<PostDTO> index(PostParamsDTO params, @RequestParam(defaultValue = "1") int
        var spec = specBuilder.build(params);
        // Возвращается Page<PostDTO>
        var posts = repository.findAll(spec, PageRequest.of(page - 1, 10));
        var result = posts.map(postMapper::map);

        return result;
    }
}
```

Что происходит в этом коде:

1. В метод приходят параметры для фильтрации и страница, которую нужно выбрать
2. На основе параметров для фильтрации формируется спецификация
3. Выполняется выборка данных по спецификации и с учетом указанной страницы данных. Из `page` вычитается единица, потому что иначе получится запрос `LIMIT 10 OFFSET 10` вместо `LIMIT 10 OFFSET 0`
4. Возвращенный результат `Page<Post>` преобразуется в `Page<PostDTO>` с помощью встроенного в `Page` метода `map()`, который работает точно так же, как `map()` в стримах

Далее →