TRATAMENTO DE EXCEÇÕES

Linguagem Orientada a Objetos

Prof. Rafael Tápia

O que é uma Exceção?

Uma **exceção** é um **problema** que ocorre durante a execução de um programa.

Exemplos:

- Divisão por zero
- Acesso a arquivos inexistentes,
- Erro de conexão ao banco de dados.

Por que tratar exceções?

Evitar que o programa quebre inesperadamente.

Garantir que o sistema **responda de forma adequada** ao usuário ou ao sistema externo.

Manter a estabilidade do código e permitir recuperação de erros.

Bloco try/catch

O bloco **try/catch** é utilizado para capturar e tratar exceções que podem ocorrer em uma parte do código.

try: Parte do código onde o erro pode acontecer.

catch: Define como a exceção será tratada, caso aconteça.

```
public void buscarClientePorId(String input) {
    try {
        int id = Integer.parseInt(input);
    } catch (NumberFormatException e) {
        System.out.println("Erro: o ID fornecido não é um número válido.");
    }
}
```

Usando throws em Métodos

Quando um método pode gerar uma exceção, usamos **throws** para indicar isso no cabeçalho do método. Essa indicação faz que o método devolva o erro para quem o chamou.

```
public Cliente buscarPorId(Integer id) throws Exception {
    Cliente cliente = clienteRepository.findById(id);
    if (cliente == null) {
        throw new Exception("Cliente não encontrado");
    }
    return cliente;
}
```

Exceções Personalizadas

Vamos criar uma exceção personalizada chamada **ClienteNaoEncontradoException** para tratar casos onde o cliente não é encontrado.

```
public class ClienteNaoEncontradoException extends Exception {
    public ClienteNaoEncontradoException(String message) {
        super(message);
    }
}
```

Uma exceção personalizada facilita o tratamento de casos específicos, como "Cliente não encontrado".

RETORNOS HTTP

RETORNOS HTTP

Em aplicações web, quando um cliente (navegador ou outro sistema) faz uma requisição, o servidor responde com um **código de status HTTP**.

Esses códigos são importantes para indicar se a operação foi bem-sucedida ou se houve algum problema.

Vamos entender como isso funciona de forma simples.

O que são os códigos HTTP?

Os códigos HTTP são números que indicam o resultado de uma requisição feita ao servidor. Eles são divididos em categorias, e cada um tem um significado específico.

Categorias de Códigos HTTP:

1. Códigos 2xx (Sucesso):
   200 OK: A requisição foi bem-sucedida e o servidor retornou o que era esperado.

2. Códigos 4xx (Erros do Cliente):
   404 Not Found: O recurso solicitado não foi encontrado.
   400 Bad Request: O servidor não entendeu a requisição.

3. Códigos 5xx (Erros do Servidor):
   500 Internal Server Error: Ocorre quando algo deu errado no servidor.

Por que os códigos HTTP são importantes?

Os códigos HTTP são essenciais para a comunicação clara entre o cliente e o servidor. Eles:

- **Informam ao cliente o que aconteceu**: Um código 200 indica que tudo correu bem, enquanto um 404 informa que o recurso solicitado não foi encontrado.
- **Ajudam na depuração**: Se você está desenvolvendo ou usando um sistema, os códigos de status podem ajudar a identificar problemas. Se você receber um 500, sabe que houve um erro interno no servidor.
- **Facilitam a padronização**: Todos os sistemas seguem essa convenção, o que facilita a comunicação entre diferentes aplicações e serviços.

Exemplo no código:

Aqui vamos ver como usamos os códigos HTTP no nosso projeto de "Clientes" com Spring Boot. Queremos garantir que o cliente receba uma resposta clara dependendo do que acontece.

```
@RestController
public class ClienteController {

    @GetMapping("/clientes/{id}")
    public ResponseEntity<Cliente> buscarCliente(@PathVariable Integer id) {
        try {
            Cliente cliente = clienteService.buscarPorId(id);
            return ResponseEntity.ok(cliente);
        } catch (ClienteNaoEncontradoException e) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).body(null);
        } catch (Exception e) {
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(null);
        }
    }
}
```

Explicação do código:

1. Se o cliente for encontrado no banco de dados, retornamos um **200 OK** com os dados do cliente.
2. Se o cliente não for encontrado, retornamos um **404 Not Found**, indicando que o recurso não existe.
3. Se ocorrer algum erro inesperado no servidor, retornamos um **500 Internal Server Error**, informando que houve um problema interno.

DICAS

5. Como o JPA Funciona?

- Inserir ou atualizar esse cliente no banco de dados.
- Sempre use o código correto para a situação. Isso melhora a comunicação com o cliente ou outro sistema que consome sua API.
- Para operações bem-sucedidas, como uma busca ou inserção, use o 200 OK.
- Para erros de recursos não encontrados, use 404 Not Found.
- Para erros de validação ou requisições incorretas, use 400 Bad Request.
- Para problemas internos, use 500 Internal Server Error.

EXERCÍCIOS

Validação de E-mail no Cadastro de Clientes

Implementar uma validação de e-mail antes de salvar o cliente no banco de dados.

Caso o e-mail seja inválido, um erro personalizado será lançado, e o sistema deverá retornar um código de erro HTTP apropriado.

Objetivo:

- Validar o formato do e-mail antes de salvar um cliente no banco de dados.
- Lançar uma exceção personalizada caso o e-mail seja inválido.
- Tratar a exceção no controller e retornar um código HTTP adequado.

Passos:

Validação de E-mail:

- Antes de salvar o cliente, verifique se o e-mail informado está no formato correto.
- Utilize a função de validação fornecida (próximo slide) para verificar se o e-mail é válido.

Lançar Erro:

- Caso o e-mail seja inválido, lance uma exceção personalizada chamada EmailInvalidoException.

Retorno de Erro HTTP:

- No controller, capture a exceção EmailInvalidoException e retorne um código HTTP 400 Bad Request com uma mensagem de erro adequada.

Função para Validar E-mail:

Use a seguinte função para validar o e-mail no serviço:

```
public boolean emailValido(String email) {
    String emailRegex =
"^[a-zA-Z0-9_+&*-]+(?:\\.[a-zA-Z0-9_+&*-]+)*@(?:[a-zA-Z0-9-]+\\.)+[a-zA-Z
]{2,7}$";
    return email != null && email.matches(emailRegex);
}
```

Exceção Personalizada:

Crie uma exceção personalizada chamada EmailInvalidoException:

```
public class EmailInvalidoException extends RuntimeException {
    public EmailInvalidoException(String mensagem) {
        super(mensagem);
    }
}
```

Modificação no Serviço de Cliente:

No serviço de cliente, adicione a validação de e-mail antes de salvar o cliente:

```
public Cliente salvarCliente(Cliente cliente) {
    if (!emailValido(cliente.getEmail())) {
        throw new EmailInvalidoException("O e-mail informado é inválido:
" + cliente.getEmail());
    }
    return clienteRepository.save(cliente);
}
```

Tratamento de Exceção no Controller:

No controller, capture a exceção **EmailInvalidoException** e retorne um erro **HTTP 400 Bad Request**:

```
@PostMapping("/clientes")
public ResponseEntity<Cliente> salvarCliente(@RequestBody Cliente cliente) {
    try {
        Cliente clienteSalvo = clienteService.salvarCliente(cliente);
        return ResponseEntity.status(HttpStatus.CREATED).body(clienteSalvo);
    } catch (EmailInvalidoException e) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(null);
    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(null);
    }
}
```

DICAS DO EXERCÍCIO

Expressão Regular: A expressão regular fornecida para a função emailValido() verifica padrões comuns de e-mails. Caso deseje, você pode modificá-la para ser mais rigorosa ou flexível.

Testes: Faça testes para garantir que e-mails válidos são aceitos e e-mails inválidos são rejeitados com a mensagem de erro apropriada.

Erro HTTP 400: Este código é utilizado para indicar que a requisição enviada pelo cliente é inválida, como no caso de um e-mail mal formatado.