01 Apresentação

Transcrição

Boas-vindas ao curso de **Spring Boot 3: aplique boas práticas e proteja uma API Rest!**

Me chamo Rodrigo Ferreira e serei o seu instrutor ao longo deste curso, em que vamos aprender como usar as ferramentas do Spring Boot, sendo um framework do Java.

Rodrigo Ferreira é uma pessoa de pele clara, com olhos castanhos e cabelos castanhos e curto. Veste camiseta rosa lisa, e está sentado em uma cadeira preta. Ao fundo, há uma parede lisa com iluminação azul gradiente.

No curso anterior...

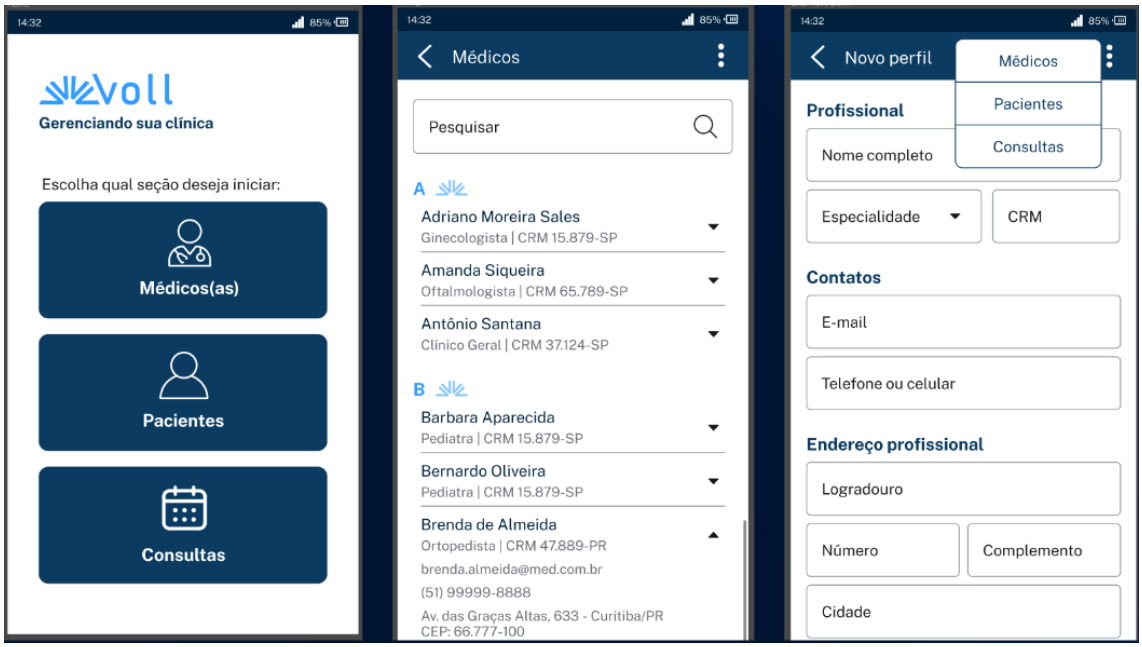
Este é o segundo curso, é importante que você tenha concluído o anterior Curso de Spring Boot 3: desenvolva uma API Rest em Java.
Nele, iniciamos o projeto que daremos continuidade.
Recapitulando, no primeiro curso aprendemos como funciona o **Spring Boot**, a criar um projeto usando **Spring initializr** e começamos a desenvolver a **API Rest**.
No caso, fizemos o **CRUD** (*Create, Read, Update e Delete*), implementamos a funcionalidade de cadastro, listagem, remoção e atualização. Aprendemos, também, a fazer validações de formulários utilizando o *bean validation*. Por fim, focamos na **paginação e ordenação**.

- Desenvolvimento de uma API Rest

- CRUD (Create, Read, Update e Delete)
- Validações
- Paginação e ordenação

Tudo isso foi desenvolvido no curso anterior, e a ideia deste presente curso é dar continuidade ao projeto iniciado. Partiremos do ponto que paramos no curso passado, e aprenderemos novos recursos do framework.

A seguir, vamos relembrar o *layout* do aplicativo mobile da nossa aplicação:



Lembrando que focamos na parte de back-end, na API Rest, e continuaremos trabalhando nesse projeto da clínica médica. Desenvolvemos o CRUD de médicos e pacientes, e daremos prosseguimento neste curso.

Objetivos

- Boas práticas na API
- Tratamento de erros
- Autenticação/Autorização
- Tokens JWT

Os objetivos deste segundo curso são: **aprender boas práticas na API** referente ao protocolo HTTP. Faremos ajustes na classe controller, para seguir as boas práticas do protocolo HTTP quanto ao retorno dos códigos HTTP e das respostas que a API devolve.
Logo após, realizaremos **tratamento de erros**. Eventualmente, pode ocorrer um erro na API, e precisamos entender o que o Spring faz ao ocorrer uma *exception* enquanto o programa é executado, o que é devolvido como resposta para o cliente da API.
Assim, vamos **personalizar** esses retornos para tratar esses erros da melhor forma possível.
Após isso, focaremos na segurança, no **controle de autenticação e de autorização** da API. No curso anterior não abordamos isso, logo a nossa API está pública - qualquer pessoa pode enviar requisições para remover, atualizar ou alterar informações da API.
Mas não é dessa forma que desejamos, precisamos ter um controle. Isso será feito na aplicação front-end, porém, na API precisamos ter um código que permite o usuário se **autenticar**, e também ter um controle de acesso de informações públicas e privadas.
Aprenderemos a aplicar isso com o **Spring Security**, sendo um módulo do Spring responsável por monitorar esse controle.
No caso, usaremos a autenticação fundamentada em tokens com o padrão **JSON Web Token (JWT)**.

São esses os objetivos do segundo curso, focaremos em boas práticas, tratamento de erros e no controle de acesso, autenticação e autorização, usando tokens.

Gostou? Caso, sim, te espero na próxima aula!

02 Preparando o ambiente - Projeto inicial

Neste curso, utilizaremos o mesmo projeto que foi finalizado no primeiro curso de Spring Boot. Você pode obter uma cópia do projeto neste repositório do GitHub: Projeto Inicial.

03 Padronizando retornos da API

Transcrição

Nesta aula faremos alguns ajustes no código do projeto, para melhorarmos algumas questões referentes às boas práticas.

No Insomnia, temos as requisições CRUD feitas na API, tanto de médicos quanto de pacientes. É justamente o que fizemos no curso anterior.

Pelo Insomnia disparamos as requisições para a API, isso para simular o aplicativo mobile ou aplicação front-end se comunicando com a API back-end.

Temos algumas considerações importantes: lembre-se que estamos desenvolvendo uma **API Rest seguindo o protocolo HTTP**. Logo, precisamos seguir às boas práticas relacionadas ao protocolo.

Uma dessas boas práticas já estamos aplicando: cada requisição que temos no Insomnia (excluir, atualizar, cadastrar e remover médico e pacientes), usamos os verbos do protocolo HTTP conforme a requisição.

Para excluir usamos o verbo `delete`, para cadastrar um médico ou paciente utilizamos o método `post`, para listar usamos o `get` e para atualizar utilizamos o `put`.

- Remover: `delete`
- Cadastrar: `post`
- Listar: `get`
- Atualizar: `put`

Porém, não são somente os verbos do protocolo HTTP e a URL que consideramos importantes, precisamos ter um **tratamento do retorno da API nas requisições**.

Por exemplo, na requisição `DEL` de excluir um médico do lado esquerdo do Insomnia. Perceba que a requisição disparará um método `delete` com a URL `http://localhost:8080/medicos/2`, sendo o ID 2.

Requisições no Insomnia:

DEL	Excluir Médico
PUT	Atualizar Médico
GET	Listagem de médicos
POST	Cadastro de Médico
POST	Cadastro de Paciente
GET	Listagem de pacientes
PUT	Atualizar Paciente
DEL	Excluir Paciente

Se clicarmos no botão "*Send*" para enviar à requisição, repare que ele nos devolve um código `200 OK`, contido em uma caixa na cor verde, do lado direito. Isso significa que a requisição foi feita com sucesso.

O código `200` é o código do protocolo HTTP que significa "OK". No corpo da resposta, na aba "*Preview*", temos uma mensagem "*No body returned for response*" (em português, "Nenhum corpo retornou para resposta"). Está vazio.

Na requisição `PUT` de atualizar médico, temos algo semelhante:

```
PUT: http://localhost:8080/medicos

{
    "id": 2,
    "telefone": "1100009999"
} COPIAR CÓDIGO
```

Perceba que disparamos a requisição levando o JSON com os dados do médico que desejamos atualizar. Ao clicarmos no botão "Send", nos

devolve o código `200 OK` e com o corpo da resposta vazio. Igual ao caso anterior.

Será que precisamos retornar `200` em todas as requisições? Ou há outros códigos no protocolo HTTP mais adequados? É justamente nesse detalhe que faremos os ajustes.

O protocolo HTTP possui diversos códigos para vários cenários e não estamos usando esse recurso de forma adequada.

Estamos devolvendo `200` quando dá certo ("OK"), ou `500` que o Spring devolve de forma automática, ou erro `400` caso ocorra um erro de validação do *bean validation*.

Vamos analisar tudo isso no projeto.

Abriremos o IntelliJ, com o projeto já importado, sendo o mesmo que finalizamos no curso anterior. Do lado esquerdo, clicamos em "src > main > java > med.voll.api > controller > MedicoController".

Lembre-se que toda requisição chega no arquivo `MedicoController`. Se analisarmos o método excluir, repare que colocamos o retorno como `void`.

```
//código omitido


 @DeleteMapping("/{id}")
 @Transactional
    public void excluir(@PathVariable Long id) {
        var medico = repository.getReferenceById(id);
```

```
        medico.excluir();
} COPIAR CÓDIGO
```

Não colocamos nenhum retorno no método de atualizar e de cadastrar. Devolvemos algo somente no método de listar, porque precisamos retornar algo, no caso, estamos usando a paginação do Spring Boot.

Porém, os outros métodos estão devolvendo um `void`. Isso é um problema!

Ao usarmos o `void`, não informamos o que o Spring precisa devolver. Por isso, por padrão, ele retorna o código `200 OK`, se a requisição for processada com sucesso.

Porém, há outros códigos HTTP mais customizados dependendo do cenário. Por exemplo, no método excluir, o mais adequado seria devolver o código `204`, que se refere à requisição processada e sem conteúdo.

Para devolver o código `204`, começaremos a padronizar esses métodos. Ao invés de usarmos o `void`, **utilizaremos uma classe do Spring** chamada `ResponseEntity`, sendo uma classe que conseguimos controlar a resposta devolvida pelo framework.

```
// código omitido


 @DeleteMapping("/{id}")
 @Transactional
    public ResponseEntity excluir(@PathVariable Long id) {
```

```
        var medico = repository.getReferenceById(id);
        medico.excluir();
} COPIAR CÓDIGO
```

Assim, o retorno não será mais vazio ("*void*") e sim um objeto do tipo `ResponseEntity`. Esse método terá erro de compilação, ainda precisamos incluir o `return`.

Na última linha do método excluir, após a exclusão, adicionaremos o `return`. Contudo, como instanciamos o objeto `ResponseEntity`? Na classe `ResponseEntity` há métodos estáticos, que podemos usar neste caso.

Logo, colocaremos `return ResponseEntity`, sendo a classe, e digitamos um ponto ("."): `return ResponseEntity.`.

Perceba que será exibido uma caixa com vários métodos que podemos devolver conforme o que desejamos,
como `ok`, `badRequest`, `noContent`, `notFound`, etc.

Clicaremos no método `noContent` para o método excluir, com isso, ficaremos com: `return ResponseEntity.noContent();`.

Perceba que aparece um sublinhado na cor vermelha no retorno, isso significa que o método `noContent` não devolve um `ResponseEntity`. Na sequência, chamaremos o `.build()`.

```
// código omitido
```

```
return ResponseEntity.noContent().build();COPIAR CÓDIGO
```

O `noContent()` cria um objeto e chamamos o `build()` para construir o objeto `ResponseEntity`.

```
//código omitido


@DeleteMapping("/{id}")
@Transactional
    public ResponseEntity excluir(@PathVariable Long id) {
        var medico = repository.getReferenceById(id);
        medico.excluir();


        return ResponseEntity.noContent().build();
    } COPIAR CÓDIGO
```

Podemos salvar clicando em "Ctrl + S".

Agora, voltaremos à Insomnia para enviarmos uma requisição para excluir um médico. Ele nos devolvia o código `200 OK`, como podemos visualizar no lado direito.

Ao dispararmos a requisição clicando no botão "Send", o código se torna `204 No Content`. Colocando o mouse por cima, é exibida uma mensagem explicando o que significa esse código.

Esse erro faz parte da categoria `200`, que são os códigos de requisição que obtiveram sucesso, porém, é específico. O `204 No Content' significa que foi processado com sucesso, mas não possui conteúdo para ser mostrado na resposta.

Para remover, essa é uma boa prática, usar o código `204` e não o `200`. Faremos essa mesma alteração nos outros métodos do arquivo `MedicoController`, para padronizarmos o controller e termos **todos** os métodos retornando um objeto `ResponseEntity`.

Nos métodos `atualizar`, `cadastrar` e `listagem`, alteramos de `void` para `ResponseEntity`.

Método atualizar

```
//código omitido


@PutMapping
@Transactional
public ResponseEntity atualizar(@RequestBody @Valid
DadosAtualizacaoMedico dados) {
        var medico = repository.getReferenceById(dados.id());
        medico.atualizarInformacoes(dados);
    }COPIAR CÓDIGO
```

Método cadastrar

```
//código omitido


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

No método de listar, incluiremos o `ResponseEntity` e no `Page<DadosListagemMedico>`, colocaremos mais um "<>" mas entre o `page` e o `DadosListagemMedico`: `public <Page<DadosListagemMedico>>`.

Método listagem

```
//código omitido


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

Vamos padronizar, todos os métodos usaram a classe `ResponseEntity` do Spring Boot.

Como esperado, será exibido o erro de compilação (sublinhado na cor vermelha do método), isso acontece porque precisamos alterar para devolver o objeto `ResponseEntity`. Faremos esses ajustes agora.

No método de listar, alteramos o `return`. Isto é, ele não devolverá diretamente o *page* e vamos colocar o `repository.findAllByAtivoTrue(paginacao)` em uma variável chamada `page`.

```
//código omitido
```

```
@GetMapping
public ResponseEntity<Page<DadosListagemMedico>>
listar(@PageableDefault(size = 10, sort = {"nome"}) Pageable
paginacao) {
        var page =
repository.findAllByAtivoTrue(paginacao).map(DadosListagemMedico::new)
;
    }

//código omitido COPIAR CÓDIGO
```

Na linha seguinte, faremos o `return ResponseEntity.ok()`, porque queremos devolver o código `200`. No parênteses do `ok()`, passamos o objeto "page".

```
//código omitido

@GetMapping
public ResponseEntity<Page<DadosListagemMedico>>
listar(@PageableDefault(size = 10, sort = {"nome"}) Pageable
paginacao) {
        var page =
repository.findAllByAtivoTrue(paginacao).map(DadosListagemMedico::new)
;
        return ResponseEntity.ok(page);
    }

//código omitido COPIAR CÓDIGO
```

Com isso, no método de listar será devolvido o código `200`, e junto na resposta vem o objeto de paginação com os dados dos médicos.

Nos métodos de cadastrar e atualizar, como funcionaria?

Analisaremos, primeiro, o método `atualizar`.

Método atualizar

```
//código omitido


@PutMapping
@Transactional
public ResponseEntity atualizar(@RequestBody @Valid
DadosAtualizacaoMedico dados) {
        var medico = repository.getReferenceById(dados.id());
        medico.atualizarInformacoes(dados);
    }


//código omitidoCOPIAR CÓDIGO
```

Neste método, usávamos `void`, ou seja, não tínhamos retorno nenhum. Porém, diferente do método de excluir, no de atualizar não podemos devolver um código `204`.

O mais interessante no método de atualizar é devolver a informação atualizada. Como é para atualizar o registro do médico, no final devolveremos os dados do médico atualizado.

No final do método digitaremos `return ResponseEntity.ok()` e dentro dos parênteses precisamos passar o objeto.

Contudo, **não podemos passar o objeto médico**, porque é uma entidade JPA ("Java Persistence API") e **não é recomendado devolver e receber entidades JPA no controller**.

Logo, precisamos devolver um DTO ("*Data Transfer Object*"). Por exemplo, temos o `DadosAtualizacaoMedico` que é o nosso DTO. Clicando em "DadosAtualizacaoMedico", conseguimos visualizar o código.

Porém, esse DTO está incompleto, ele possui somente o `id`, `nome`, `telefone` e `endereço`. Sendo o DTO que representa os dados da atualização de um médico, isto é, os dados que o aplicativo mobile enviará para atualizar as informações.

No caso, queremos devolver **todas as informações do médico**. Por isso, criaremos outro DTO para representar esses dados do médico que estão sendo atualizados.

Voltando ao arquivo `MedicoController`, dentro do parênteses do `ok.()` colocaremos `new DadosDetalhamentoMedico()`. Agora, sim, passamos o objeto médico: `new DadosDetalhamentoMedico(medico)`.

```
//código omitido


@PutMapping
@Transactional
public ResponseEntity atualizar(@RequestBody @Valid
DadosAtualizacaoMedico dados) {
        var medico = repository.getReferenceById(dados.id());
```

```
        medico.atualizarInformacoes(dados);

        return ResponseEntity.ok(new DadosDetalhamentoMedico(medico));
    }

//código omitido COPIAR CÓDIGO
```

Perceba que `DadosDetalhamentoMedico` está escrito em uma cor vermelha bem forte, isso significa que ocorreu erro de compilação. Para ajustar isso, selecionaremos "Alt + Enter" no teclado, será exibido um *pop-up* com diversas opções, escolheremos a "`Create record 'DadosDetalhamentoMedico'`". Isso para ele criar esse DTO para nós.

Na caixa exibida, temos o título "*Create Record DadosDetalhamentoMedico*", com os campos "*Destination package*" e "*Target destination directory*".

No primeiro campo "*Destination package*" consta "med.voll.api.controller", alteraremos para "med.voll.api.medico", para mudar o pacote. Logo após, clicaremos no botão "Ok", no canto inferior direito.

Será criado o arquivo `DadosDetalhamentoMedico`:

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

public record DadosDetalhamentoMedico(Medico medico) {

} COPIAR CÓDIGO
```

Porém, teremos um `record` não recebendo o objeto médico e sim as informações do médico.

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


public record DadosDetalhamentoMedico(Long id, String nome, String
email, String crm, String telefone, Especialidade especialidade,
Endereco endereco) {


} COPIAR CÓDIGO
```

No método estamos instanciando um objeto médico, logo podemos criar um construtor que recebe um objeto do tipo `medico`. Esse construtor chama o construtor principal do `record` passando os parâmetros.

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


import med.voll.api.endereco.Endereco;


public record DadosDetalhamentoMedico(Long id, String nome, String
email, String crm, String telefone, Especialidade especialidade,
Endereco endereco) {
        public DadosDetalhamentoMedico (Medico medico) {
            this(medico.getId(), medico.getNome(), medico.getEmail(),
medico.getCrm(), medico.getTelefone(), medico.getEspecialidade(),
medico.getEndereco());


        }
} COPIAR CÓDIGO
```

Com isso, temos o `record` que recebe como parâmetro os dados do médico, e um construtor para passarmos o médico como parâmetro, que será usado no controller.

O método de cadastrar será um pouco diferente, porque existe o código `201` do protocolo HTTP que **significa que a requisição foi processada e o novo recurso foi criado**. Esse código possui um tratamento especial.

E usaremos o objeto `DadosDetalhamentoMedico`. Inclusive, podemos criar um novo *endpoint* (ou método) no controller, dado que temos somente os quatro métodos do CRUD (*Create*, *Read*, *Update* e *Delete*). Porém, faltou o método para detalhar.

Na listagem estamos devolvendo somente algumas informações dos médicos, mas e se quisermos detalhar trazendo todos os dados de um médico específico? Faltou esse método no controller.

No método de cadastrar, usaremos o código HTTP `201`, o mais adequado. E usaremos esse DTO, também.

Mas aprenderemos isso na sequência. Te espero na próxima aula!

04 Devolvendo o código HTTP 201

Transcrição

Na aula anterior, começamos a realizar as alterações no controller, nos métodos de excluir, atualizar e listar, para padronizar os retornos

para responder `ResponseEntity`. No entanto, faltou o método cadastrar.

O método de cadastrar possui um detalhe a mais devido ao protocolo HTTP, e faremos mais uma requisição para detalhar o médico.

Conforme o protocolo HTTP, ao cadastrarmos uma informação ou recurso em uma API, o código HTTP que deve ser devolvido, neste cenário, é o código `201` chamado *created*. Esse código significa que um registro foi criado na API.

Código 201: devolve no corpo da resposta os dados do novo recurso/registro criado e um cabeçalho do protocolo HTTP (Location).

Porém, esse código `201` possui algumas regras. Na resposta, devemos colocar o código `201` e no corpo da resposta os dados do novo registro criado e, também, um cabeçalho do protocolo HTTP.

Esse cabeçalho mostra o endereço para que o front-end, ou aplicativo mobile consiga acessar o recurso cadastrado. Logo, no cadastro não devolvemos apenas o código `200 OK` e nem apenas o `201`.

Precisa ser o código `201`, com os dados no formato JSON e um cabeçalho, na resposta. Para fazer isso, usaremos alguns recursos do Spring Boot.

```
//código omitido


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

```
//código omitidoCOPIAR CÓDIGO
```

No método de cadastrar do arquivo `MedicoController`, note que ele já aponta um erro de compilação, já que alteramos o retorno de `void` para `ResponseEntity`.

Após chamarmos o `repository.save`, como fazemos para acionar o código `201`? Incluiremos `return ResponseEntity` com o método `.created()`, que passaremos como parâmetro a `uri`: `return ResponseEntity.created(uri)`.

Essa `uri` representa o endereço, e o Spring cria o cabeçalho *location* de forma automática conforme a `uri` que passamos como parâmetro.

Ainda vamos criar essa URI.

```
//código omitido


@PostMapping

@Transactional

public ResponseEntity cadastrar(@RequestBody @Valid

DadosCadastroMedico dados) {

        repository.save(new Medico(dados));


                return ResponseEntity.created(uri)

}


//código omitidoCOPIAR CÓDIGO
```

Na sequência colocamos `.body()`, para incluir as informações que queremos devolver no corpo da resposta, como parâmetro colocamos `dto`. Assim, ele cria o objeto `ResponseEntity`.

```
//código omitido


@PostMapping
```

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

                return ResponseEntity.created(uri).body(dto);
}

//código omitidoCOPIAR CÓDIGO
```

Temos o erro de compilação em `uri` e `dto`, isso porque essas variáveis não existem neste método cadastrar.

Para criarmos o objeto `uri`, na linha de cima do `return`, vamos criar a variável, `var uri =`. **A `URI` deve ser o endereço da API**, no caso é o `http://localhost:8080/medicos/id`, sendo o ID do médico que acabou de ser criado no banco de dados.

Lembrando que está rodando local e ainda faremos o `deploy` para rodar no servidor. Logo, não será mais `http://localhost:8080`, será alterado.

Para não precisarmos ter que dar muita atenção para esse ponto no controller, o Spring possui uma classe que **encapsula o endereço da API**. Essa classe realiza a construção da URI de forma automática.

Para usarmos essa classe, incluiremos mais um parâmetro no método cadastrar.

Atualmente, estamos recebendo o `DadosCadastroMedico`.

Colocaremos uma vírgula (",") e, na sequência, um objeto do tipo `UriComponentsBuilder`, sendo a classe que gera a URI.

Chamaremos essa classe de `uriBuilder`.

```
//código omitido
```

```
@PostMapping
@Transactional
public ResponseEntity cadastrar(@RequestBody @Valid
DadosCadastroMedico dados, UriComponentsBuilder uriBuilder) {
        repository.save(new Medico(dados));

                var uri =

                return ResponseEntity.created(uri).body(dto);
}

//código omitidoCOPIAR CÓDIGO
```

Basta recebermos a classe como parâmetro no método controller, que o Spring fica responsável por passar esse parâmetro de forma automática.

Voltando para a variável `uri`, após o sinal de igual podemos colocar `uriBuilder.`, e chamaremos o método `path` para passarmos o complemento da URL: `var uri = uriBuilder.path()`. Isso porque ele cria somente a URI `localhost:8080`, e precisamos incluir o complemento `/medicos/id`.
Portanto, no parênteses do método `path` vamos passar `"/medicos/{id}"`. O `id` entre chaves é um parâmetro dinâmico.

```
//código omitido

@PostMapping
@Transactional
```

```
public ResponseEntity cadastrar(@RequestBody @Valid
DadosCadastroMedico dados, UriComponentsBuilder uriBuilder) {
        repository.save(new Medico(dados));

                var uri = uriBuilder.path("/medicos/{id}")

                return ResponseEntity.created(uri).body(dto);
}

//código omitido COPIAR CÓDIGO
```

Perceba que é semelhante ao que fizemos no método de excluir: `@DeleteMapping("/{id}")`. O Spring sabe que o `/{id}` é um parâmetro dinâmico.

Voltando ao método de cadastrar, na sequência do `path`, precisamos substituir esse ID pelo ID do médico que foi criado no banco de dados.

Para isso, digitamos `.buildAndExpand()`. Nele, precisamos passar, como parâmetro, o ID do médico criado no banco de dados. Esse ID está na linha anterior, no `repository.save` que chamamos para salvar no banco de dados.

```
repository.save(new Medico(dados)); COPIAR CÓDIGO
```

Contudo, passamos como parâmetro o `new Medico` para o método `save`. Vamos precisar desse médico na linha seguinte, por isso, criaremos uma variável para o médico: `var medico =`.

Em seguida, vamos extrair a linha que estamos passando como parâmetro para o método `save` e colaremos na variável `medico`.

```
var medico = new Medico(dados)

repository.save(); COPIAR CÓDIGO
```

No `repository.save()` passamos o `medico` como parâmetro, e no `buildAndExpand()` o `medico.getId()`. O ID será gerado pelo banco de dados na linha anterior de forma automática. Logo após o *build and expand*, colocamos `.toUri()` para criar o objeto URI.

```
//código omitido


@PostMapping

@Transactional

public ResponseEntity cadastrar(@RequestBody @Valid

DadosCadastroMedico dados, UriComponentsBuilder uriBuilder) {

                var medico = new Medico(dados)

                repository.save(medico);


                var uri =

uriBuilder.path("/medicos/{id}").buildAndExpand(medico.getId()).toUri(

);


                return ResponseEntity.created(uri).body(dto);

}


//código omitido COPIAR CÓDIGO
```

Criamos o objeto URI. Agora, no `return`, perceba que o parâmetro `dto` está na cor vermelha, significa que precisamos criá-lo. Usaremos o mesmo `dto` que utilizamos no método de atualizar, o `new DadosDetalhamentoMedico(medico)`.

Copiaremos esse método e colaremos no parâmetro do método `.body()`..

```
return ResponseEntity.created(uri).body(new
DadosDetalhamentoMedico(medico)); COPIAR CÓDIGO
```

Dessa forma, temos:

```
//código omitido

@PostMapping
@Transactional
public ResponseEntity cadastrar(@RequestBody @Valid
DadosCadastroMedico dados, UriComponentsBuilder uriBuilder) {
                var medico = new Medico(dados)
                repository.save(medico);

                var uri =
uriBuilder.path("/medicos/{id}").buildAndExpand(medico.getId()).toUri(
);

                return ResponseEntity.created(uri).body(new
DadosDetalhamentoMedico(medico));
```

```
}


//código omitido COPIAR CÓDIGO
```

O método de cadastrar possui diversos detalhes, porque precisamos devolver o código `201`, o cabeçalho *location* com a URI e no corpo da resposta é necessário ter uma representação do recurso recém criado.

Agora, vamos testar o cadastro, listagem, atualização e exclusão!

Alteramos para retornar o `ResponseEntity` nos métodos e salvamos essas modificações. Como o projeto já estava sendo executado, ele detecta as alterações de forma automática devido ao *DevTools*.

Voltando ao Insomnia, no método excluir médico, clicaremos no botão "Send" para disparar a requisição. Note que retornou o código `204 No Content`, deu certo.

Agora, clicamos do lado esquerdo do IntelliJ em "`PUT` atualizar médido". Neste método, alteraremos no corpo do JSON o ID de "2" para "1", porque desejamos atualizar o médico que contém o ID 1 e o telefone:

```
PUT: http://localhost:8080/medicos

{
    "id": 1,
    "telefone": "2111112222"
} COPIAR CÓDIGO
```

Logo após, clicamos no botão "Send". Antes nos devolvia o código `200 OK`, sem um corpo no retorno. Agora, temos o código `200 OK`, mas com o seguinte corpo na resposta:

Preview

```
{
        "id": 1,
        "nome": "Rodrigo Ferreira",
        "email": "rodrigo.ferreira@voll.med",
        "crm": "123456",
        "telefone": "2111112222",
        "especialidade": "ORTOPEDIA",
        "endereco": {
            "logradouro": "rua 1",
            "bairro": "bairro",
            "cep": "12345678",
            "numero": "1",
            "complemento": null,
            "cidade": "Brasil",
            "uf": "DF"
        }
} COPIAR CÓDIGO
```

Esses são os dados atualizados do médico. Perceba que retorna **todos** os dados, não somente o `id` e o `telefone` - que foram os que alteramos.

Do lado esquerdo do IntelliJ, selecionaremos "`GET` Listagem de médicos" e depois o botão "Send". Perceba que este método não

mudou, continua devolvendo o código `200 OK` com JSON no corpo da resposta com os dados da paginação.

Ou seja, alteramos para `ResponseEntity` mas não alterou nada. Isso era esperado.

Vamos verificar se foi feita a alteração no cadastro. Para isso, clicaremos em "`POST` Cadastro de Médico" e no JSON, alteraremos os campos com as informações do "Renato" para "Juliana Queiroz".

Antes:

```
{
        "nome": "Renato Amoedo",
        "email": "renato.amoedo@voll.med",
        "crm": "233444",
        "telefone": "61999998888",
        "especialidade": "ORTOPEDIA",
        "endereco": {
            "logradouro": "rua 1",
            "bairro": "bairro",
            "cep": "12345678",
            "complemento": null,
            "cidade": "Brasil",
            "uf": "DF"
        }
} COPIAR CÓDIGO
```

Depois

```
{
        "nome": "Juliana Queiroz",
```

```
        "email": "juliana.queiroz@voll.med",
        "crm": "233444",
        "telefone": "61999998888",
        "especialidade": "ORTOPEDIA",
        "endereco": {
            "logradouro": "rua 1",
            "bairro": "bairro",
            "cep": "12345678",
            "complemento": null,
            "cidade": "Brasil",
            "uf": "DF"
        }
} COPIAR CÓDIGO
```

Após essas alterações, clicamos no botão "Send". Antes o código estava `500 Internal Server Error`, agora está como `201 Created`, com as seguintes informações no corpo da resposta em formato JSON:

```
{
        "id": 6,
        "nome": "Juliana Queiroz",
        "email": "juliana.queiroz@voll.med",
        "crm": "233444",
        "telefone": "2111112222",
        "especialidade": "ORTOPEDIA",
        "endereco": {
            "logradouro": "rua 1",
            "bairro": "bairro",
            "cep": "12345678",
```

```
            "numero": null,
            "complemento": null,
            "cidade": "Brasil",
            "uf": "DF"
        }
} COPIAR CÓDIGO
```

Do lado direito da aba "*Preview*", temos a aba "*Header*", sendo os cabeçalhos que foram devolvidos. Ao clicarmos na aba "Header", temos:

Name	Value
Location	http://localhost:8080/medicos/6
Content-Type	application/json
Transfer-Encoding	chunked
Date	Thu, 20 Oct 2022 19:49:00 GMT

Dessa forma, temos o cabeçalho *location* com o endereço `http://localhost:8080/medicos/6`, sendo o número 6 o ID do médico que acabamos de cadastrar no banco de dados.

Se tentarmos entrar no endereço `/medicos/6`, será devolvido o código `404`. Isso porque **não configuramos uma requisição para detalhar os dados de um médico**.

Por enquanto, temos somente o CRUD (Create, Read, Update e Delete), mas em uma API geralmente temos uma quinta funcionalidade para detalhar as informações de um médico.

Na próxima aula, vamos aprender a criar essa funcionalidade.

Te espero no próximo vídeo!

05 Header Location

Você se depara com o seguinte método, em uma classe Controller:

```
@PostMapping
@Transactional
public ResponseEntity cadastrar(@RequestBody @Valid
DadosCadastroProduto dados) {
    var produto = new Produto(dados);
    repository.save(produto);

    var uri = new URI("/produtos/{id}");

    return ResponseEntity.ok(new DadosDetalhamentoProduto(produto));
}
```

COPIAR CÓDIGO

Levando em consideração as boas práticas de retorno em uma requisição HTTP do tipo **POST**, escolha as alternativas que indicam os problemas do código anterior:

- Não será devolvida uma representação do recurso criado na API.
- Alternativa correta

  O código anterior não devolverá o código 201 como resposta.

  Será devolvido o código 200.
- Alternativa correta

  O cabeçalho *Location* será criado de maneira incorreta.

  No código anterior, não foi utilizada a classe `UriComponentsBuilder`, do Spring, para a criação da URI; em vez disso, a URI foi criada manualmente e de maneira incorreta. Além disso, a URI nem foi adicionada na resposta a ser devolvida pela API.
- Alternativa correta

  A validação das informações não será executada.

Parabéns, você acertou!

06 Para saber mais: códigos do protocolo HTTP

O **protocolo HTTP** (*Hypertext Transfer Protocol*, RFC 2616) é o protocolo responsável por fazer a comunicação entre o cliente, que normalmente é um *browser*, e o servidor. Dessa forma, a cada "requisição" feita pelo cliente, o servidor responde se ele obteve sucesso ou não. Se não obtiver sucesso, na maioria das vezes, a resposta do servidor será uma sequência numérica acompanhada por uma mensagem. Se não soubermos o que significa o código de resposta, dificilmente saberemos qual o problema que está acontecendo, por esse motivo é muito importante saber quais são os códigos HTTP e o que significam.

Categoria de códigos

Os códigos HTTP (ou HTTPS) possuem três dígitos, sendo que o primeiro dígito significa a classificação dentro das possíveis cinco categorias.

1XX: *Informativo* – a solicitação foi aceita ou o processo continua em andamento;
2XX: *Confirmação* – a ação foi concluída ou entendida;
3XX: *Redirecionamento* – indica que algo mais precisa ser feito ou precisou ser feito para completar a solicitação;
4XX: *Erro do cliente* – indica que a solicitação não pode ser concluída ou contém a sintaxe incorreta;
5XX: *Erro no servidor* – o servidor falhou ao concluir a solicitação.

Principais códigos de erro

Como dito anteriormente, conhecer os principais códigos de erro HTTP vai te ajudar a identificar problemas em suas aplicações, além de permitir que você entenda melhor a comunicação do seu navegador com o servidor da aplicação que está tentando acessar.

Error 403

O código 403 é o erro “Proibido”. Significa que o servidor entendeu a requisição do cliente, mas se recusa a processá-la, pois o cliente não possui autorização para isso.

Error 404

Quando você digita uma URL e recebe a mensagem *Error 404*, significa que essa URL não te levou a lugar nenhum. Pode ser que a aplicação não exista mais, a URL mudou ou você digitou a URL errada.

Error 500

É um erro menos comum, mas de vez em quando ele aparece. Esse erro significa que há um problema com alguma das bases que faz uma aplicação rodar. Esse erro pode ser, basicamente, no servidor que mantém a aplicação no ar ou na comunicação com o sistema de arquivos, que fornece a infraestrutura para a aplicação.

Error 503

O erro 503 significa que o serviço acessado está temporariamente indisponível. Causas comuns são um servidor em manutenção ou sobrecarregado. Ataques maliciosos, como o DDoS, causam bastante esse problema.

Uma dica final: dificilmente conseguimos guardar em nossa cabeça o que cada código significa, portanto, existem sites na internet que possuem todos os códigos e os significados para que possamos consultar quando necessário. Existem dois sites bem conhecidos e utilizados por pessoas desenvolvedoras, um para cada preferência: se você gosta de gatos, pode utilizar o HTTP Cats; já, se prefere cachorros, utilize o HTTP Dogs.

07 Detalhando dados na API

Transcrição

Padronizamos os retornos para `ResponseEntity` no controller, e para cada operação do CRUD devolver o código HTTP adequado.

Header do método Cadastro de médico:

Name	Value
Location	http://localhost:8080/medicos/6
Content-Type	application/json
Transfer-Encoding	chunked
Date	Thu, 20 Oct 2022 19:49:00 GMT

Temos o CRUD e ficamos com a funcionalidade de detalhes de um médico pendente.

O cabeçalho *location*, ao cadastrarmos um médico, ele nos devolve o endereço para acessarmos o recurso criado na API, sendo o `http://localhost:8080/medicos/6`.

No entanto, se criarmos uma nova requisição no Insomnia disparando para esse endereço, vai retornar erro. Vamos simular isso no Insomnia.

Para isso, do lado esquerdo, criaremos mais uma requisição clicando no botão "+". Será exibido uma caixa com as seguintes opções:

- HTTP Request (Ctrl + N)
- Graph Request
- gRPC Request
- New Folder (Ctrl + Shift + N)

Clicaremos na primeira opção "HTTP Request". Perceba que do lado esquerdo foi gerado uma nova aba "`GET` New Request". Como estamos trazendo registros da API, é uma requisição do tipo `GET`.
Na aba "`GET` New Request",
digitaremos `"http://localhost:8080/medicos/1"` no campo de endereço, sendo o número 1 o ID.

```
http://localhost:8080/medicos/1COPIAR CÓDIGO
```

Se dispararmos essa requisição, por não termos mapeado esse endereço no controller, vai gerar algum problema. Isto é, não temos `/medicos/1` mapeado como requisição do tipo `get`.

Clicando no botão "Send" para testar, perceba que o código devolvido é o `405 Method Not Allowed`. Isso significa que essa URL pode existir, está mapeada no controller.

Voltando ao arquivo `MedicoController`, descendo a página do código, conseguimos visualizar que está mapeado para o método de excluir: `@DeleteMapping("/{id}")`.

```
// código omitido


**@DeleteMapping("/{id}")**
@Transactional
    public ResponseEntity excluir(@PathVariable Long id) {
        var medico = repository.getReferenceById(id);
        medico.excluir();


        return ResponseEntity.noContent().build();
    }COPIAR CÓDIGO
```

Por isso, no Insomnia, devolveu o código `405` e não `404`, justamente porque o endereço está mapeado. Porém, para a requisição do tipo `delete`, disparamos a requisição do tipo `get`.

Esse código `405` nos informa que ele aceita somente a requisição do tipo `delete` e não `get` para esse disparo.

Agora, implementaremos a funcionalidade de detalhar os registros dos médicos. Será esse endereço, porém, o tipo da requisição será `get` e não `delete`.

Voltando ao IntelliJ, antes do último fechamento de chaves do arquivo `MedicoController` criaremos o nosso método de detalhar. Podemos até copiar o método `delete` e colar mais para baixo para fazer os ajustes, já que será bem semelhante.

Método `delete` para fazermos os ajustes.

```
//código omitido



@DeleteMapping("/{id}")
@Transactional
public ResponseEntity excluir(@PathVariable Long id) {
        var medico = repository.getReferenceById(id);
        medico.excluir();

        return ResponseEntity.noContent().build();
    }COPIAR CÓDIGO
```

Ao invés de `@DeleteMapping` usaremos `@GetMapping` com o complemento `/{id}`. Vamos remover o `@Transactional`, porque é um método de leitura, e alteramos o nome do método para `detalhar`.

```
//código omitido


@GetMapping("/{id}")
public ResponseEntity detalhar(@PathVariable Long id) {
        var medico = repository.getReferenceById(id);
        medico.excluir();

        return ResponseEntity.noContent().build();
    }COPIAR CÓDIGO
```

Na implementação, precisamos carregar o médico do banco de dados e, por isso, a linha com a variável `medico` permanece e a linha `medico.excluir()` podemos remover.

```
//código omitido


@GetMapping("/{id}")
```

```
public ResponseEntity detalhar(@PathVariable Long id) {
        var medico = repository.getReferenceById(id);

        return ResponseEntity.noContent().build();
    }COPIAR CÓDIGO
```

Por fim, no `return` precisamos devolver um conteúdo. Em razão disso, alteramos de `.noContent()` para `.ok()` e removeremos o `.build()`.

```
//código omitido

@GetMapping("/{id}")
public ResponseEntity detalhar(@PathVariable Long id) {
        var medico = repository.getReferenceById(id);
        return ResponseEntity.ok();
    }COPIAR CÓDIGO
```

Como parâmetro do método `ok()`, criamos um DTO, sendo o mesmo de detalhamento: `new DadosDetalhamentoMedico()`.
Em `DadosDetalhamentoMedico()` vamos passar como parâmetro o médico que acabamos de carregar no banco de dados.

```
//código omitido

@GetMapping("/{id}")
public ResponseEntity detalhar(@PathVariable Long id) {
        var medico = repository.getReferenceById(id);
        return ResponseEntity.ok(new DadosDetalhamentoMedico(medico));
    }COPIAR CÓDIGO
```

Já tínhamos tudo implementado, até o DTO `DadosDetalhamentoMedico`. Logo, tivemos somente que usar o mesmo DTO utilizado no cadastro e na atualização.

Note que, eventualmente, podemos reaproveitar um DTO, podendo ser usado em diversos métodos no controller.

Salvaremos essas alterações e voltaremos ao Insomnia. Podemos alterar o nome do arquivo de "*New Request*" para "Detalhar Médico". Basta clicar duas vezes em cima do nome da pasta que deseja alterar. Em seguida, clicamos no botão "Send" para enviar uma requisição para detalhar o médico. O código devolvido foi o `200 OK`, com os seguintes campos no corpo da resposta:

```
{
        "id": 1,
        "nome": "Rodrigo Ferreira",
        "email": "rodrigo.ferreira@voll.med",
        "crm": "123456",
        "telefone": "2111112222",
        "especialidade": "ORTOPEDIA",
        "endereco": {
            "logradouro": "rua 1",
            "bairro": "bairro",
            "cep": "12345678",
            "numero": "1",
            "complemento": null,
            "cidade": "Brasil",
            "uf": "DF"
        }
}COPIAR CÓDIGO
```

No caso, se trata sobre os dados do médico que corresponde ao ID número 1.

No endereço, vamos alterar o ID de número 1, para o número 6 após a última barra "/":

```
http://localhost:8080/medicos/6COPIAR CÓDIGO
```

Logo após alterarmos, clicaremos no botão "Send". Temos como retorno o código `200 OK` e no corpo da resposta as informações sobre a Juliana.

```
{
                "id": 6,
        "nome": "Juliana Queiroz",
        "email": "juliana.queiroz@voll.med",
        "crm": "233444",
        "telefone": "61999998888",
        "especialidade": "ORTOPEDIA",
        "endereco": {
            "logradouro": "rua 1",
            "bairro": "bairro",
            "cep": null,
            "complemento": null,
            "cidade": "Brasil",
            "uf": "DF"
        }
}COPIAR CÓDIGO
```

A funcionalidade de detalhar está funcionando!

Podemos aplicar a mesma lógica para a funcionalidade de pacientes. Inclusive, é uma atividade que vamos deixar para vocês.

Voltaremos ao IntelliJ, no arquivo do controller. Conseguimos fazer as alterações necessárias para a melhoria do nosso código, todos os métodos estão padronizados retornando o `ResponseEntity`.
Além disso, em cada método devolvemos o código HTTP mais adequado para aquela determinada situação. Por exemplo, o código `201` para o cadastro, o código `200` para o método de listagem, atualizar e detalhar e `204` para o método de excluir.

Em cada um desses códigos, podemos ter um corpo da resposta e cabeçalhos do protocolo HTTP.

Deixaremos um Para saber mais: códigos do protocolo HTTP, para você ler com mais detalhes sobre esses códigos HTTP.

Assim, conseguimos aplicar a primeira melhoria no código do nosso projeto. Na sequência, faremos outras melhorias.

Te espero no próximo capítulo!

08 Faça como eu fiz: ResponseEntity

Agora é com você! Faça o mesmo procedimento que eu fiz na aula, implementando o `ResponseEntity`, porém, para as funcionalidades do **CRUD de pacientes**.

-
-

Você precisará alterar todos os métodos da classe `PacienteController` para que eles retornem o

objeto `ResponseEntity`, da mesma forma que foi demonstrado na aula para a classe `MedicoController`:

```
@PostMapping
@Transactional
public ResponseEntity cadastrar(@RequestBody @Valid
DadosCadastroPaciente dados, UriComponentsBuilder uriBuilder) {
    var paciente = new Paciente(dados);
    repository.save(paciente);

    var uri =
uriBuilder.path("/pacientes/{id}").buildAndExpand(paciente.getId()).to
Uri();
    return ResponseEntity.created(uri).body(new
DadosDetalhamentoPaciente(paciente));
}

@GetMapping
public ResponseEntity<Page<DadosListagemPaciente>>
listar(@PageableDefault(size = 10, sort = {"nome"}) Pageable
paginacao) {
    var page =
repository.findAllByAtivoTrue(paginacao).map(DadosListagemPaciente::ne
w);
    return ResponseEntity.ok(page);
}

@PutMapping
@Transactional
```

```
public ResponseEntity atualizar(@RequestBody @Valid
DadosAtualizacaoPaciente dados) {
    var paciente = repository.getReferenceById(dados.id());
    paciente.atualizarInformacoes(dados);

    return ResponseEntity.ok(new DadosDetalhamentoPaciente(paciente));
}

@DeleteMapping("/{id}")
@Transactional
public ResponseEntity excluir(@PathVariable Long id) {
    var paciente = repository.getReferenceById(id);
    paciente.excluir();

    return ResponseEntity.noContent().build();
}COPIAR CÓDIGO
```

Além disso, você deverá criar mais um método nesse controller, que será responsável por devolver os detalhes de um paciente:

```
@GetMapping("/{id}")
public ResponseEntity detalhar(@PathVariable Long id) {
    var paciente = repository.getReferenceById(id);
    return ResponseEntity.ok(new DadosDetalhamentoPaciente(paciente));
}COPIAR CÓDIGO
```

E também precisará criar o DTO `DadosDetalhamentoPaciente`:

```
public record DadosDetalhamentoPaciente(String nome, String email,
String telefone, String cpf, Endereco endereco) {
    public DadosDetalhamentoPaciente(Paciente paciente) {
```

```
        this(paciente.getNome(), paciente.getEmail(),
paciente.getTelefone(), paciente.getCpf(), paciente.getEndereco());
    }
}
```

09 O que aprendemos?

Nessa aula, você aprendeu como:

- Utilizar a classe `ResponseEntity`, do Spring, para personalizar os retornos dos métodos de uma classe Controller;
- Modificar o código HTTP devolvido nas respostas da API;
- Adicionar cabeçalhos nas respostas da API;
- Utilizar os códigos HTTP mais apropriados para cada operação realizada na API.