

FRAMEWORKS JAVA, .NET &
WEBSERVICES

WEB SERVICES **COM SPRING BOOT**



6A

LISTA DE FIGURAS

Figura 1 – Retornar usuários – API fMock	10
Figura 2 – Retornar usuário de id 1 – API fiap-on SandBox	10
Figura 3 – JSON trafegar via HTTP/HTTPs	13
Figura 4 – Retornar usuário de id 1 – API fiap-on SandBox	19
Figura 5 – Visualizar HTTP Headers no Chrome – API fiap-on SandBox	21
Figura 6 – Criar a Collection no Postman	22
Figura 7 – Nomear a Collection no Postman	23
Figura 8 – Criar a requisição GET no Postman	23
Figura 9 – Salvar a requisição GET na Collection Fiap Usuários	24
Figura 10 – Criar a requisição PUT no Postman.....	25
Figura 11 – Enviar a requisição PUT no Postman	25
Figura 12 – Criar e enviar uma requisição POST no Postman	27
Figura 13 – Headers HTTP e Status Code da resposta ao POST no Postman	27
Figura 14 – Criar e enviar uma requisição DELETE no Postman	28
Figura 15 – Criar e enviar uma requisição de teste do DELETE no Postman ..	29
Figura 16 – Enviar a requisição GET de teste no Postman	29
Figura 17 – Spring Framework.....	40
Figura 18 – Página de criação de projeto no Spring Initializr.....	41
Figura 19 – Estrutura do Projeto	42
Figura 20 – Tela de inicialização do projeto Spring Boot.....	44
Figura 21 – Tela de login do H2	45
Figura 22 – Interface Web do H2	45
Figura 23 – Teste do GET no Postman.....	55
Figura 24 – Teste do POST no Postman	56
Figura 25 – Teste do GET no Postman.....	57
Figura 26 – Teste do PUT no Postman.....	58
Figura 27 – Teste do GET por ID no Postman.....	58
Figura 28 – Teste do DELETE no Postman	59
Figura 29 – Teste do DELETE no Postman (Parte 2).....	60

LISTA DE QUADROS

Quadro 1 – Status Code.....	30
Quadro 2 – Annotations de Validação.....	47

EXEMPLO

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Lista de usuários representado em JSON	14
Código-fonte 2 – Lista de usuários representado em XML	14
Código-fonte 3 – Lista de usuários em JSON sem o array usuários	15
Código-fonte 4 – Mensagem de requisição HTTP	18
Código-fonte 5 – Mensagem de resposta HTTP	19
Código-fonte 6 – Configuração do H2 Database no Spring Boot.....	44
Código-fonte 7 – Código da Entidade Produto.....	46
Código-fonte 8 – Código do ProdutoRepository.....	48
Código-fonte 9 – Instanciação do ProdutoRepository	49
Código-fonte 10 – Código do ProdutoResource	53
Código-fonte 11 – JSON exemplo com um produto.....	55
Código-fonte 12 – JSON exemplo de um novo produto.....	57
Código-fonte 13 – Código da Entidade Categoria.....	61
Código-fonte 14 – Código do CategoriaRepository.....	61
Código-fonte 15 – Código da Entidade Produto com Categoria	62

SUMÁRIO

1 WEB SERVICES COM SPRING BOOT.....	6
1.1 O que é REST?	6
1.2 Restrições REST	7
1.2.1 Interface Uniforme	8
1.2.2 Crie soluções baseadas em recursos	8
1.2.3 Manipular os recursos utilizando representações	11
1.3 JSON	11
1.3.1 Por que JSON?	13
1.4 Manipular os recursos através de operações padronizadas.....	16
1.4.1 HTTP Headers	17
1.4.2 Media Types.....	20
1.4.3 HTTP Methods	21
1.4.3.1 GET	21
1.4.3.2 PUT	24
1.4.3.3 POST.....	26
1.4.3.4 DELETE	28
1.4.4 Respostas padronizadas.....	30
1.4.4.1 HTTP Status Code	30
1.4.4.2 Uso Geral	31
1.4.4.3 GET	32
1.4.4.4 POST ou PUT	33
1.4.4.5 PUT	33
1.4.4.6 DELETE	33
1.4.4.7 POST, PUT e DELETE	33
1.4.4.8 Use mensagens auto descritivas	34
2 NÃO MANTENHA O ESTADO – STATELESS	35
2.1 Utilizar o Cache	36
2.3 Utilizar a arquitetura Client-Server	36
2.2 Arquitetar o sistema em camadas.....	37
2.3 Forneça código sobre demanda.....	37
2.4 Aplicações RESTful.....	37
3 BACK-END COM SPRING BOOT	38
3.1 Frameworks.....	38
3.2 Spring Framework	39
3.3 Spring Boot.....	40
3.4 Banco de Dados (H2 Database)	43
3.5 Spring Data JPA.....	45
3.6 REST Controller	49
3.7 Teste da API REST	54
3.8 Relacionamentos One-to-Many e/ou Many-to-Many.....	60
CONCLUSÃO.....	63
REFERÊNCIA	64

1 WEB SERVICES COM SPRING BOOT

Vamos dar início à saga do *Web Service* com padrão REST. Você descobrirá como funciona e qual é sua ligação com o protocolo web HTTP. Agora você tem mais uma carta para usar dentro do seu projeto, vamos lá?

1.1 O que é REST?

REST é um estilo arquitetural para sistemas distribuídos baseados em recursos da web. É uma abordagem popular para a construção de serviços web que utiliza protocolos HTTP e os conceitos do modelo de maturidade Richardson para criar APIs simples e escaláveis. Em um sistema REST, cada recurso é representado por uma URL única, que pode ser acessada por diferentes tipos de clientes, incluindo navegadores da web, aplicativos móveis e outros serviços web. O uso de padrões abertos e amplamente adotados, como HTTP e JSON, torna as APIs REST mais fáceis de integrar com outros sistemas e plataformas. Além disso, o REST é altamente escalável, permitindo que o processamento e o armazenamento de dados sejam distribuídos em diferentes servidores, tornando o sistema mais eficiente e tolerante a falhas. Se você está interessado em desenvolver serviços web ou consumir APIs, é importante entender os princípios e as práticas do REST.

Traduzindo livremente “*Representational State Transfer*” ou REST, temos Transferência de Estado Representativo. O termo REST foi introduzido e definido em 2000 na tese de doutorado de Roy Fielding para a University of California, Irvine.

Nesta tese, resumidamente, primeiro Roy determina uma metodologia de classificação para estilos arquiteturais. Depois, apresenta diversos estilos, cada um com o conjunto das regras que os definem. Na sequência, demonstra os requisitos da arquitetura WWW e os problemas enfrentados na concepção e avaliação das melhorias propostas aos principais protocolos de comunicação web. Por fim, estabelece o estilo arquitetônico REST como um modelo de como a Web moderna deveria funcionar para sistemas hipermídia distribuídos.

Fielding participou efetivamente no desenvolvimento de muitos dos protocolos Web, incluindo HTTP (*HyperText Transfer Protocol*) e URI (*Uniform Resource*

Identifier), sendo, assim, não é muito difícil compreendermos a razão do estilo REST está tão alinhado à arquitetura Web.

Segundo Fielding, o REST fornece um conjunto de restrições arquitetônicas que, quando aplicadas como um todo, enfatizam a escalabilidade das interações dos componentes, a generalidade das interfaces e a implantação independente de componentes e componentes intermediários para reduzir a latência de interação, reforçar a segurança e encapsular sistemas legados.

Então, antes de desenvolvermos serviços RESTful em Java, vamos conhecer os itens que compõem este conjunto de restrições e descobrir como cada um ajuda as aplicações RESTful a serem simples, leves e rápidas.

1.2 Restrições REST

O REST (Representational State Transfer) é um conjunto de princípios arquiteturais que guiam o design de sistemas baseados em recursos da web para criar APIs simples, escaláveis e flexíveis. Roy Fielding os definiu em sua dissertação de doutorado em 2000 e são amplamente usados na indústria de desenvolvimento de software. As restrições REST incluem seis princípios: Client-Server, Stateless, Cacheable, Layered System, Code on Demand (opcional) e Uniform Interface. A separação das responsabilidades entre o cliente e o servidor é fundamental para a escalabilidade e a evolução do sistema. Cada solicitação do cliente para o servidor deve conter todas as informações necessárias para o servidor compreender e responder a essa solicitação, sem depender de nenhum contexto armazenado no servidor. As respostas do servidor devem ser explicitamente marcadas como cacheáveis ou não-cacheáveis para permitir que os clientes possam armazenar em cache as respostas e melhorar a performance do sistema. O sistema deve ser composto por camadas hierárquicas e a interface deve ser uniforme e consistente, utilizando recursos identificáveis por URLs, mensagens autoexplicativas, métodos HTTP padronizados e representações de recursos em formatos comuns (como XML, JSON ou HTML). O servidor pode enviar código executável (como applets ou scripts) para o cliente, que pode ser executado localmente e estender a funcionalidade da aplicação.

O cumprimento dessas restrições REST é fundamental para a criação de APIs simples, escaláveis e flexíveis, que possam ser facilmente integradas com outros sistemas e plataformas. Se você está interessado em desenvolver APIs RESTful, é importante entender e aplicar esses princípios em seu projeto.

1.2.1 Interface Uniforme

A restrição de interface uniforme define a interface entre clientes e servidores. Esta restrição é muito importante e, muitas vezes, a mais citada nos materiais e documentos sobre REST. Os princípios contidos nesta restrição simplificam e desacoplam a arquitetura, fazendo com que cada parte evolua de forma independente.

Os princípios orientadores da interface uniforme são:

1.2.2 Crie soluções baseadas em recursos

Um recurso é um elemento abstrato que nos permite mapear qualquer informação do mundo real, como um elemento para acesso via Web. Uma solução RESTful geralmente expõe um conjunto de recursos que identificam os objetivos da interação entre o serviço e os seus consumidores.

Os recursos são identificados e acessados por URIs que compõem um namespace global e único. Neste contexto, os próprios recursos são separados das representações que serão devolvidas ao cliente consumidor. Por exemplo, imagine que queremos acessar uma lista de usuários (informações do mundo real) que estão em um banco de dados em algum lugar do mundo. O servidor não enviará o banco de dados, mas sim uma representação dos usuários em algum formato. No exemplo que faremos mais à frente, a representação dos usuários será retornada em JSON.

Pois bem, como vamos encontrar esse servidor e obter uma representação do recurso que queremos?

Para tanto, vamos utilizar uma *Uniform Resource Identifier* ou URI, que nada mais é do que uma sequência de caracteres usados para identificar um recurso de forma global e única. Utilizaremos a URL <

<https://63fd457c677c4158731bddb4.mockapi.io/usuarios> >, que endereça uma API construída em uma ferramenta de Mock exclusivamente para exemplificarmos os conceitos que estamos vendo.

Você já ouviu falar de ferramentas de Mock como o extinto Sandbox, mockapi ou postman? São plataformas web que permite criar mocks de APIs rapidamente e de forma simples. Mas o que são mocks de APIs?

Mock é uma simulação da API real, que é usada durante o desenvolvimento para testar funcionalidades ou integrações sem precisar acessar a API real. É como se fosse um "dublê" real que pode ser configurado para retornar dados específicos em cada requisição.

As ferramentas oferecem interfaces fáceis de usar para visualizar e testar as respostas da API mock criada. Com isso, você pode simular diferentes cenários de respostas e garantir que sua aplicação seja capaz de lidar com essas situações de forma adequada.

Para mais, acesse: < <https://mockapi.io/> ou <https://learning.postman.com/docs/designing-and-developing-your-api/mocking-data/setting-up-mock/> .

Podemos dizer que uma URL (*Uniform Resource Locator*) é uma URI que, além de identificar unicamente um recurso na web, especifica o protocolo, o servidor, o contexto da aplicação e o meio para acessar a representação do recurso desejado.

Vamos entender como isso acontece esmiuçando a URL:

< <https://63fd457c677c4158731bddb4.mockapi.io/usuarios>

Protocolo: http

Servidor na rede: <https://63fd457c677c4158731bddb4.mockapi.io>

Meio de acesso à representação do recurso: /usuarios

Representação do recurso: no Chrome, instale a extensão JSONView (Validate and view JSON documents), digite a URL e você obterá do servidor a representação em JSON do recurso:



Figura 1 – Retornar usuários – API fMock
Fonte: Elaborado pelo autor (2017)

Como podemos perceber, uma URL deve ser única por recurso, ou seja, quando queremos receber a lista de usuários, representada pela notação JSON, devemos sempre utilizar a URL <<https://63fd457c677c4158731bddb4.mockapi.io/usuarios/>>.

E se quisermos acessar apenas o primeiro usuário?

A URL deve ser diferente e novamente única. Sendo, assim, uma implementação simples e intuitiva para obtermos o primeiro usuário naturalmente seria a URL <<https://63fd457c677c4158731bddb4.mockapi.io/usuarios/1>> e foi desta forma que codificamos a API no Sandbox. Testando a URL no Chrome, obtemos:

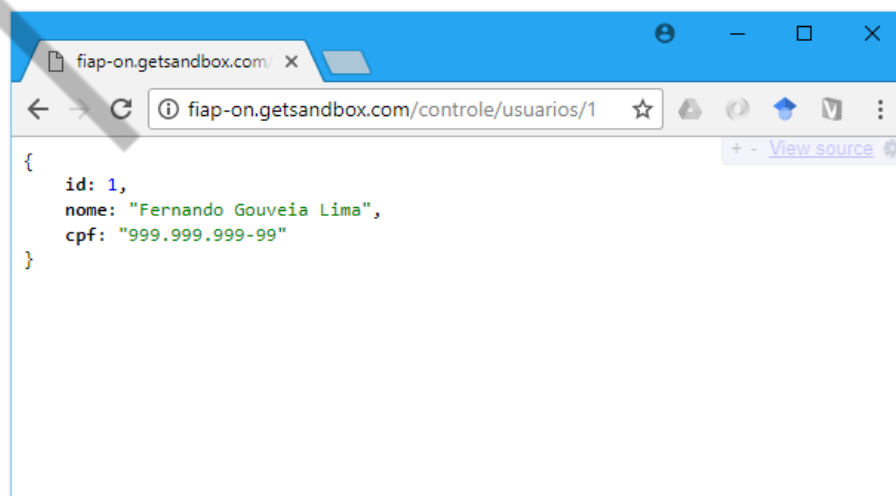


Figura 2 – Retornar usuário de id 1 – API fiap-on SandBox
Fonte: Elaborado pelo autor (2017)

Note que o conceito de URI/URL como identificador único facilita muito o entendimento e o consumo da API. Se você enviar os links para alguma pessoa desinformada sobre a aplicação, creio que, por si só, facilmente concluirá que se trata de um aplicativo web para o controle dos usuários, que lista um usuário específico ou todos, de acordo com a URL empregada. Concorda?

1.2.3 Manipular os recursos utilizando representações

Quando um cliente tem uma representação de um recurso, possui informações suficientes para incluir, modificar ou excluir o recurso no servidor, desde que tenha permissão para fazê-lo. Vamos entender melhor este conceito mais à frente.

Em nossos exemplos e, na maioria dos serviços RESTful, a Notação de Objetos JavaScript ou JSON (*JavaScript Object Notation*) é largamente utilizada como opção ao XML (*eXtensible Markup Language*).

1.3 JSON

Segundo o json.org, JSON é uma formatação leve de troca dos dados. Para seres humanos, é fácil de ler e escrever. Para máquinas, é fácil de interpretar e gerar. Está baseada em um subconjunto da linguagem de programação JavaScript, Standard ECMA-262 3ª Edição – Dezembro – 1999. JSON é um formato em texto, completamente independente de linguagem, pois usa convenções familiares a diversas linguagens.

JSON, basicamente, é construído em duas estruturas:

- Uma coleção de pares nome / valor.
- Uma lista ordenada dos valores.

Como a estrutura do JSON é muito simples, em uma situação que precisamos trocar dados entre uma aplicação cliente e um serviço web, seria muito interessante podermos transformar um objeto em JSON (*Marshalling*) e enviá-lo para o serviço, bem como recebermos um JSON e transformá-lo em um objeto no cliente

(*Unmarshalling*), concorda? Faremos o envio no tópico a seguir, utilizando os verbos HTTP PUT e POST.

Como JSON é independente de plataforma, podemos enviar e receber informações via Web em diversas linguagens e em ambientes de hardware e software totalmente heterogêneos, utilizando Web Services.

Em um cenário hipotético, poderíamos ter aplicações clientes em hardwares distintos, SOs e linguagens de programação diferentes, transformando objetos JavaScript (Browsers), objetos Java (App) e objetos .Net (Desktop Application) em JSON. E enviá-lo para um único serviço, em um Servidor Web, responsável por armazená-lo em um Banco de Dados ou em um Servidor de Arquivos.

Ainda em nosso cenário hipotético, não seria impossível imaginar os diversos clientes consumindo o mesmo serviço para receber um JSON formatado a partir de objetos instanciados no Servidor Web com dados do Banco de Dados. Ou para receber o JSON de um arquivo de texto físico, formatado em JSON, que o Servidor Web busca no Servidor de Arquivos.

Da mesma forma que os clientes podem estar em ambientes de hardware e software diferentes sem impactar a comunicação, o Web Service executado no Internet Server também pode ser codificado nas mais diversas linguagens, tais como, Java, .NET, PHP, entre outras. Ele pode ser executado também em SOs completamente diferentes dos clientes, sem nada impactar a comunicação entre consumidor e serviço web.

Na essência, JSON é um texto formatado, tanto quanto o XML. Portanto, ambos possibilitam o tráfego suave através de firewalls e podem utilizar a porta 80 do protocolo HTTP, possibilitando a chamada de serviços em um Servidor de Internet, sem o bloqueio de um Firewall.

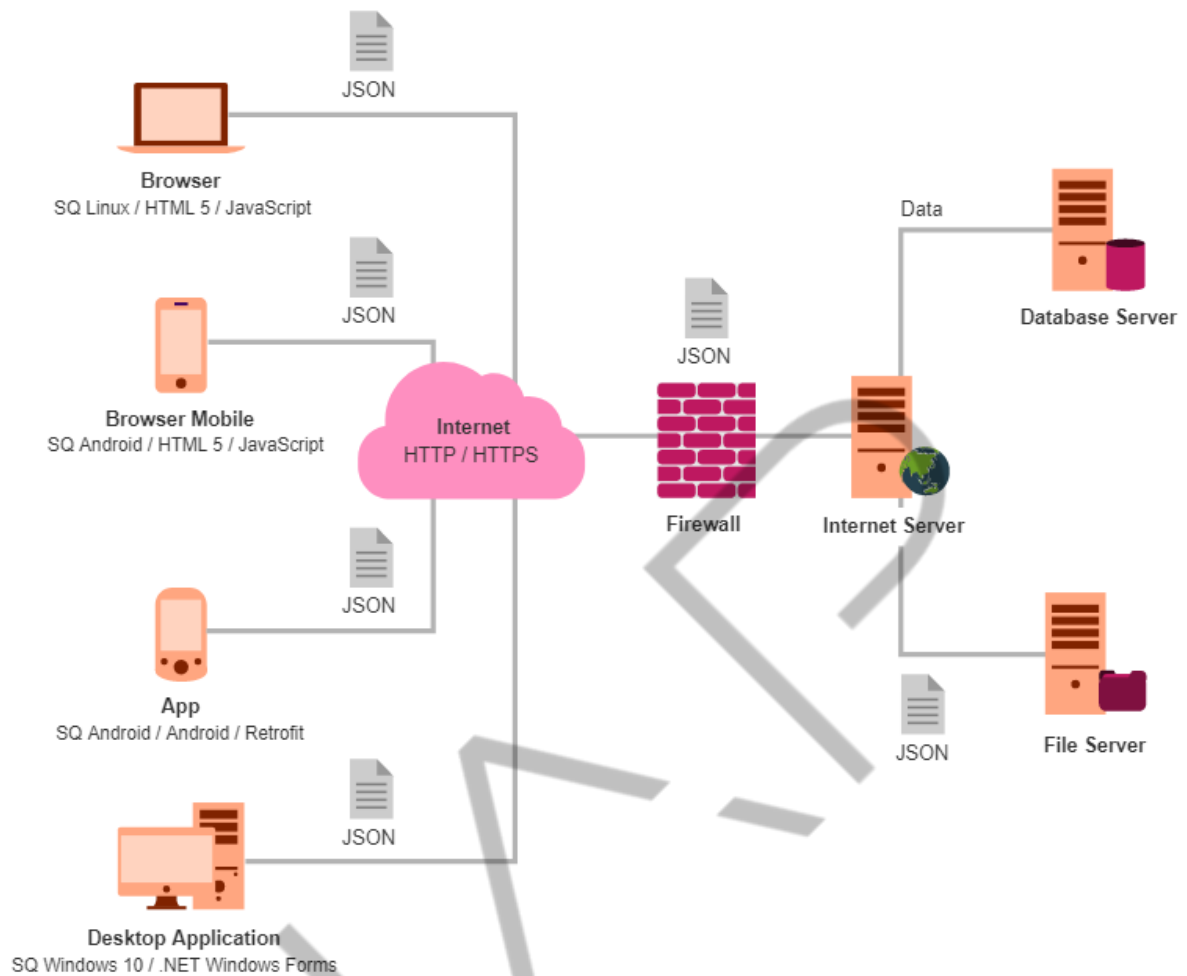


Figura 3 – JSON trafegar via HTTP/HTTPS
Fonte: Elaborado pelo autor (2017)

1.3.1 Por que JSON?

Para responder a esta pergunta, vamos olhar o mesmo recurso, uma lista contendo dois usuários, representada por ambas as formatações.

JSON:

```
{ "id": 1,  
  "nome": "Fernando Gouveia Lima",  
  "cpf": "999.999.999-99"},  
{ "id": 2,  
  "nome": "Fernando Barbosa Lima",  
  "cpf": "111.111.111-11" }
```

Código-fonte 1 – Lista de usuários representado em JSON
Fonte: Elaborado pelo autor (2017)

XML:

```
<usuarios>  
  <usuario>  
    <id>1</id>  
    <nome>Fernando Gouveia Lima</nome>  
    <cpf>999.999.999-99</cpf>  
  </usuario>  
  <usuario>  
    <id>2</id>  
    <nome>Fernando Barbosa Lima</nome>  
    <cpf>111.111.111-11</cpf>  
  </usuario>  
</usuarios>
```

Código-fonte 2 – Lista de usuários representado em XML
Fonte: Elaborado pelo autor (2017)

Analisando os dois códigos, podemos perceber facilmente que ambas as representações são auto descritivas e possuem uma hierarquia fácil de entender. Em ambas as representações, as quebras de linha e os espaços não são obrigatórios e foram empregados apenas para facilitar a visualização.

Da mesma forma, não é difícil percebermos que a representação JSON emprega array e não usa tags de fechamento para delimitar um dado de outro. Portanto, tem um tamanho menor do que uma representação em XML e, na maioria dos casos, é mais rápida de se escrever e ler.

Array "usuarios": []	X	Tag <usuarios></usuarios> End Tag
-----------------------------	----------	---

name/value "id": 1	X	Tag <id> Value 1 </id> End Tag
---------------------------	----------	---

Ainda podemos acrescentar mais uma diferença, o JSON não tem a obrigatoriedade de ter um elemento-raiz como o XML, ou seja, o nome "usuarios": é opcional. Enquanto as TAGs <usuarios></usuarios>, que determinam o elemento-raiz do XML, são obrigatórias. Apesar de menos intuitivo, muitas aplicações processam o JSON para representar o recurso lista dos usuários, estruturando-o da seguinte forma:

```
{ [
  {"id": 1,
    "nome": "Fernando Gouveia Lima",
    "cpf": "999.999.999-99"},
  {"id": 2,
    "nome": "Fernando Barbosa Lima",
    "cpf": "111.111.111-11"} ] }
```

Código-fonte 3 – Lista de usuários em JSON sem o array usuários
Fonte: Elaborado pelo autor (2017)

Dando dois passos para trás, enfatizamos que “na maioria dos casos, JSON é mais rápido de ler e escrever” e não que “em todos os casos”, pois a linguagem de

programação e a escolha das bibliotecas e da técnica de análise (*parsing*) utilizada impactam diretamente na velocidade da escrita e leitura em ambas as formas de representação. Em nossos estudos, empregaremos Java como a linguagem de programação e faremos *Marshalling* e *Unmarshalling* utilizando bibliotecas desenvolvidas para Java.

De qualquer maneira, podemos afirmar que aplicações aderentes às restrições impostas pelo estilo REST e que utilizam JSON como representação se saem muito bem em situações em que há limitação dos recursos ou quando um baixo consumo de largura de banda é necessário. Ou seja, JSON é muito empregado em soluções nas quais o tamanho do fluxo dos dados entre o cliente e o servidor é de grande importância. JSON é utilizado por grandes empresas que enfrentam desafios com estas características, tais como, Google, Facebook, Twitter, entre outras.

Para saber mais sobre JSON, acesse:

<<http://www.json.org> e https://www.w3schools.com/js/js_json_intro.asp>

Apesar da atenção especial que demos em JSON e XML, precisamos compreender que um recurso pode ter várias representações, ou seja, podemos desenvolver aplicações que retornem representações diferentes de acordo com cenários ou necessidades específicas de cada cliente. Como exemplo, se o cliente final for uma aplicação Mainframe, nada nos impede de retornar uma representação em CSV ao invés de JSON ou XML.

1.4 Manipular os recursos através de operações padronizadas

Quando falamos sobre manipular os recursos de forma padronizada, queremos dizer sobre como enviar e receber informações via rede de forma organizada e unificada. Podemos afirmar que esse é o principal papel do protocolo HTTP. Também podemos dizer que aplicações RESTful devem se aproveitar das características deste protocolo e dos demais padrões Web para aderir às restrições REST.

1.4.1 HTTP Headers

Em uma comunicação simples utilizando HTTP, o cliente sempre solicita algo e, então, o servidor responde. Sendo assim, a responsabilidade do uso correto do protocolo é tanto do cliente quanto do servidor.

Mensagens HTTP trocadas entre eles são formadas obrigatoriamente por uma linha inicial, conhecida como Request Line (em solicitações do cliente) e reconhecida como Status Line (em respostas do servidor). Depois, por um conjunto de Header Fields (Request or Response) e, opcionalmente, por um Body para envio ou retorno de conteúdo.

Como vimos nas requisições que retornaram uma lista dos usuários e, depois, um usuário específico, fizemos ambas as solicitações sem enviar nada no Body, já que as operações não precisavam. Você se lembra?

Uma linha de solicitação ou Request Line contém o verbo HTTP, a URL do recurso e a versão do protocolo HTTP que está sendo usada. Os campos do cabeçalho incluem metadados descritos em pares (nome e valor). Cada campo engloba um nome e um valor com uma informação específica sobre a solicitação.

Os clientes usam a primeira linha e os campos do cabeçalho de uma mensagem de solicitação para processar o recurso, para se identificar e para informar como a representação deve ser retornada pelo servidor.

Como exemplo, vamos dar uma olhada na mensagem de solicitação HTTP que fizemos no Chrome para < <https://63fd457c677c4158731bddb4.mockapi.io/usuarios/3> >.

```
GET /usuarios/1 HTTP/1.1  
Host: 63fd457c677c4158731bddb4.mockapi.io  
Accept: */*  
Accept-Language: pt-BR,pt;  
User-Agent: Chrome/61.0.3163.100  
Accept-Encoding: gif, deflate
```

Diagrama de uma mensagem de requisição HTTP. A primeira linha, "GET /usuarios/1 HTTP/1.1", é rotulada como "Linha de Solicitação". As linhas subsequentes, "Host: 63fd457c677c4158731bddb4.mockapi.io", "Accept: */*", "Accept-Language: pt-BR,pt;", "User-Agent: Chrome/61.0.3163.100" e "Accept-Encoding: gif, deflate", são agrupadas por uma chave de corchete à direita e rotuladas como "Cabeçalho".

Código-fonte 4 – Mensagem de requisição HTTP
Fonte: Elaborado pelo autor (2017)

Nesta mensagem, podemos ver que uma solicitação com o método **GET** foi feita para o servidor **63fd457c677c4158731bddb4.mockapi.io** na URL **/usuarios/3**, aceitando qualquer tipo de representação retorno ***/*** na língua **pt-BR,pt**. A solicitação foi feita do navegador **Chrome/61.0.3163.100** que está apto para processar o conteúdo de resposta comprimido, usando os algoritmos **gzip, deflate**.

E o Body?

Nesta mensagem de solicitação, não foi necessário, mas, em outras mensagens, será. Em solicitações PUT (Update) e POST (Create), não teremos de enviar para o servidor uma representação do recurso que queremos atualizar ou criar por meio dele?

Nestes casos, o Body conterà as informações que precisamos enviar para o Servidor e as processará de acordo com o verbo HTTP e o URL que estiverem na Request Line da mensagem.

Vimos uma mensagem de requisição GET. Vamos olhar a resposta?

HTTP/1.1 200 OK

} Linha de Solicitação

Server: nginx

Date: Mon, 30 Oct 2017 15:36:31 GMT

Content-Type: application/json

Content-Encoding: gzip

Código-fonte 5 – Mensagem de resposta HTTP
Fonte: Elaborado pelo autor (2017)

A primeira linha de uma mensagem de resposta do protocolo HTTP é conhecida como Status Line e consistida pela versão de protocolo, seguida por um código de status numérico e uma frase explicativa sobre o código. O serviço fiap-on utilizou-a para informar o sucesso do processamento do método GET através do Status Code 200.

Nos campos do cabeçalho informam que o servidor web que processou o pedido é um Nginx, a data e o horário do processamento no servidor, o tipo do conteúdo retornado no Body e como foi comprimido. Aqui está o Body da nossa resposta em JSON. Você se lembra?



Figura 4 – Retornar usuário de id 1 – API fiap-on SandBox
Fonte: Elaborado pelo autor (2017)

Cada requisição de um cliente feita a um serviço está condicionada a um verbo HTTP e comentamos que os serviços RESTful agem de forma diferente de acordo com eles, vamos entender mais um pouco sobre os mais utilizados em APIs REST.

1.4.2 Media Types

Os protocolos da Internet foram criados para permitir que diferentes tipos de informação, como texto, imagens, vídeos e áudios, possam ser transmitidos de forma eficiente. Para isso, foi desenvolvido um mecanismo chamado Media Types, que rotula o tipo de conteúdo que está sendo enviado através da rede. Os Media Types são categorizados em seis tipos principais: texto, aplicação, imagem, vídeo, áudio e vnd. O tipo de Media Type utilizado depende da informação que está sendo transmitida. Por exemplo, um documento de texto seria rotulado como "text", enquanto uma imagem seria rotulada como "imagem". O uso dos Media Types é importante para garantir que o conteúdo seja transmitido corretamente e para que os programas de computador possam reconhecer o tipo de conteúdo que estão recebendo. Além disso, os Media Types também permitem que os desenvolvedores criem novos tipos de conteúdo e os rotulem de forma apropriada, para que eles possam ser transmitidos de forma segura e eficiente pela rede.

Os Media Types são divididos em tipos/subtipos e acrescidos de parâmetros quando uma informação adicional é necessária. Um exemplo seria "text/html; charset=utf-8".

Quando fizemos uma solicitação para `<https://63fd457c677c4158731bddb4.mockapi.io/usuarios/3>`, informamos à nossa API, via cabeçalho HTTP Accept, que esperávamos qualquer tipo de conteúdo de retorno, ou seja, media type `/*/*`.

Na resposta, a API retornou no cabeçalho Content-Type o Media Type `application/json`, para que o cliente saiba que o retorno deve ser processado como JSON. Caso tivéssemos solicitado via Accept um conteúdo em XML e a API não pudesse retorná-lo, o Status Code 415 deveria ser retornado pela API. APIs RESTful utilizam vários tipos de Media Types, mas os mais comumente usados são `application/json` e `application/xml`.

Entenderemos mais à frente sobre HTTP Status Code e faremos exemplos com Media Type diferentes.

1.4.3 HTTP Methods

1.4.3.1 GET

Em nossos dois exemplos de pesquisa sobre informações de usuários, utilizamos uma das operações padronizadas pelo HTTP, a operação GET.

Para confirmar essa afirmação, pressione F12 no Chrome, busque novamente por < <https://63fd457c677c4158731bddb4.mockapi.io/usuarios/>, selecione Network e depois usuarios. Então você verá em Headers que o método de requisição foi o GET e o código de sucesso foi o 200, conforme imagem abaixo.

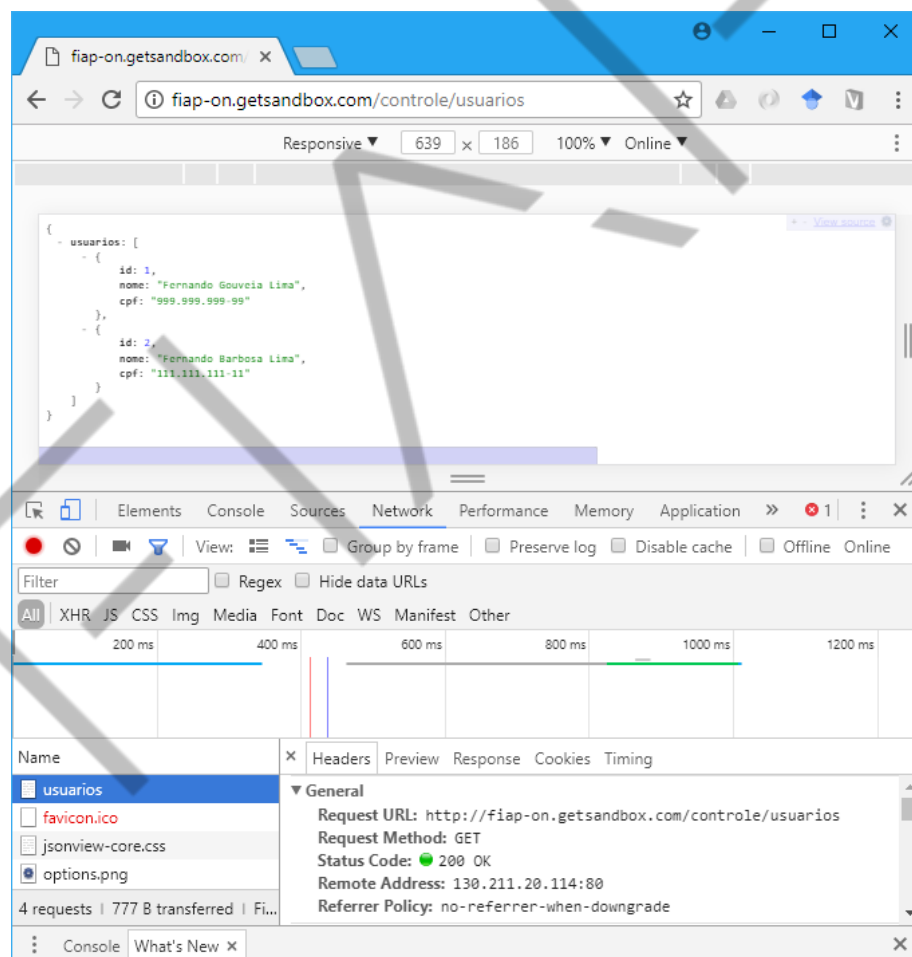


Figura 5 – Visualizar HTTP Headers no Chrome – API fiap-on SandBox
Fonte: Elaborado pelo autor (2017)

Como comentamos, o protocolo HTTP nos fornece uma interface de operações padronizadas. Esta interface é muito utilizada por aplicações RESTful para operações CRUD.

Além do verbo HTTP GET para leitura (R), aplicações RESTful geralmente utilizam os verbos POST para criação (C), PUT para atualização (U) e DELETE para excluir (D) um determinado recurso via rede. O protocolo HTTP possui mais verbos, além dos quatro citados, mas, neste momento, vamos focar nestes.

Você se lembra quando mencionamos que se o cliente possui permissão e uma representação do recurso, pode incluir, modificar ou excluir esse recurso no servidor?

Uma aplicação RESTful vale-se das informações contidas em uma mensagem de solicitação HTTP para identificar e realizar a operação que o cliente deseja. Os exemplos a seguir possuem como premissa a instalação do Postman, uma ferramenta utilizada por desenvolvedores e analistas de testes para auxílio na criação, testes e documentação de APIs.

Para instalar a ferramenta de acordo com a sua plataforma, baixe da seguinte URL <<https://www.getpostman.com>>. Após os detalhes de criação de conta superados, ao executar o Postman, selecione Collection e siga os demais passos:

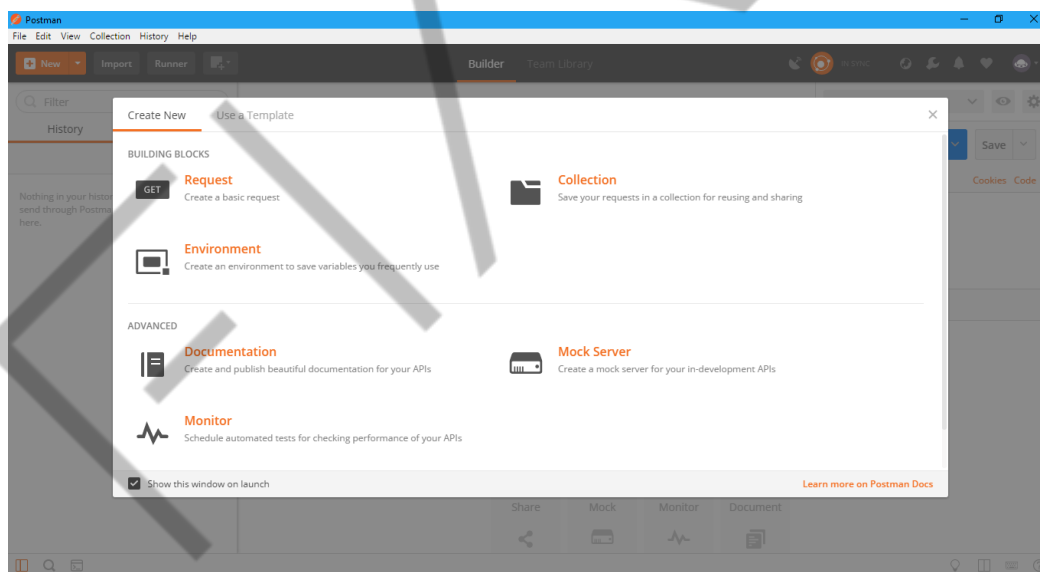


Figura 6 – Criar a Collection no Postman
Fonte: Elaborado pelo autor (2017)

Nomeie a Collection como FIAP Usuários:

Web Services com Spring Boot

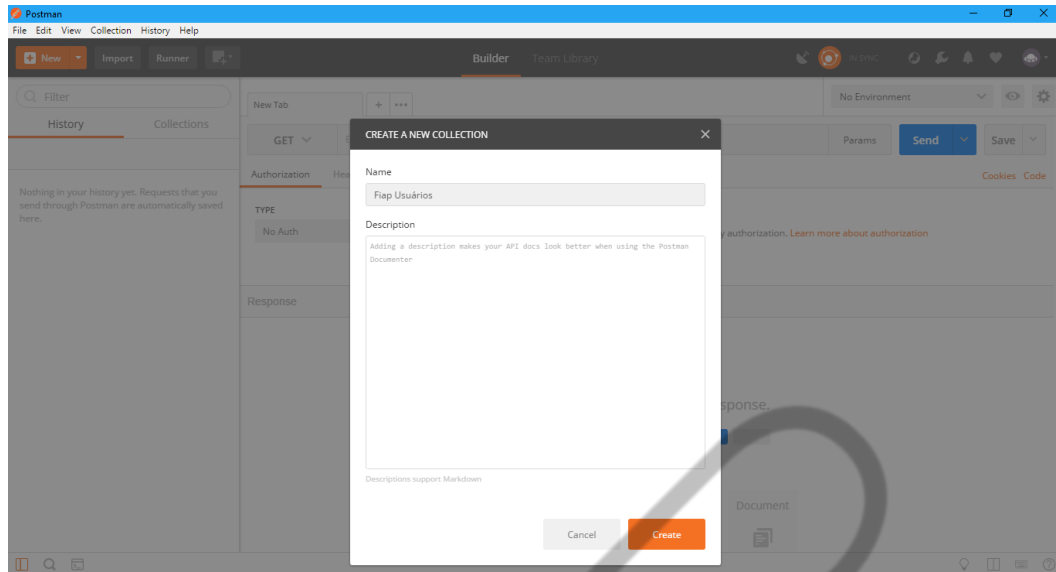


Figura 7 – Nomear a Collection no Postman
Fonte: Elaborado pelo autor (2017)

Crie o primeiro método da Coleção utilizando o método GET. Digite a URL <<https://63fd457c677c4158731bddb4.mockapi.io/usuarios/>> e clique em Send. Nossa API retornará a lista de usuários em JSON, conforme imagem abaixo.

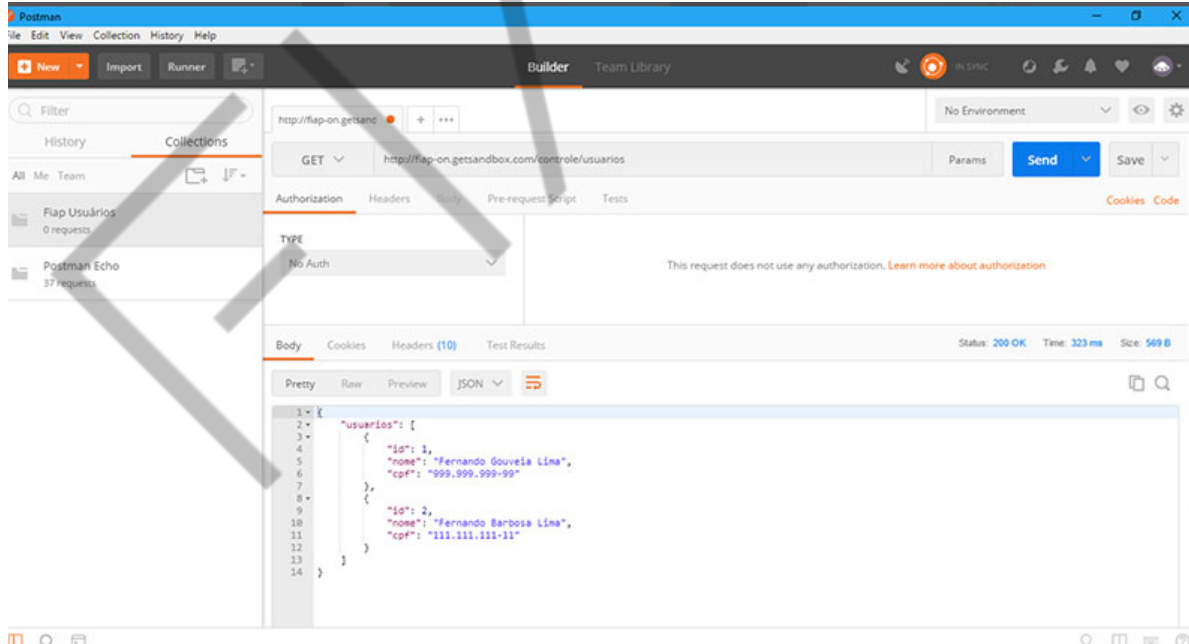


Figura 8 – Criar a requisição GET no Postman
Fonte: Elaborado pelo autor (2017)

Explore a aba Headers e veja o Status 200 OK e o tempo de resposta da API. Na sequência, clique em Save e salve essa requisição na coleção FIAP Usuários, conforme imagem abaixo:

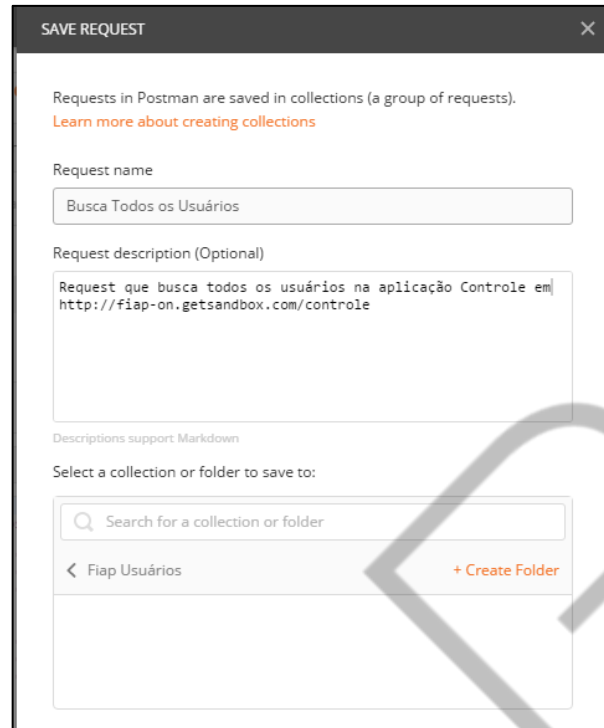


Figura 9 – Salvar a requisição GET na Collection Fiap Usuários
Fonte: Elaborado pelo autor (2017)

Fazendo isso, deixe a pesquisa que realizamos gravada na ferramenta para ser identificada, documentada e separada por coleções, geralmente ligadas a uma aplicação ou a um módulo de uma aplicação maior que se pretende testar.

Caso queira, repita estas ações para a URL que retorna apenas um usuário. No momento, este não é o nosso foco. Vamos mirar nos demais métodos CRUD da nossa API, utilizando os verbos HTTP PUT, POST e DELETE. Apesar de não ser a melhor prática, executaremos estas três operações e, depois, realizaremos uma nova consulta com GET para aferir os resultados.

1.4.3.2 PUT

Primeiro vamos atualizar o usuário 1 utilizando o Postman. Pois bem, para realizar esta operação, precisamos enviar a representação do recurso que queremos alterar para a aplicação RESTful, utilizando o verbo PUT.

Vamos lá, no Postman, adicione uma nova aba, utilize a URL da imagem e altere o método de GET para PUT. Em Headers, adicione Key e Value, conforme

Web Services com Spring Boot

imagem abaixo. Precisamos adicionar ao cabeçalho a informação de que enviaremos em nossa requisição o conteúdo do Body em JSON.

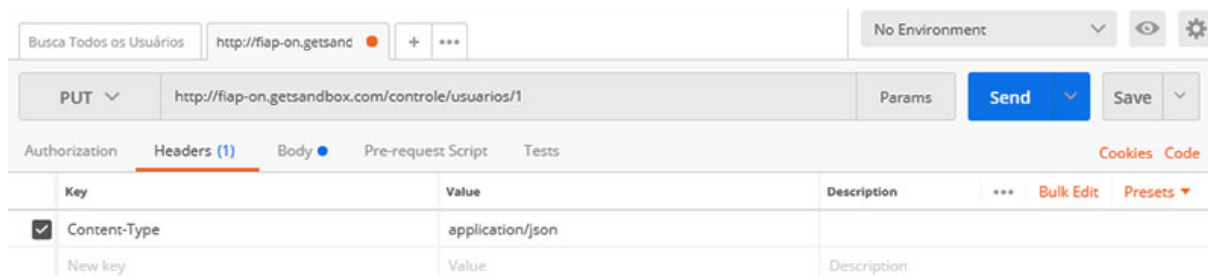


Figura 10 – Criar a requisição PUT no Postman
Fonte: Elaborado pelo autor (2017)

Em Body, selecione raw e escreva a representação do recurso que queremos alterar. Clique em Send e obterá o status OK, conforme imagem abaixo:

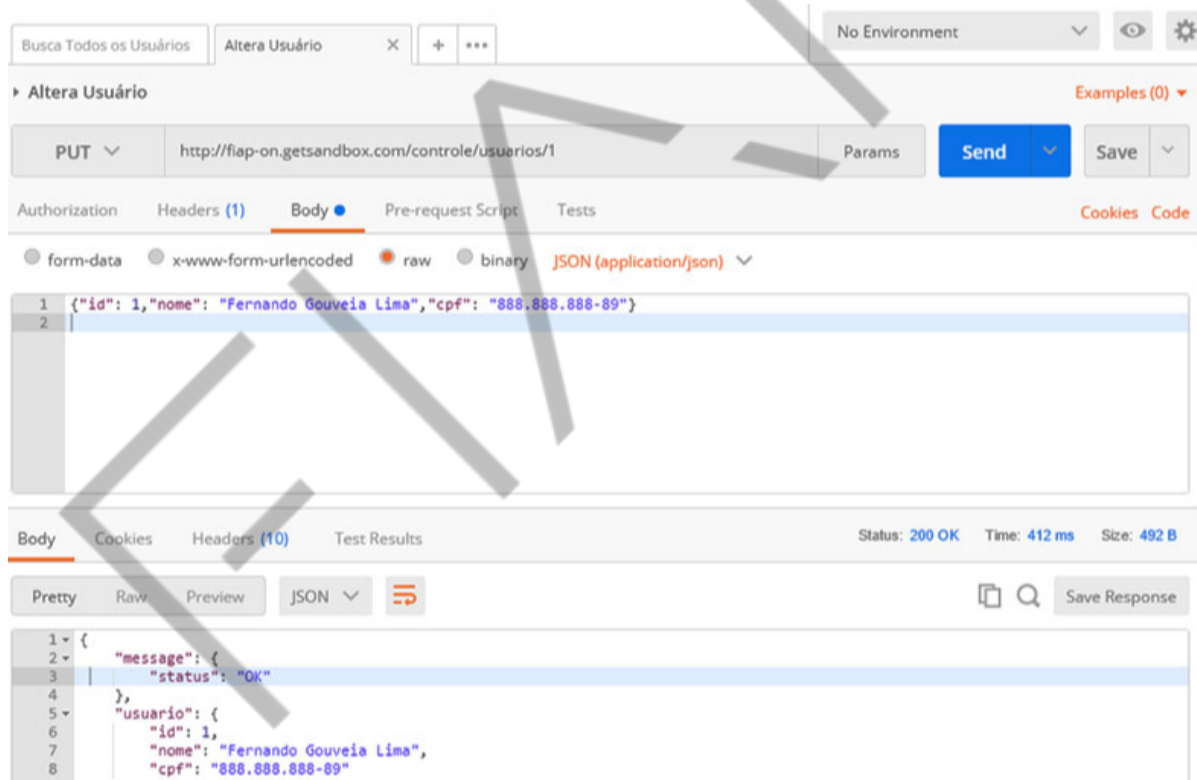


Figura 11 – Enviar a requisição PUT no Postman
Fonte: Elaborado pelo autor (2017)

A operação obteve sucesso, retornou o Status Code 200 OK e, como parte da mensagem de retorno, o recurso que foi atualizado para evitar que o consumidor do serviço tenha que fazer uma nova solicitação via GET para obter a representação atualizada do usuário de id = 1. Essa é uma boa prática de desenvolvimento

empregada pelo REST, tanto em alteração de recursos (PUT) quanto em criação (POST).

Salve essa operação na Collection Fiap Usuários com o nome que desejar.

O método PUT é utilizado para alterar um recurso enviando a sua representação completa. Caso queira implementar uma atualização enviando uma representação parcial, estude a abordagem, utilizando o verbo PATCH.

A RFC 7231 orienta que o método PUT altere o recurso, quando encontrado e quando não encontrado, crie o recurso. Em REST, desconheço a obrigatoriedade de implementarmos uma API CRUD desta forma. Sugiro que isso fique a critério do design da solução e seja documentado.

As restrições que considero sobre este assunto são:

- Se um recurso for atualizado com PUT, retorne 200 OK, quando retorná-lo no body; 204 No Content, se atualizá-lo, mas nada for retornado no body.
- Se um recurso for criado com PUT, retorne o Status Code 201 Created, a URI do novo recurso, no cabeçalho Location, e uma representação do novo recurso criado no body.

Algumas implementações fazem ainda o uso do verbo PATCH para alterações onde apenas uma determinada propriedade de um registro muda. A diferença entre PUT e PATCH é que no PUT sempre precisamos enviar todas as propriedades do registro, mesmo as que não serão alteradas. No PATCH, enviamos apenas as propriedades que sofreram alteração.

O mock FIAP utilizou o PUT apenas para alterações dos recursos.

1.4.3.3 POST

Para criar um recurso através da nossa API, adicione uma nova aba no Postman, utilize a URL da imagem abaixo e altere o método para POST.

Em Headers, adicione Key e Value, conforme fizemos no método PUT.

Web Services com Spring Boot

Em Body, selecione raw e escreva a representação do recurso que desejamos criar. Clique em Send e obterá o status OK, conforme imagem abaixo:

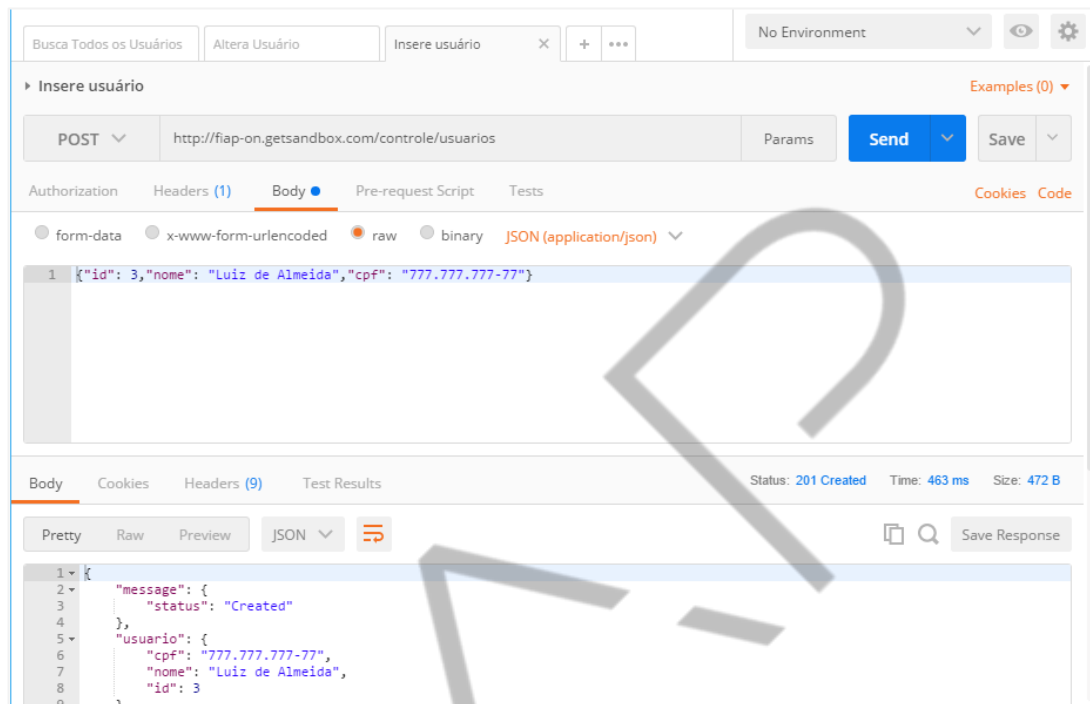


Figura 12 – Criar e enviar uma requisição POST no Postman
Fonte: Elaborado pelo autor (2017)

Salve a operação na Collection. Observe que, para criar um usuário, utilizamos a mesma URL que retorna todos os usuários, mas, desta vez, empregamos o método POST em vez do GET e enviamos uma representação do usuário para criação. Simples, não?

Repare que a API retornou o usuário criado e utilizou o Status Code 201 – Created para informar o sucesso da operação e não mais o Status Code 200 – OK. Ainda temos mais um uso relevante do protocolo HTTP, a API utilizou o cabeçalho Location para retornar a URI completa do novo recurso.

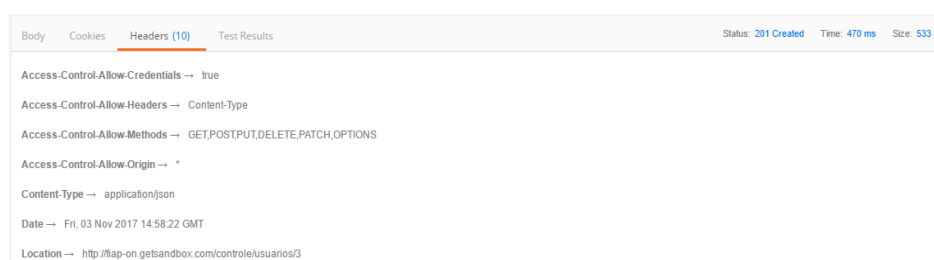


Figura 13 – Headers HTTP e Status Code da resposta ao POST no Postman
Fonte: Elaborado pelo autor (2017)

Creio que neste ponto ficou mais fácil perceber como a nossa aplicação RESTful aproveita as características do HTTP em seu benefício.

1.4.3.4 DELETE

Agora vamos excluir um usuário através da nossa API. Para tanto, adicione outra aba no Postman, utilize a URL da imagem abaixo e altere o método para DELETE.

Clique em Send e obterá o status OK, conforme imagem abaixo.

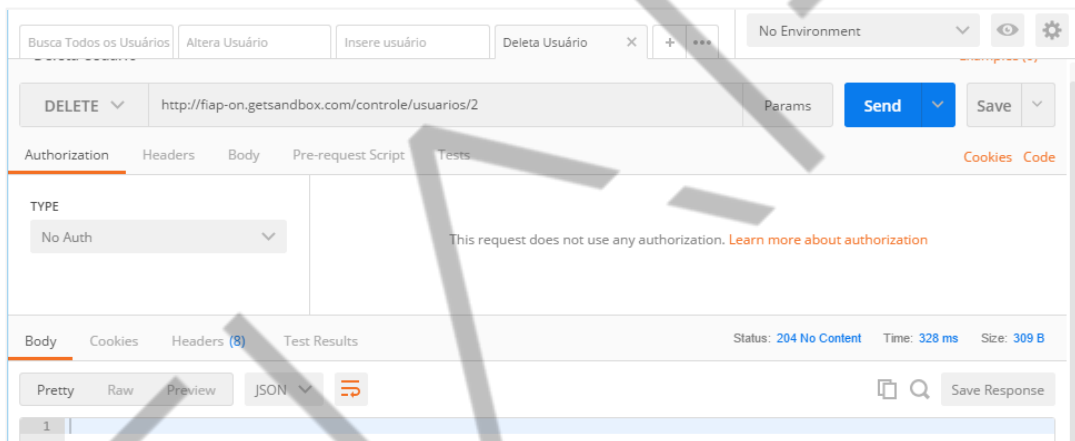


Figura 14 – Criar e enviar uma requisição DELETE no Postman
Fonte: Elaborado pelo autor (2017)

Importante comentar que o DELETE pode disparar uma ação na API que exclui o recurso ou apenas a marca para não ser mais disponibilizado via GET. Isso dependerá da regra de negócio e da forma como o *back-end* está desenvolvido.

Como resultado, receberemos da API o Status Code 204, que indica que o servidor cumpriu com êxito a solicitação e que não há conteúdo adicional no corpo da resposta.

Se fizermos uma busca ao usuário excluído em <https://63fd457c677c4158731bddb4.mockapi.io/usuarios/3>, teremos como resposta o Status Code 404 Not Found e uma mensagem tratada pela API, conforme imagem abaixo:

Web Services com Spring Boot

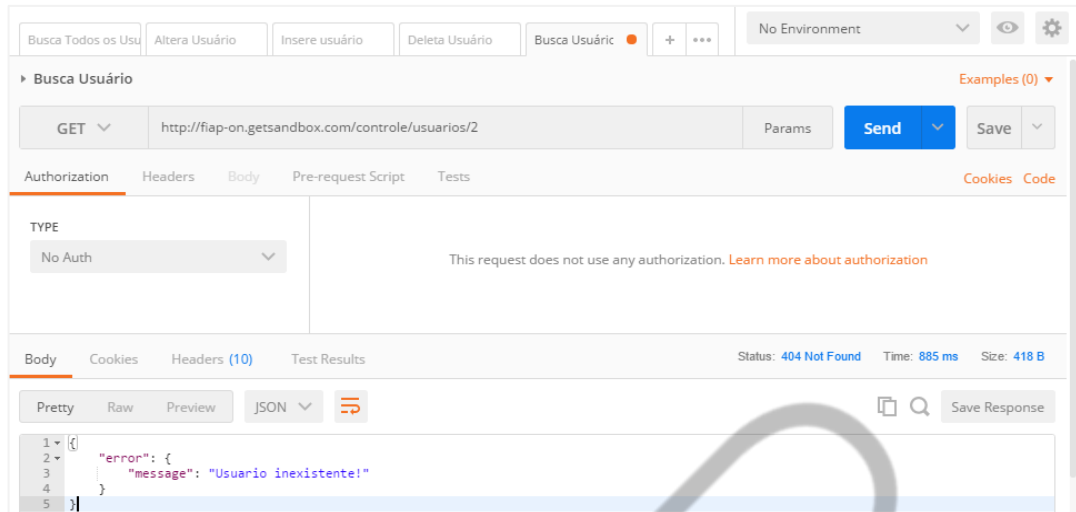


Figura 15 – Criar e enviar uma requisição de teste do DELETE no Postman
Fonte: Elaborado pelo autor (2017)

Por fim, vamos testar se as operações Update (PUT), Create (POST) e Delete (DELETE) deram certo. Volte na aba Busca Todos os Usuários e clique em Send.

Utilizando GET, obterá um JSON com o usuário de id 3 criado e sem o usuário 2, que excluimos no passo anterior.

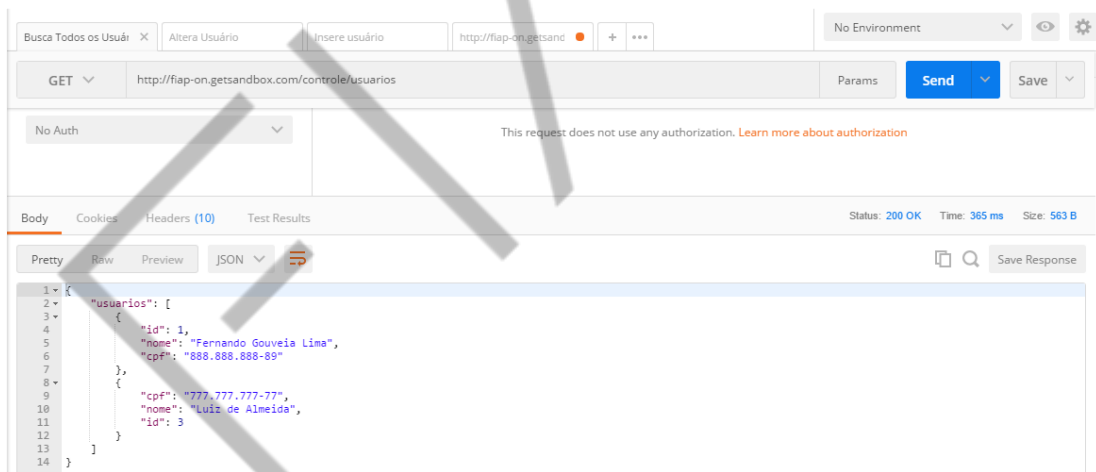


Figura 16 – Enviar a requisição GET de teste no Postman
Fonte: Elaborado pelo autor (2017)

Novamente, recebemos o Status Code 200 OK, informando que a requisição foi processada com sucesso e que há informação contida no Body da resposta.

1.4.4 Respostas padronizadas

Como podemos perceber, em todas as respostas da nossa API recebemos um código de status. Nas requisições PUT (Update) e POST (Create), incluímos no cabeçalho HTTP da solicitação uma informação importante, o formato da representação contida no Body. Em nossos exemplos, utilizamos o campo Content-Type com o valor application/json para informar à API que enviamos uma representação JSON para criar ou alterar um recurso.

Desta forma, o cliente e o serviço utilizaram propriedades do protocolo HTTP para trocar informações adicionais ao conteúdo das mensagens e, se analisarmos bem, informações muito relevantes.

Vamos entender um pouco mais sobre esse contexto e descobrir como o estilo REST novamente se beneficia da arquitetura Web e do protocolo HTTP.

1.4.4.1 HTTP Status Code

Como toda requisição enviada a um Servidor Web via HTTP retorna um código de status, os serviços RESTful utilizam parte da mensagem de resposta do protocolo para informar ao cliente sobre o resultado da solicitação que fez.

O HTTP define dezenas de códigos de status que podem ser usados para retornarmos tal resultado. Em face da quantidade, os Status Codes estão divididos em cinco categorias, apresentadas a seguir:

CATEGORIA	DESCRIÇÃO
1xx: informativo	Esta categoria dos códigos de status indica uma resposta provisória de solicitação recebida, dando continuidade ao processo.
2xx: Sucesso	A categoria 2xx indica que a ação solicitada pelo cliente foi recebida, compreendida, aceita e processada com êxito.
3xx: Redirecionamento	Indica que o cliente deve tomar alguma ação adicional para completar seu pedido.
4xx: Erro do cliente	Esta categoria retorna códigos para os casos em que o cliente pode ter cometido um erro.
5xx: Erro do servidor	A categoria 5xx indica que o servidor assumiu um erro ao processar a solicitação.

Quadro 1 – Status Code
Fonte: Elaborado pelo autor (2017)

Conhecer os códigos de status HTTP e empregá-los corretamente, de acordo com o status do processamento, é uma responsabilidade importante que cabe aos desenvolvedores de APIs RESTful.

Sendo assim, vamos conhecer alguns dos principais códigos e dos empregos mais comuns, baseados na RFC7231 <https://tools.ietf.org/html/rfc7231>.

1.4.4.2 Uso Geral

- **301 Moved Permanently:**

Significa que o recurso solicitado foi realocado permanentemente. A resposta deve conter um cabeçalho Location com a nova URL.

- **303 See Other:**

A requisição foi processada, mas a resposta ao pedido pode ser encontrada em URI diferente e deve ser recuperada usando um método GET nesse recurso. O cabeçalho Location informa onde a resposta do processamento está.

- **307 Temporary Redirect:**

Semelhante ao 301, só que indicando que o redirecionamento é temporário.

- **400 Bad Request:**

O pedido não pôde ser entendido pelo servidor devido à sintaxe com problemas.

- **401 Unauthorized:**

O pedido requer autenticação de usuário, as credenciais de autenticação estão faltando ou, se fornecidas, foram recusadas para acessar o recurso.

- **403 Forbidden:**

O servidor entendeu o pedido, mas está se recusando a cumpri-lo. A autorização não ajudará e o pedido não deve ser repetido.

- **404 Not Found:**

O servidor não encontrou nada que corresponda ao Request-URI.

- **405 Method Not Allowed:**

O método especificado não é permitido para o recurso identificado.

- **409 Conflict:**

O pedido não pôde ser concluído devido a um conflito com o estado atual do recurso. Este código é permitido em situações em que se espera que o usuário possa resolver o conflito e reenviar a solicitação.

Um exemplo comum é a criação de um recurso que deveria ser único, mas entra em conflito com um recurso semelhante, já existente.

- **410 Gone:**

O recurso não está disponível e é provável que a condição seja permanente.

- **415 Unsupported MediaType:**

Aponta que o servidor se recusa a atender ao pedido porque a solicitação está em um formato não suportado pelo método no recurso. Exemplo: solicitação de um JSON e o retorno é um recurso PDF.

- **500 Internal Server Error:**

Indica que o servidor encontrou uma condição inesperada que impediu o cumprimento do pedido.

- **503 Service Unavailable:**

Mostra que, para o servidor, atualmente não é possível lidar com o pedido devido a uma sobrecarga temporária ou manutenção programada, o que, provavelmente, será atenuado após algum tempo.

- **504 Gateway Timeout:**

Exibe que o servidor, enquanto atuava como um gateway ou proxy, não recebeu uma resposta a tempo de um servidor que precisava acessar para completar o pedido ou a solicitação não pode ser atendida devido a alguma outra falha que causou timeout.

1.4.4.3 GET

- **200 OK:**

O pedido foi bem-sucedido e a resposta contém a representação solicitada.

- **302 Found:**

Indica que o recurso de destino reside temporariamente em um URI diferente e funciona como uma resposta de redirecionamento em que a URI temporária pode ser obtida no cabeçalho da resposta.

- **304 Not Modified:**

Aponta que uma solicitação GET condicional foi recebida e resultaria em uma resposta 200 (OK) se não fosse pelo fato de não ser necessário retorno, pois o cliente possui uma representação válida do recurso solicitado.

1.4.4.4 POST ou PUT

- **201 OK:**

Exibe que o pedido foi cumprido e resultou em um ou mais novos recursos criados. Tipicamente, a identificação do recurso criado e uma representação dele são retornadas.

1.4.4.5 PUT

- **200 OK:**

O pedido foi bem-sucedido e o corpo da resposta contém a representação atualizada.

1.4.4.6 DELETE

- **204 No Content:**

O código de status 204 (Sem Conteúdo) indica que o servidor cumpriu com êxito a solicitação e que não há conteúdo adicional no corpo da resposta.

1.4.4.7 POST, PUT e DELETE

- **202 Accept:**

O pedido foi aceito para processamento, mas não foi concluído. Não há garantias sobre o resultado do processamento posterior. O propósito deste código é permitir ao

servidor receber e confirmar solicitações para outros processamentos. Não há funcionalidades no HTTP para reenviar um status de forma assíncrona quando o processamento posterior finalizar.

1.4.4.8 Use mensagens auto descritivas

Cada mensagem de uma aplicação RESTful deve incluir informações suficientes para descrever como deve ser processada. Os recursos são desacoplados de sua representação para que seu conteúdo possa ser acessado em vários formatos, como o JSON, que vimos em nossos exemplos, ou XML, HTML, TXT, entre outros Media Types.

Metadados sobre o recurso estão disponíveis e devem ser usados, por exemplo, para a aplicação avisar erros ou sucessos das operações (Status Code), informar o formato de representação apropriado (Key e Value) e executar autenticação ou controle de acesso (Authorization) que veremos mais à frente.

62 NÃO MANTENHA O ESTADO – STATELESS

Você sabe o que é a arquitetura REST? Ela é um estilo de comunicação entre cliente e servidor que não mantém o estado no servidor. Diferentemente de outras abordagens, que usam um container web para manter o estado dos dados trocados, a arquitetura REST troca mensagens sem manter o estado no servidor.

Mas o que isso significa? Significa que todas as informações necessárias para lidar com uma solicitação devem estar contidas dentro da própria solicitação, como parte da URL, dos parâmetros enviados, do corpo (body) ou dos cabeçalhos HTTP da mensagem.

A URL é utilizada para identificar um recurso no servidor de forma exclusiva, e o corpo da mensagem contém o estado atual ou a mudança de estado desse recurso. Quando o servidor retorna uma resposta, ela é enviada para o cliente através de cabeçalhos, código de status e corpo da resposta. Após o processamento da mensagem, nenhuma informação é armazenada em variáveis de memória no servidor.

A comunicação sem estado proporcionada pela arquitetura REST traz algumas vantagens, como maior escalabilidade. Isso acontece porque o servidor não precisa manter, atualizar ou comunicar nenhum tipo de estado do cliente, tornando as comunicações consecutivas independentes do servidor que as atende. Além disso, os balanceadores de carga da rede não precisam se preocupar com configurações adicionais para controlar sessões.

Em resumo, a arquitetura REST é uma forma de comunicação entre cliente e servidor que não mantém estado no servidor. Todas as informações necessárias para lidar com uma solicitação estão contidas dentro da própria solicitação, trazendo vantagens como escalabilidade e independência do servidor que atende as comunicações. Se você precisa de uma comunicação simples e escalável, a arquitetura REST pode ser a solução ideal para o seu projeto.

2.1 Utilizar o Cache

Como na World Wide Web, os clientes podem armazenar respostas em cache. As respostas implícitas ou explícitas são definidas como cacheáveis ou não e servem para impedir que os clientes reutilizem dados obsoletos ou inapropriados em resposta aos pedidos adicionais. Se bem gerenciado, o armazenamento em cache reduz ou elimina algumas interações cliente-servidor, melhorando ainda mais a escalabilidade e o desempenho das soluções REST.

2.3 Utilizar a arquitetura Client-Server

Como podemos perceber, a interface uniforme separa os clientes dos servidores. Tal separação permite que os clientes não fiquem preocupados com o processamento e a persistência dos dados, que são responsabilidades dos servidores, possibilitando que a portabilidade do código cliente seja evoluída livremente.

Por sua vez, os servidores não se preocupam com a interface do usuário ou com a sessão do usuário, de modo que os servidores possam ser simples, performáticos e escaláveis.

Em nossos exemplos, para o cliente Postman, pouco importa se a API RESTful foi desenvolvida em Java, .NET ou JavaScript. Da mesma forma, nossa API JavaScript pouco se importa se a requisição vem de uma aplicação cliente como o Postman ou de um Browser.

Nessa arquitetura, servidores e clientes podem ser desenvolvidos, mantidos e substituídos livremente, desde que a maneira como trocam as mensagens não seja alterada. Perceba que, em certo ponto, voltamos a comentar sobre Stateless.

Como exemplo, se um servidor que acabou de responder a um cliente for desligado para manutenção, como cada mensagem de solicitação deve conter o estado, outro servidor pode assumir uma solicitação subsequente e dar andamento no atendimento ao cliente, sem impacto algum para ambos.

Da mesma forma, se incluirmos um novo servidor de aplicação na Farm com a nossa API, estará apto a atender à mesma solicitação subsequente, pois não depende dos históricos mantidos em sessão.

Importante para escalabilidade, não?

2.2 Arquitetar o sistema em camadas

Um cliente que utilizar um sistema em camadas normalmente não pode dizer se está trocando mensagens com o servidor final ou um servidor intermediário ao longo do caminho. Servidores intermediários podem melhorar a escalabilidade do sistema, possibilitando o balanceamento de carga e fornecendo caches.

Além disso, camadas adicionais podem ser utilizadas para impor políticas de segurança à solução, como é feito pelo firewall.

2.3 Forneça código sobre demanda

Segundo Fielding, essa restrição REST permite que a capacidade funcional do cliente seja ampliada, baixando e executando código na forma de applets ou scripts. Ainda, segundo ele, permitir que os recursos sejam baixados após a implantação melhora a extensibilidade do sistema, mas também reduz a visibilidade e, portanto, é uma restrição opcional dentro do estilo REST.

2.4 Aplicações RESTful

Para ser considerada como RESTful, a aplicação deve estar aderente às cinco restrições obrigatórias do estilo REST que comentamos: Uniform Interface, Stateless, Cacheable, Client-Server e Layered System, a única considerada como opcional é a Code on Demand.

Claro que as características de hardware, SO e linguagem de programação influenciam, mas podemos afirmar que a conformidade com o estilo fará com que qualquer API adquira propriedades desejáveis, como melhora no desempenho, maior escalabilidade, simplicidade, facilidade de manutenção, portabilidade e confiabilidade.

3 BACK-END COM SPRING BOOT

3.1 Frameworks

Você viu nos capítulos anteriores que frameworks nada mais são do que uma abstração ou biblioteca, contendo um conjunto de códigos e módulos cujo comportamento-padrão pode ser alterado por meio de uma configuração ou de um código adicional.

Em suma, empregamos frameworks em projetos para não termos de escrever todo o código a partir do zero. Para isso, utilizamos módulos já criados e validados pelo mercado, de forma que apenas seja necessário configurá-los e, eventualmente, customizar alguma operação.

Vale citar que existem algumas diferenças entre um framework e uma biblioteca (*library*), entre as quais:

- ***Inversion of Control (IoC)***: o controle do fluxo de execução do código é feito pelo framework, não pela aplicação principal.
- ***Extensibilidade***: é possível expandir as capacidades de um framework por meio de um *overriding* de métodos ou de acréscimo de classes.
- ***Imutabilidade***: enquanto é possível estender um framework, não é possível modificar (pelo menos não é fácil fazer isso) o seu código interno.

O uso de um framework acelera o processo de construção das partes mais básicas de um sistema e permite que determinadas etapas do processo possam ser padronizadas. Além disso, a própria comunicação e documentação de projeto se tornam mais “fáceis”, no sentido de bastar a menção aos frameworks utilizados e suas respectivas versões.

Entretanto, alguns cuidados precisam ser tomados com o uso de um framework. Por isso é necessário citar alguns pontos fracos que devem ser considerados:

- ***Manutenção***: caso ocorra algum *bug* interno ao framework, não será prático fazer a sua correção, visto que o código interno não pode ser alterado.

- **Migração:** se um determinado framework possui uma evolução rápida, em algumas situações, gasta-se mais tempo testando a migração para a nova versão do que na hipótese de não haver uso de framework.
- **Performance:** em aplicações de missão crítica, o uso de frameworks pode causar um *overhead* (processamento adicional), simplesmente pelo fato de que o framework acrescenta mais etapas intermediárias de processamento, resultando em menor *performance* do sistema.

No mundo Java, pensando em *stacks* completos (nesse caso, que atendam ao padrão arquitetural MVC), o mais utilizado é o Spring Boot com o Spring MVC.

3.2 Spring Framework

Anteriormente, apresentamos o Spring Framework com o objetivo de tornar o desenvolvimento de aplicações mais rápido.

Ao longo de sua evolução, o framework passou não só a integrar outros componentes do mundo Java, mas também desenvolveu os seus próprios módulos.

Existem módulos e integrações desde a parte de persistência até a parte web, incluindo a parte de APIs RESTFul.

Veremos com mais detalhes um módulo chamado Spring Boot, que facilitará bastante na construção de nossa aplicação MVC.

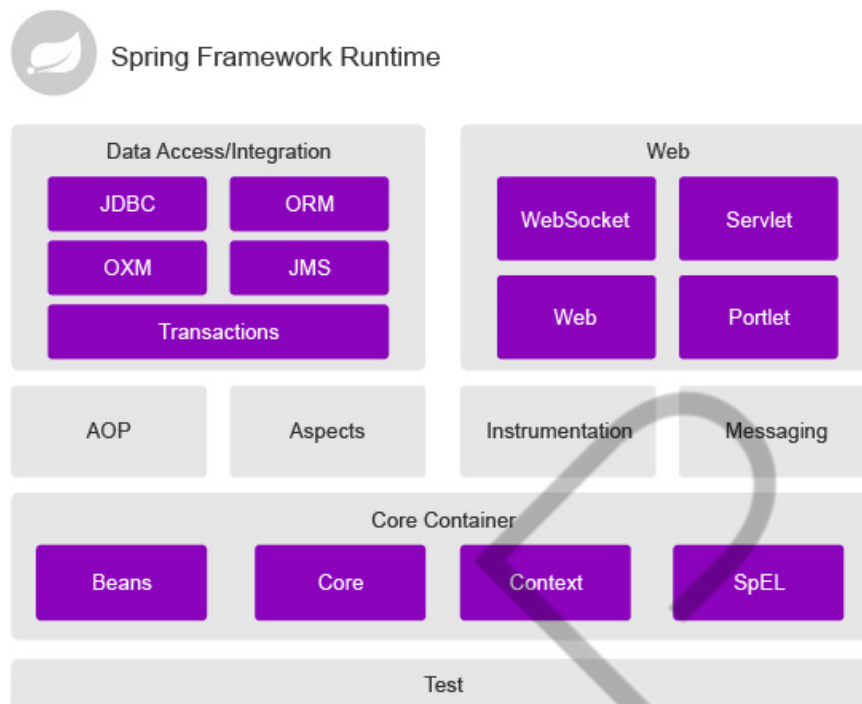


Figura 17 – Spring Framework
Fonte: Spring (2019)

3.3 Spring Boot

É possível ver rapidamente a lista de módulos que o ecossistema do Spring fornece por meio deste link: <<https://spring.io/projects>>. São em torno de 30 módulos, cada um deles com uma extensa possibilidade de configurações.

Parece ser complexo demais para quem quer começar a explorar o Spring, não é? É por esse motivo que foi criado um projeto chamado Spring Boot, que utiliza algumas convenções sobre como construir uma aplicação Spring, acelerando bem o aprendizado inicial sobre o framework.

Para criar um projeto Spring Boot, existe uma ferramenta muito prática, a Spring Initializr, e há dois caminhos para utilizá-la:

- **Via Website:** utilizar o site <<https://start.spring.io/>> e parametrizar o seu projeto. No final do assistente, é gerado um arquivo zipado com os arquivos de projeto e configurações-padrão.
- **Via IDE:** utilizar o IntelliJ IDEA Ultimate Edition, que tem um assistente embutido, mas outras IDEs, como o Eclipse e o NetBeans, possuem

também plugins que permitem a criação de um projeto por meio de um assistente.

Vamos criar um projeto agora?

Para criarmos um projeto utilizando o website, basta entrarmos com os dados de projeto como na Figura “Página de criação de projeto no Spring Initializr”:

The screenshot shows the Spring Initializr web interface. On the left, under 'Project', 'Maven Project' is selected. Under 'Language', 'Java' is selected. Under 'Spring Boot', version '2.6.3' is selected. Under 'Project Metadata', the Group is 'br.com.flap', Artifact is 'example', Name is 'example', Description is 'Demo project for Spring Boot', and Package name is 'br.com.flap.example'. Under 'Packaging', 'Jar' is selected, and under 'Java', version '11' is selected. On the right, under 'Dependencies', several options are listed: 'Spring Web' (WEB), 'Rest Repositories' (WEB), 'Thymeleaf' (TEMPLATE ENGINES), 'Spring Data JPA' (SQL), and 'H2 Database' (SQL). Each dependency has a brief description. A button 'ADD DEPENDENCIES... CTRL + B' is at the top right of the Dependencies section.

Figura 18 – Página de criação de projeto no Spring Initializr
Fonte: Elaborado pelo autor (2022)

No lado esquerdo da página Spring Initializr, devemos colocar os módulos adicionais que queremos como dependências no Maven e, nesse caso, os seguintes módulos devem ser ativados:

- Web => Spring Web
- Web => Rest Repositories
- Template Engines => Thymeleaf
- SQL => Spring Data JPA
- SQL => H2 Database

Depois, basta clicarmos em *Generate* e pronto, teremos o arquivo zipado com o esqueleto de projeto criado.

Para o projeto que construiremos a seguir, utilizaremos a estrutura apresentada na Figura “Estrutura do Projeto”:

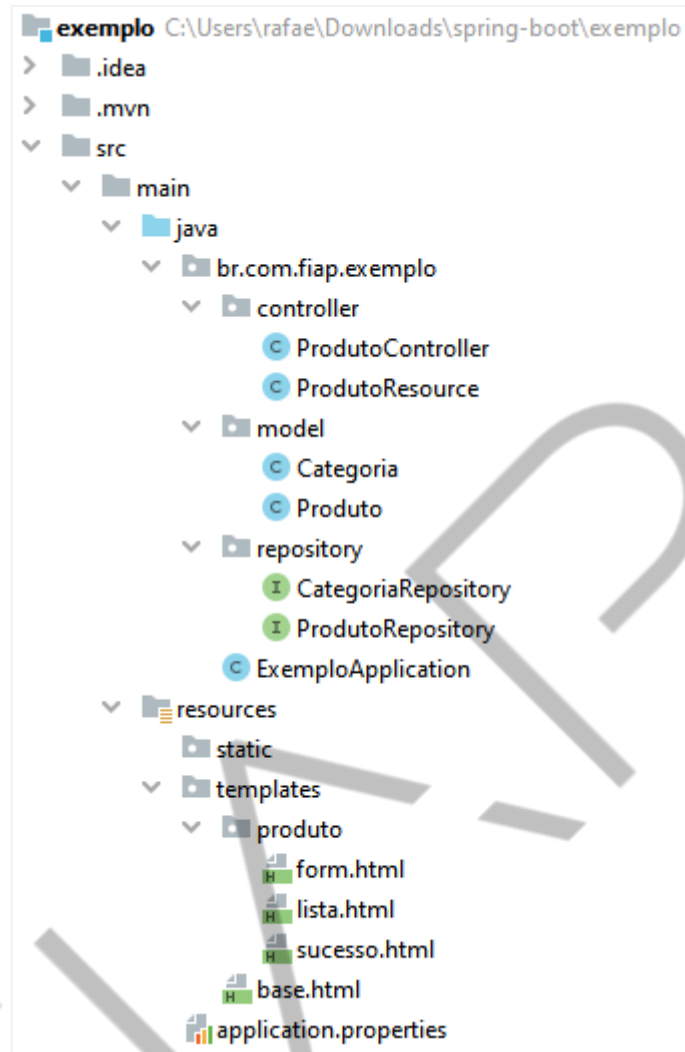


Figura 19 – Estrutura do Projeto
Fonte: Elaborado pelo autor (2019)

Não se preocupe se o seu projeto não possuir todos os arquivos listados. No final do hands-on, teremos todos esses arquivos e saberemos o papel de cada um deles no projeto.

Vale comentar sobre três arquivos antes de começarmos a construir o nosso primeiro projeto em Spring Boot:

- **ExemploApplication.java:** este arquivo será o nosso *main*, no qual a aplicação será inicializada. Tirando situações em que configurações adicionais avançadas devam ser feitas, não se costuma mexer neste arquivo.
- **application.properties:** este é o arquivo de *properties* que mantém as configurações de conectores ao banco de dados e demais configurações globais de projeto.

- **pom.xml (não está constando na figura):** é o arquivo do Maven que cuida, principalmente, das dependências. Se você habilitou os módulos no Spring Initializr, essas dependências serão listadas neste arquivo automaticamente.

Neste capítulo, vamos trabalhar a construção do projeto na parte de *backend* do Spring Boot, focando, principalmente, no *Model* e em parte do *Controller*, ficando o capítulo a seguir com o final do *Controller* e a *View* (assim, construindo toda a arquitetura MVC).

Nas próximas seções, construiremos parte a parte do *back-end* do projeto, começando pelo banco de dados, seguido pelo Spring Data JPA (que faz a integração entre o banco de dados e o projeto) e terminando com o REST *Controller* (que externaliza um acesso RESTFul).

3.4 Banco de Dados (H2 Database)

A configuração de uma conexão de banco de dados é muito prática no Spring Boot. Basta listar as configurações no arquivo **application.properties** (de maneira similar ao realizado no **persistence.xml** ou no **hibernate.properties**).

Vamos colocar, inicialmente, o snippet de configuração para o H2 Database, por ser um banco de dados muito prático de se utilizar para teste, pois ele sobe localmente e possui interface web integrada ao Spring Boot. Mas caso queira outro banco de dados, é muito fácil encontrar as configurações na Internet.

```
# URL de acesso ao banco, neste caso, um arquivo chamado
fiap
spring.datasource.url=jdbc:h2:file:~/fiap
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=user
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update

# Console Web Habilitado
```

```
spring.h6.console.enabled=true

# URL para acessar o console web
spring.h6.console.path=/h2
```

Código-fonte 6 – Configuração do H2 Database no Spring Boot
Fonte: Elaborado pelo autor (2019)

Essas configurações cuidam de listar qual é o Database Driver a ser utilizado e as credenciais de acesso. O DDL-Autoatualiza o schema do banco de dados conforme as *annotations* das *entities* (mais à frente, teremos mais informações) e as duas configurações finais, que veremos logo a seguir.

Colocaremos essa configuração no **application.properties** e executaremos o projeto (por meio do **ExemploApplication.java**).

Repare que não foi necessário configurar um container Tomcat por conta de o módulo Spring Web já ter embutido um Tomcat no projeto Spring Boot, como na Figura “Tela de inicialização do projeto Spring Boot”:

```
main] org.hibernate.Version : HHH000412: Hibernate Core {5.3.11.Final}
main] org.hibernate.cfg.Environment : HHH000206: hibernate.properties not found
main] o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations {5.0.4.Final}
main] org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate.dialect.H2Dialect
main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
main] aWebConfiguration$JpaWebMvcConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries
main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
main] br.com.fiap.exemplo.ExemploApplication : Started ExemploApplication in 14.907 seconds (JVM running for 19.368)
```

Figura 20 – Tela de inicialização do projeto Spring Boot
Fonte: Elaborado pelo autor (2019)

Com o sistema no ar, vamos acessar a URL: <http://localhost:8080/h2> e utilizar as credenciais que estão listadas no **application.properties** (inclusive, o DataSource URL).

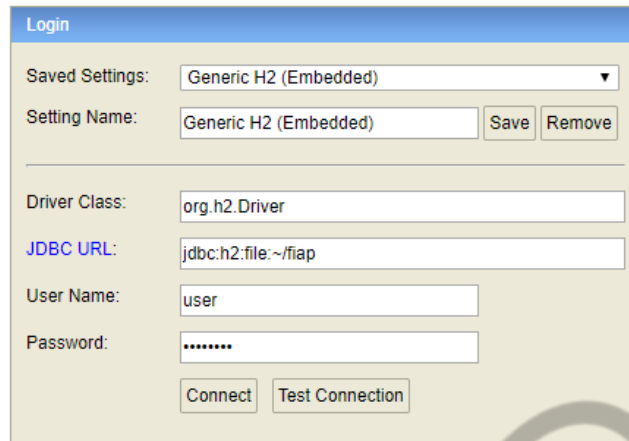


Figura 21 – Tela de login do H2
Fonte: Elaborado pelo autor (2019)

Conseguindo logar, chega-se à interface web do H2 Database, na qual podemos disparar *queries* e consultar o banco de dados. Prático, não?

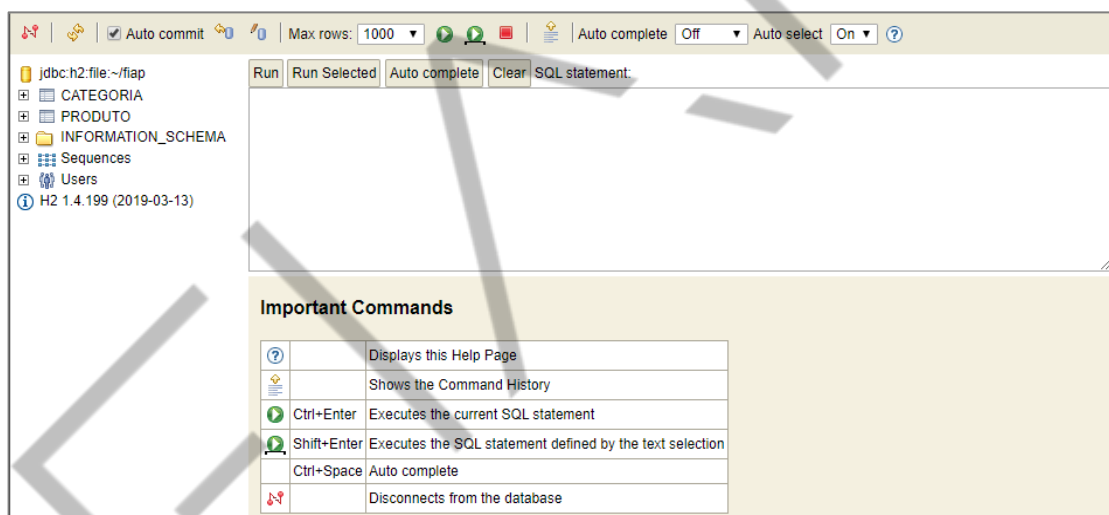


Figura 22 – Interface Web do H2
Fonte: Elaborado pelo autor (2019)

Agora que temos o banco de dados no ar, podemos partir para a configuração do Spring Data JPA, que fará o papel de intermediário entre o Spring Boot e o H2 Database.

3.5 Spring Data JPA

Spring Data é um módulo do Spring Framework responsável por fazer a integração com diversas classes de banco de dados. Vale a pena conferir a

diversidade de conectores possíveis com ele em: <<https://spring.io/projects/spring-data>>.

No caso do nosso projeto, vamos utilizar o Spring Data por cima do JPA (biblioteca de persistência para bancos relacionais). Além de reutilizar as *annotations* do JPA, ele automatiza o processo de criação de repositórios (*repositories*).

Inicialmente, configuraremos a entidade Produto (colocado dentro de um subpacote model, vide figura sobre a estrutura de projeto), com o Código-fonte “Código da Entidade Produto”:

```
@Entity
@SequenceGenerator(name = "produto", sequenceName =
"SQ_PRODUTO", allocationSize = 1)
public class Produto {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
generator = "produto")
    private int codigo;

    @NotBlank(message = "Nome obrigatório!")
    @Size(max = 50)
    private String nome;

    @Min(value=0, message = "Preço não pode ser negativo")
    private double preco;

    private boolean novo;

    @Past
    private LocalDate dataFabricacao;

    // criar getters e setters aqui.
}
```

Código-fonte 7 – Código da Entidade Produto
Fonte: Elaborado pelo autor (2019)

A classe Produto nada mais é do que um *Bean* com *annotations* JPA (mapeando a entidade, depois definindo a chave primária com uma *sequence*

associada) e as variáveis que refletirão em colunas na tabela Produto no banco de dados. Lembre-se de criar os *getters* e *setters* dele.

Além das *annotations* do JPA, são colocadas *annotations* associadas à validação de campos. No Spring MVC, é possível fazer a validação em apenas um local. Nesse caso, foram utilizadas as seguintes validações:

- “**nome**”: não pode ser em branco (@NotBlank) e tamanho máximo de 50 caracteres (@Size com valor 50).
- “**preco**”: não pode ser negativo (@Min com valor zero).
- “**dataFabricacao**”: data tem que ser no passado (@Past).

É possível também colocar mensagens customizadas de erro para cada validação que falhar, por meio do parâmetro *message*. Veremos o teste da validação no próximo capítulo, quando construirmos a interface.

A Tabela “Código da Entidade Produto” traz as *annotations* de validação mais utilizadas:

Anotação	Descrição
@AssertFalse	O valor deve ser falso.
@AssertTrue	O valor deve ser verdadeiro.
@DecimalMax	Valor deve ser menor ou igual.
@DecimalMin	Valor deve ser maior ou igual.
@Future	A data deve estar no futuro.
@Past	A data deve estar no passado.
@Min	O valor deve ser maior ou igual.
@Max	O valor deve ser menor ou igual.
@NotNull	O valor não pode ser nulo.
@Size	A quantidade de elementos deve estar entre o min e max.
@Null	O valor deve ser nulo.
@Pattern	O valor deve obedecer à expressão regular.

Quadro 2 – Annotations de Validação
Fonte: Elaborado pelo autor (2019)

Agora vamos configurar o `ProdutoRepository` (colocado dentro de um subpacote chamado `repository`, vide figura sobre a estrutura do projeto), com o “Código do `ProdutoRepository`”.

```
@Repository
public interface ProdutoRepository extends
JpaRepository<Produto, Integer> {

    List<Produto> findByName(String prod);

    List<Produto> findByNovo(boolean novo);

    List<Produto> findByNameAndNovo(String prod, boolean
novo);

    List<Produto> findByPrecoGreaterThan(double preco);
}
```

Código-fonte 8 – Código do `ProdutoRepository`
Fonte: Elaborado pelo autor (2019)

Repare que não foram colocadas as assinaturas referentes aos métodos de CRUD para o banco de dados devido ao `JpaRepository<Entity, Id>` já injetar o código necessário para essa realização. Portanto, teremos o seguinte mapeamento do CRUD:

- SELECT: `findAll()` ou `findById()`
- INSERT ou UPDATE: `save()`
- DELETE: `deleteById()`

Isso torna a criação de repositórios comuns (ou um DAO – *Data Access Object*) muito rápida.

Existem alguns métodos cuja assinatura faz com que o Spring Data injete um código especial, como o `findByName`, que fará a busca no campo nome da tabela `Produto`. É possível também combinar lógica condicional para a busca, como nos demais exemplos no código, lembrando que existe uma nomenclatura que deve ser seguida.

Mais informações em: <<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.query-methods.details>>.

Só falta um detalhe para finalizarmos a parte do Spring Data, que é a instanciação do `ProdutoRepository`. Para os desenvolvedores que já estão acostumados com as técnicas clássicas de inicialização de um repositório, tradicionalmente, se criaria um `EntityManager` ou um `EntityManagerFactory`. Mas no caso do Spring Data, isso não será necessário. Basta declarar o `ProdutoRepository` na classe em que será utilizada e utilizar a annotation `@Autowired`, como no código “Instanciação do `ProdutoRepository`”.

```
public class ProdutoResource {  
  
    @Autowired  
    private ProdutoRepository produtoRepository;  
  
    // demais métodos  
  
}
```

Código-fonte 9 – Instanciação do `ProdutoRepository`
Fonte: Elaborado pelo autor (2019)

A annotation `@Autowired` instancia e injeta o repositório na variável correta. Com isso, não há necessidade de nos preocuparmos em criar uma factory ou um singleton para a instanciação do repositório, o Spring Data cuida disso.

Agora podemos plugar o Spring Data com alguma classe ou serviço que possa prover ou consumir informação. Na próxima seção, faremos isso via API, por meio de um REST Controller. E, em outro capítulo, faremos isso via web (HTML).

3.6 REST Controller

No Spring Boot, é muito prático externalizar uma rota para a Internet, mas antes de mostrar como fazer isso, é importante saber como funciona um webservice RESTFul.

REST (*Representational State Transfer*) é um padrão sucessor do SOAP (*Simple Object Access Protocol*), utilizado em webservices modernos devido a uma série de vantagens arquiteturais, sendo algumas delas:

- O uso do JSON (*JavaScript Object Notation*) como formato de conteúdo, que é compacto e menos verboso do que o padrão XML (*eXtensible Markup Language*).
- A utilização do conceito de verbos como heurística da ação que pode ser realizada em determinada rota.
- O conceito de webservices leves e simples, retirando a necessidade do uso do WSDL (*WebService Description Language*) como protocolo de descrição do serviço.

Os webservices que suportam REST também costumam ser chamados de RESTFul, contendo verbos (ações) possíveis, entre os quais, quatro que são muito utilizados:

- GET: para obtenção de informação.
- POST: para inserir uma informação.
- PUT/PATCH: para atualizar uma informação.
- DELETE: para remover uma informação.

Note que esses verbos são análogos às operações de SELECT, INSERT, UPDATE e DELETE do SQL, respectivamente.

Com isso, podemos falar sobre o conceito de Resource, que é uma classe responsável por externalizar as rotas de acesso a um webservice (ou API).

Agora vamos para a parte prática: criaremos um arquivo chamado ProdutoResource dentro do subpackage controller (vide figura de estrutura do projeto), com o “Código-fonte do ProdutoResource”.

```
package br.com.fiap.example.resources;

import java.util.List;

import
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Controller;
```

```
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import
org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import
org.springframework.web.bind.annotation.ResponseStatus;
import
org.springframework.web.bind.annotation.RestController;

import br.com.fiap.example.domain.Produto;
import br.com.fiap.example.repository.ProdutoRepository;

@RestController
@RequestMapping("produto")
public class ProdutoResource {

    @Autowired
    private ProdutoRepository produtoRepository;

    @GetMapping
    public List<Produto> listar() {
        return produtoRepository.findAll();
    }
}
```

```
@GetMapping("{codigo}")
public Produto buscar(@PathVariable int codigo) {
    return produtoRepository.findById(codigo).get();
}

@ResponseStatus(code = HttpStatus.CREATED)
@PostMapping
public Produto cadastrar(@RequestBody Produto produto) {
    return produtoRepository.save(produto);
}

@PutMapping("{id}")
public Produto atualizar(@RequestBody Produto produto,
    @PathVariable int id) {
    produto.setCodigo(id);
    return produtoRepository.save(produto);
}

@DeleteMapping("{codigo}")
public void remover(@PathVariable int codigo) {
    produtoRepository.deleteById(codigo);
}

@GetMapping("pesquisa")
```

```
public List<Produto> buscar(@RequestParam(required =
false) String nome, @RequestParam(defaultValue = "false")
boolean novo) {
    return nome != null ?
    produtoRepository.findByNomeAndNovo(nome, novo) :
    produtoRepository.findByNovo(novo);
}

}
```

Código-fonte 9 – Código do ProdutoResource
Fonte: Elaborado pelo autor (2019)

Vamos, agora, estudar as annotations utilizadas neste código:

- **@RestController:** é a *annotation* que habilita a externalização dessa classe como um websevice RESTFul.
- **@RequestMapping:** habilita uma rota com caminho específico (produto, nesse caso) para todas as sub-rotas dentro da classe, ou seja, todas as sub-rotas deverão conter um /produto/ na URL.
- **@GetMapping:** habilita uma rota do verbo GET, que nada mais é do que obter a listagem de todos os elementos de um repository.
- **@GetMapping({id}):** habilita uma rota do verbo GET que recebe um ID de elemento a ser buscado.
- **@PathVariable:** é uma forma de avisar ao Spring MVC que uma determinada parte da URL virá como se fosse uma variável dentro do método.
- **@ResponseStatus:** contém o código HTTP de resposta e varia conforme o tipo de operação realizada. Vide a listagem de códigos HTTP em: <https://en.wikipedia.org/wiki/List_of_HTTP_status_codes>.
- **@PostMapping:** habilita uma rota do verbo POST para inserir um elemento.

- **@RequestBody**: faz o mapeamento dos dados que virão do *request* em um *Bean*. Nesse caso, quando os dados são recebidos, eles são colocados dentro de um *Bean* para uso dentro do método.
- **@PutMapping({id})**: habilita uma rota do verbo PUT para atualizar um elemento, passando o ID dele via URL.
- **@DeleteMapping({id})**: habilita uma rota do verbo DELETE para remover um elemento, passando o seu ID via URL.

Para testar as rotas, vamos utilizar um *plugin* chamado *Postman*, que será mostrado a seguir.

3.7 Teste da API REST

Agora que estamos com a API REST construída, vamos testar algumas das rotas utilizando um plugin do Chrome chamado Postman.

<<https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcdcbncdddmop?hl=en>> ou direto pelo serviço web <<https://www.getpostman.com/>>.

Abrindo o plugin, inicialmente, vamos testar o verbo GET (que faz a listagem de todos os produtos). Então, basta colocar a rota **http://localhost:8080/produto** no campo de endereços, selecionar GET no *dropdown* da esquerda e enviar uma requisição, conforme a Figura “Teste do GET no Postman”:

Web Services com Spring Boot

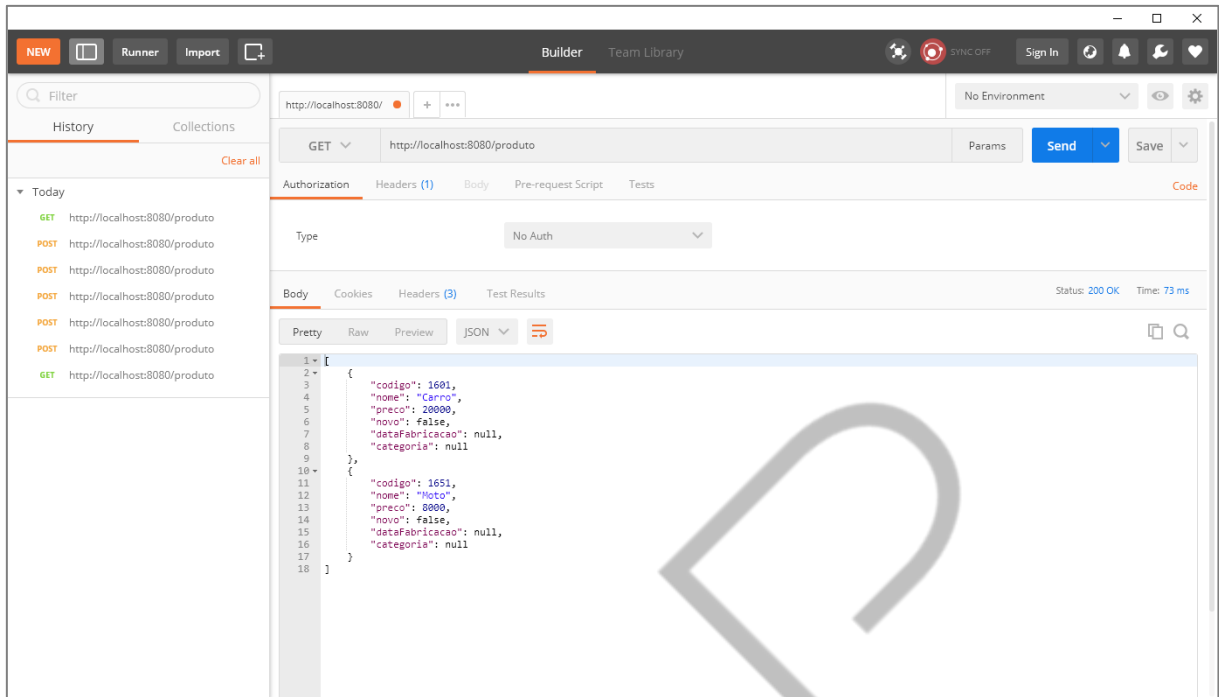


Figura 23 – Teste do GET no Postman
Fonte: Elaborado pelo autor (2019)

Caso a resposta tenha sido apenas dois colchetes [], não se preocupe, pois isso quer dizer que o JSON de resposta veio vazio (porque a tabela Produto está vazia), o que é esperado.

Vamos testar agora a inserção dos dados de um produto, que será realizada por meio de um JSON direto no verbo POST. Para isso, podemos manter o mesmo endereço no Postman (http://localhost:8080/produto) e trocar para POST no *dropdown* da esquerda.

Como vamos inserir as informações do produto via JSON. Precisamos ir na aba Body, colocar no formato *raw* e selecionar o formato JSON (application/json) para a requisição ser realizada no formato correto.

Assim, podemos enviar um JSON com os dados do produto a ser inserido, como no Código Fonte “JSON exemplo com um produto”:

```
{ "nome": "Lancha",  
  "preco": 18000.00 }
```

Código-fonte 11 – JSON exemplo com um produto
Fonte: Elaborado pelo autor (2019)

Web Services com Spring Boot

Esse produto tem as informações de nome e preço; os demais campos serão preenchidos com valores-padrão e o código do produto será preenchido automaticamente (lembre-se da *sequence* na *entity*).

Na Figura “Teste do POST no Postman”, mostramos a tela com as configurações do POST e o resultado de sua execução.

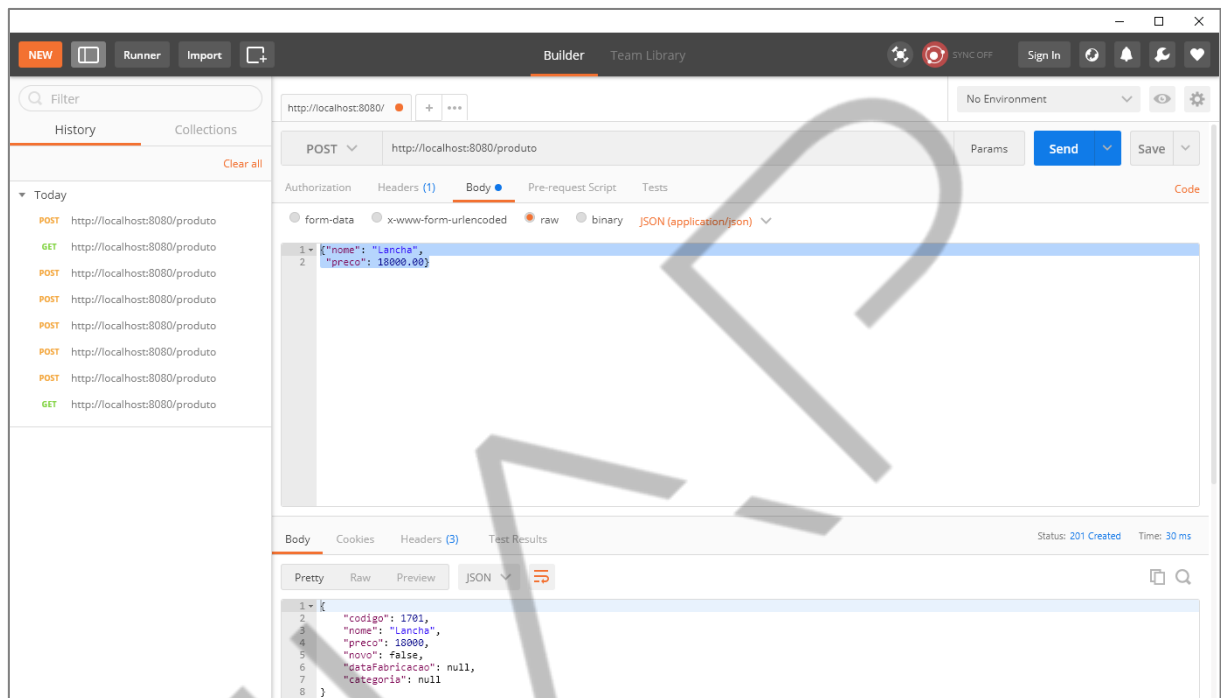


Figura 24 – Teste do POST no Postman
Fonte: Elaborado pelo autor (2019)

Repare que, na parte inferior da figura, é retornado o JSON com os dados que foram persistidos no banco ao se criar o registro; e, à direita, o código *201 Created*, informando que a operação foi realizada com sucesso.

Com isso, podemos testar o GET novamente para ver se aparece o produto que acabamos de inserir:

Web Services com Spring Boot

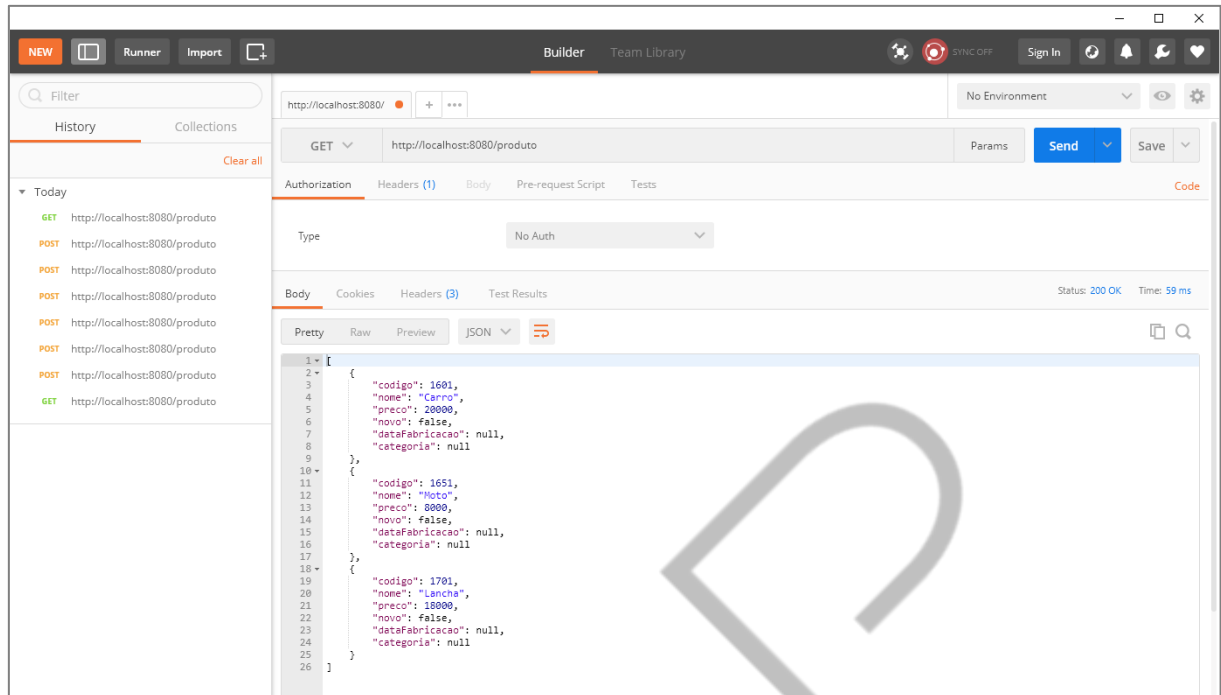


Figura 25 – Teste do GET no Postman
Fonte: Elaborado pelo autor (2019)

Agora, temos o registro da Lancha no banco de dados. Falta testarmos a alteração de um registro. Para isso, vamos mudar o preço e o nome da Lancha (com o código 1701, neste caso). Vamos alterar o verbo para PUT no *dropdown* e, na rota, precisamos informar o código do recurso a ser modificado, portanto, a URL passa a ser: <http://localhost:8080/produto/1701>.

No JSON, precisamos informar um novo nome e preço, como escrito no Código Fonte “JSON exemplo de um novo produto”:

```
{ "nome": "Nova Lancha",  
  "preco": 28000.00 }
```

Código-fonte 12 – JSON exemplo de um novo produto
Fonte: Elaborado pelo autor (2019)

Enviando a requisição pelo Postman, temos a seguinte tela:

Web Services com Spring Boot

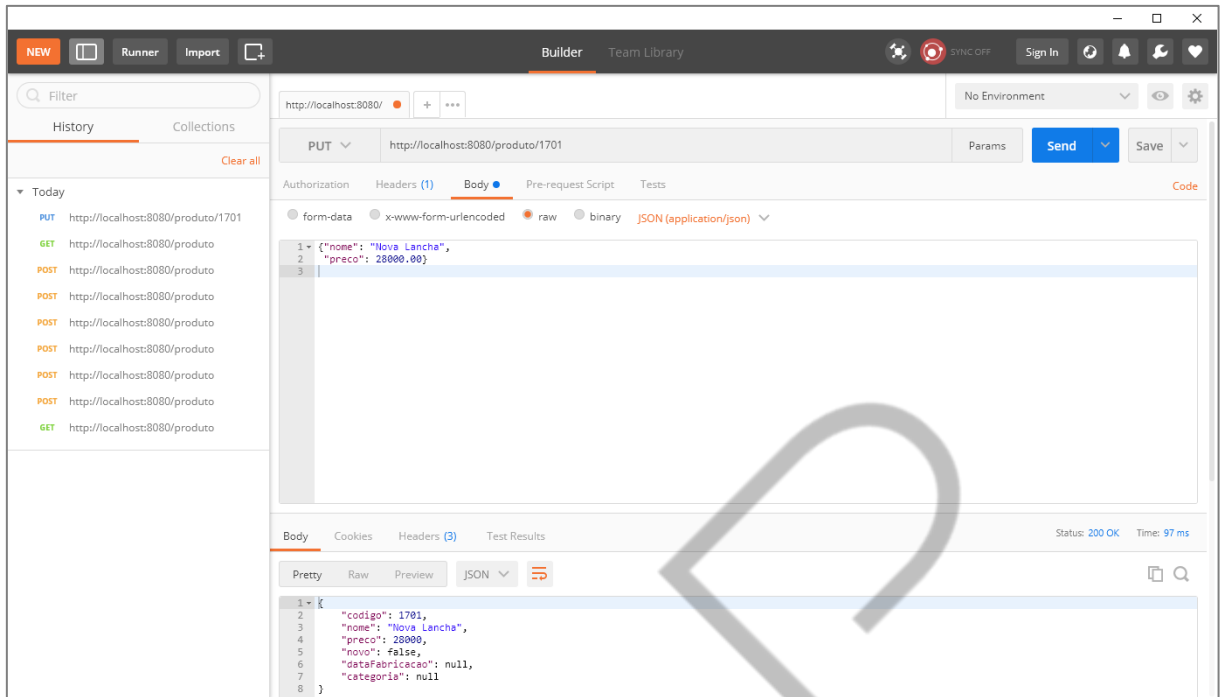


Figura 26 – Teste do PUT no Postman
Fonte: Elaborado pelo autor (2019)

Aproveitando para testar o GET por ID, vamos trocar para GET no *dropdown* da esquerda e manter a rota `<http://localhost:8080/1701>` para obter os dados apenas do código de produto apresentado na Figura “Teste do GET por ID no Postman”.

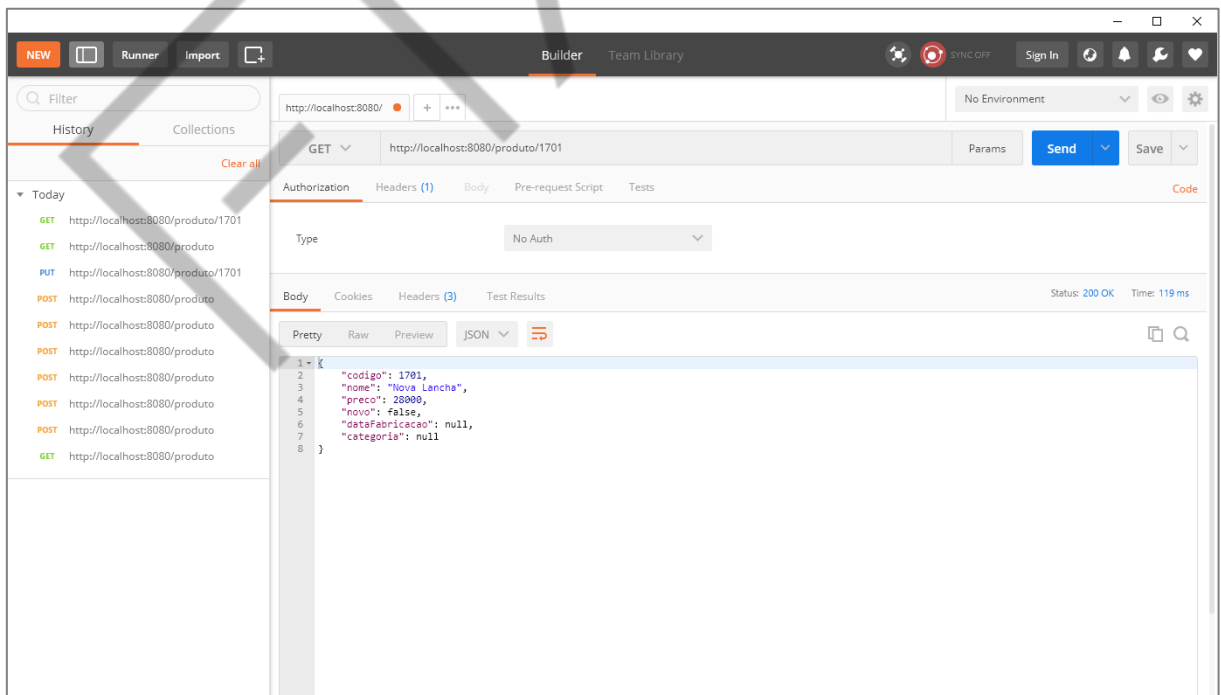


Figura 27 – Teste do GET por ID no Postman
Fonte: Elaborado pelo autor (2019)

Conseguimos ver que a alteração do nome e do preço foi realizada com sucesso.

E para finalizar, precisamos testar o verbo DELETE. Então, basta trocarmos o verbo para DELETE no *dropdown* da esquerda, mantermos a rota <http://localhost:8080/produto/1701> e dispararmos a requisição, conforme a Figura “Teste do DELETE no Postman”:

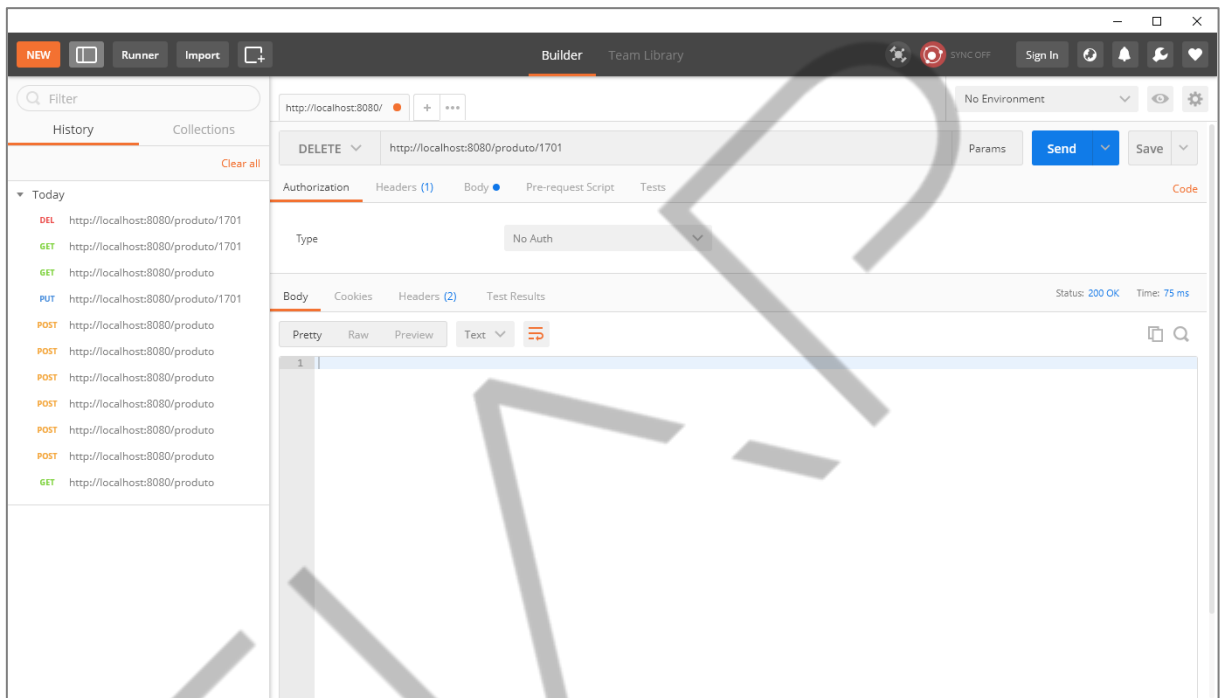


Figura 28 – Teste do DELETE no Postman
Fonte: Elaborado pelo autor (2019)

Assim, caso a requisição tenha sido feita corretamente, não haverá nenhuma mensagem de resposta além do *200 OK* à direita.

Vamos verificar fazendo um GET final. Devemos trocar o verbo para GET no *dropdown* da esquerda e colocar a rota original <http://localhost:8080/produto>:

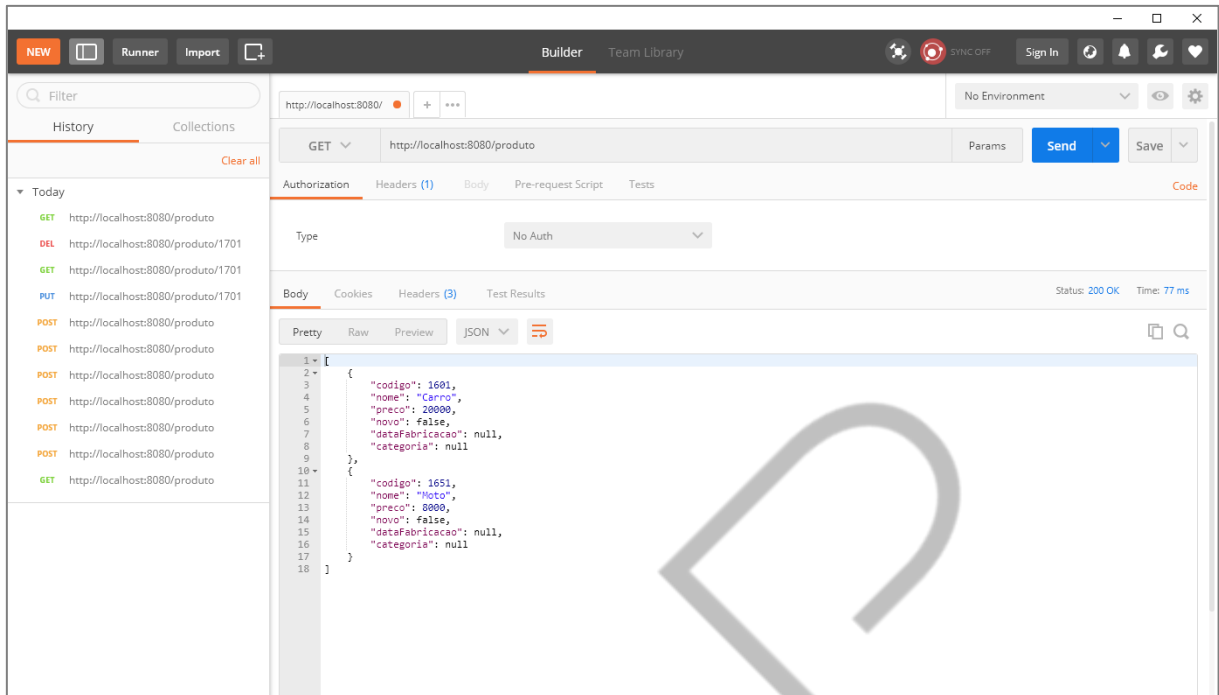


Figura 29 – Teste do DELETE no Postman (Parte 2)
Fonte: Elaborado pelo autor (2019)

Observe, na Figura “Teste do DELETE no Postman (Parte 2)” que o código 1701 não consta na listagem, ou seja, a remoção foi realizada com sucesso.

Com isso, testamos os quatro verbos principais de uma API REST em nosso projeto.

Falta um tópico para encerrarmos esta parte do *back-end* no projeto: fazer o mapeamento de relacionamentos entre tabelas do banco de dados.

3.8 Relacionamentos One-to-Many e/ou Many-to-Many

Agora vamos estruturar um relacionamento *One-to-Many* (um para muitos), imaginando que um produto pode pertencer a uma categoria e uma categoria pode conter diversos produtos. Para isso, precisamos, inicialmente, definir a entidade Categoria, conforme o “Código da Entidade Categoria”:

```
@Entity
@SequenceGenerator(name = "categoria", sequenceName =
"SQ_T_CATEGORIA", allocationSize = 1)
public class Categoria {

    @Id
```

```
@GeneratedValue(strategy = GenerationType.SEQUENCE,
generator = "categoria")
private int codigo;

private String nome;

public int getCodigo() {
    return codigo;
}

public void setCodigo(int codigo) {
    this.codigo = codigo;
}

public String getNome() {
    return nome;
}

public void setNome(String nome) {
    this.nome = nome;
}
}
```

Código-fonte 13 – Código da Entidade Categoria
Fonte: Elaborado pelo autor (2019)

Assim como o seu repositório (*repository*), em “Código do CategoriaRepository”:

```
public interface CategoriaRepository extends
JpaRepository<Categoria, Integer> {
}
```

Código-fonte 14 – Código do CategoriaRepository
Fonte: Elaborado pelo autor (2019)

Na entidade Produto, precisamos mapear e informar o relacionamento, conforme “Código da Entidade Produto com Categoria”:

```
@Entity
@SequenceGenerator(name = "produto", sequenceName =
"SQ_PRODUTO", allocationSize = 1)
public class Produto {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
generator = "produto")
    private int codigo;

    @NotBlank(message = "Nome obrigatório!")
}
```

```
@Size(max = 50)
private String nome;

@Min(value=0, message = "Preço não pode ser negativo")
private double preco;

private boolean novo;

@Past
private LocalDate dataFabricacao;

@ManyToOne
private Categoria categoria;

public Categoria getCategoria() {
    return categoria;
}

public void setCategoria(Categoria categoria) {
    this.categoria = categoria;
}

// criar getters e setters aqui.
}
```

Código-fonte 15 – Código da Entidade Produto com Categoria
Fonte: Elaborado pelo autor (2019)

Note que só foi acrescentado o código (em negrito) relacionado à variável categoria, utilizando-se da annotation **@ManyToOne** para informar sobre o relacionamento entre Produto e Categoria (de natureza muitos para um), além do getter e setter dessa categoria.

O projeto da API de exemplo em SpringBoot pode ser baixado em:

Git: <https://github.com/FIAP/example-springboot-api>

Zip: <https://github.com/FIAP/example-springboot-api/archive/refs/heads/master.zip>

CONCLUSÃO

O objetivo deste capítulo foi apresentar o módulo Spring Boot (em conjunto com o Spring MVC), assim como construir um primeiro projeto *hands-on* com a parte de persistência e externalização de uma API REST, cuidando de toda a parte de *back-end* do projeto.

Você deve ter reparado que, na Figura “Estrutura do Projeto”, existem mais arquivos do que falamos até agora. Esses arquivos serão o foco da segunda parte do projeto (*front-end*).

Veremos que o Spring Boot possui módulos que automatizam muitas etapas no processo de criação da camada de *View* do sistema.

REFERÊNCIA

SPRING. **Introduction to the Spring Framework.** Disponível em:
<<https://docs.spring.io/spring/docs/3.0.x/spring-framework-reference/html/overview.html>>. Acesso em: 13 jan. 2021.

EMANIP