04

Simplificando o JSON

Transcrição

[00:00] Vamos continuar então. No último vídeo vimos como fazer validação de formulário com Bean validation. Testamos no Postman e vimos que funcionou. Quando você manda o JSON com todos os dados certos, dá ok, normal, ele cria o recurso. Agora, quando tem algum campo faltando, inválido, o servidor devolve o erro "400 Bad Request". Só que no corpo da resposta, por padrão, o Spring gera um JSON gigantesco.

[00:40] Na aula de hoje vamos ver como simplificar esse JSON que é mandado de resposta pelo Spring. Vamos voltar para o Eclipse, no nosso projeto `TopicosController.java`. Como isso vai ser tipo um tratamento de erros, e esse tratamento não vai valer só para o `TopicosController.java`. Qualquer `Controller` do meu projeto quero que tenha esse tratamento de erros. A lógica para tratar a validação e as mensagens não vai ficar no `TopicosController.java`. Temos que colocar isso em outro lugar isolado para que valha para qualquer tipo de `Controller` do nosso projeto.

[01:20] O Spring tem uma solução para esse tipo de cenário. A solução é criar uma espécie de interceptador.

```
@GetMapping
public List<TopicoDto> lista(String nomeCurso) {
    if (nomeCurso == null) {
            List<Topico> topicos = topicoRepository.findAll();
```

```
            return TopicoDto.converter(topicos);
        } else {
            List<Topico> topicos =
topicoRepository.findByCursoNome(nomeCurso);
            return TopicoDto.converter(topicos);
        }
}
```

COPIAR CÓDIGO

Toda vez que acontecer uma *exception*, em qualquer método de qualquer `Controller`, o Spring automaticamente vai chamar esse interceptador, e nós faremos o tratamento apropriado. Esse interceptador é chamado de *Controller Advice*. A ideia é criarmos uma classe e transformá-la em um *Controller Advice*, onde vamos fazer o tratamento do erro.

[01:48] Precisamos criar uma nova classe no projeto. Para isso, pressionaremos "br.com.alura.forum" e abriremos uma nova tela, onde selecionaremos, em "Create a new Java class", "Class" e "Next". Na próxima tela, preencheremos "Package" com "br.com.alura.config.validacao", colocando em um pacote `config.validacao`, porque depois teremos outras configurações no projeto. Vamos nomear essa classe de "ErroDeValidacaoHandler" (essa classe que vai fazer o *Handler*, o tratamento dos erros de validação) e finalizar apertando "Finish".

[02:23] No `"ErroDeValidacaoHandler".java`, preciso dizer para o Spring que a classe `ErroDeValidacaoHandler` é um *Controller Advice*. Existe a anotação `@ControllerAdvice`, mas, no nosso caso, usaremos `@RestControllerAdvice`, porque estamos usando `RestController` da API REST.

```
package br.com.alura.forum.config.validacao;

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

```
@RestControllerAdvice
public class ErroDeValidacaoHandler {

}
```

COPIAR CÓDIGO

Agora, precisamos ensinar para o Spring que esse *Controller Advice* vai fazer tratamento de erros, para quando aparecer uma exceção. Por exemplo, invalidação de formulários.

[02:55] Para isso, vamos criar um método usando `void` por enquanto, `public void handle()`, sendo que é o método `handle()` que fará o tratamento do erro. Preciso dizer para o Spring que esse método deve ser chamado quando houver uma exceção dentro de algum `Controller`. Para falar isso para o Spring, em cima do método, vamos colocar uma anotação `@ExceptionHandler`, que é do próprio Spring.

```
package br.com.alura.forum.config.validacao;

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

@RestControllerAdvice
public class ErroDeValidacaoHandler {

    @ExceptionHandler()
    public void handle() {

    }

}
```

COPIAR CÓDIGO

E temos que passar como parâmetro que tipo de exceção que, quando acontecer dentro do `Controller`, o Spring vai direcionar para o método `handler()`. No nosso caso, é exceção de validação de formulário. Quando dá um erro de validação de formulário, usando Bean Validation, que exceção que o Spring lança? Ele manda uma *exception* chamada `MethodArgumentNotValidException`. Temos que passar o nome da classe, isto é, `MethodArgumentNotValidException.class`. Agora o Spring sabe que se acontecer essa *execption* em qualquer `RestController`, ele vai cair no método `@ExceptionHandler()`.

```
@ExceptionHandler(MethodArgumentNotValidException.class)
public void handle() {

}
```

COPIAR CÓDIGO

[04:05] No método `handle()`, preciso pegar a exceção que aconteceu, para pegar as mensagens, fazer o tratamento. Esse método recebe como parâmetro um objeto do tipo `MethodArgumentNotValidException`, que é o mesmo *Handler* que estou utilizando no `@ExceptionHandler()`. Chamarei o parâmetro de `exception`.

```
@ExceptionHandler(MethodArgumentNotValidException.class)
public void handle(MethodArgumentNotValidException exception) {

}
```

COPIAR CÓDIGO

[04:32] Toda vez que acontecer alguma *exception* desse tipo em qualquer `RestController` do projeto, o Spring vai chamar esse método `handle()` passando como parâmetro a *exception* que aconteceu. Tendo esse interceptador, o Spring considera que deu um erro, mas que não vai devolver o código 400 para o cliente. vai chamar o interceptador e, aqui dentro, você faz o tratamento. Ele considera que

fizemos o tratamento e, por padrão, ele vai devolver 200 para o cliente. Mas não quero que ele devolva 200, quero que ele devolva 400. Então, em cima do método, escreverei `@ResponseStatus()` para falar para ele qual o status que ele vai devolver. O parâmetro será `code = HttpStatus.BAD_REQUEST`

```
package br.com.alura.forum.config.validacao;

import org.springframework.http.HttpStatus;

@RestControllerAdvice
public class ErroDeValidacaoHandler {

    @ResponseStatus(code = HttpStatus.BAD_REQUEST)
        @ExceptionHandler(MethodArgumentoNotValidException.class)
        public void handle(MethodArgumentNotValidException
exception) {

        }

}
```

COPIAR CÓDIGO

[05:20] Por mais que eu tenha tratado o erro, não é para devolver 200. É para continuar devolvendo 400, porque meu tratamento é só para mudar as mensagens. Agora, dentro do código, eu faço o tratamento.

[05:33] Só que o Spring, além de devolver o `@ResponseStatus`, devolve o que o método estiver retornando. Então, nosso método não pode ser `void`. Ele tem que devolver alguma coisa, que, no nosso caso, vai ser uma lista com as mensagens de erro. Mas ao invés de ser aquela mensagem bizarra, enorme, do Spring, vai ser uma mensagem que nós vamos personalizar.

[05:55] Esse método vai devolver um `List<>` . Para representar o erro, vou criar um novo DTO que serve para representar um erro de formulário. Vou criar uma classe chamada `ErroDeFormularioDto` .

```
@ResponseStatus(code = HttpStatus.BAD_REQUEST)
@ExceptionHandler(MethodArgumentNotValidException.class)
public List<ErroDeFormularioDto>
handle(MethodArgumentNotValidException exception) {

}
```

COPIAR CÓDIGO

Essa classe não existe, então vou criá-la selecionando, no atalho, "Create class 'ErroDeFormularioDto". Depois, em "Create a New Java Class" - considerando que essa classe faz parte da configuração de validação - a colocarei no mesmo pacote "br.com.alura.forum.config.validacao". Essa classe `ErroDeFormularioDto` que vai representar um erro de validação. O JSON, que vai ser devolvido para o cliente, não vai ser mais aquele imenso do Spring. Vai ser o JSON representado por essa classe.

[06:43] Nessa classe, o que preciso ter de informações para mandar para o JSON? Só duas: primeiro, qual foi o campo que deu erro (se foi título, mensagem, nome do curso), então vou criar um atributo chamado `campo` ; e o segundo, é uma `String` também, correspondente à mensagem - qual o erro - logo, chamarei de `erro` .

```
package br.com.alura.forum.config.validacao;

public class ErroDeFormularioDto {

    public String campo;
     private String erro;

}
```

COPIAR CÓDIGO

[07:23] E aí vou declarar um construtor. Para isso selecionarei "Generate Construtor using Fields" usando o atalho. Na próxima tela, selecionados "campo" e "erro", basta apertar "Generate". Agora, quando der `new` nessa classe, já passarei campo e mensagem de erro. Usarei de novo o atalho para gerar os *Getters* e *Setters*, selecionando "Generate Getters and Setters". Só que no caso, como já estou recebendo os parâmetros do construtor, só vou gerar os *Getters*. Sendo assim, seleciono "campo" e "erro" e, em seguida, "Select Getters".

[07:37] O DTO, `ErroDeFormularioDto`, representa o erro de algum campo. Ele tem o nome do campo, `private String campo`, e uma mensagem de erro, `private String erro`. Era só o que eu queria: para cada campo, o nome do campo e a mensagem de erro.

```
package br.com.alura.forum.config.validacao;

public class ErroDeFormularioDto {

    private String campo;
        private String erro;

        public ErroDeFormularioDto(String campo, String erro) {
            this.campo = campo;
                this.erro = erro;
      }

        public String getCampo() {
            return campo;
        }

        public String getErro() {
```

```
            return erro;
        }

        }
```

COPIAR CÓDIGO

[07:52] Voltando ao nosso `ErroDeValidacaoHandler.java`. Esse que vai ser o retorno: `List<ErroDeFormularioDto> handle(MethodArgumentNotValidException exception)`. Isto é, ele vai devolver uma lista com cada um dos erros que aconteceram.

[08:02] Como eu descubro que erros aconteceram? Por isso o método `handle()` recebe o tal do `(MethodArgumentNotValidException exception)`. Dentro desse objeto tem todos os erros que aconteceram.

[08:41] Se pegarmos o parâmetro `exception`, seguido do método `getBindingResult()`, com o resultado das validações, e do método `getfieldErrors()` que contém todos erros "*Field*", de formulário.

```
exception.getBindingResult().getFieldErrors();
```

COPIAR CÓDIGO

com o comando "Ctrl + 1", usarei o atalho para criar uma variável local que chamarei de `fieldErrors`. Essa variável tem os erros de formulário. Só que não quero devolver essa lista, `List<>`, de `FieldError`. Quero devolver uma lista de `ErroDeFormulárioDto`. Então:

```
List<ErroDeFormularioDto> dto = new ArrayList<>()
```

COPIAR CÓDIGO

O que vou devolver nesse método é justamente o DTO, `return dto`. Estou devolvendo uma lista de erro de formulário DTO. Só que no momento essa lista está

vazia.

[09:24] Agora que peguei os `fieldErrors`, vou ter que percorrer essa lista para cada *field error* criar um objeto *erro dto* e guardar nessa lista representada pela minha variável `dto` (`List<ErroDeFormularioDto> dto = new ArrayList<>()`).

Agora vou usar os recursos de *stream*, de interação, do Java8. Então, vou chamar direto o `forEach()`, (`fieldErrors.forEach()`) e passar um lâmbda, indicando: para cada erro, o que quero fazer, isto é, `fieldErrors.forEach(e -> {`. Neste caso, quero criar um erro de formulário DTO, então, `ErroDeFormularioDto erro = new ErroDeFormularioDto(campo, erro)`.

```
    List<FieldError> fieldErrors =
exception.getBindingResult().getFieldErrors();
    fieldErrors.forEach(e -> {
        ErroDeFormularioDto erro = new ErroDeFormularioDto(campo,
erro);
     });

     return dto;
  }

}
```

COPIAR CÓDIGO

[10:00] Lembre-se que, quando eu dou `new` no `ErroDeFormularioDto()`, tenho que passar uma mensagem e qual o campo que deu erro. O campo pego pelo `e.getField()`. O próprio erro do `FieldError` já tem qual é o nome do campo.

```
    List<FieldError> fieldErrors =
exception.getBindingResult().getFieldErrors();
    fieldErrors.forEach(e -> {
```

```
        ErroDeFormularioDto erro = new
ErroDeFormularioDto(e.getField(), erro);
      });

      return dto;
   }

}
```

COPIAR CÓDIGO

Agora, para pegar a mensagem, começaremos esrevendo `String mensagem =` . Como podemos ter aquele esquema de internacionalização, isto é, de mensagens em vários idiomas, o Spring nos ajuda com uma classe chamada `MessageSource` . Vou injetar com o `@Autowired` , nessa classe `Handler` um atributo do tipo `messageSource` . Essa classe `MessageSource` te ajuda a pegar mensagens de erro, de acordo com o idioma que o cliente requisitar.

```
@RestControllerAdvice
public class ErroDeValidacaoHandler {

    @Autowired
        private MessageSource messageSource;

        @ResponseStatus(code = HttpStatus.BAD_REQUEST)
        @ExceptionHandler(MethodArgumentNotValidException.class)
        public List<ErroDeFormularioDto> dto = new ArrayList<>();

        List<FiledError> fieldErrors =
exception.getBindingResult().getFieldErrors():
        fieldErrors.forEach(e -> {
           String mensagem =
             ErroDeFormularioDto erro = new
ErroDeFormularioDto(e.getField(), mensagem);
      });
```

```
        return dto;
    }
```

COPIAR CÓDIGO

[11:15]Para pegar mensagens de erro, `String mensagem = messageSource.getMessage()` e vou passar como parâmetro o erro `e`, e uma classe do Spring chamada `localeContextHolder.getLocaLe()`, para ele descobrir qual o "*locale*", qual o local atual para pegar a mensagem no idioma correto.

```
String mensagem = messageSource.getMessage(e,
localeContextHolder.getLocaLe());
```

COPIAR CÓDIGO

Agora falta só pegar o `dto`, que é minha lista de erros, e adicionar uma mensagem de erro `add(erro)`.

```
    List<FieldError> fieldErrors =
exception.getBindingResult().getFieldErrors();
    fieldErrors.forEach(e -> {
            String mensagem = messageSource.getMessage(e,
localeContextHolder.getLocaLe());
        ErroDeFormularioDto erro = new
ErroDeFormularioDto(e.getField(), mensagem);
                dto.add(erro);
     });

     return dto;
  }
```

COPIAR CÓDIGO

Essa vai ser a lógica do nosso `ErroDeValidacaoHandler.java` . É um pouco estranha, mas a ideia do *Handler* é justamente isso. Pense que o *Handler* é um interceptador, em que estou configurando o Spring para, sempre que houver um erro - alguma *exception* em algum método de qualquer `Controller` - ele chamar automaticamente o interceptador, passando a lista com todos os erros que aconteceram. Sendo assim, eu pego essa lista e transformo no meu objeto `ErroDeFormularioDto` , que só tem o nome do campo e a mensagem para simplificar.

[12:33] Pronto. É só isso. Você só precisa criar esse *Handler*. Não precisa mexer nos `Controllers` . Automaticamente o Spring sabe que é para chamar esse erro em qualquer `Controller` quando houver um erro de validação.

[12:47] Vamos testar no Postman. Ao abrir o Postman, encontrei aquela útlima requisição, a que tinha disparado na última aula e que tem um JSON do Spring com inúmeras informações. Vou disparar novamente e agora volta o JSON só com campo e nome da mensagem. Não vem aquele JSON gigantesco.

```
[
    {
            "campo": "titulo",
                "erro": "tamanho deve estar entre 5 e 214783647"
        },
        {
            "campo": "titulo",
                "erro": "não pode estar vazio"
        }
]
```

COPIAR CÓDIGO

Então, nesse meu exemplo eu mandei o título vazio, `"titulo":""` . Na resposta, portanto, aparece que o erro é no campo título, em que o `"tamanho deve estar er 5 e 2147483647"` . Um outro erro, no próprio título, é que ele `"não pode estar`

`vazio"` . Esses são os dois problemas: estar vazio e, por isso, ser menor que 5 caracteres. Logo, funcionou corretamente.

[13:38] Só para mostrar o esquema da internacionalização, no centro esquerdo da tela, vamos localizar a aba "Headers". Lembram que mandamos um Header que é o Content-Type, para dizer que estamos mandando no JSON? Podemos adicionar outro cabeçalho. Existe um chamado "Accept-Language", para dizermos qual é a linguagem que nós aceitamos como resposta. No campo a frente, "Value", posso passar "en-US", para dizer que é em inglês americano.

[14:08] Se eu mandar esse cabeçalho, o Spring detecta que o cabeçalho está vindo, e por causa da classe `LocaleContextHolder.getLocaLe()` , ele sabe que é para pegar as mensagens em inglês. Vamos fazer o teste pressionando "Send" e disparando de novo a requisição. Teremos:

```
[
    {
            "campo": "titulo",
                "erro": "lenght must be between 5 and 2147483647"
        },
        {
            "campo": "titulo",
                "erro": "must not be empty"
        }
]
```

COPIAR CÓDIGO

[14:48] Com isso, atingimos nosso objetivo: simplificamos o JSON que é devolvido para o cliente no caso de uma requisição inválida com valores dos parâmetros enviados pelo cliente com valores inválidos. Na próxima aula vamos complementar nossa API fazendo a lógica para detalhar, excluir, alterar um tópico que foi cadastrado.