01 Produção de dados na API

Transcrição

Nas aulas anteriores finalizamos a funcionalidade de cadastro de médicos e agora criaremos a funcionalidade de listagem de médicos.

Essa funcionalidade deverá exibir nome, email, *CRM* e especialidade de cada médico, de maneira ordenada. Começando pelo nome, de maneira crescente, bem como ser paginada, com 10 registros por página.

Vamos acessar "src > main > java > med.voll.api > controller > MedicoController". Até então, havíamos implementado apenas o método `cadastrar`. Agora criaremos o método `public`, responsável pela listagem.

O retorno dele será `List<Medico> listar ()`. Acima do método, vamos adicionar a anotação `@GetMapping`, para informar o verbo do protocolo *HTTP*.

Como precisamos acessar o banco de dados, passaremos a classe `repository.findaAll()` como retorno:

```
@RestController
@RequestMapping("medicos")
public class MedicoController {

    @Autowired
```

```
    private MedicoRepository repository;


    @PostMapping
    @Transactional
    public void cadastrar(@RequestBody @Valid DadosCadastroMedico
dados) {
        repository.save(new Medico(dados));
    }


    @GetMapping
    public List<Medico> listar() {
        return repository.findAll();
    } COPIAR CÓDIGO
```

Porém, não podemos devolver uma lista de `Medico`, porque não queremos devolver todos os atributos dela, apenas nome, email, *CRM* e especialidade.

Por isso, criaremos um *DTO* que devolverá dados da *API*. Substituiremos `Medico` por `DadosListagemMedico`. Vamos usar o atalho "Alt + Enter e selecionar a opção "Create record 'DadosListagemMedico'". Em "Destination package", informaremos "med.voll.api.medico" e clicaremos em "OK".

Agora precisamos informar as propriedades que serão trabalhadas pelo *DTO*, que serão `String nome`, `String email`, `String crm` e `Especialidade especialidade`:

```
package med.voll.api.medico;
```

```
public record DadosListagemMedico(String nome, String email, String
crm, Especialidade especialidade) { COPIAR CÓDIGO
```

Vamos salvar e voltar para "MedicoController.java", onde encontraremos um erro de compilação na linha do *return* do método `listar`. Vamos fazer a conversão de `Medico` para `DadosListagemMedico`.

Faremos isso chamando o método `.stream().map()`. Dentro dos parâmetros de `.map`, passaremos `DadosListagemMedico::new`, chamando o construtor do *DTO* `DadosListagemMedico`:

```
@RestController
@RequestMapping("medicos")
public class MedicoController {

    @Autowired
    private MedicoRepository repository;

    @PostMapping
    @Transactional
    public void cadastrar(@RequestBody @Valid DadosCadastroMedico
dados) {
        repository.save(new Medico(dados));
    }

    @GetMapping
    public List<DadosListagemMedico> listar() {
```

```
        return
repository.findAll().stream.map(DadosListagemMedico::new);
    } COPIAR CÓDIGO
```

Vamos voltar para `DadosListagemMedico.java` e declarar um construtor:

```
package med.voll.api.medico;


public record DadosListagemMedico(String nome, String email, String
crm, Especialidade especialidade) {


    public DadosListagemMedico(Medico medico) {
        this(medico.getNome(), medico.getEmail(), medico.getCrm(),
medico.getEspecialidade());
    }


} COPIAR CÓDIGO
```

Vamos voltar para "MedicoController.java", que ainda indica erro, porque precisamos adicionar `.toList()` ao retorno, para converter em uma lista:

```
@RestController
@RequestMapping("medicos")
public class MedicoController {


    @Autowired
    private MedicoRepository repository;
```

```
    @PostMapping
    @Transactional
    public void cadastrar(@RequestBody @Valid DadosCadastroMedico
dados) {
        repository.save(new Medico(dados));
    }

    @GetMapping
    public List<DadosListagemMedico> listar() {
        return
repository.findAll().stream().map(DadosListagemMedico::new).toList();
    } COPIAR CÓDIGO
```

No próximo vídeo, executaremos um teste no *Insomnia*.

02 Testando a listagem

Transcrição

Vamos acessar o *Insomnia* para testar nossa *API*.

Como não queremos mais testar o cadastro de médicos, precisaremos criar uma nova requisição. No painel à esquerda, clicaremos em "+ > Http Request". Vamos renomear a nova requisição de "New Request" para "Listagem de médicos".

O verbo será o padrão, "GET". Em seguida, digitaremos a *URL* da requisição para o mesmo endereço do cadastro: "http://localhost:8080/medicos".

Obs: Não haverá conflito porque os verbos das duas requisições da mesma *URL* são diferentes.

O *body* da requisição irá vazio, porque não estamos cadastrando informações. Clicando no botão "Send", vamos disparar a requisição. No painel, receberemos os dados devolvidos pela *API*.

O *Spring Boot* assumiu automaticamente que queremos converter a lista para um *JSON*. Isso faz com que eles no devolva um arquivo desse tipo, contendo um *array* listando todos os médicos cadastrados no banco de dados.

No próximo vídeo, aprenderemos a ordenar a lista por nome, paginá-la e fazê-la exibir apenas 10 resultados por página.

03 Para saber mais: DTOs ou entidades?

Estamos utilizando DTOs para representar os dados que recebemos e devolvemos pela API, mas você provavelmente deve estar se perguntando “Por que ao invés de criar um DTO não devolvemos diretamente a entidade JPA no Controller?”. Para fazer isso, bastaria alterar o método `listar` no Controller para:

```
@GetMapping
public List<Medico> listar() {
    return repository.findAll();
} COPIAR CÓDIGO
```

Desse jeito o código ficaria mais enxuto e não precisaríamos criar o DTO no projeto. Mas, será que isso realmente é uma boa ideia?

Os problemas de receber/devolver entidades JPA

De fato é muito mais simples e cômodo não utilizar DTOs e sim lidar diretamente com as entidades JPA nos controllers. Porém, essa abordagem tem algumas desvantagens, inclusive causando vulnerabilidade na aplicação para ataques do tipo **Mass Assignment**.

Um dos problemas consiste no fato de que, ao retornar uma entidade JPA em um método de um Controller, o Spring vai gerar o JSON contendo **todos** os atributos dela, sendo que nem sempre esse é o comportamento que desejamos.

Eventualmente podemos ter atributos que não desejamos que sejam devolvidos no JSON, seja por motivos de segurança, no caso de dados *sensíveis*, ou mesmo por não serem utilizados pelos clientes da API.

Utilização da anotação `@JsonIgnore`

Nessa situação, poderíamos utilizar a anotação `@JsonIgnore`, que nos ajuda a ignorar certas propriedades de uma classe Java quando ela for serializada para um objeto JSON.

Sua utilização consiste em adicionar a anotação nos atributos que desejamos ignorar quando o JSON for gerado. Por exemplo, suponha

que em um projeto exista uma entidade JPA `Funcionario`, na qual desejamos ignorar o atributo `salario`:

```
@Getter
@NoArgsConstructor
@EqualsAndHashCode(of = "id")
@Entity(name = "Funcionario")
@Table(name = "funcionarios")
public class Funcionario {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nome;
    private String email;

    @JsonIgnore
    private BigDecimal salario;

    //restante do código omitido…
} COPIAR CÓDIGO
```

No exemplo anterior, o atributo `salario` da classe `Funcionario` não será exibido nas respostas JSON e o problema estaria solucionado.

Entretanto, pode acontecer de existir algum outro endpoint da API na qual precisamos enviar no JSON o salário dos funcionários, sendo que nesse caso teríamos problemas, pois com a anotação `@JsonIgnore` tal atributo **nunca** será enviado no JSON, e ao remover a anotação o

atributo **sempre** será enviado. Perdemos, com isso, a flexibilidade de controlar quando determinados atributos devem ser enviados no JSON e quando não.

DTO

O padrão DTO (*Data Transfer Object*) é um padrão de arquitetura que era bastante utilizado antigamente em aplicações Java distribuídas (arquitetura cliente/servidor) para representar os dados que eram enviados e recebidos entre as aplicações cliente e servidor.

O padrão DTO pode (e deve) ser utilizado quando não queremos expor todos os atributos de alguma entidade do nosso projeto, situação igual a dos salários dos funcionários mostrado no exemplo de código anterior. Além disso, com a flexibilidade e a opção de filtrar quais dados serão transmitidos, podemos poupar tempo de processamento.

Loop infinito causando `StackOverflowError`

Outro problema muito recorrente ao se trabalhar diretamente com entidades JPA acontece quando uma entidade possui algum autorrelacionamento ou relacionamento bidirecional. Por exemplo, considere as seguintes entidades JPA:

```
@Getter
@NoArgsConstructor
@EqualsAndHashCode(of = "id")
@Entity(name = "Produto")
@Table(name = "produtos")
public class Produto {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nome;
    private String descricao;
    private BigDecimal preco;

    @ManyToOne
    @JoinColumn(name = "id_categoria")
    private Categoria categoria;

    //restante do código omitido…
} COPIAR CÓDIGO
@Getter
@NoArgsConstructor
@EqualsAndHashCode(of = "id")
@Entity(name = "Categoria")
@Table(name = "categorias")
public class Categoria {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nome;

    @OneToMany(mappedBy = "categoria")
    private List<Produto> produtos = new ArrayList<>();
```

```
    //restante do código omitido…
} COPIAR CÓDIGO
```

Ao retornar um objeto do tipo `Produto` no Controller, o Spring teria problemas para gerar o JSON desse objeto, causando uma exception do tipo `StackOverflowError`. Esse problema ocorre porque o objeto produto tem um atributo do tipo `Categoria`, que por sua vez tem um atributo do tipo `List<Produto>`, causando assim um loop infinito no processo de serialização para JSON.

Tal problema pode ser resolvido com a utilização da anotação `@JsonIgnore` ou com a utilização das anotações `@JsonBackReference` e `@JsonManagedReference`, mas também poderia ser evitado com a utilização de um DTO que representa apenas os dados que devem ser devolvidos no JSON.

04 Paginação

Transcrição

Agora vamos cuidar da paginação e da ordenação.

Começaremos pela paginação. Queremos trazer apenas 10 registros por página. Vamos configurar uma requisição para a *API* para passar de página quando necessário.

Como paginação é algo comum, o *Spring* já tem um mecanismo para fazer isso. Em "MedicoController.java", no método `listar`, passaremos o parâmetro `Pageable`.

Obs: Cuidado na hora de importar. Selecione `Pageable org.springframework.data.domain` e não `Pageable java.awt.print`. A segunda não é aplicável ao *Spring Framework*.

Daremos o nome `paginacao` ao parâmetro. Vamos passar o novo parâmetro dentro do método `.findAll`. Com isso, o *Spring* montará a *query* automaticamente com o esquema de paginação.

Substituiremos, também, o retorno do método. Não será mais `List`, e sim `Page`. Vamos alterar também o `return`, que agora não precisará mais da chamada do método `.stream()`. `.toList()` também não será mais necessário:

```
    @GetMapping
    public Page<DadosListagemMedico> listar(Pageable paginacao) {
        return
repository.findAll(paginacao).map(DadosListagemMedico::new);
    } COPIAR CÓDIGO
```

Agora vamos salvar e voltar ao *Insomnia* para disparar a requisição. O JSON devolvido, agora, terá o atributo `content`, com o *array* da lista de médicos dentro dele. Ao final, encontraremos novas informações relacionadas à paginação.

Agora vamos controlar o número de registros exibidos. Para isso, passaremos, na *URL*, o parâmetro `?size`. Se o igualarmos a 1, teremos a exibição de apenas um registro na tela:

http://localhost:8080/medicos?size=1 COPIAR CÓDIGO

Obs: Se não passarmos o parâmetro `size`, o Spring devolverá 20 registros por padrão.

Para trazermos a página, vamos passar outro parâmetro na *URL*, após usar um `&`. Será o parâmetro `page`. Como a primeira página é representada por `page=0`, para trazer o próxima, traremos `page=1`. E assim sucessivamente.

Com esse dois parâmetros, controlamos a paginação.

05 Ordenação

Transcrição

Agora vamos cuidar da ordenação.
Para fazer isso, vamos remover os parâmetros `size` e `page`, adicionados por nós na aula anterior. Para mudar a ordenação, também usaremos um parâmetro na *URL*, chamado `sort`.

Junto dele, passamos o nome do atributo na entidade. Se quisermos ordenar pelo nome, por exemplo, passaremos a *URL* `http://localhost:8080/medicos?sort=nome`.

Se dispararmos a aquisição, veremos que a exibição dos registros será feita em ordem alfabética. Se quiser ordenar por outro parâmetro, basta substituir a informação depois de `sort`.

Por padrão, a ordenação acontece de maneira crescente. Mas é possível inverter isso, ordenando por ordem decrescente. Para isso, basta adicionar `,desc` à *URL*.

É possível combinar com os parâmetro que vimos no vídeo anterior. Basta adicioná-los na *URL*, sempre conectando-os com um `&`, como no exemplo abaixo:

```
http://localhost8080/medicos?sort=crm,desc&size=2&page=1 COPIAR CÓDIGO
```

Por padrão, o nome dos parâmetros é escrito em inglês. Porém, consguimos customizar esses parâmetros no arquivo "application.properties".

Vamos voltar à *IDE*. O parâmetro `Pageable`, que usamos em `lista`, é opcional. Se voltarmos para o *Insomnia* e disparmos a requisição sem nenhum parâmetro na *URL*, ela vai carregar todos os registros usando o padrão do *Spring*.

O padrão é 20 resultados por página, e na ordem em que cadastramos a informação no banco de dados. É possível, porém, alterar esse padrão.

Em "MedicoController.java", podemos trocar o padrão da paginação adicionando uma anotação no parâmetro `Pageable`. O nome dela é `@PageableDefault`. Na sequência, abrimos parênteses e passamos os atributos `size`, `page` e `sort`. Podemos escolher o atributo que guiará a ordenação, passando entre chaves duplas.

Por exemplo, se passarmos `lista(@PageableDefault(size = 10, sort = {"nome"})`, isso significa que, caso não passemos parâmetros

na *URL*, no *Insomnia*, o novo padrão será a exibição de 10 resultados por página, ordenados a partir do nome.

Vamos salvar e, depois disso, podemos testar no *Insomnia*. Se executarmos sem passar parâmetros na *URL*, veremos que tivemos sucesso em definir o nmovo padrão.

Se adicionarmos parâmetros na *URL*, o *Insomnia* usará as informações da *URL*.

Caso queiramos saber como a *query* está sendo feita no banco de dados, podemos configurar para que isso seja exibido para nós. A configuração é feita em "src > main > resources > application.properties".

Lá, vamos adicionar `spring.jpa.show-sql=true`. Com isso, os *SQLs* disparados no banco de dados serão impressos.
Com isso, conseguiremos ver as informações das queries na *IDE* após fazermos a requisição no *Insomnia*.

O parâmetro é difícil de ser visualizado, porque é exibido numa linha só. Para facilitar a visualização, vamos adicionar outro parâmetro, que passa as informações com quebras de linha.

De volta a "application.properties", passaremos a propriedade `spring.jpa.properties.hibernate.format_sql=true`.

Depois que salvarmos, veremos que tivemos sucesso.

Conseguimos implementar nossa funcionalidade de listagem de médicos usando paginação, ordenação e usando o *log* para ver o que está sendo disparado.

A próxima funcionalidade, que veremos no próximo vídeo, será a atualização de médicos.

06 Para saber mais: parâmetros de paginação

Conforme aprendemos nos vídeos anteriores, por padrão, os parâmetros utilizados para realizar a paginação e a ordenação devem se chamar `page`, `size` e `sort`. Entretanto, o Spring Boot permite que os nomes de tais parâmetros sejam modificados via configuração no arquivo `application.properties`.

Por exemplo, poderíamos traduzir para português os nomes desses parâmetros com as seguintes propriedades:

```
spring.data.web.pageable.page-parameter=pagina

spring.data.web.pageable.size-parameter=tamanho

spring.data.web.sort.sort-parameter=ordem COPIAR CÓDIGO
```

Com isso, nas requisições que utilizam paginação, devemos utilizar esses nomes que foram definidos. Por exemplo, para listar os médicos de nossa API trazendo apenas 5 registros da página 2, ordenados pelo e-mail e de maneira decrescente, a URL da requisição deve ser:

```
http://localhost:8080/medicos?tamanho=5&pagina=1&ordem=email,desc

COPIAR CÓDIGO
```

07 Limitando dados

Um colega de trabalho está tendo dificuldades em utilizar o recurso de paginação do Spring Boot e pediu sua ajuda.

Ao analisar a classe Controller que ele criou, você identificou o seguinte método:

```
@GetMapping
public void carregarProdutosCadastrados(Pageable paginacao) {
    repository.findAll().stream().map(DadosListagemProduto::new);
} COPIAR CÓDIGO
```

Quais problemas no código anterior você identifica? Selecione até duas alternativas.

- O parâmetro paginação não está sendo utilizado.

  O parâmetro paginação foi declarado corretamente no método do Controller, entretanto, ele não foi utilizado na chamada ao método do repository.

- Alternativa correta

  Faltou adicionar o parâmetro de paginação na anotação `@GetMapping`.

- Alternativa correta

  O retorno do método está como `void`.

  Para devolver as informações da API, o método não pode ter `void` como retorno.

- Alternativa correta

  O código anterior não tem problemas, então a paginação não deve estar funcionando por algum problema na requisição sendo disparada.

Parabéns, você acertou!

08 Faça como eu fiz: listagem de pacientes

Agora é com você! Faça o mesmo procedimento que eu fiz na aula, porém, para a funcionalidade de **listagem de pacientes**.

Opinião do instrutor

-
-

Você precisará adicionar um novo método no Controller de paciente:

```
@GetMapping
public Page<DadosListagemPaciente> listar(@PageableDefault(page = 0,
size = 10, sort = {"nome"}) Pageable paginacao) {
    return
repository.findAll(paginacao).map(DadosListagemPaciente::new);
} COPIAR CÓDIGO
```

Também precisará criar o DTO `DadosListagemPaciente`:

```
public record DadosListagemPaciente(String nome, String email, String
cpf) {
    public DadosListagemPaciente(Paciente paciente) {
        this(paciente.getNome(), paciente.getEmail(),
paciente.getCpf());
    }
} COPIAR CÓDIGO
```

E, caso queira ver os comandos SQL disparados no banco de dados, vai precisar adicionar as seguintes propriedades no arquivo `application.properties`:

```
spring.jpa.show-sql=true
```

```
spring.jpa.properties.hibernate.format_sql=true COPIAR CÓDIGO
```

09 O que aprendemos?

Nessa aula, você aprendeu como:

- Utilizar a anotação `@GetMapping` para mapear métodos em Controllers que produzem dados;
- Utilizar a interface `Pageable` do Spring para realizar consultas com paginação;
- Controlar a paginação e a ordenação dos dados devolvidos pela API com os parâmetros `page`, `size` e `sort`;
- Configurar o projeto para que os comandos SQL sejam exibidos no console.