SENAI

Serviço Nacional de Aprendizagem Industrial

PELO FUTURO DO TRABALHO

Serviços

Desenvolvimento de Sistemas

Prof. Me. Reneilson Santos

Abril/2024

SENAI
PELO FUTURO DO TRABALHO

Agenda

- Serviços
  - Por que usar classes de serviço?
  - Injeção de Repositório
  - Operações de Escrita
  - Mapeamento de Entrada e Saída
- Serviço no Controlador

Serviços

Serviços

As classes de serviços são classes “opcionais” do Spring Boot que deixam o nosso código mais limpo.

Com a utilização de serviços conseguimos criar mais uma camada dentro do nosso projeto que **trata da regra de negócio** e deixamos o nosso controlador mais limpo, sendo responsável apenas pela entrada/saída de dados, ou seja, uma camada de **apresentação** de fato.

Apesar de ser possível criar regras de negócio e lógica no controlador, não é considerado uma boa prática.

Serviços

Os serviços são componentes do Spring Boot com uma anotação própria que permite, portanto, o uso da injeção de dependências automatizada dentro da classe.

As classes de serviços facilitam também os testes dos controladores, deixando os testes da lógica de negócio sendo feito sobre as classes de serviços.

Criando classes de Serviço

A classe de serviço precisa ter a anotação @Service.

É uma boa prática fazer o CRUD dentro da classe de serviço, com o tratamento de validações relacionadas às regras de negócio.

```
@Service
public class EventoService { ... }
```

Usando classes de Serviço

A classe de serviço, assim como de repositório, é injetada automaticamente no nosso controlador usando o @Autowired.

```
@RestController
@RequestMapping("eventos")
public class EventoController {
    @Autowired
    private EventoService service;
    ...
}
```

Por que usar classes de Serviços?

Por quê?

Lógica de Negócios: Os serviços são responsáveis por conter a lógica de negócios da aplicação. Eles encapsulam operações e regras de negócios específicas do domínio da aplicação. Isso ajuda a manter o código limpo, modular e fácil de manter.

Separação de Responsabilidades: O uso de serviços ajuda a manter a separação de responsabilidades em seu aplicativo, seguindo princípios de arquitetura, como o princípio de responsabilidade única (SRP). Isso significa que os serviços têm uma única responsabilidade bem definida.

Por quê?

Reutilização de Código: Os serviços podem ser reutilizados em várias partes da aplicação. Isso promove a modularidade e evita a duplicação de código, permitindo que você chame os mesmos serviços de diferentes partes do aplicativo.

Testabilidade: Os serviços podem ser facilmente testados unitariamente (com testes de unidade) porque eles encapsulam a lógica de negócios. Isso facilita a criação de testes automatizados para garantir que o comportamento da aplicação seja correto.

Por quê?

Transações: Muitas vezes, os serviços também estão envolvidos na gestão de transações. O Spring Boot oferece suporte para anotações de controle de transação, como @Transactional, que podem ser usadas em métodos de serviço para garantir a consistência dos dados durante operações de banco de dados.

Integração com Controladores: Os serviços são frequentemente usados em conjunto com controladores (classes que lidam com as requisições HTTP em uma aplicação web). Os controladores podem chamar métodos de serviço para processar solicitações do cliente e, em seguida, retornar as respostas apropriadas.

Camada de Serviço: Em muitas arquiteturas de aplicativos baseados em Spring Boot, os serviços fazem parte da camada de serviço (service layer), que fica entre a camada de controle (controller layer) e a camada de acesso a dados (data access layer). Essa separação de camadas ajuda a manter a aplicação organizada e escalável.

Injeção de Repositórios

Injeção de Repositórios

Para injetar o repositório, portanto, basta fazermos uso da anotação **@Autowired**, que será usada para injeção de dependência inversa.

Como essa classe de controlador possui uma anotação do Spring Boot (**@Service**), a nossa aplicação consegue fazer a injeção de dependência sem precisarmos fazer nada além da declaração da variável.

```
@Service
public class PetService {
   @Autowired
   private PetRepository repository;
   ...
}
```

```
@Service
public class UserService {
   @Autowired
   private UserRepository repository;
   ...
}
```

Operações de Escrita

Operações de Escrita

A anotação **@Transactional** é utilizada para avisar à nossa aplicação que o dado só pode ser salvo se não houver nenhum problema (exceção) durante a execução daquele código, além de permitir que as alterações feitas no objeto vão direto para o banco de dados, caso isso seja realizado ainda dentro da operação transacional.

Ela é, portanto, utilizada para ações de escrita no banco de dados (métodos de criação, atualização e remoção).

O Transactional vem do Spring, e não do Jakarta.

```
import org.springframework.transaction.annotation.Transactional;
```

Mapeamento de Entrada e Saída

Mapeamento de E/S

Para mudar o tipo de entrada/saída para usar um DTO (Data Transfer Object), você geralmente precisará mapear a entidade persistente para o DTO correspondente.

Isso pode ser feito usando mapeamento manual ou ferramentas de mapeamento de objetos, como o **ModelMapper** ou o **MapStruct**.

ModelMapper

```
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>2.4.4</version>
</dependency>
```

ModelMapper

```
private PetOutput mapOutput(Pet pet) {
    ModelMapper modelMapper = new ModelMapper();
    return modelMapper.map(pet, PetOutput.class);
}

private Pet mapInput(PetInput input) {
    ModelMapper modelMapper = new ModelMapper();
    return modelMapper.map(input, Pet.class);
}
```

ModelMapper

Se o **ModelMapper** estiver pedindo um construtor sem argumentos e público, você pode precisar adicionar um construtor sem argumentos explicitamente à sua classe Lombok, mesmo que o Lombok já forneça um construtor por padrão.

```
@NoArgsConstructor
```

Com a anotação @NoArgsConstructor, o Lombok irá adicionar um construtor sem argumentos à sua classe, atendendo assim aos requisitos do ModelMapper.

```java
public List<PetOutput> list() {
    return repository
            .findAll()
            .stream()
            .map(item -> mapOutput(item))
            .toList();
}

public PetOutput create(PetInput input) {
    Pet pet = repository.save(mapInput(input));
    return mapOutput(pet);
}

public PetOutput read(Long id) {
    Pet pet = repository.findById(id).orElse(null);
    return mapOutput(pet);
}
```

```java
public PetOutput update(Long id, PetInput input) {
    if (repository.existsById(id)) {
        Pet pet = mapInput(input);
        pet.setId(id);
        pet = repository.save(pet);
        return mapOutput(pet);
    }
    return null;
}

public void delete(Long id) {
    repository.deleteById(id);
}
```

Serviço no Controlador

```java
@RestController
@RequestMapping("pets")
public class PetController {
    @Autowired
    private PetService service;

    @GetMapping
    public ResponseEntity<List<PetOutput>> list() {
        return ResponseEntity.ok(service.list());
    }

    @PostMapping
    public ResponseEntity<PetOutput> create(@ResponseBody PetInput input){
        return ResponseEntity.status(201).body(service.create(input));
    }

    @DeleteMapping("/{id}")
    public ResponseEntity delete(Long id){
        service.delete(id);
        return ResponseEntity.noContent().build();
    }
```

```java
    @PutMapping("/{id}")
    public ResponseEntity<PetOutput> update(Long id, @ResponseBody PetInput input){
        PetOutput petUpdated = service.update(id, input);
        if(petUpdated != null){
            return ResponseEntity.ok(petUpdated);
        }else{
            return ResponseEntity.notFound().build();
        }
    }
    @GetMapping("/{id}")
    public ResponseEntity<PetOutput> read(Long id){
        PetOutput output = service.read(id);
        if(output != null){
            return ResponseEntity.ok(output);
        }else{
            return ResponseEntity.notFound().build();
        }
    }
}
```

Atividade Serviços

Criar os serviços para as classes:

- Order
- Category
- Tag

Bibliografia

Bibliografia

- JPA Repositories (https://docs.spring.io/spring-data/jpa/docs/1.5.0.RELEASE/reference/html/jpa.repositories.html)
- Accessing Data with JPA (https://spring.io/guides/gs/accessing-data-jpa/)
- Spring Data Partial Update (https://www.baeldung.com/spring-data-partial-update)
-