02. Lidando com erros

Date	@31/01/2023
Categoria	Java API
Curso	Spring Boot 3: aplique boas práticas e proteja uma API Rest

Tópicos

- Projeto da aula anterior
- Lidando com erros na API
- Para saber mais: propriedades do Spring Boot
- Tratando erro 404
- Tratando erro 400
- Tratamento de exceptions
- Para saber mais: personalizando mensagens de erro
- Faça como eu fiz: RestControllerAdvice
- O que aprendemos?

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.

Tratando erro 400

Indica que o servidor não conseguiu processar uma requisição por erro de validação nos dados enviados pelo cliente.

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() {
    }

}
```

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.
- A classe não foi anotada com `@Configuration`.
    - Não é necessário adicionar essa anotação em classes de tratamento de *exceptions*.
- 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`.
- A *exception* passada na anotação `@ExceptionHandler` é do tipo ***unchecked***.

- O tratamento de *exceptions* do Spring funciona tanto para *checked exceptions*, quanto para *unchecked exceptions*.

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) {}
```

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
```

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) {}
```

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*.