01 Projeto da aula anterior

Caso queira, você pode baixar aqui o projeto do curso no ponto em que paramos na aula anterior.

02Lidando com erros na API

Transcrição

Fizemos as alterações no controller, para devolver os códigos do protocolo HTTP de forma adequada, conforme a funcionalidade. E, também, padronizamos para retornar `Response Entity` nos métodos do controller.

Continuaremos com as melhorias referente à questão do protocolo HTTP e dos códigos HTTP devolvidos. Contudo, agora, vamos analisar outros cenários.

No Insomnia, temos a requisição "Detalhar médico" em que disparamos uma requisição do tipo `get` para o endereço `http://localhost:8080/medicos/6`, passando como parâmetro o ID.

Por exemplo, no caso, temos a médica Juliana Queiroz como sendo a médica correspondente ao ID número 6 do banco de dados. Logo, se dispararmos essa requisição `/medicos/6`, nos retorna o JSON com as informações da Juliana, em "Preview".

```
{
```

```
        "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
```

Porém, e se passarmos um ID que não possui registro no banco de dados após o `medicos/`? Por exemplo, vamos alterar o ID para `6999`.

```
http://localhost:8080/medicos/6999COPIAR CÓDIGO
```

Em seguida, clicamos no botão "Send". Note que o código devolvido foi o `500 Internal Server Error` e, em "Preview", temos uma mensagem com os campos `timestamp`, `status`, `error` e `trace`.

```
{
    "timestamp": "2022-10-21T17:59:29.080+00:00",
    "status": 500,
    "error": "Internal Server Error",
    "trace":
```

```
"jakarta.persistence.EntityNotFoundException: Unable to find
med.voll.api.medico.Medico with id
6999\n\tatorg.hibernate.jpa.boot.internal.EntityManagerFactoryBuilderi
mpl$JpanEntityNoFoundDelegate"


//retorno omitido
} COPIAR CÓDIGO
```

Código ou erro 500: Erro no servidor interno.

No caso, seria um erro na API back-end. Contudo, o que o Spring faz quando ocorre um erro no código do back-end? Por padrão, ele retorna essas informações em formato JSON que podemos visualizar na aba "Preview".

Nas informações do JSON há um objeto com algumas propriedades. A primeira é o `timestamp` que comunica data e hora do erro, o `status` que é o código HTTP `500` e o `error` é o nome do erro referente ao código.

Por fim, a propriedade `trace`, sendo a *stack trace* do erro - sendo a mesma exibida no console, informando qual a *exception* que ocorreu. Nesta propriedade, temos a mensagem "*Unable to find med.voll.api.medico.Medico with id 6999*", que significa que o ID que digitamos não existe no banco de dados.

Esse é o JSON devolvido. Porém, não é uma boa prática retornarmos um *stack trace* para o usuário da API, seja em uma aplicação front-end ou aplicativo mobile que está consumindo a nossa API.

Isso porque estamos expondo informações sensíveis e desnecessárias, o que pode se tornar uma brecha de segurança.

Neste caso da requisição `get`, poderíamos informar somente o status e qual o erro. Vamos aprender a fazer esse tratamento no retorno.

Desejamos padronizar as mensagens devolvidas pela API quando ocorre erro. Há diversas situações de erro que podem ocorrer da nossa API, nem sempre tudo funcionará conforme o esperado.

Por isso, em situações de erros é uma boa prática colocar respostas mais apropriadas para esses cenários.

Para ajustar isso, voltaremos ao Insomnia para analisarmos uma coisa. Perceba que o próprio tratamento do Spring já traz as informações que desejamos, porém, a propriedade `trace` nos devolve dados desnecessários e sensíveis.

O JSON devolvido no Insomnia é o padrão do Spring Boot, mas podemos configurar no projeto para não retornar os dados do campo `trace`.

Voltando ao IntelliJ, no arquivo `application.properties` em "src > main > resources".

application.properties

```
spring.datasource.url=jdbc:mysql://localhost/vollmed_api

spring.datasource.username=root

spring.datasource.password=root
```

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

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

Neste arquivo, temos as propriedades que configuramos de banco de dados e de gerar o SQL toda vez que a API vai ao banco de dados. Agora, incluiremos mais uma propriedade para o Spring não enviar a *stack trace* em caso de erro.

Para sabermos quais as propriedades do Spring Boot, **basta consultar a documentação**.

Common Application Properties

Na documentação, do lado esquerdo, temos um menu com o agrupamento das propriedades conforme a categoria, como propriedade Web, JPA, TomCat, etc.

Por exemplo, em "*Data properties*", são as propriedades do Spring Data, as propriedades de banco de dados, de controle de transação, entre outras tecnologias referente a dados.

No nosso caso, a propriedade que desejamos está relacionada com **o servidor**. Por isso, clicaremos no item 11 em "*Server Properties*", do lado esquerdo da documentação.

Em "*Server Properties*", temos uma tabela com os campos: *name*, *description* e *default value*. Nela, temos a propriedades ideal para o nosso caso, sendo a`server.error.include-stacktrace`.

Name	Description	Default Value
server.error.include-stacktrace	When to include the "trace" attribute.	never

Após copiar essa propriedade, voltaremos ao IntelliJ. Nele, colaremos o `server.error.include-stacktrace` com o valor padrão "never", na última linha do arquivo `application.properties`.

application.properties

```
spring.datasource.url=jdbc:mysql://localhost/vollmed_api

spring.datasource.username=root

spring.datasource.password=root


spring.jpa.show-sql=true

spring.jpa.properties.hibernate.format_sql=true


server.error.include-stacktrace=never COPIAR CÓDIGO
```

Salvaremos essa configuração e abriremos a aba "Rund", no canto inferior esquerdo. Perceba que ele reiniciou a execução, tudo certo, por enquanto.

Vamos voltar ao Insomnia, na requisição detalhar médico e clicar no botão "Send". Lembrando que no endereço estamos passando o ID de um médico que não existe: `http://localhost:8080/medicos/6999`.

Após clicarmos no botão, note que retornou o código `500 Internal Server Error`, mas que em "Preview" não temos mais o campo `trace`:

```
{
```

```
    "timestamp": "2022-10-21T18:06:01.320+00:00",
    "status": 500,
    "error": "Internal Server Error",
    "message": "Unable to find med.voll.api.medico.Medico with id
6999",
    "path": "/medicos/6999"
} COPIAR CÓDIGO
```

Assim, conseguimos aplicar um tratamento mais apropriado para esse cenário. Caso dê erro `500` na API, devolvemos um JSON (sendo que o próprio Spring fez isso, não escrevemos nenhum código) e somente informamos para não ser exibido o *stack trace*.

O usuário da API precisa saber somente que ocorreu um erro, qual o erro, o endereço que ocorreu e uma mensagem informando o motivo.

Desse modo, configuramos uma melhoria na devolutiva desse erro. Porém, na verdade, esta situação não deveria gerar um código `500`. Se estamos disparando uma requisição com um número de ID inexistente, o código HTTP mais adequado é o `404`.

Código ou erro 404: Recurso não encontrado.

O código `500` ocorreu porque estamos usando o Spring Data, e fazendo uma consulta no banco de dados com a interface *repository*. E o padrão do *Spring Data JPA* ao passar um ID inexistente é retornar uma *exception*. Assim, ao gerar uma excepction, nos retorna o código ou erro `500`.

Podemos configurar isso no nosso projeto. Em casos de determinadas *exceptions*, não retornar erro `500` e sim o `404`. Vamos personalizar o tratamento de erros na API.

Na sequência, vamos aprender como fazer essa personalização.

Te espero no próximo vídeo!

03 Para saber mais: propriedades do Spring Boot

Ao longo dos cursos, tivemos que adicionar algumas propriedades no arquivo `application.properties` para realizar configurações no projeto, como, por exemplo, as configurações de acesso ao banco de dados.

O Spring Boot possui centenas de propriedades que podemos incluir nesse arquivo, sendo impossível memorizar todas elas. Sendo assim, é importante conhecer a documentação que lista todas essas propriedades, pois eventualmente precisaremos consultá-la.

Você pode acessar a documentação oficial no link: Common Application Properties.

04 Tratando erro 404

Transcrição

Na aula anterior, tiramos a *stack trace* do erro `500` que o Spring é responsável pelo tratamento. Porém, neste cenário em específico deveria devolver o código `404`.

Para ajustar isso, voltaremos ao IntelliJ no arquivo `MedicoController`. O método que está sendo chamado nessa requisição é o `detalhar`.

```
//código omitido


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

Neste método, ele chama o `repository.getReferenceById(id)`. É justamente nesta linha que está o problema. O método `getReferenceById()`, ao passarmos um ID inexistente na tabela do banco de dados, ele envia uma *exception* do tipo *EntityNotFoundException*.

E perceba que não fizemos o tratamento, não temos o `try-catch`. Assim, a *exception* aconteceu nessa linha e foi lançada para o Spring. O Spring, por padrão, se ocorrer uma *exception* que não foi tratada no código, ele trata como uma exceção gerando o erro `500`.

Por padrão, exceções não tratadas no código são interpretadas pelo Spring Boot como erro 500.

No caso, não desejamos esse comportamento padrão, queremos que ao dar essa *exception* em específico, seja devolvido o erro `404`. Uma forma de lidar com essa situação, seria isolar o código do método de

detalhar, dentro de um `try-catch`. Faremos a captura da *exception*, e devolveremos o código `404`.

Ao fazermos isso, estamos tratando o erro somente no método de detalhar da classe `MedicoController`. Porém, essa exceção pode acontecer em outros métodos e controllers.
Por isso, ao invés de duplicarmos o `try-catch` no código, podemos usar outro recurso do Spring para isolar esse tipo de tratamento de erros.

A ideia é criarmos uma classe e nela termos o método responsável por tratar esse erro em específico.

Antes disso, faremos um ajuste. No pacote `med.voll.api`, temos quatro sub-pacotes: `controller`, `endereco`, `medico` e `paciente`.

Os sub-pacotes `endereco`, `medico` e `paciente` são referentes ao domínio da aplicação. Portanto, vamos isolar esses três pacotes em um único chamado `domain`.

Para isso, vamos selecionar os três pacotes segurando a tecla "Ctrl" e clicar com o botão direito do mouse. Nas caixas seguintes exibidas, escolhemos as opções "Refactor > Move packages or directories".

Será exibido um pop-up, em que vamos escolher para onde desejamos mover esses três pacotes que selecionamos. Para mover, no final do caminho vamos alterar de "endereco" para "domain".

```
/home/rodrigo/Desktop/api/src/main/java/med/voll/api/domain COPIAR CÓDIGO
```

Após isso, podemos clicar no botão "Refactor", no canto inferior direito. Assim, ficamos com as seguintes pastas do lado esquerdo do IntelliJ:

- `med.voll.api`
    - `controller`
    - `domain`

Se clicarmos em "domain", temos:

- `domain`
    - `endereco`
    - `medico`
    - `paciente`

Perceba que o arquivo `MedicoController` está com um sublinhado na cor vermelha. Isso significa que ocorreu um erro no Refactor, nos controllers ele não conseguiu renomear nos *imports*. Perceba que na linha 7 do `import`, ele ainda está apontando para o pacote antigo:

```
MedicoController

import med.voll.api.medico.*; COPIAR CÓDIGO
```

Vamos alterar para:

```
import med.voll.api.domain.medico.*; COPIAR CÓDIGO
```

Faremos o mesmo ajuste no arquivo `PacienteController`.

```
PacienteController
```

```
import med.voll.api.paciente.*; COPIAR CÓDIGO
```

Modificaremos para:

```
import med.voll.api.domain.paciente.*; COPIAR CÓDIGO
```

Pronto! Agora a estrutura está mais organizada. No pacote principal temos os pacotes `controller` e o `domain`. Além desses dois pacotes, criaremos um terceiro chamado "infra", em que ficarão os códigos relacionados à infraestrutura.

Para isso, clicamos com o botão direito do mouse na pasta `med.voll.api` e escolhemos a opção "Package". Na caixa exibida digitaremos o nome "infra", que ficará: `med.voll.api.infra` . Selecionando o pacote `infra`, usaremos o atalho "Alt + Insert" e escolheremos a opção "Java Class". No pop-up exibido, digitaremos "TratadorDeErros", sendo a classe que vai fazer o isolamento do tratamento de erros.

TratadorDeErros

```
package med.voll.api.infra;


public class TratadorDeErros {


} COPIAR CÓDIGO
```

Por enquanto, é uma classe em Java e não há nada em Spring. Isto é, o Spring não irá carregar essa classe automaticamente quando recarregarmos o projeto. Para ele carregar essa classe, é necessário termos à anotação `@RestControllerAdvice`.

TratadorDeErros

```
package med.voll.api.infra;


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


@RestControllerAdvice
public class TratadorDeErros {


} COPIAR CÓDIGO
```

Nesta classe, criaremos um método responsável por lidar com a exceção `EntityNotFoundException`. Para criar o método digitaremos `public void`, que chamaremos de `tratarErro404()` e, depois, abrimos e fechamos chaves ("{}").

TratadorDeErros

```
package med.voll.api.infra;


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


@RestControllerAdvice
public class TratadorDeErros {


        public void tratarErro404() {


        }


} COPIAR CÓDIGO
```

Uma linha anterior ao método, precisamos informar ao Spring para **qual exceção esse método será chamado**. No caso, usaremos à anotação `@ExceptionHandler`.

TratadorDeErros

```
package med.voll.api.infra;


import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;


@RestControllerAdvice
public class TratadorDeErros {


        @ExceptionHandler
        public void tratarErro404() {


        }


} COPIAR CÓDIGO
```

Acrescentaremos um abre e fecha parênteses "()" na anotação e dentro vamos especificar a classe exception:

```
@ExceptionHandler(EntityNotFoundException.class) COPIAR CÓDIGO
```

Código completo até o momento:

TratadorDeErros

```
package med.voll.api.infra;
```

```
import jakarta.persistence.EntityNotFoundException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

@RestControllerAdvice
public class TratadorDeErros {

        @ExceptionHandler(EntityNotFoundException.class)
        public void tratarErro404() {

        }

} COPIAR CÓDIGO
```

Dessa forma, o Spring sabe que se em qualquer controller do projeto for lançado uma exceção `EntityNotFoundException`, é para chamar o método `tratarErro404()`. E o que devolvermos, é o que será devolvido como resposta na requisição.

Por isso, vamos alterar de `void` para `ResponseEntity`:

TratadorDeErros

```
package med.voll.api.infra;

import jakarta.persistence.EntityNotFoundException;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;
```

```
@RestControllerAdvice
public class TratadorDeErros {

        @ExceptionHandler(EntityNotFoundException.class)
        public ResponseEntity tratarErro404() {

        }

} COPIAR CÓDIGO
```

No método `tratarErro404()`, colocaremos um `return ResponseEntity.notFound()`. No final, incluiremos um `.buil()` para ele criar o objeto `Response Entity`.

TratadorDeErros

```
package med.voll.api.infra;

import jakarta.persistence.EntityNotFoundException;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

@RestControllerAdvice
public class TratadorDeErros {

        @ExceptionHandler(EntityNotFoundException.class)
        public ResponseEntity tratarErro404() {
            return ResponseEntity.notFound().build();
```

```
        }


} COPIAR CÓDIGO
```

Note que é simples criar uma classe com métodos que trataram exceções não tratadas no controller. Logo, esse código fica isolado e não precisamos ter `try-catch` nos controllers. Estes nem percebem que há uma classe externa tratando o erro.

Salvaremos o arquivo, e o *DevTools* já reiniciou a execução. Agora, vamos testar. Para isso, voltaremos ao Insomnia para disparar a requisição de detalhar médico com um ID inexistente, clicando no botão "Send".

Endereço da requisição detalhar médico: http://localhost:8080/medicos/6999

Perceba que o código devolvido é o `404 Not Found`, escrito em uma caixa na cor laranja do lado direito do botão "Send". Funcionou!

Disparamos a requisição para detalhar o médico, a requisição chamará o `MedicoController`, vai cair na linha `repository.getReferenceById()` e será lançada a *exception*.

Contudo, essa *exception* não será mais tratada pelo Spring e, sim, na classe `TratadorDeErros` no método `tratarErro404()`. Neste método, comunicamos que é para devolver o código `404`.

É simples termos uma classe para tratar exceções no Spring Boot, deixando o código do controller enxuto. Sem nenhum tratamento de erros.

Assim, temos o tratamento do erro `500` - feito pelo próprio Spring, nós somente configuramos no `application.properties` - e a classe que trata o erro `404`. Podemos ter mais métodos tratando outros códigos de erros.

Na próxima aula, faremos o tratamento do erro `400`, usado quando disparamos uma requisição para cadastrar um médico ou paciente com erros na validação.

Aprenderemos como no próximo vídeo. Te espero lá!

05 Tratando erro 400

Transcrição

Por fim, o último erro que vamos tratar na API será o erro `400`. Este código ocorre quando o cliente da nossa API dispara uma requisição com dados inválidos.

Código ou erro 400: indica que o servidor não conseguiu processar uma requisição por erro de validação nos dados enviados pelo cliente.

Voltando ao Insomnia, isso pode acontecer tanto no cadastro de médicos quanto de pacientes.

Vamos clicar em "Cadastro de médico" do lado esquerdo. Neste método temos o endereço `http://localhost:8080/medicos` com o verbo `post`, e no corpo do JSON os seguintes dados:

```
{
        "nome": "Renato Amoedo",
        "email": "renato.amoedo@voll.med",
        "crm": "333444",
        "telefone": "61999998888",
        "especialidade": "ORTOPEDIA",
        "endereco": {
            "logradouro": "rua 1",
            "bairro": "bairro",
            "cep": "12345678",
            "cidade": "Brasilia",
            "uf": "DF"
        }
} COPIAR CÓDIGO
```

Estamos usando o *bean validation* na API para realizar as validações dos campos obrigatórios, entre outros dados.

Logo, se removermos os campos `nome`, `email`, `crm` e `telefone` do JSON e dispararmos a requisição com os campos de `especialidade` e `endereco`, deveria retornar o código `400`.

Isso porque todos os campos excluídos são obrigatórios, como *not null*.

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

Após essa remoção, clicaremos no botão "Send". Note que foi devolvido o erro `400 Bad Request`. Por padrão, ao acontecer um erro `400`, o Spring faz a integração com o *bean validation*, e roda as validações.

Caso ocorra alguma falha, é devolvido o código 400 e no JSON da resposta nos envia vários campos,
como `timestamp`, `status`, `error`, `message`, entre outros campos. Há, também, um objeto chamado `errors`, que contém o array com cada erro de validação ocorrido.

No entanto, esse *array* possui um objeto complexo com várias informações.

```
// Retorno omitido


{
        "codes": [
```

```
                "NotBlank.dadosCadastroMedico.telefone",
                "NotBlank.telefone",
                "NotBlank.java.lang.string",
                "NotBlank"
        ],

// Retorno omitido

} COPIAR CÓDIGO
```

Temos os nomes das anotações do *bean validation*, o que chegou, qual a mensagem padrão, o código, entre outras informações. Contudo, esse retorno poderia ser mais simplificado.

Podemos retornar um JSON com uma lista dos campos que geraram erro e, para cada campo, devolver qual o campo e a mensagem de erro. Por exemplo: "campo nome é obrigatório", "o e-mail está com formato inválido", etc.

Esses dados bastam para o usuário da nossa API, já que ele saberá qual o campo inválido e o motivo. Esse será o tratamento que faremos na API.

Voltando ao IntelliJ, incluiremos um novo método na classe `TratadorDeErros`.

TratadorDeErros

```
package med.voll.api.infra;


import jakarta.persistence.EntityNotFoundException;
import org.springframework.http.ResponseEntity;
```

```
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

@RestControllerAdvice
public class TratadorDeErros {

        @ExceptionHandler(EntityNotFoundException.class)
        public ResponseEntity tratarErro404() {
            return ResponseEntity.notFound().build();
        }

} COPIAR CÓDIGO
```

Após o penúltimo fecha chaves do `return` do método `tratarErro404()`, criaremos mais um método chamado `tratarErro400()`.

```
TratadorDeErros
//código omitido

@RestControllerAdvice
public class TratadorDeErros {

        @ExceptionHandler(EntityNotFoundException.class)
        public ResponseEntity tratarErro404() {
            return ResponseEntity.notFound().build();
        }

        public ResponseEntity tratarErro400() {
```

```
        }
} COPIAR CÓDIGO
```

Acima do método precisamos ter uma anotação, copiaremos a que consta no método da anotação anterior.

```
TratadorDeErros
//código omitido

@RestControllerAdvice
public class TratadorDeErros {

        @ExceptionHandler(EntityNotFoundException.class)
        public ResponseEntity tratarErro404() {
            return ResponseEntity.notFound().build();
        }

        @ExceptionHandler(EntityNotFoundException.class)
        public ResponseEntity tratarErro400() {

        }
} COPIAR CÓDIGO
```

Agora, qual a exceção que o método `tratarErro400()` vai tratar? Qual a *exception* que é lançada pelo Spring ou *bean validation*?

Há uma específica para esse erro, do *bean validation*, o `MethodArgumentNoValidExcpetion.class`.

```
TratadorDeErros

//código omitido


@ExceptionHandler(MethodArgumentNoValidExcpetion.class)

public ResponseEntity tratarErro400() {


} COPIAR CÓDIGO
```

Essa é a exceção que o *bean validation* lança quando há campo inválido. Assim, o Spring já sabe que se em alguma requisição ou controller ocorrer uma *exception* do tipo `Method Argument Not Valid Exception`, é para cair no erro `400`.

No retorno, colocaremos `return ResponseEntity.badRequest().build()`.

```
TratadorDeErros

//código omitido


@ExceptionHandler(MethodArgumentNoValidExcpetion.class)

public ResponseEntity tratarErro400() {

    return ResponseEntity.badRequest().build();


} COPIAR CÓDIGO
```

Bem semelhante ao que montamos no método anterior. Contudo, vamos testar para visualizar o que o Spring nos devolve se deixarmos assim.

Para testar, salvaremos o arquivo e voltaremos ao IntelliJ. No canto inferior esquerdo, clicamos na aba "Run" para verificar se reiniciou. Caso, sim, voltaremos ao Insomnia.

No Insomnia, na requisição "Cadastro de Médico", selecionaremos o botão "Send". Lembrando que removemos alguns campos do JSON para fazermos alguns testes.

Ao dispararmos, nos é devolvido o código `400 Bad Request`. No entanto, em "Preview", não retornou nenhum dado, somente a mensagem "*No body returned for response*". Conseguimos fazer o tratamento personalizado, mas faltou o corpo da resposta.

Isso porque quando o cliente recebe o erro da API, é exibido o erro `400` informando que é um dado inválido, mas não especificamos qual. Neste erro, nós precisamos retornar alguns dados no corpo.

Para isso, voltaremos ao IntelliJ. Para levar um corpo com as informações, no método `tratarErro400()`, precisamos passar o objeto que desejamos retornar no parênteses do `bad request()`.
Porém, diferente do código `404`, no `400` é necessário sabermos quais erros ocorreram. Portanto, precisamos capturar a exceção lançada, pois é nela que teremos acesso a quais campos estão inválidos conforme as regras do *bean validation*.

Para isso, na assinatura do método `tratarErro400()`, receberemos como parâmetro a *exception* lançada. Essa exceção deve ser a mesma da anotação, podemos copiá-la.

Logo após, colaremos no parênteses do método `tratarErro400()` e depois nomearemos o parâmetro de `ex` (abreviação de exception).

```
TratadorDeErros

//código omitido


@ExceptionHandler(MethodArgumentNoValidExcpetion.class)
public ResponseEntity tratarErro400(MethodArgumentNoValidExcpetion ex)
{


    return ResponseEntity.badRequest().build();


} COPIAR CÓDIGO
```

Se quisermos, podemos receber a *exception* lançada no método que estamos tratando o erro, basta declarar como parâmetro na assinatura do método.

Para capturar os erros que ocorreram no *bean validation*,

Esse objeto *exception* possui um método que retorna a lista com os campos inválidos. Vamos colocar isso em uma variável chamada `erros` dentro do método, antes do `return`.

```
TratadorDeErros

//código omitido


@ExceptionHandler(MethodArgumentNoValidExcpetion.class)
```

```
public ResponseEntity tratarErro400(MethodArgumentNoValidExcpetion ex)
{
    var erros = ex.getFieldErrors();
    return ResponseEntity.badRequest().build();

} COPIAR CÓDIGO
```

O `get field errors()` lista cada erro ocorrido em cada campo. E esse é o objeto que desejamos retornar no corpo da resposta. Porém, vamos verificar o que acontece se devolvermos esse objeto de forma direta.

Guardamos o `getFieldErrors` na variável `erros`, e para enviar um corpo na resposta incluímos o `.body()` no `return`. No método `body()` passamos a lista de `erros`, e podemos remover o `.build()`, já que o *body* nos retorna o objeto `ResponseEntity`.

```
TratadorDeErros
//código omitido

@ExceptionHandler(MethodArgumentNoValidExcpetion.class)
public ResponseEntity tratarErro400(MethodArgumentNoValidExcpetion ex)
{
    var erros = ex.getFieldErrors();

    return ResponseEntity.badRequest().body(erros);

} COPIAR CÓDIGO
```

Após essas alterações, salvaremos o arquivo. Se devolvermos essa lista no corpo da resposta, ele nos devolve o código 400 com um JSON na resposta.

Para testar, voltaremos ao Insomnia e clicaremos no botão "Send", do método "Cadastro de Médico". Note que nos retornou o código `400 Bad Request`, com um corpo em "Preview".

Preview

```
[
    {
    "codes": [
    "NotBlank.dadosCadastroMedico.nome",
    "NotBlank.nome",
    "NotBlank.java.lang.String",
    "NotBlank"
    ],

    //retorno omitido

    }
] COPIAR CÓDIGO
```

Todavia, é o mesmo JSON que foi devolvido anteriormente, com informações desnecessárias. No caso, desejamos retornar somente o nome do campo e a mensagem.

Para isso, vamos personalizar aquela lista, transformando-a em um DTO. Assim como fizemos na listagem de médicos, precisaremos criar

um DTO que contém somente os campos e as mensagens. E teremos que converter a lista do *bean validation*, para a lista do nosso DTO.

DTO: *data transfer object*, em português, objeto de transferência de dados.

Voltaremos ao IntelliJ para criar o DTO *record*. Contudo, ao invés de criarmos em um arquivo separado, montaremos dentro da classe `TratadorDeErros`.

Isso porque vamos usar o DTO somente dentro dessa classe. Logo, podemos criar como um *record* interno. Para isso, após o fechamento de chaves do método `tratarErro400`, vamos declarar o *record* chamado `DadosErroValidacao()`.

```
TratadorDeErros

//código omitido


private record DadosErroValidacao() {


} COPIAR CÓDIGO
```

No caso, o DTO terá somente dois parâmetros: campo e mensagem.

```
TratadorDeErros

//código omitido


private record DadosErroValidacao(String campo, String mensagem) {


} COPIAR CÓDIGO
```

Agora, no `body` do método `tratarErro400` não passaremos mais a lista de `erros`, precisamos convertê-la para uma lista de dados, erro e validação.

No parêntese do `body` passaremos `erros.stream().map()`, vamos chamar os recursos do Java 8 para convertemos uma lista para outra. Essa parte solicita para erros me dê um *stream* e mapeie cada objeto `FieldError` para um objeto `DadosErroValidacao`.

Em `DadosErroValidacao` acrescentamos `::new` para chamar o construtor. Por fim, usaremos o método `.toList()` para convertermos para uma lista.

```
TratadorDeErros

//código omitido


@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity tratarErro400(MethodArgumentNotValidException
ex) {
    var erros = ex.getFieldErrors();

    return
ResponseEntity.badRequest().body(erros.stream().map(DadosErroValidacao
::new).toList());

} COPIAR CÓDIGO
```

Perceba que abaixo de "DadosErroValidacao" do retorno, temos uma linha na cor vermelha. Isso quer dizer que tivemos um erro de

compilação, isso acontece porque no nosso DTO `DadosErroValidacao` precisaremos incluir outro construtor que receberá o objeto `FieldError`.

Para isso, vamos declarar um construtor em `record`. Na linha seguinte colocaremos `public DadosErroValidacao()` que vai receber um objeto do tipo `FieldError` chamado `erro`.

```
//código omitido


private record DadosErroValidacao(String campo, String mensagem) {
        public DadosErroValidacao(FieldError erro) {


        }
} COPIAR CÓDIGO
```

Na próxima linha, chamaremos o construtor padrão do `record` passando `erro.getField()` e `erro.getDefaultMessage()`.

- **erro.getField()**: nos devolve o nome do campo
- **erro.getDefaultMessage()**: nos devolve a mensagem para um campo específico.

```
//código omitido


private record DadosErroValidacao(String campo, String mensagem) {
        public DadosErroValidacao(FieldError erro) {
            this(erro.getField(), erro.getDefaultMessage());
        }
} COPIAR CÓDIGO
```

Perceba que o método `tratarErro400` está compilando, agora.

Deste modo, conseguimos converter a lista de `FieldError` para uma lista de `DadosErroValidacao`. Podemos salvar e voltar ao Insomnia para disparar uma requisição no método "Cadastro de Médico".

Note que temos o código `400 Bad Request`, com o seguinte corpo na resposta:

```
[
    {
    "campo": "telefone",
    "mensagens": "must not be blank"
    },
    {
    "campo": "email",
    "mensagens": "must not be blank"
    },
        {
    "campo": "nome",
    "mensagens": "must not be blank"
    },
        {
    "campo": "crm",
    "mensagens": "must not be blank"
    }
] COPIAR CÓDIGO
```

Temos um array com cada campo inválido e sua respectiva mensagem.

Vamos tentar cadastrar um médico, incluindo os campos que removemos anteriormente e alterando os campos nome, email e crm para:

```
{
        "nome": "Bruna Silva",
        "email": "bruna.silva@voll.med",
        "crm": "124580",
        "telefone": "61999998888",
        "especialidade": "ORTOPEDIA",
        "endereco": {
            "logradouro": "rua 1",
            "bairro": "bairro",
            "cep": "12345678",
            "cidade": "Brasil",
            "uf": "DF"
        }
} COPIAR CÓDIGO
```

Agora, clicaremos no botão "Send". Foi devolvido o código `201 Created`, com o seguinte corpo da resposta:

```
{
        "id": 7,
        "nome": "Bruna Silva",
        "email": "bruna.silva@voll.med",
        "crm": "124580",
```

```
        "telefone": "2111112222",
        "especialidade": "ORTOPEDIA",
        "endereco": {
            "logradouro": "rua 1",
            "bairro": "bairro",
            "cep": "12345678",
            "numero": null,
            "complemento": null,
            "cidade": "Brasil",
            "uf": "DF"
        }
} COPIAR CÓDIGO
```

Pronto! Tudo certo.

Se removermos o campo `crm` do JSON e clicarmos no botão "Send", temos:

```
[
    {
    "campo": "crm",
    "mensagens": "must not be blank"
    }
] COPIAR CÓDIGO
```

Tudo funcionou conforme o esperado.

Conseguimos aplicar mais um tratamento de erro, em que devolvemos um JSON simplificado para facilitar a leitura do usuário da nossa API.

O usuário dispara a requisição, e se tiver algum dado inválido ele recebe o erro `400`, com o campo e a mensagem na resposta.

Na sequência, seguiremos desenvolvendo o nosso projeto.

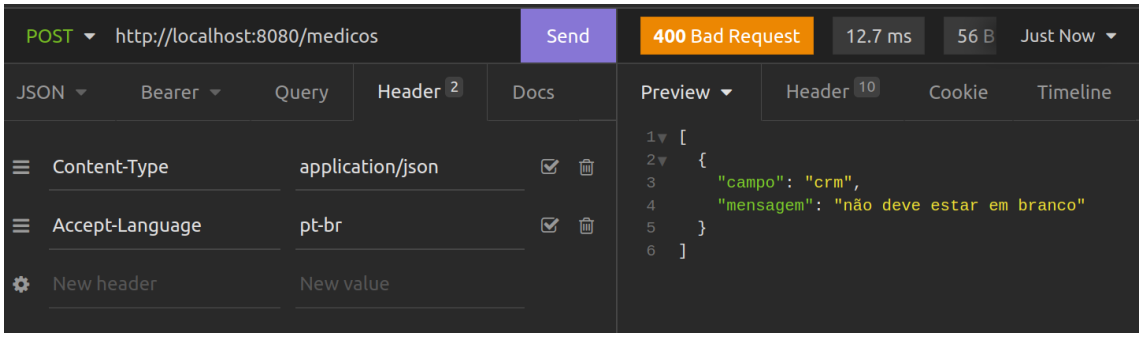
Até mais!

06 Para saber mais: mensagens em português

Por padrão o Bean Validation devolve as mensagens de erro em inglês, entretanto existe uma tradução dessas mensagens para o português já implementada nessa especificação.

No protocolo HTTP existe um cabeçalho chamado **Accept-Language**, que serve para indicar ao servidor o idioma de preferência do cliente disparando a requisição. Podemos utilizar esse cabeçalho para indicar ao Spring o idioma desejado, para que então na integração com o Bean Validation ele busque as mensagens de acordo com o idioma indicado.

No Insomnia, e também nas outras ferramentas similares, existe uma opção chamada **Header** que podemos incluir cabeçalhos a serem enviados na requisição. Se adicionarmos o header **Accept-Language** com o valor **pt-br**, as mensagens de erro do Bean Validation serão automaticamente devolvidas em português.



Obs: O Bean Validation tem tradução das mensagens de erro apenas para alguns poucos idiomas.

07 Tratamento de exceptions

Em um projeto de uma API Rest com Spring Boot, o tratamento personalizado de **Erro 404** não está sendo realizado corretamente, apesar de existir a seguinte classe nesse projeto:

```
@RestController
public class ExceptionHandler {

    @ExceptionHandler(EntityNotFoundException.class)
    public void tratarErro404() {
    }

} COPIAR CÓDIGO
```

Por qual motivo o método `tratarErro404` dessa classe não está sendo executado?

- O retorno do método foi declarado como `void`.

  Embora o ideal seja devolver alguma informação, deixar o método sem retorno não impede que ele seja chamado pelo spring.

- Alternativa correta

  A classe não foi anotada com `@Configuration`.

- Alternativa correta

  A classe foi anotada de maneira incorreta.

  Em APIs Rest, classes de tratamento de *exceptions* devem ser anotadas com o `@RestControllerAdvice` e não com o `@RestController`.

- Alternativa correta

> A *exception* passada na anotação `@ExceptionHandler` é do tipo ***unchecked***.

Parabéns, você acertou!

08 Para saber mais: personalizando mensagens de erro

Você deve ter notado que o *Bean Validation* possui uma mensagem de erro para cada uma de suas anotações. Por exemplo, quando a validação falha em algum atributo anotado com `@NotBlank`, a mensagem de erro será: ***must not be blank***.

Essas mensagens de erro não foram definidas na aplicação, pois são mensagens de erro **padrão** do próprio *Bean Validation*. Entretanto, caso você queira, pode personalizar tais mensagens.

Uma das maneiras de personalizar as mensagens de erro é adicionar o atributo `message` nas próprias anotações de validação:

```
public record DadosCadastroMedico(

    @NotBlank(message = "Nome é obrigatório")

    String nome,


    @NotBlank(message = "Email é obrigatório")

    @Email(message = "Formato do email é inválido")

    String email,


    @NotBlank(message = "Telefone é obrigatório")

    String telefone,


    @NotBlank(message = "CRM é obrigatório")
```

```
    @Pattern(regexp = "\\d{4,6}", message = "Formato do CRM é
inválido")
    String crm,

    @NotNull(message = "Especialidade é obrigatória")
    Especialidade especialidade,

    @NotNull(message = "Dados do endereço são obrigatórios")
    @Valid DadosEndereco endereco) {} COPIAR CÓDIGO
```

Outra maneira é isolar as mensagens em um arquivo de propriedades, que deve possuir o nome ***ValidationMessages.properties*** e ser criado no diretório `src/main/resources`:

```
nome.obrigatorio=Nome é obrigatório
email.obrigatorio=Email é obrigatório
email.invalido=Formato do email é inválido
telefone.obrigatorio=Telefone é obrigatório
crm.obrigatorio=CRM é obrigatório
crm.invalido=Formato do CRM é inválido
especialidade.obrigatoria=Especialidade é obrigatória
endereco.obrigatorio=Dados do endereço são obrigatórios COPIAR CÓDIGO
```

E, nas anotações, indicar a chave das propriedades pelo próprio atributo `message`, delimitando com os caracteres `{` e `}`:

```
public record DadosCadastroMedico(
    @NotBlank(message = "{nome.obrigatorio}")
    String nome,
```

```
    @NotBlank(message = "{email.obrigatorio}")
    @Email(message = "{email.invalido}")
    String email,

    @NotBlank(message = "{telefone.obrigatorio}")
    String telefone,

    @NotBlank(message = "{crm.obrigatorio}")
    @Pattern(regexp = "\\d{4,6}", message = "{crm.invalido}")
    String crm,

    @NotNull(message = "{especialidade.obrigatoria}")
    Especialidade especialidade,

    @NotNull(message = "{endereco.obrigatorio}")
    @Valid DadosEndereco endereco) {} COPIAR CÓDIGO
```

09 Faça como eu fiz: RestControllerAdvice

Agora é com você! Faça o mesmo procedimento que eu fiz na aula, criando uma classe responsável por tratar as *exceptions* que podem ocorrer nas classes Controller.

Opinião do instrutor

-
-

Você precisará criar uma classe similar a esta:

```
@RestControllerAdvice
public class TratadorDeErros {
```

```
@ExceptionHandler(EntityNotFoundException.class)
public ResponseEntity tratarErro404() {
    return ResponseEntity.notFound().build();
}

@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity tratarErro400(MethodArgumentNotValidException
ex) {
    var erros = ex.getFieldErrors();
    return
ResponseEntity.badRequest().body(erros.stream().map(DadosErroValidacao
::new).toList());
}

private record DadosErroValidacao(String campo, String mensagem) {
    public DadosErroValidacao(FieldError erro) {
        this(erro.getField(), erro.getDefaultMessage());
    }
}
} COPIAR CÓDIGO
```

Além disso, você também deve adicionar a seguinte propriedade no arquivo `application.properties`, para evitar que a stacktrace da exception seja devolvida no corpo da resposta:

```
server.error.include-stacktrace=never COPIAR CÓDIGO
```

10 O que aprendemos?

Nessa aula, você aprendeu como:

- Criar uma classe para isolar o tratamento de *exceptions* da API, com a utilização da anotação `@RestControllerAdvice`;
- Utilizar a anotação `@ExceptionHandler`, do Spring, para indicar qual *exception* um determinado método da classe de tratamento de erros deve capturar;
- Tratar erros do tipo **404 (*Not Found*)** na classe de tratamento de erros;
- Tratar erros do tipo **400 (*Bad Request*)**, para erros de validação do *Bean Validation*, na classe de tratamento de erros;
- Simplificar o JSON devolvido pela API em casos de erro de validação do *Bean Validation*.