



INSTITUTO FEDERAL
Triângulo Mineiro
Campus Uberlândia Centro

Projeto Backend

Microserviços e NoSQL

Finalizando a Implementação de UserAPI

Prof. Lucas Montanheiro
lucasmontanheiro@iftm.edu.br

Buscando usuário por CPF

- Com a aula anterior, já temos um serviço que retorna uma lista de usuários, vamos criar agora um que recebe o identificador de um usuário e retorna apenas um usuário específico.
- Esse serviço também utiliza o verbo HTTP **GET**, com a diferença de que ele recebe um parâmetro na URL para filtrar o usuário a ser retornado.
- Para implementar esse método, a mesma lista de usuários vai ser utilizada, porém, agora, o retorno é apenas um dos usuários da lista ou uma exceção dizendo que o usuário não foi encontrado.

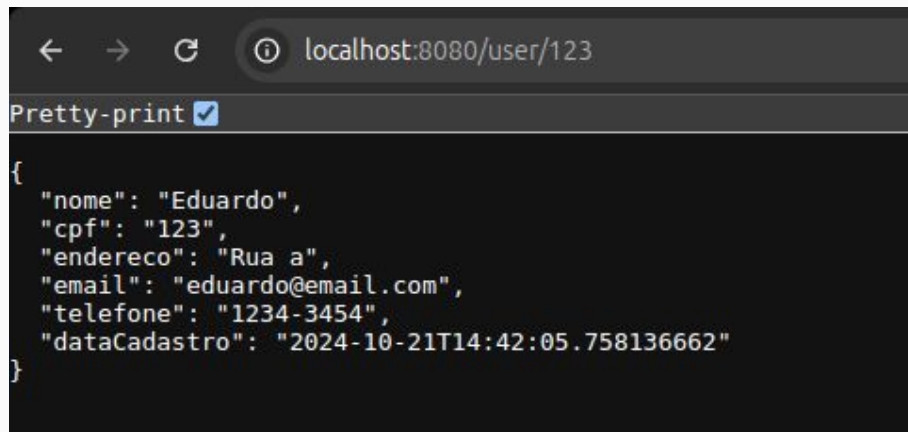
Buscando usuário por CPF

- Em nosso UserController, vamos adicionar a busca por o parâmetro na URL.
- Para isso vamos colocar o valor **{cpf}** na definição da rota e também adicionar a anotação **@PathVariable** no parâmetro **cpf**.
- Para funcionar corretamente tanto o valor na URL quanto o parâmetro tem que possuir o mesmo nome.

```
30 @GetMapping("/{cpf}")
31 public UserDTO getUsersFiltro(@PathVariable String cpf) {
32     return usuarios
33         .stream()
34         .filter(userDTO -> userDTO.getCpf().equals(cpf))
35         .findFirst()
36         .orElseThrow(() -> new RuntimeException("User not found."));
37 }
```

Buscando usuário por CPF

- Esse serviço terá duas possíveis respostas: a primeira é caso o método encontre algum usuário com o CPF utilizado como filtro.
- Por exemplo, se a chamada para o serviço for **http://localhost:8080/user/123**, a resposta será:



A screenshot of a web browser window. The address bar shows the URL `localhost:8080/user/123`. Below the address bar, there is a checkbox labeled "Pretty-print" which is checked. The main content area of the browser displays a JSON object representing a user record.

```
{
  "nome": "Eduardo",
  "cpf": "123",
  "endereco": "Rua a",
  "email": "eduardo@email.com",
  "telefone": "1234-3454",
  "dataCadastro": "2024-10-21T14:42:05.758136662"
}
```

Buscando usuário por CPF

- A segunda é caso nenhum usuário seja encontrado.
- Por exemplo, para a chamada **http://localhost:8080/user/000**, a resposta será um erro indicando que nenhum usuário foi encontrado.
- A mensagem do erro não é a melhor possível, pois ela não indica claramente qual o problema. Veremos em breve como melhorar essa mensagem.



Adicionando um novo Usuário

- O próximo serviço salvará as informações de um novo usuário na lista de usuários.
- É possível utilizar um método **GET** para enviar dados para o servidor, porém isso não é recomendável.
- O correto é enviar os dados em uma requisição **POST** e no corpo da requisição, não na URL.

Adicionando um novo Usuário

- Em um método **POST**, a anotação utilizada é a **@PostMapping**, também recebendo como parâmetro o caminho para a requisição.
- Para o método receber dados no corpo da requisição, é necessário utilizar a anotação **@RequestBody**.
- O seguinte método usa essas anotações para receber as informações de um usuário e inseri-lo na lista:

```
39     @PostMapping
40     @ResponseStatus(HttpStatus.CREATED)
41     public UserDTO inserir(@RequestBody UserDTO userDTO) {
42         userDTO.setDataCadastro(LocalDateTime.now());
43         usuarios.add(userDTO);
44         return userDTO;
45     }
```

Atenção aos Códigos HTTP

- Um outro fator importante desse método é o código de retorno.
- No REST, um dos dados que vão na resposta para o usuário é o código de retorno de uma rota.
- Por padrão, esse código é o **200 OK**, mas podemos mudar esse retorno dependendo do tipo de método.
- Normalmente um método POST retorna um código **201 CREATED**, que indica que um novo recurso foi criado no servidor.
- A definição do código pode ser feita usando a anotação **@ResponseStatus**.

Códigos HTTP

- Na resposta HTTP, um dado bastante importante é o código da resposta.
- Esse código indica resumidamente se a requisição foi tratada corretamente e o que aconteceu no servidor.
- Ela é muito utilizada para a comunicação entre serviços, para que o cliente saiba o que fazer depois de chamar o serviço.

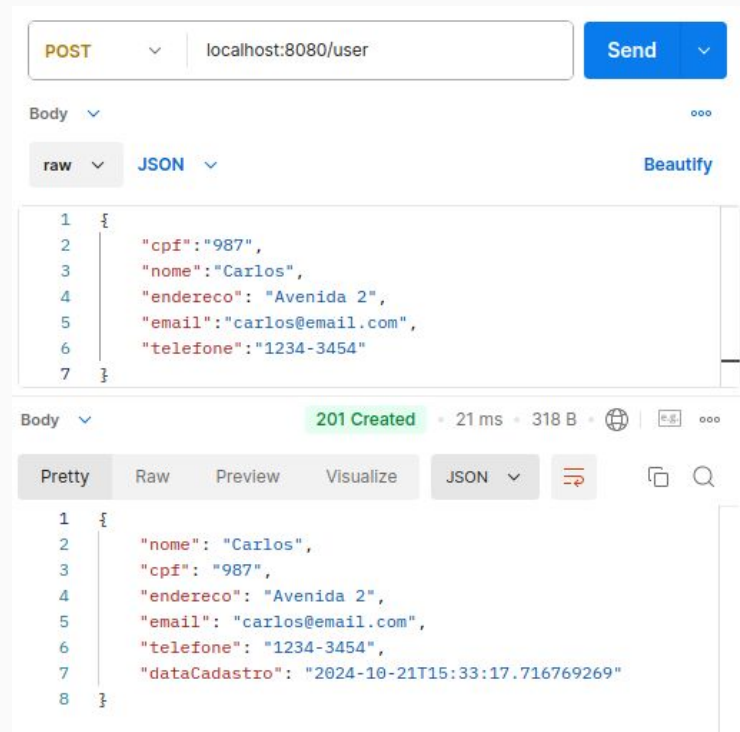
Códigos HTTP

Os erros são divididos por faixa, que são:

- **200:** indica que a requisição foi tratada corretamente. Alguns dos códigos mais utilizados são o 200 OK , 201 CREATED , 202 ACCEPTED e 204 NO_CONTENT
- **300:** indica um redirecionamento. Os códigos mais conhecidos nessa faixa são o 301 MOVED_PERMANENTLY e o 308 PERMANENT_REDIRECT
- **400:** indica erros do cliente (por exemplo, o cliente enviou dados no formato errado). Alguns dos códigos mais utilizados são o 400 BAD_REQUEST , 401 UNAUTHORIZED , 404 NOT_FOUND e 408 TIMEOUT .
- **500:** indica que o servidor está com algum problema e não pode tratar a requisição. Os erros mais conhecidos nesta faixa são o 500 INTERNAL_SERVER_ERROR e 502 BAD_GATEWAY .

Adicionando um novo Usuário

- Para testar uma requisição utilizando o verbo HTTP **POST** é necessário utilizar alguns programas específicos, como: Postman, Insomnia ou a Extensão do VSCode Thunder Client.
- Basta configurar a requisição como **POST** e passar as informações de usuário no corpo da requisição, utilizando o formato **JSON**.



Adicionando um novo Usuário

- Note que a data de cadastro não foi passada no JSON, pois ela é preenchida automaticamente com a data do servidor no método inserir.
- Se chamarmos novamente o método que lista todos os usuários da lista, receberemos a seguinte resposta agora:

```
1  [
2    {
3      "nome": "Eduardo",
4      "cpf": "123",
5      "endereco": "Rua a",
6      "email": "eduardo@email.com",
7      "telefone": "1234-3454",
8      "dataCadastro": "2024-10-21T15:31:26.943654741"
9    },
10   {
11     "nome": "Luiz",
12     "cpf": "456",
13     "endereco": "Rua b",
14     "email": "luiz@email.com",
15     "telefone": "1234-3454",
16     "dataCadastro": "2024-10-21T15:31:26.943705002"
17   },
18   {
19     "nome": "Bruna",
20     "cpf": "678",
21     "endereco": "Rua c",
22     "email": "bruna@email.com",
23     "telefone": "1234-3454",
24     "dataCadastro": "2024-10-21T15:31:26.943727673"
25   },
26   {
27     "nome": "Carlos",
28     "cpf": "987",
29     "endereco": "Avenida 2",
30     "email": "carlos@email.com",
31     "telefone": "1234-3454",
32     "dataCadastro": "2024-10-21T15:33:17.716769269"
33   }
34 ]
```

Adicionando um novo Usuário

- Nesse tipo de rota, outro requisito importante é fazer a validação dos dados de entrada.
- Por exemplo, podemos definir que o nome, o CPF e o e-mail são dados obrigatórios para o cadastro de usuário, mas do jeito como a rota foi implementada, isso não está sendo validado.
- O Spring possui um mecanismo bastante simples para fazer esse tipo de validação básica.
- Podemos fazer isso diretamente na classe **UserDTO** apenas adicionando a anotação **@NotBlank**.

Adicionando um novo Usuário

- Nesse tipo de rota, outro requisito importante é fazer a validação dos dados de entrada.
- Por exemplo, podemos definir que o **nome**, o **CPF** e o **e-mail** são dados obrigatórios para o cadastro de usuário, mas do jeito como a rota foi implementada, isso não está sendo validado.

Adicionando um novo Usuário

- Inclua a seguinte dependência:

```
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-validation</artifactId>  
</dependency>
```

Adicionando um novo Usuário

- O Spring possui um mecanismo bastante simples para fazer esse tipo de validação básica.
- Podemos fazer isso diretamente na classe **UserDTO** apenas adicionando a anotação **@NotBlank**.
- Para funcionar é necessário ter a importação da dependência:

```
import jakarta.validation.constraints.NotBlank;
```

```
12 @Getter
13 @Setter
14 @NoArgsConstructor
15 @AllArgsConstructor
16 public class UserDTO {
17
18     @NotBlank(message = "Nome é obrigatório")
19     private String nome;
20     @NotBlank(message = "CPF é obrigatório")
21     private String cpf;
22     private String endereco;
23     @NotBlank(message = "E-mail é obrigatório")
24     private String email;
25     private String telefone;
26     private LocalDateTime dataCadastro;
27
28 }
```


Adicionando um novo Usuário

- Outra pequena mudança que deve ser feita para a validação funcionar é adicionar a anotação **@Valid** no parâmetro que recebe os dados na rota **POST**.
- Isso deve ser feito para indicar para o Spring que as validações que estão definidas no DTO devem ser realizadas antes da execução do método do controlador.
- Para funcionar é necessário ter a importação da dependência:

```
import jakarta.validation.Valid;
```

```
41     @PostMapping
42     @ResponseStatus(HttpStatus.CREATED)
43     public UserDTO inserir(@RequestBody @Valid UserDTO userDTO) {
44         userDTO.setDataCadastro(LocalDateTime.now());
45         usuarios.add(userDTO);
46         return userDTO;
47     }
```

Adicionando um novo Usuário

- Caso algum dado esteja incorreto na chamada para a rota, o retorno será uma mensagem indicando que alguma coisa de errado aconteceu.
- Aqui, a resposta ainda não está muito boa também, pois ela não indica exatamente o que aconteceu de errado, mais para a frente, veremos como melhorar essa mensagem.

The screenshot shows a REST client interface with the following details:

- Method:** POST
- URL:** localhost:8080/user
- Body (Request):**

```
{  "nome": "Carlos",  "endereço": "Avenida 2",  "email": "carlos@email.com",  "telefone": "1234-3454"}
```
- Response:** 400 Bad Request (211 ms, 239 B)
- Response Body (JSON):**

```
{  "timestamp": "2024-10-21T19:23:28.081+00:00",  "status": 400,  "error": "Bad Request",  "path": "/user"}
```

Excluindo um Usuário

- O último serviço será para excluir um usuário da lista e será bem parecido com o serviço de busca.
- Ele também receberá um CPF como parâmetro na URL e fará uma busca pelo usuário.
- Caso ele seja encontrado, ele será removido da lista e o serviço retornará **true**; caso contrário, retornará **false**.

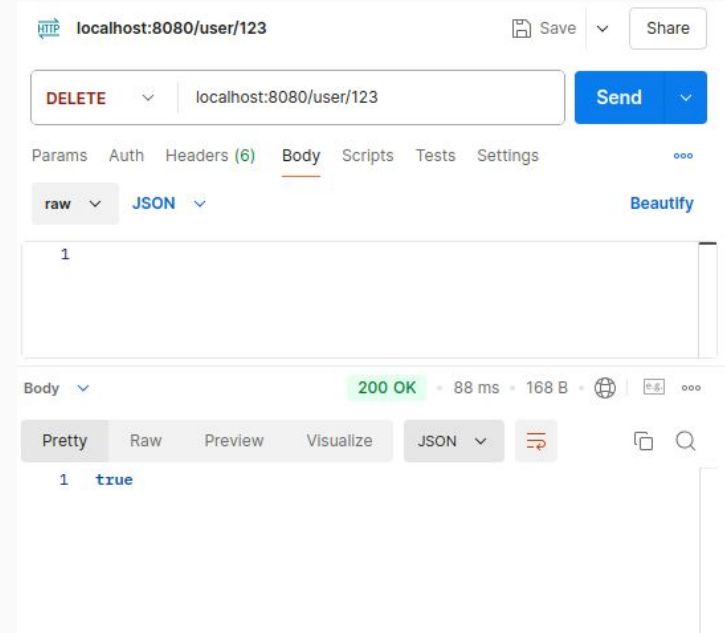
Excluindo um Usuário

- Uma diferença é que esse método usará o verbo **DELETE** do protocolo HTTP.
- A anotação **@DeleteMapping** cria um serviço com o verbo **DELETE** e a anotação **@PathVariable** funciona da mesma forma que no serviço **GET** anterior, o CPF será passado para o serviço na URL.

```
49     @DeleteMapping("/{cpf}")
50     public boolean remover(@PathVariable String cpf) {
51         return usuarios
52             .removeIf(userDTO -> userDTO.getCpf().equals(cpf));
53     }
```

Excluindo um Usuário

- Ao fazer um chamada, utilizando o verbo **DELETE** e passando como parâmetro o usuário com CPF **123**, ele será excluído o resultado será **true**.
- Exemplo: **http://localhost:8080/user/123**



Excluindo um Usuário

- Com essas implementações temos nossa primeira API já funcional;
- Ainda falta bastante trabalho para a API ficar pronta para produção;
- Chegou a hora de integrar com o Banco de Dados, utilizando Spring Data.