01 Enviando dados para a API

Transcrição

Com a aplicação criada, vamos desenvolver as funcionalidades do projeto. Quais as funcionalidades do projeto? O que vamos precisar implementar? Quais as validações e regras de negócio?
Para facilitar a descrição dessas funcionalidades, criamos um quadro no Trello , com cada cartão contendo a definição de uma funcionalidade.

À esquerda, na coluna "To do", temos o primeiro cartão chamado "Cadastro de médicos". Vamos arrastá-la para a coluna "Doing", para mostrar que essa é a funcionalidade que estamos fazendo no momento. Ao finalizarmos, podemos arrastar para a coluna "Done" ("Feito").

Clicando no cartão "Cadastro de médicos", é aberta uma pop-up com a descrição da funcionalidade:

Funcionalidade "Cadastro de médicos":

```
O sistema deve possuir uma funcionalidade de cadastro de médicos, na
qual as seguintes informações deverão ser preenchidas:


 Nome
 E-mail
 Telefone
 CRM
 Especialidade (Ortopedia, Cardiologia, Ginecologia ou Dermatologia)
```

```
 Endereço completo (logradouro, número, complemento, bairro, cidade,
UF e CEP)


Todas as informações são de preenchimento **obrigatório**, exceto o
número e o complemento do endereço.COPIAR CÓDIGO
```

Temos os campos obrigatórios e a regra de negócio que devem ser aplicados no projeto. Para facilitar o entendimento, dado que vamos desenvolver somente API Back-end (não teremos interface gráfica), vamos disponibilizar os protótipos das telas do aplicativo no Figma.

Layout das telas no Figma

Nesta página, temos os protótipos das telas do nosso aplicativo mobile desta aplicação.

Agora, vamos focar na funcionalidade de cadastrar médicos.

As pessoas que trabalham nessa clínica terão esse aplicativo instalado no celular. Esse aplicativo vai ter um menu para acessar a página para

cadastrar um novo médico, com os campos descritos no cartão do Trello.

Ao clicarmos no botão "Concluir cadastro", o aplicativo mobile irá enviar uma requisição para a nossa API. Nela, receberemos, validaremos e salvaremos essas informações em um banco de dados.

Com esse layout, conseguimos visualizar melhor o funcionamento, já que não implementaremos a parte das telas. Com as funcionalidades descritas e os layouts nos auxiliando, vamos iniciar a implementação pela parte de disparar as requisições.

Como não temos um aplicativo mobile e nem aplicação Front-end, como testaremos a API? Como vamos enviar as requisições?

Usaremos uma ferramenta de testes de API, as duas mais usadas são:

- Postman
- Insomnia

Neste curso, utilizaremos a ferramenta Insomnia.

Abrindo o Insomnia, na tela inicial temos um menu no canto superior esquerdo as opções: *Application*, *Edit*, *View*, *Window*, *Tools* e *Help*. Abaixo, temos o símbolo da ferramenta Insomnia com uma seta para expandir, em que mais à direita dela temos um ícone de engrenagem e outro com a silhueta do rosto de uma pessoa.

Mais abaixo, temos a seção "Dashboard", em que temos um campo de busca à direita e um botão "Create ▾".

Criaremos um projeto para configurarmos as requisições e deixarmos tudo agrupado, de forma mais simples. Para isso, na parte superior esquerda da página, clicaremos em "Insomnia ▾" e depois na opção "*Create new project*" ("+ Criar novo projeto").
Será aberta uma pop-up com o título "Create New Project" com um campo para digitarmos o nome do projeto abaixo e um botão "*Create*", no canto inferior direito. O nome do nosso projeto será "API Voll.med", e após digitar o nome no campo podemos selecionar o botão "Create".

Seremos redirecionados para a página do Dashboard, mas perceba que no lugar que estava escrito "Insomnia" agora está com o nome "API Voll.med", no canto superior esquerdo.

Agora, no canto superior direito vamos clicar no botão "Create ▾". Serão exibidas duas seções, sendo elas `New` e `Import from`. Na primeira, temos as opções "*Design Document*" e "*Request Collection*", já na segunda temos: "*File*", "URL" e "*Clipboard*". Vamos clicar na opção "Request Collection", para criarmos uma coleção de requisições que enviaremos para a API.
Será aberta uma pop-up com o título "*Create New Request Collection*" ("Criar nova coleção de solicitações") com um campo para digitarmos o nome da *request collection* abaixo e um botão "*Create*", no canto inferior direito. Chamaremos de "Requisições" e depois clicaremos no botão "Create".
Agora, sim, estamos na tela principal do Insomnia. Nela, temos à esquerda, a seção "*No Environment*" ("Sem ambiente") com um campo para filtrarmos informações abaixo e à esquerda deste campo temos um botão com um símbolo de mais e uma seta apontando para baixo (para expandir). E a direita temos as opções com os seguintes atalhos:

- *New Request* ("Novo pedido"): (Ctrl + N)
- *Switch Requests* ("Solicitações de troca"): (Ctrl + P)
- *Edit Environments* ("Editar ambientes"): (Ctrl + E)

E abaixo, mais dois botões:

- *Import from File* ("Importar do arquivo")
- *New Request* ("Novo pedido")

Com isso, conseguimos simular uma requisição para a API.

À esquerda, clicaremos no botão com o ícone de mais, à direita do campo de filtrar. As alternativas exibidas são:

- HTTP Request
- Graph Request
- gRPC Request
- New Folder

Selecionaremos "HTTP Request" ou podemos usar o atalho "Ctrl + N". Agora, à esquerda do Insomnia, temos "New Request" e à direita, na parte superior do painel central, um campo para selecionarmos o método e incluirmos um endereço com um botão "Send". Abaixo, temos as opções: *Body*, *Auth*, *Query*, *Header* e *Docs*.

Dessa forma, conseguimos passar as informações da requisição que desejamos enviar para a API.

Do lado esquerdo, vamos renomear de "New Request" para "Cadastro de Médico", dando dois cliques e digitando o nome desejado. No

painel central, precisamos escolher qual o método que vamos disparar essa requisição.

Por ser um cadastro, não é um método `get` - usado para leitura para receber dados da API). O que desejamos fazer é ao contrário, queremos enviar dados para API. Por isso, vamos clicar em "Get ▼" e no menu que será expandido, escolheremos o método `POST`.
Do lado direito do método, precisamos inserir a URL da API. Digitaremos `http://localhost:8080/`, e precisamos ter alguma URL mapeada no projeto. Como estamos trabalhando com as funcionalidades de cadastro de médicos, colocaremos /medicos.

```
http://localhost:8080/medicosCOPIAR CÓDIGO
```

Assim, temos uma requisição do tipo `post` para a URL `http://localhost:8080/medicos`. Mas falta um detalhe, nesta requisição precisamos enviar os dados para API. Onde passamos esses dados no Insomnia?

Abaixo do método post, temos uma aba chamada "Body ▼", sendo justamente para passarmos o corpo da requisição. Clicando em "Body ▼", escolheremos a opção "JSON". Normalmente, em API Rest, os dados são usados no formato JSON.

Note que foi aberto um campo para digitarmos esses dados, em que digitaremos as seguintes informações:

```
{
"nome": "Rodrigo Ferreira",
"email": "rodrigo.ferreira@voll.med",
"crm": "123456",
"especialidade": "ortopedia",
```

```
"endereco": {
    "logradouro": "rua 1",
    "bairro": "bairro",
    "cep": "12345678",
    "cidade": "Brasilia",
    "uf": "DF",
    "numero": "1",
    "complemento": "complemento"
    }
}COPIAR CÓDIGO
```

É um JSON que representa os dados da funcionalidade de cadastro de médicos. Note que são os mesmos campos do nosso protótipo e do *card* do Trello. Com a requisição montada, podemos clicar no botão "Send".

Após selecionar o botão para enviar a requisição, à direita será exibido o erro `404 Not Found` com os seguintes dados na aba "Preview":

```
{
    "timestamp": "2022-10-13T00:43:01.850+00:00"
    "status": 404,
    "error": "Not Found",
    "message": "No message available",
    "path": "/medicos"
}COPIAR CÓDIGO
```

Isso aconteceu porque estamos enviando uma requisição para o endereço "/medicos" que não está mapeado no controller. Lembrando que temos no projeto somente o "/hello".

Agora que entendemos qual o processo para enviar as requisições para a nossa API usando o Insomnia, podemos começar a escrever os códigos e implementar a funcionalidade no Back-end.

Mas vamos aprender isso na sequência. Até a próxima aula!

02 Preparando o ambiente: Trello e Figma

Durante o curso eu utilizarei um quadro do Trello contendo cartões que descrevem cada uma das funcionalidades da aplicação. Você pode acessar esse quadro neste link:

- Trello - Curso de Spring Boot

Além disso, também vou exibir o layout mobile da aplicação, que pode ser acessado neste link da ferramenta Figma:

- Layout mobile da aplicação Voll.med

03 Recebendo dados na API

Transcrição

Na aula anterior aprendemos a enviar requisições pelo Insomnia, mas está retornando erro 404. Isso porque ainda não implementamos o endereço `/medicos` no Back-end.

Neste vídeo, vamos aprender como implementar essa funcionalidade usando o Spring Boot.

Voltando ao IntelliJ, precisamos mapear a URL `/medicos` no projeto. Para isso, é necessário criarmos uma classe controller, seguindo o padrão do Spring MVC, o controller é o arquivo que mapeamos as requisições enviadas para nossa API.
Já temos o arquivo `HelloController`, em que estamos tratando a funcionalidade de médicos e, por isso, vamos gerar outro controller. Para esse objetivo, à esquerda selecionaremos o pacote `Controller` e usaremos o atalho "Alt + Insert". Será exibido um menu, onde vamos escolher a opção "Java Class", no pop-up seguinte com o título "New Java Class" digitaremos como nome "MedicoController".

MedicoController:

```
package med.voll.api.controller;


public class MedicoController {

}COPIAR CÓDIGO
```

No momento é uma classe Java que não está associada ao Spring, ele não vai carregar essa classe no projeto. Precisamos incluir a anotação,

```
package med.voll.api.controller;


import org.springframework.web.bind.annotation.RestController;


@RestController
public class MedicoController {


}COPIAR CÓDIGO
```

Com isso, estamos comunicando o Spring que esta é uma classe *Restcontroller* que precisa ser carregada durante a inicialização

do projeto. Desse modo, o Spring irá carregar a classe `MedicoController` toda vez que o projeto for inicializado.
Além dessa anotação, incluiremos a `@RequestMapping("")`, passando a URL deste controller, no caso `("medicos")`, `@RequestMapping("medicos")`.

```
package med.voll.api.controller;


import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;


@RestController

@RequestMapping("medicos")

public class MedicoController {


}COPIAR CÓDIGO
```

Assim, o Spring sabe que esta é uma classe controller devido à anotação `@RestController`, que está mapeando a URL `/medicos`. Isto é, ao chegar uma requisição para `/medicos` o Spring vai detectar que deverá chamar o `MedicoController`.
Dentro do controller precisamos ter métodos que representam as funcionalidades. Vamos declarar um método chamado `cadastrar()` para a funcionalidade de cadastro de médicos, com o retorno `void` (significa que não teremos retorno).

```
//código omitido


public void cadastrar() {

    }


//código omitidoCOPIAR CÓDIGO
```

Por enquanto está vazio. Acima do método, é necessário especificarmos o verbo do protocolo HTTP que ele vai lidar. No caso, estamos enviando as requisições via verbo `post`, por isso, incluiremos a anotação `@PostMapping`.

```
package med.voll.api.controller;


import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;


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


        @PostMapping
         public void cadastrar() {
    }


}COPIAR CÓDIGO
```

Estamos comunicando o Spring que ao chegar uma requisição do tipo `post` para a URL `/medicos`, ele deve chamar o método `cadastrar` da classe `MedicoController`. É isso que acabamos de mapear. Podemos salvar o arquivo clicando em "Ctrl + S". Voltando ao Insomnia, estamos usando o método `POST` com a URL `http://localhost:8080/medicos`. Em "Body", temos:

```
{
"nome": "Rodrigo Ferreira",
"email": "rodrigo.ferreira@voll.med",
"crm": "123456",
```

```
"especialidade": "ortopedia",
"endereco": {
    "logradouro": "rua 1",
    "bairro": "bairro",
    "cep": "12345678",
    "cidade": "Brasilia",
    "uf": "DF",
    "numero": "1",
    "complemento": "complemento"
    }
}COPIAR CÓDIGO
```

E na aula anterior, tivemos um erro `404 Not Found`, em que à direita, em "Preview", foi exibido:

```
{
    "timestamp": "2022-10-13T00:43:01.850+00:00"
    "status": 404,
    "error": "Not Found",
    "message": "No message available",
    "path": "/medicos"
}COPIAR CÓDIGO
```

Após lembrarmos disso, podemos disparar a requisição clicando no botão "Send", à direita da URL. Perceba que agora retornou `200 OK` e, em "Preview", temos a mensagem: "*No body returned for response*" (em português, "Nenhum corpo retornado para resposta"). Isso significa que deu certo e que a requisição foi processada com sucesso!

Porém, nesta requisição, estamos levando um JSON no corpo. Como recebemos esse JSON no método `cadastrar` do `MedicoController`?

Voltando para o IntelliJ, no método `cadastrar` do arquivo `MedicoController`, um método no Java recebe parâmetros e o nosso está vazio.
Vamos inserir no parêntese do método `cadastrar` o`String json` e dentro dele, colocaremos `System.out.println()`, passando o `json` como parâmetro, para verificar se está chegando da forma correta.

```
package med.voll.api.controller;


import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;


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


        @PostMapping
         public void cadastrar(String json) {
              System.out.println(json);
    }


}COPIAR CÓDIGO
```

Podemos salvar clicando em "Ctrl + S" e na parte inferior direita do IntelliJ, selecionaremos a aba "Run". Nela, perceba que a reinicialização automática está sendo feita devido ao DevTools.

Agora, voltaremos à Insomnia para enviar a requisição, clicando no botão "Send". À direita, continuamos com o status `200 OK` e a mensagem "*No body returned for responde*".
Vamos verificar se no `system.out.println()` vai chegar o JSON que inserimos no Insomnia, enviado no controller. Para isso, voltaremos ao terminal do IntelliJ.
Na aba "Run", note que ele exibiu `null`. Isso significa que não é bem assim que imprimimos os dados. Logo, como recebemos os dados da requisição?
É dessa forma que fizemos, mas faltou informarmos ao Spring que o parâmetro `string json` do método `cadastrar` é para pegar do corpo da requisição.
Para informarmos isso para o Spring, incluiremos a anotação `@RequestBody` neste parâmetro.

```
package med.voll.api.controller;


import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestBody;


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


        @PostMapping
         public void cadastrar(@RequestBody String json) {
              System.out.println(json);
    }
```

```
}COPIAR CÓDIGO
```

Agora o Spring sabe que o parâmetro JSON do método `cadastrar` é para ele puxar do corpo da requisição. Salvaremos essa alteração feita no arquivo clicando em "Ctrl + S".

Logo após, voltaremos ao Insomnia e selecionaremos o botão "Send". Seguimos com o status `200 OK` e a mensagem "*No body returned for responde*".

Na aba "Run" do IntelliJ, vamos verificar se o `system.out.println()` vai exibir o JSON de forma correta:

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

Nos retornou exatamente o JSON que incluímos no corpo da requisição. Assim, as informações chegaram para o método `cadastrar` do controller da forma esperada.

Essa é uma das maneiras de recebermos dados nos métodos dos controllers: declarando como string e anotando como request body, no caso de requisições do tipo `post`.
Porém, essa não é a melhor forma, porque estamos recebendo esses campos **como uma string literal**. Por exemplo, se quisermos imprimir o CEP 12345678, teríamos que fazer um `parse` em cima dessa string para chegar na string do CEP. Seria um processo bem trabalhoso.
Será que conseguimos receber uma `string cep` e renomearmos de `json` para `cep` no `system out`?

```
        @PostMapping
         public void cadastrar(@RequestBody String cep) {
              System.out.println(cep);
    }COPIAR CÓDIGO
```

Com essas alterações, será que o Spring consegue entender que dentro do JSON há um campo chamado `cep` e que por ele estar recebendo somente este campo, será enviado somente o `12345678`?

Para testar, clicaremos em "Ctrl + S" para salvar. Depois, vamos clicar na aba "Run", na parte inferior do IntelliJ. Perceba que já reiniciou, podemos ir no Insomnia e selecionar o botão "Send", para enviar a requisição.

Voltando novamente para a aba "Run" do IntelliJ, retornou exatamente o JSON completo novamente:

```
{
"nome":"Rodrigo Ferreira",
"email":"rodrigo.ferreira@voll.med",
"crm":"123456",
"especialidade":"ortopedia",
```

```
"endereco":{
    "logradouro":"rua 1",
    "bairro":"bairro",
    "cep:"12345678",
    "cidade":"Brasilia,
    "uf":"DF",
    "numero":"1",
    "complemento":"complemento",
    }
}COPIAR CÓDIGO
```

Com isso, percebemos que ao receber uma string no parâmetro do método `cadastrar` do controller e anotá-la com `@RequestBody`, o Spring exibirá o corpo todo do JSON e passará para essa string.

Se quisermos receber os campos separados, não podemos receber como string. Será necessário criarmos uma classe, e nela declarar os atributos com os mesmos nomes que estão sendo recebidos pelo JSON.

Faremos isso, pois queremos trabalhar com cada campo de forma separada.

Vamos aprender como aplicar isso no próximo vídeo, usando recursos do Java. Te espero lá!

04 Para saber mais: JSON

JSON (*JavaScript Object Notation*) é um formato utilizado para representação de informações, assim como XML e CSV.

Uma API precisa receber e devolver informações em algum formato, que representa os recursos gerenciados por ela. O JSON é um desses formatos possíveis, tendo se popularizado devido a sua leveza, simplicidade, facilidade de leitura por pessoas e máquinas, bem como seu suporte pelas diversas linguagens de programação.

Um exemplo de representação de uma informação no formato XML seria:

```
<produto>
    <nome>Mochila</nome>
    <preco>89.90</preco>
    <descricao>Mochila para notebooks de até 17 polegadas</descricao>
</produto> COPIAR CÓDIGO
```

Já a mesma informação poderia ser represetada no formato JSON da seguinte maneira:

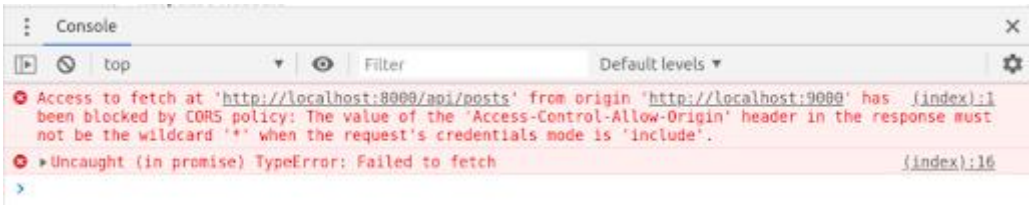
```
{
    "nome" : "Mochila",
    "preco" : 89.90,
    "descricao" : "Mochila para notebooks de até 17 polegadas"
} COPIAR CÓDIGO
```

Perceba como o formato JSON é muito mais compacto e legível. Justamente por isso se tornou o formato universal utilizado em comunicação de aplicações, principalmente no caso de APIs REST.

Mais detalhes sobre o JSON podem ser encontrados no site JSON.org.

05 Para saber mais: lidando com CORS

Quando desenvolvemos APIs e queremos que todos os seus recursos fiquem disponíveis a qualquer cliente HTTP, uma das coisas que vem à nossa cabeça é o CORS (*Cross-Origin Resource Sharing*), em português, "compartilhamento de recursos com origens diferentes". Se ainda não aconteceu com você, fique tranquilo, é normal termos erros de CORS na hora de consumir e disponibilizar APIs.



Mas afinal, o que é CORS, o que causa os erros e como evitá-los em nossas APIs com Spring Boot?

CORS

O CORS é um mecanismo utilizado para adicionar cabeçalhos HTTP que informam aos navegadores para permitir que uma aplicação Web seja executada em uma origem e acesse recursos de outra origem diferente. Esse tipo de ação é chamada de *requisição cross-origin HTTP*. Na prática, então, ele informa aos navegadores se um determinado recurso pode ou não ser acessado.

Mas por que os erros acontecem? Chegou a hora de entender!

Same-origin policy

Por padrão, uma aplicação Front-end, escrita em JavaScript, só consegue acessar recursos localizados na mesma origem da solicitação. Isso acontece por conta da política de mesma origem (*same-origin policy*), que é um mecanismo de segurança dos Browsers que restringe a maneira de um documento ou script de uma origem interagir com recursos de outra origem. Essa política possui o objetivo de frear ataques maliciosos.

Duas URLs compartilham a mesma origem se o protocolo, porta (caso especificado) e host são os mesmos. Vamos comparar possíveis variações considerando a URL `https://cursos.alura.com.br/category/programacao`:

URL	Resultado	Motivo
https://cursos.alura.com.br/category/front-end	Mesma origem	Só o caminho difere
http://cursos.alura.com.br/category/programacao	Erro de CORS	Protocolo diferente (http)
https://faculdade.alura.com.br:80/category/programacao	Erro de CORS	Host diferente

Agora, fica a dúvida: o que fazer quando precisamos consumir uma API com URL diferente sem termos problemas com o CORS? Como, por exemplo, quando queremos consumir uma API que roda na porta 8000 a partir de uma aplicação React rodando na porta 3000. Veja só!

Ao enviar uma requisição para uma API de origem diferente, a API precisa retornar um header chamado **Access-Control-Allow-Origin**. Dentro dele, é necessário informar as diferentes origens que serão

permitidas para consumir a API, em nosso caso: `Access-Control-Allow-Origin: http://localhost:3000`.
É possível permitir o acesso de qualquer origem utilizando o símbolo *(asterisco): `Access-Control-Allow-Origin: *`. Mas isso não é uma medida recomendada, pois permite que origens desconhecidas acessem o servidor, a não ser que seja intencional, como no caso de uma API pública. Agora vamos ver como fazer isso no Spring Boot de maneira correta.

Habilitando diferentes origens no Spring Boot

Para configurar o CORS e habilitar uma origem específica para consumir a API, basta criar uma classe de configuração como a seguinte:

```
@Configuration
public class CorsConfiguration implements WebMvcConfigurer {


    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
            .allowedOrigins("http://localhost:3000")
            .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS",
"HEAD", "TRACE", "CONNECT");
    }
}
COPIAR CÓDIGO
```

http://localhost:3000 seria o endereço da aplicação Front-end e **.allowedMethods** os métodos que serão permitidos para serem

executados. Com isso, você poderá consumir a sua API sem problemas a partir de uma aplicação Front-end.

06 DTO com Java Record

Transcrição

MedicoController:

```
package med.voll.api.controller;


import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestBody;


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


        @PostMapping
         public void cadastrar(@RequestBody String json) {
              System.out.println(json);
    }


}
```

COPIAR CÓDIGO

Conseguimos receber as informações enviadas pelo Insomnia no controller, mas precisamos encontrar uma maneira de não

recebermos esses dados como string e sim receber o JSON inteiro como string.

Uma forma de recebermos cada campo isoladamente, é não usar uma string como parâmetro do método `cadastrar` e sim uma classe. Nela, declaramos os atributos com os mesmos nomes que constam no JSON. Por isso, no método `cadastrar` vamos alterar o parâmetro de `string json` para uma classe que criaremos para representar os dados enviados pela requisição. Chamaremos essa classe de `DadosCadastroMedico`, nomearemos o parâmetro de `dados` e no `system out` ao invés de `json` será `dados`.

```
public void cadastrar(@RequestBody DadosCadastroMedico dados)
```

COPIAR CÓDIGO

`MedicoController` completo:

```
package med.voll.api.controller;


import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestBody;


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


        @PostMapping
        public void cadastrar(@RequestBody DadosCadastroMedico dados)
{
              System.out.println(dados);
```

```
    }


}
```
COPIAR CÓDIGO

Note que `DadosCadastroMedico` está escrito na cor vermelha, isso significa que está com erro de compilação, porque ainda não criamos essa classe.

Vamos criá-la, para isso podemos selecionar `DadosCadastroMedico` e depois usar o atalho "Alt + Enter". Será exibido um menu com várias opções, em que clicaremos na "create record 'DadosCadastroMedico'".

Esses dados que estão chegando na API, usaremos o recurso de `record` (disponível nas últimas versões do Java). Este recurso funciona como se fosse uma classe imutável, para deixarmos o código simples.

Isso para não usarmos uma classe tradicional, pois seria necessário digitarmos os métodos `getters` e `setters`, criar construtor, e todas as outras verbosidades do Java.

Voltando, após clicarmos na opção "create record 'DadosCadastroMedico'", será aberta uma pop-up com o título "*Create Record DadosCadastroMedico*", com dois campos: "*Destination package*" ("Pacote de destino") e "*Target destination directory*" ("Diretório de destino").

Create Record DadosCadastroMedico:

- **Destination package**: med.voll.api.controller
- **Target destination directory**: .../src/main/java/med/voll/api/controller

Nessa pop-up é para escolhermos em qual pacote desejamos criar essa classe record. Alteraremos o pacote para a classe não ficar no controller, usaremos médico.

- **Destination package**: med.voll.api.medico

Podemos clicar no botão "Ok", no canto inferior direito. Seremos redirecionados para o arquivo que acabamos de gerar.

`DadosCadastroMedico`:

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


public record DadosCadastroMedico() {


}
```

COPIAR CÓDIGO

No parêntese do método `record`, precisamos inserir os campos enviados pela requisição:

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


import med.voll.api.endereco.DadosEndereco;


public record DadosCadastroMedico(String nome, String email, String
crm) {
}
```

COPIAR CÓDIGO

Como o campo "especialidade" é fixo (temos quatro opções para a pessoa selecionar), não usaremos string e sim um `enum`. Podemos colocar após o `String crm`, o `Especialidade especialidade`.

Perceba que a palavra "Especialidade" está na cor vermelha, isso significa que não existe esse `enum` especialidade. Selecionando "Especialidade" usaremos o atalho "Alt + Enter", que exibirá um menu com diversas opções. Nele, escolheremos a opção "*Create enum 'Especialidade'*".

No pop-up seguinte, com o título "Create Enum Especialidade", podemos simplesmente clicar no botão "Ok".

```
Especialidade:

package med.voll.api.medico;


public enum Especialidade {


}
COPIAR CÓDIGO
```

Dentro das chaves do `enum`, vamos declarar as constantes, que são as opções do campo especialidade:

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


public enum Especialidade {


    ORTOPEDIA,

    CARDIOLOGIA,

    GINECOLOGIA,

    DERMATOLOGIA;


}
COPIAR CÓDIGO
```

Após isso, voltaremos para o arquivo `DadosCadastroMedico` para incluir os outros campos no parâmetro do método `cadastrar`. No campo "endereço" temos contido diversos campos nele, por isso, criaremos outro `record` para representar os dados do endereço.

```
DadosEndereco endereco

COPIAR CÓDIGO
```

Repare que após incluir esse campo no parâmetro, "DadosEndereco" está na cor vermelha porque não existe esse `record`.
Selecionando "DadosEndereco", pressionaremos as teclas "Alt + Enter", que exibirá um menu com diversas opções. Nele, escolheremos a opção "*Create record 'DadosEndereco'*".
No pop-up seguinte, com o título "Create Record DadosEndereco", vamos alterar o pacote de `medico` para `endereco`. Lembrando que o endereço também será usado ao cadastrar pacientes, por isso colocamos em outro pacote isolado.

- **Destination package**: med.voll.api.endereco

Logo após, podemos clicar no botão "Ok". Seremos redirecionados para o arquivo `DadosEndereco` inicial:

```
package med.voll.api.endereco;


public record DadosEndereco() {

}

COPIAR CÓDIGO
```

No parêntese, vamos inserir os campos do endereço:

```
package med.voll.api.endereco;


public record DadosEndereco(String logradouro, String bairro, String

cep, String cidade, String uf, String complemento, String numero) {
```

```
}
```

COPIAR CÓDIGO

Salvaremos o arquivo clicando em "Ctrl + S" e voltaremos para o arquivo `DadosCadastroMedico`, em que atualmente está como:

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


import med.voll.api.endereco.DadosEndereco;


public record DadosCadastroMedico(String nome, String email, String
crm, Especialidade especialidade, DadosEndereco endereco) {
}
```

COPIAR CÓDIGO

Com isso, o Java vai criar uma classe imutável, em que cada um desses campos vai virar atributos com os métodos `getters` e com os construtores, sem precisarmos fazer isso manualmente. O código fica mais simplificado ao usarmos `record` e o Spring cria de forma automática para nós.
Voltando para `MedicoController`, no método `cadastrar` estamos recebendo como parâmetro `DadosCadastroMedico` - sendo um `record`. Em seguida, efetuamos um `system out println` nesses dados.

Vamos salvar o arquivo e depois abrir a aba "Run", ele já reiniciou. No insomnia, clicaremos novamente no botão "Send" para enviar a requisição, assim podemos verificar como chegaram os dados.

Note que à direita, em "Preview", retornou um erro `400 Bad Request` e abaixo os campos *timestamp, status, error* e *trace*. Este último possui um grande stack. Isso quer dizer que algum campo foi enviado de forma errada.

Vamos analisar a mensagem do campo *"trace"* para tentarmos entender o problema.

Parte do erro selecionado pelo instrutor:
Cannot deserialize value of type 'med.voll.api.medico.Especialidade' from String "ortopedia"

Isso significa que não foi possível desserializar o campo "Especialidade" da string ortopedia. Criamos o `enum` para especialidade com a constante ortopedia.

Porém, as constantes do `enum` são escritas com letras maiúsculas. À esquerda, no JSON, note que estamos enviando com tudo em letra minúscula. Por isso, vamos alterar de "ortopedia" para "ORTOPEDIA".

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

Após isso, podemos enviar novamente a requisição clicando no botão "Send". Agora, sim, retornou `200 OK` com a mensagem "No body returned for response", em "Preview".
Vamos voltar ao IntelliJ para verificar se o `system out` exibiu corretamente, na aba "Run".

```
DadosCadastroMedico [nome=Rodrigo Ferreira,
email=rodrigo.ferreira@voll.med, crm=123456,
especialidade=ORTOPEDIA,
endereco=DadosEndereco[logradouro=rua 1, bairro=bairro,
cep=12345678, cidade=Brasilia, uf=DF, numero=1,
complemento=complemento]]
```

Chegou! Toda vez que aplicarmos `system out` em um objeto do tipo record, por padrão, o Java exibe dessa forma que consta no retorno do terminal.

Os campos "número" e "complemento" são opcionais, vamos remover o "complemento" do JSON no Insomnia para verificarmos o que vai acontecer.

```
{
"nome": "Rodrigo Ferreira",
"email": "rodrigo.ferreira@voll.med",
"crm": "123456",
"especialidade": "ORTOPEDIA",
"endereco": {
    "logradouro": "rua 1",
    "bairro": "bairro",
    "cep": "12345678",
    "cidade": "Brasilia",
    "uf": "DF",
```

```
        "numero": "1"
    }
}
```
COPIAR CÓDIGO

Novamente selecionaremos o botão "Send" e vamos voltar para o IntelliJ, na aba "Run". O campo “complemento” vem como `null`.

```
DadosCadastroMedico [nome=Rodrigo Ferreira,
email=rodrigo.ferreira@voll.med, crm=123456,
especialidade=ORTOPEDIA,
endereco=DadosEndereco[logradouro=rua 1, bairro=bairro,
cep=12345678, cidade=Brasilia, uf=DF, numero=1,
complemento=null]]
```

Logo, quando não preenchemos um campo que consta no `record`, o Spring insere nulo.

Desse modo, conseguimos receber os dados na API. Mas ao invés de recebermos como string, usamos o `record`. Esse tipo de classe Java ou `record` chamamos de padrão *DTO - Data Transfer Object* ("Objeto de transferência de dados").

É um padrão usado em APIs para representar os dados que chegam na API e também os dados que devolvemos dela.

Usaremos bastante esse padrão DTO nos pontos de entrada e saída. Isto é, sempre que precisarmos receber ou devolver dados da API, criaremos um DTO - sendo uma classe ou record que contém apenas os campos que desejamos receber ou devolver da API.

Assim, recebemos os dados na nossa API e agora precisamos pegá-los para executarmos as validações, isso para verificar se os campos estão

chegando corretamente, e depois salvar essas informações em um banco de dados.

Mas isso é assunto da próxima aula. Te espero lá!

07 Para saber mais: Java Record

Lançado oficialmente no Java 16, mas disponível desde o Java 14 de maneira experimental, o **Record** é um recurso que permite representar uma classe imutável, contendo apenas atributos, construtor e métodos de leitura, de uma maneira muito simples e enxuta.

Esse tipo de classe se encaixa perfeitamente para representar classes DTO, já que seu objetivo é apenas representar dados que serão recebidos ou devolvidos pela API, sem nenhum tipo de comportamento.

Para se criar uma classe DTO imutável, sem a utilização do Record, era necessário escrever muito código. Vejamos um exemplo de uma classe DTO que representa um telefone:

```
public final class Telefone {


    private final String ddd;
    private final String numero;


    public Telefone(String ddd, String numero) {
        this.ddd = ddd;
```

```
        this.numero = numero;
    }

    @Override
    public int hashCode() {
        return Objects.hash(ddd, numero);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        } else if (!(obj instanceof Telefone)) {
            return false;
        } else {
            Telefone other = (Telefone) obj;
            return Objects.equals(ddd, other.ddd)
              && Objects.equals(numero, other.numero);
        }
    }

    public String getDdd() {
        return this.ddd;
    }

    public String getNumero() {
        return this.numero;
    }
} COPIAR CÓDIGO
```

Agora com o Record, todo esse código pode ser resumido com uma única linha:

```
public record Telefone(String ddd, String numero){}COPIAR CÓDIGO
```

Muito mais simples, não?!

Por baixo dos panos, o Java vai transformar esse Record em uma classe imutável, muito similar ao código exibido anteriormente.

Mais detalhes sobre esse recurso podem ser encontrados na documentação oficial.

08 JSON e DTO

Em uma classe Controller existe o seguinte método declarado:

```
@PostMapping
public void cadastrar(DadosCadastroProduto dados) {
    System.out.println(dados);
}COPIAR CÓDIGO
```

E nesse projeto foi criado também o DTO `DadosCadastroProduto`:

```
public record DadosCadastroProduto(String nome, String descricao,
BigDecimal preco){}COPIAR CÓDIGO
```

Você dispara uma requisição POST, enviando no corpo dela o seguinte JSON:

```
{
    "preco" : 399.99,
    "descricao" : "Wireless. Cor: branca",
    "nome" : "Fone de ouvido"
} COPIAR CÓDIGO
```

Mas, ao verificar o console da IDE, percebe que os dados estão chegando todos como **null**.

Escolha a alternativa CORRETA que indica o porquê os dados estão retornado como nulos:

- Faltou mapear no método do Controller que a requisição vai receber dados no formato JSON.

- Alternativa correta

  A ordem dos campos no JSON não é a mesma ordem declarada no DTO.

- Alternativa correta

  Faltou anotar o parâmetro `dados`, recebido no método `cadastrar` do Controller, com **@RequestBody**.
  Sem essa anotação o Spring não vai ler o corpo da requisição e mapear os campos dele para o DTO recebido como parâmetro.

Parabéns, você acertou!

09 Faça como eu fiz: controller de pacientes

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

-
-

Você precisará criar uma classe Controller:

```
@RestController
@RequestMapping("pacientes")
public class PacienteController {

    @PostMapping
    public void cadastrar(@RequestBody DadosCadastroPaciente dados) {
        System.out.println("dados recebido: " +dados);
    }

}COPIAR CÓDIGO
```

Também precisará criar um DTO:

```
public record DadosCadastroPaciente(

        String nome,

        String email,

        String telefone,

        String cpf,

        DadosEndereco endereco

) {

} COPIAR CÓDIGO
```

Já o DTO `DadosEndereco` será o mesmo utilizado na funcionalidade de cadastro de médicos.

10 O que aprendemos?

Nessa aula, você aprendeu como:

- Mapear requisições POST em uma classe Controller;
- Enviar requisições POST para a API utilizando o Insomnia;
- Enviar dados para API no formato JSON;
- Utilizar a anotação `@RequestBody` para receber os dados do corpo da requisição em um parâmetro no Controller;
- Utilizar o padrão **DTO (*Data Transfer Object*)**, via Java Records, para representar os dados recebidos em uma requisição POST.