



Introdução a Web Services REST com Spring Framework

Parte 2 → Verbos HTTP

Prof. Me. Jorge Luís Gregório

www.jlgregorio.com.br



Agenda

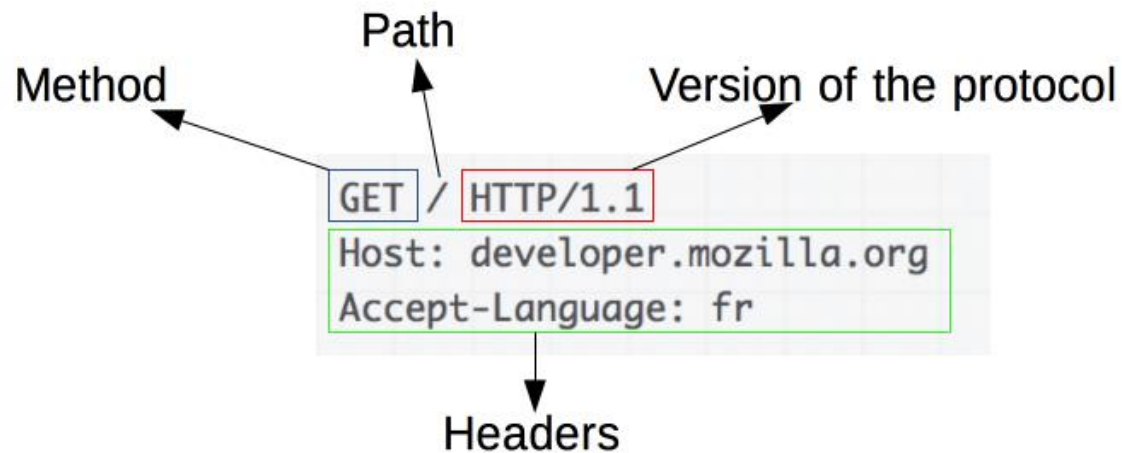
- Códigos de *status* do protocolo HTTP
- Sobre os verbos HTTP
 - GET
 - POST
 - PUT
 - PATCH
 - DELETE
- Criando um CRUD (sem banco de dados)

A large, stylized green leaf graphic is positioned on the left side of the slide, partially overlapping the green box. It has a light green fill and a darker green outline, with a pointed tip and a curved shape.

VERBOS HTTP

Verbos, comandos ou operações?

- Na literatura, é usado o termo **verbo HTTP**, porém dependendo da documentação da tecnologia utilizada, é possível que você se depare com os termos **comandos HTTP** ou **operações HTTP**; Na prática, todos são a mesma coisa;



O verbo GET

- Usado para operações de recuperação ou leitura de dados;
- É o verbo mais comum, pois a maioria das requisições em uma aplicação distribuída é do tipo “traga uma informação pra mim”;
- No melhor cenário, um serviço REST que recebe uma requisição GET retorna um conteúdo em formato XML ou JSON, além de um *status code* 200 (OK);
- Em um cenário que possui erros, o retorno mais comum é o status 404 (***not found***) ou 400 (***bad request***);
- Suporta parâmetros via **URL** (*path params* ou *query params*) e via **HEADER** (cabeçalho);
- Único verbo que não suporta parâmetros via BODY,

O verbo POST

- O verbo **POST** é usado para a criação de um novo recurso;
- **Exemplo:** inserir um novo registro na base de dados; enviar um arquivo ao servidor (upload);
- No cenário ideal, quando a operação requisitada é executada com sucesso, os status codes retornados são 200 (**OK**) ou 201 (**CREATED**);
- Suporta parâmetros via URL (path params e query params), via HEADER e via BODY (recomendado via body)

O verbo PUT

- O verbo **PUT** é usado para modificar ou atualizar um recurso já existente;
- **Exemplo:** operações de *update* em um banco de dados;
- É colocado um recurso existente no BODY (ou corpo da requisição) contendo novas informações. Assim o recurso original é atualizado.
- No cenário ideal, quando a operação requisitada é executada com sucesso, o *status code* retornado é 200 (**OK**) ou 204 (**NO CONTENT**);
- Quando não retorna nenhum conteúdo, o status code é **204**;
- Recomenda-se retornar no body os valores atualizados;
- Suporta parâmetros via URL (*path params* e *query params*), via HEADER e via BODY (recomendado via body)

O verbo DELETE

- O verbo DELETE é usado para excluir um recurso identificado por uma URI (*Uniform Resource Identifier*);
- **Exemplo:** excluir um registro na base de dados ou arquivo no servidor;
- Quando a operação é realizada com sucesso, o *status code* retorno é 200 (OK);
- Também é possível retornar no *response body* o item deletado – não recomendado;
- Recomenda-se retornar o status code 200, confirmando que a operação foi realizada, ou 204 (**NO CONTENT**);
- Suporta parâmetros via URL (*path params* e *query params*), via HEADER e via BODY (recomendado via body)
- Recomenda-se informar via *path param* o identificador do recurso a ser removido.

VERBOS MENOS USUAIS

- O verbo **PATCH** é usado para atualizar parcialmente um recurso
 - **Exemplo:** atualizar parcialmente um registro na base de dados, considerando apenas alguns campos – enviar o objeto inteiro pode acarretar uso excessivo de banda;
 - Pode acontecer colisões entre múltiplas operações de **PATCH**, quando tentam atualizar o mesmo recurso. Isso pode corromper o recurso utilizado ou gerar problemas de consistência;
- O verbo **HEAD** é similar ao verbo GET, porém a resposta contém apenas uma *response line* e *headers*, sem um *body*.
- O verbo **TRACE** é usado para recuperar o conteúdo de uma requisição, sendo possível usá-lo para *debugar* a aplicação;
- O verbo **OPTIONS** é usado pelos clientes para descobrir quais operações HTTP são suportadas pelo servidor.
- Verbo **CONNECT** é usado para estabelecer uma conexão persistente com um servidor.

REST - http verbs



Client



Retrieve all users

Create a user

Retrieve one User

Update a user

Delete a User

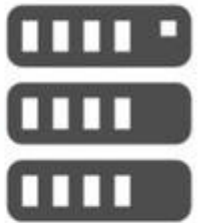
⇒ GET /users

⇒ POST /users

⇒ GET /users/{id}

⇒ PUT /users{id}

⇒ DELETE /users/{id}



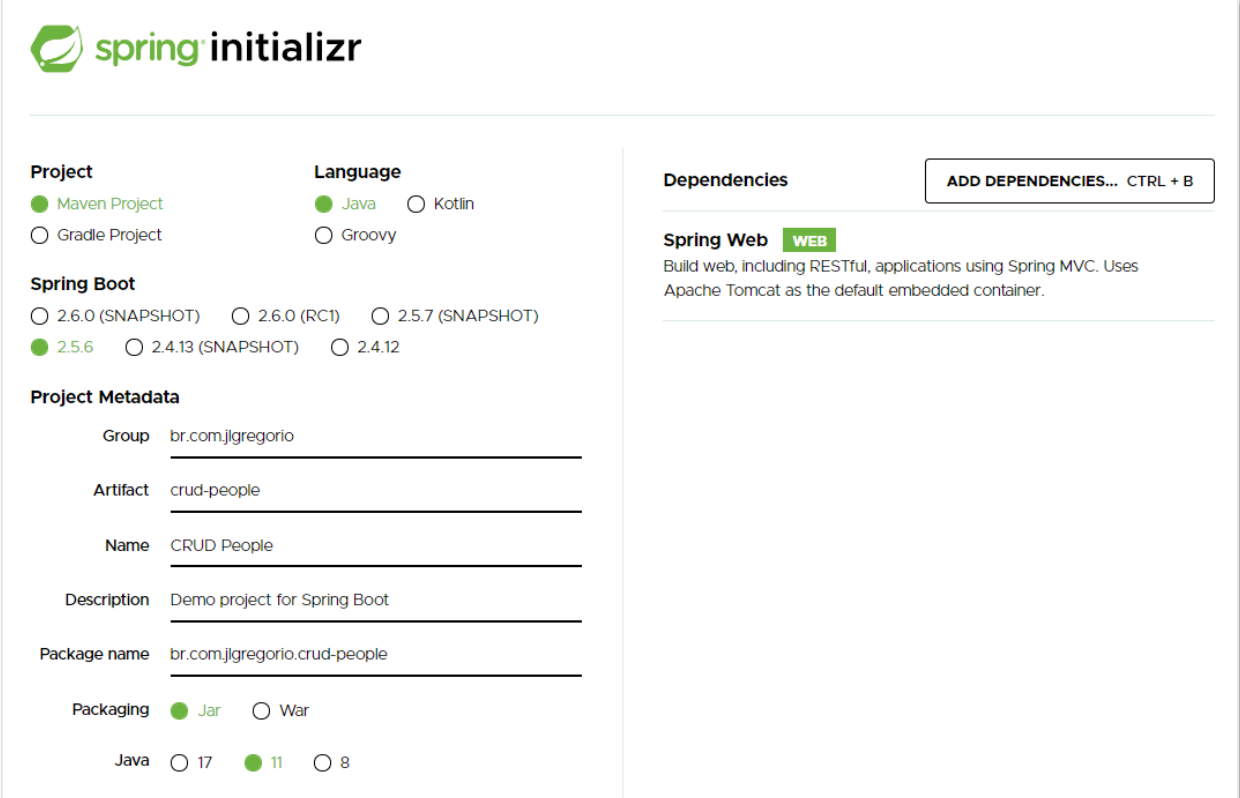
Server

A large, stylized green leaf graphic is positioned on the left side of the image, partially overlapping the green rectangular box. The leaf is light green with a darker green outline and a central vein.

CRIANDO UM PROJETO CRUD

Criando o projeto via Spring Initializr

- Abra o Spring Initializr e crie um novo projeto de acordo com as configurações ao lado da figura ao lado:
 - Maven Project
 - Java
 - Spring Boot 2.5.6
 - Packaging: Jar
 - Java: 11
- Vamos implementar uma calculadora simples, ao mesmo tempo em que aprendemos novos conceitos sobre Spring, REST e padrões de projeto. Bora!



The screenshot shows the Spring Initializr web interface. The 'Project' section has 'Maven Project' selected. The 'Language' section has 'Java' selected. The 'Spring Boot' section has '2.5.6' selected. The 'Project Metadata' section has 'Group' as 'br.com.jlgregorio', 'Artifact' as 'crud-people', 'Name' as 'CRUD People', 'Description' as 'Demo project for Spring Boot', and 'Package name' as 'br.com.jlgregorio.crud-people'. The 'Packaging' section has 'Jar' selected. The 'Java' version section has '11' selected. The 'Dependencies' section has 'Spring Web' selected, with a description: 'Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.' There is an 'ADD DEPENDENCIES... CTRL + B' button.

Project

☒ Maven Project
☐ Gradle Project

Language

☒ Java ☐ Kotlin
☐ Groovy

Spring Boot

☐ 2.6.0 (SNAPSHOT) ☐ 2.6.0 (RC1) ☐ 2.5.7 (SNAPSHOT)
☒ 2.5.6 ☐ 2.4.13 (SNAPSHOT) ☐ 2.4.12

Project Metadata

Group

Artifact

Name

Description

Package name

Packaging

☒ Jar ☐ War

Java

☐ 17 ☒ 11 ☐ 8

Dependencies ADD DEPENDENCIES... CTRL + B

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Criando a model

- Crie um pacote model no pacote principal do projeto;
- Crie uma classe Java chamada ***Person***;
- Adicione os construtores (padrão e com parâmetros);
- Adicione os métodos *getters* e *setters*;
- Adicione os métodos *hashCode* e *equals*, gerados automaticamente pela IDE.

```
1  package br.com.jlgregorio.crudpeople.model;
2  import ...
3
4
5  public class Person implements Serializable {
6
7      private long id;
8      private String firstName;
9      private String lastName;
10     private String profession;
11
12     public Person() {
13     }
14
15     public Person(long id, String firstName, String lastName, String profession) {
16         this.id = id;
17         this.firstName = firstName;
18         this.lastName = lastName;
19         this.profession = profession;
20     }
21 }
```

Adicionando a camada Service

- Dentro do pacote principal do projeto, crie um novo pacote chamado **services**;
- Agora crie uma nova classe chamada **PersonServices** (Código ao lado)
- A camada Service é usada para desacoplar o **model** do **controller**, portanto, use a anotação **@Service** para definir essa classe;
- Isso possibilitará que ela seja injetada no controlador, sem a necessidade de ser instanciada;

```
// defines as a Service - Dependency Injection
@Service
public class PersonServices {
    // an id generator
    public final AtomicLong genId = new AtomicLong(initialValue: 3);

    // a list of people
    private List<Person> people;

    // mocking people
    private List<Person> mockPeople(){
        List<Person> list = new ArrayList<>();
        Person person = new Person(id: 1, firstName: "Uzumaki", lastName: "Naruto", profession: "Hokage");
        list.add(person);
        person = new Person(id: 2, firstName: "Shikamaru", lastName: "Nara", profession: "Ninja");
        list.add(person);
        person = new Person(id: 3, firstName: "Sasuke", lastName: "Uchiha", profession: "Ninja");
        list.add(person);
        return list;
    }
}
```

Continuação da classe Service...

- Os métodos mostrados no Código ao lado servem respectivamente para:
- Construtor;
- Encontrar uma pessoa pelo id;
- Retornar todas as pessoas;

```
// mocking on construct
public PersonServices(){
    people = mockPeople();
}

// method to find a Person using the id
public Person findById(String id) {
    List<Person> list = mockPeople();
    Person person = null;
    for (Person p : list
        ) {
        if(p.getId() == Long.parseLong(id)){
            person = p;
        }
    }
    return person;
}

// return all people
public List<Person> findAll(){
    return people;
}
```

Continuação da classe Service...

- Os métodos mostrados no Código ao lado servem respectivamente para:
- Criar uma nova pessoa e adicioná-la na lista;
- Remover uma pessoa pelo id

```
// create a new people
public Person create (Person person){
    person.setId(genId.incrementAndGet());
    people.add(person);
    return person;
}

// delete an object from list
public List<Person> delete(long id){
    people.removeIf(p → p.getId() == id);
    return people;
}
```


Finalização da classe Service...

- Os métodos mostrados no Código ao lado servem respectivamente para:
- Pegar o índice de uma Pessoa dentro da lista;
- Atualizar os dados de uma Pessoa.

```
// get the index of an object - useful to update
public int getIndex(Person person){
    int index = -1;
    for (Person p : people
        ) {
        if (p.getId() == person.getId()){
            index = people.indexOf(p);
        }
    }
    return index;
}

// update an object
public Person update(Person person){
    people.set(getIndex(person), person);
    return person;
}
```

A decorative graphic of a green leaf, partially visible on the left side of the slide, with a light green outline and a darker green fill.

**Usando o verbo GET para
recuperar informações**

Adicionando a camada Controller

- No pacote principal da aplicação, crie um pacote model;
- Crie uma classe chamada PersonController (Código ao lado);

```
//annotations here
@RestController //this is a Rest Controller
@RequestMapping("/people") //the request mapping starts with people
public class PersonController {

    //dependency injection with @Autowired
    @Autowired
    private PersonServices services;

    //request mapping using path params and expliciting request method and response
    @RequestMapping(value="/{id}",
                    method = RequestMethod.GET,
                    produces = MediaType.APPLICATION_JSON_VALUE)
    public Person findById(@PathVariable("id") String id){
        //using the services with dependency injection
        return services.findById(id);
    }
}
```

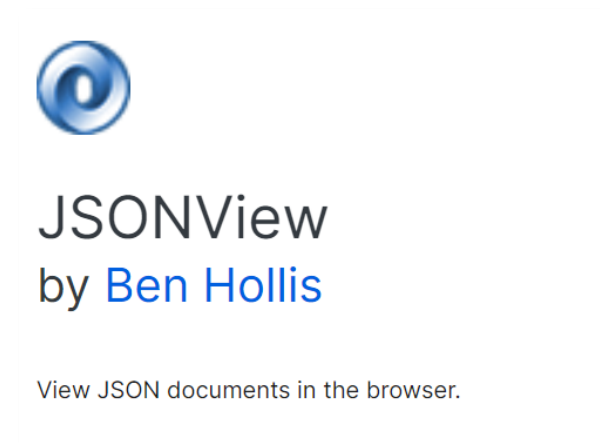
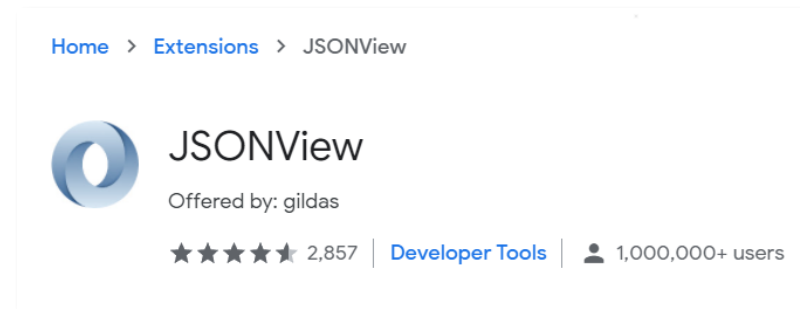
Testando a aplicação

- Abra o navegador em <http://localhost:8080/people/5>



Dica

- Instale a extensão **JSON View** no navegador Chrome ou no Firefox para que os arquivos JSON sejam exibidos de maneira mais organizada, como nos exemplo mostrados anteriormente.
- Chrome: <https://chrome.google.com/webstore/detail/jsonview/chklaanhfefbnpoihckbnefhakgolnmc?hl=en>
- Firefox: <https://addons.mozilla.org/en-US/firefox/addon/jsonview/>



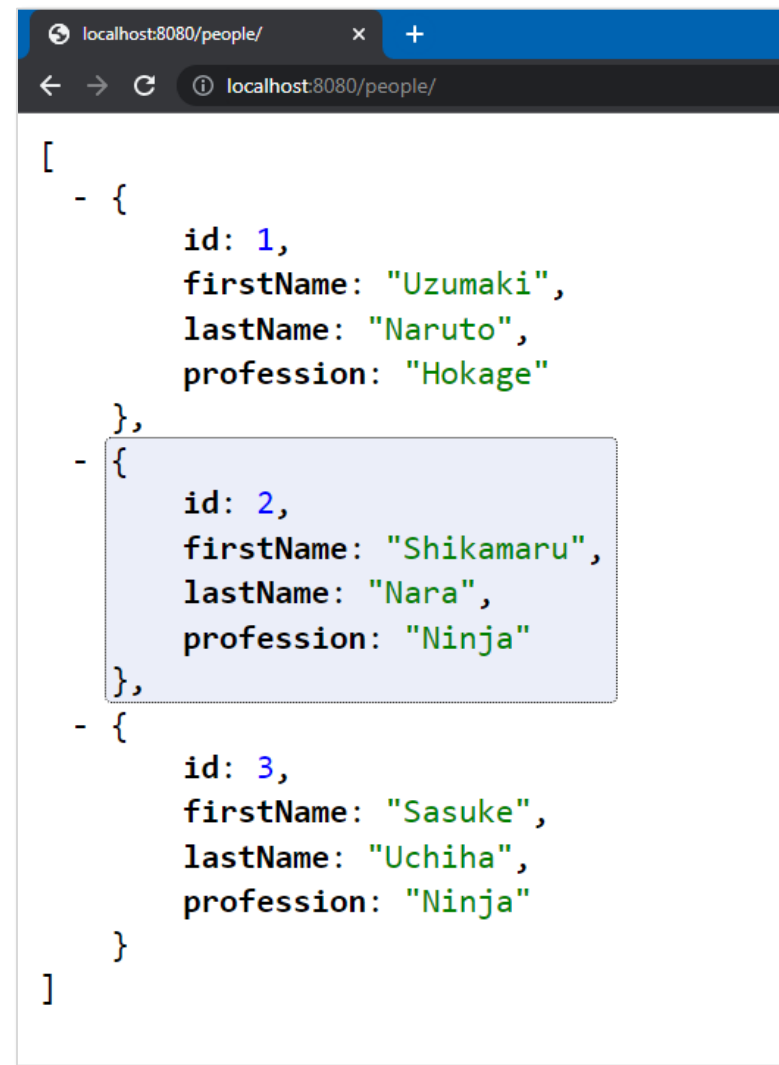
Cria o método findAll() no controlador

- Abra o **PersonController** e adicione o método ao lado;
- Note que o parâmetro value da anotação **@RequestMapping** é apenas uma barra, logo ela pode ser retirada, considerando que quando não há mapeamento explícito, o **RestController** assume o que está definido nele, nesse caso: “/people”

```
// request mapping using "/" to retrieve all people
@RequestMapping(value = "/",
    method = RequestMethod.GET,
    produces = MediaType.APPLICATION_JSON_VALUE)
public List<Person> findAll() { return services.findAll(); }
```

Testando a requisição

- Abra o navegador em <http://localhost:8080/people>



A screenshot of a web browser window with the address bar showing 'localhost:8080/people/'. The browser displays a JSON array of three objects, each representing a person. The first object is for Uzumaki Naruto (id: 1, profession: Hokage). The second object is for Shikamaru Nara (id: 2, profession: Ninja) and is highlighted with a light blue background. The third object is for Sasuke Uchiha (id: 3, profession: Ninja). The JSON is formatted with syntax highlighting.

```
[  
  - {  
    id: 1,  
    firstName: "Uzumaki",  
    lastName: "Naruto",  
    profession: "Hokage"  
  },  
  - {  
    id: 2,  
    firstName: "Shikamaru",  
    lastName: "Nara",  
    profession: "Ninja"  
  },  
  - {  
    id: 3,  
    firstName: "Sasuke",  
    lastName: "Uchiha",  
    profession: "Ninja"  
  }  
]
```

A large, stylized green leaf graphic is positioned on the left side of the slide, partially overlapping the green text box. It has a light green fill and a darker green outline, with a central vein and several smaller veins branching out.

Usando o verbo POST para criar recursos

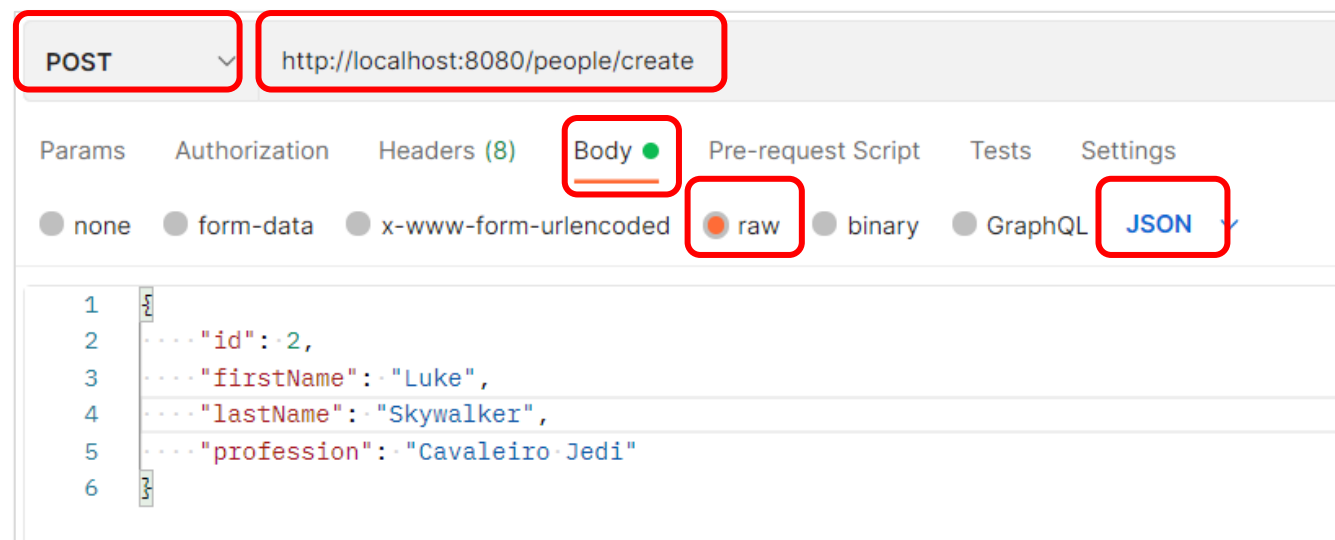
Criando o método create

- O método **create** vai simular a criação de um novo recurso;
- Agora abra **PersonController** e crie o método create junto com um **RequestMapping**, como no Código ao lado:
- Note que o método agora é do tipo **POST** e ele consome e produz **JSON**;
- O parâmetro do método é um **RequestBody**, ou seja, um objeto complexo, não sendo possível informá-lo via **GET**

```
// request mapping using "/", but with POST method - data passed by request body
@RequestMapping(method = RequestMethod.POST,
    consumes = MediaType.APPLICATION_JSON_VALUE,
    produces = MediaType.APPLICATION_JSON_VALUE)
public Person create(@RequestBody Person person){
    return services.create(person);
}
```

Testando via Postman

- O método POST não passa parâmetros via URL;
- Os dados são enviados em um **Request Body**;
- Para simular uma requisição POST, vamos usar o Postman;
 - Selecione o método POST;
 - A URL é <http://localhost:8080/people/create>
 - Abra a guia **Body**, depois raw e Json, como destacado na figura ao lado;
 - Clique em Send para fazer a requisição
- A Resposta deve ser o mesmo objeto





**Usando o verbo PUT para
atualizar recursos**

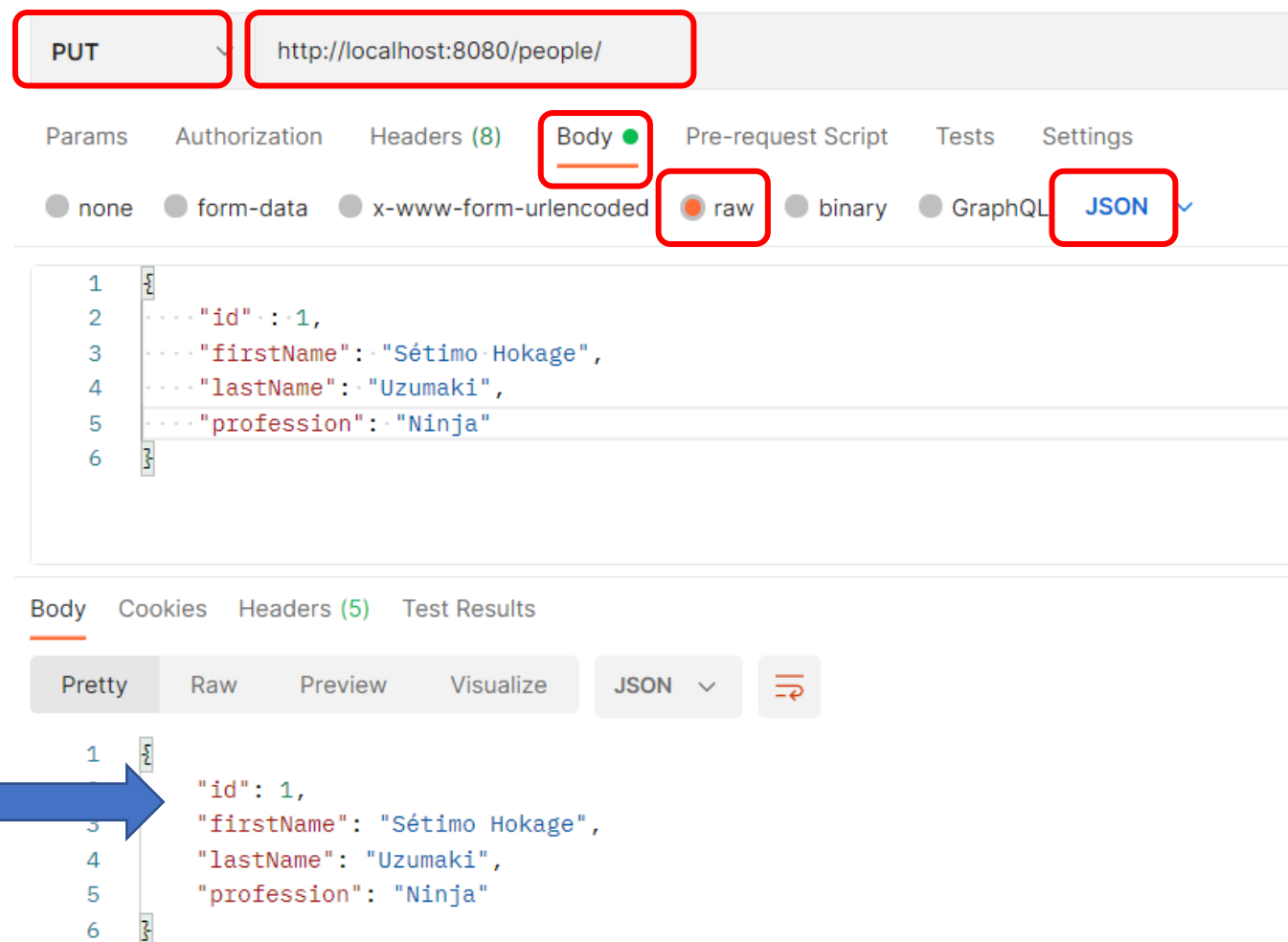
Criando o método update

- O método **update** vai simular a atualização de um recurso;
- Ele é bem similar ao método create;
- Agora abra **PersonController** e crie o método create junto com um **RequestMapping**, como no Código abaixo:
- Note que o método agora é do tipo **PUT** e ele consome e produz **JSON**;
- O parâmetro do método é um **RequestBody**, ou seja, um objeto complexo, não sendo possível informá-lo via **GET**

```
// request mapping using "/", but with PUT method - data passed by request body
@RequestMapping(method = RequestMethod.PUT,
    consumes = MediaType.APPLICATION_JSON_VALUE,
    produces = MediaType.APPLICATION_JSON_VALUE)
public Person update(@RequestBody Person person){
    return services.update(person);
}
```

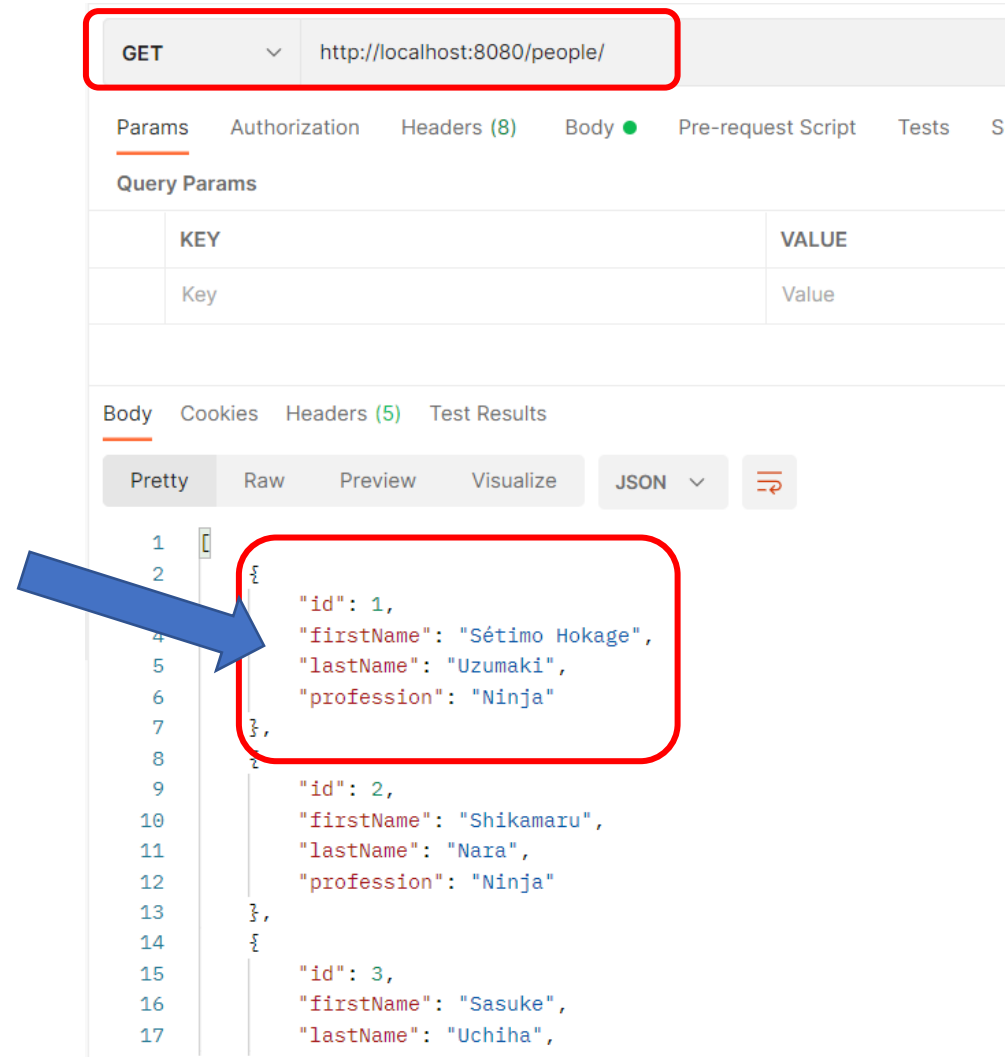
Testando via Postman

- O método POST não passa parâmetros via URL;
- Os dados são enviados em um **Request Body**;
- Para simular uma requisição PUT, vamos usar o Postman;
 - Selecione o método POST;
 - A URL é <http://localhost:8080/people/update>
 - Abra a guia **Body**, depois raw e Json, como destacado na figura ao lado;
 - Clique em Send para fazer a requisição
- A Resposta deve ser o mesmo objeto



Verifique se foi atualizado!

- Agora, vamos fazer um get na URL people/ para ver se os dados foram atualizados:





**Usando o verbo DELETE para
excluir recursos**

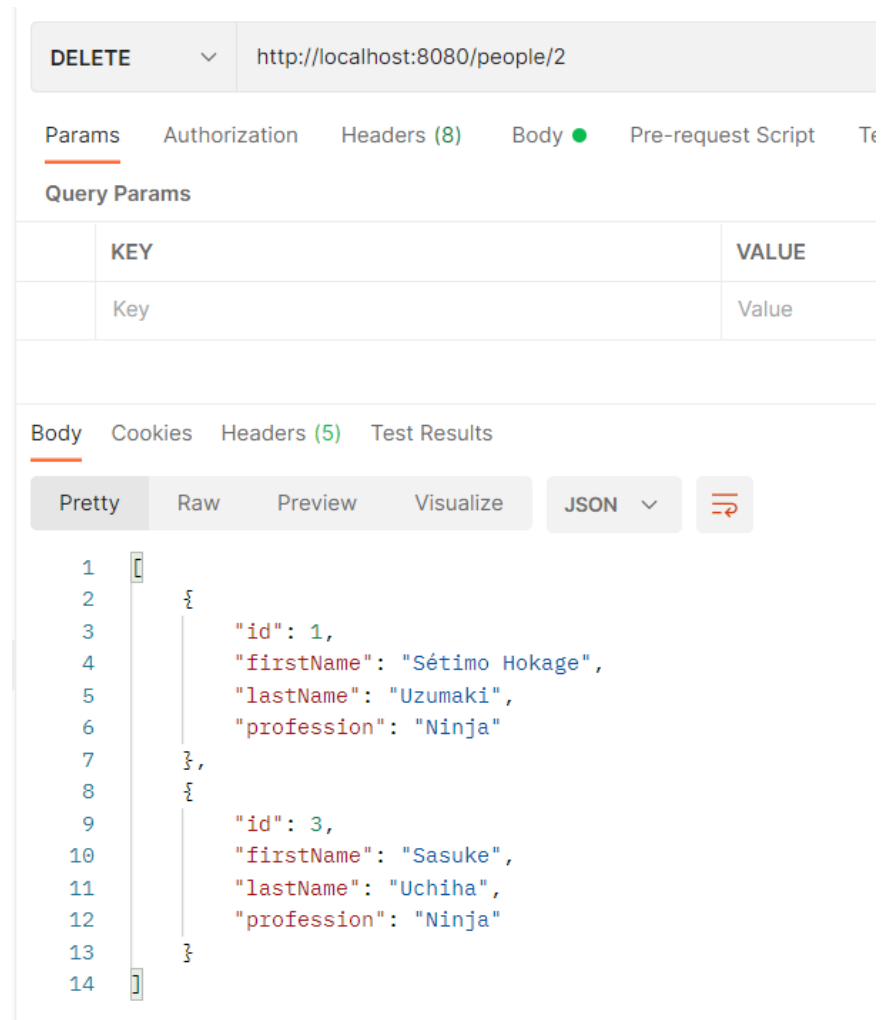
Criando o método delete

- O método ***delete*** vai simular a exclusão de um recurso;
- Agora abra **PersonController** e crie o método delete junto com um **RequestMapping**, como no Código abaixo:
- Note que o método agora é do tipo **DELETE** e ele não consome, nem produz JSON;
- O parâmetro do método é passado via PathParam ({id}), já que o método é o GET;
- O retorno é a lista atualizada;

```
// request mapping using a path param and DELETE method
@RequestMapping(value =("/{id}", method = RequestMethod.DELETE)
public List<Person> delete(@PathVariable long id){
    return services.delete(id);
}
```


Testando via Postman

- No Postman, precisamos mudar o método da requisição e informar o PathParam na URL;
- O resultado é a lista atualizada, sem o registro que foi excluído;



Atividades

- Crie um novo projeto simples, similar a esse, considerando todos os métodos;
- Sugestão: cadastro de veículos;

Sobre mim

JORGE LUÍS GREGÓRIO

- Professor da Faculdade de Tecnologia “Prof. José Camargo” – Fatec Jales, e da Escola Técnica Estadual Dr. José Luiz Viana Coutinho – Etec Jales;
 - Articulista do Jornal de Jales – Coluna “Fatecnologia”;
 - Apresentador do Tech Trends, podcast oficial da Fatec Jales;
 - Bacharel em Sistemas de Informação; Especialista em Desenvolvimento de Software para Web e Mestre em Ciência da Computação.
 - Trabalha com tecnologia desde 1998, tendo atuado como analista de suporte; administrador de redes de computadores; desenvolvedor de software, *webdesigner* e professor.
-
- Site oficial: www.jlgregorio.com.br
 - Perfil do LinkedIn: www.linkedin.com/in/jlgregorio81
 - Currículo Lattes: <http://lattes.cnpq.br/3776799279256689>

