BandTec
DIGITAL SCHOOL

Curso de Tecnologia em Análise e Desenvolvimento de Sistemas

Técnicas de Programação Web

Introdução ao REST

Sumário

Introdução ao REST e REST APIs

O REST (***RE**presentational **S**tate **T**ransfer*) é um estilo arquitetural que usa HTTP ou HTTPS como protocolo de comunicação. Foi proposto em 2000, na tese de doutorado de **Roy Fielding**, que, aliás, também é um dos pais do HTTP.

Quando uma API é criada adotando-se REST, dizemos que se trata de uma **RESTful API**. Porém, é muito comum vermos os termos **REST API, Web API** e **APIs para Web** para o mesmo conceito.

Até meados de 2011, a arquitetura mais usada no mundo para implementar APIs era **SOAP**, mas desde então, REST passou a ser mais adotada. Ross Mason, fundador da *MuleSoft* escreveu um artigo falando sobre o fato de REST ter ultrapassado SOAP na integração de sistemas (vide em **https://www.infoq.com/articles/rest-soap**). Ao recorrermos ao *Google Trends*, veremos que o interesse ao longo do tempo sobre REST (termo de computação) ultrapassou o de SOAP (termo de computação) exatamente em junho de 2011 e continuou crescendo, conforme a Figura 1.

Figura 1: Interesse por SOAP e REST em pesquisas no Google.

Todas as principais plataformas de desenvolvimento da atualidade (Java, .Net, Python, JavaScript/Node.js, Ruby, Go etc.) possuem frameworks e/ou bibliotecas para a implementação de RESTful APIs. E, se olharmos as grandes empresas de tecnologia como: Google, Amazon, Microsoft, IBM, Facebook, operadoras de cartão de crédito etc. todas oferecem uma coleção de RESTful APIs para o consumo de seus serviços. Isso tudo demonstra que, conforme Ross Mason antecipou em 2011, essa é a abordagem mais usada atualmente na integração de sistemas.

Uma RESTful API pode ser implementada em praticamente qualquer linguagem de programação moderna. Porém, independentemente da plataforma de desenvolvimento usada na criação trata-se de um serviço que usa o HTTP/HTTPS para enviar **requisições** (ou **solicitações**) e receber **respostas**.

Requisições HTTP

As requisições HTTP são usadas para que um cliente (ou *consumidor*) inicie uma comunicação com uma RESTful API. Uma requisição HTTP é composta basicamente de: **Método**, **URL**, **Parâmetros**, **Cabeçalhos** e **Corpo**.

Parte	Breve Descrição
Método (GET, POST, PUT, DELETE etc…)	Operação que será realizada
URL	Endereço completo do EndPoint
Parâmetros	Informações adicionais e dinâmicas à URL
Cabeçalhos (ou ***Headers***)	Informações que não vão na URL. Tudo aquilo que não é endereço mas também não é corpo. Normalmente informações de segurança e de tipo de dados que serão enviados na requisição.
Corpo (nem todos os métodos HTTP enviam)	Conteúdo da Requisição

Método: é um verbo que indica a natureza da chamada. Os métodos (ou métodos) HTTP mais usados em RESTful APIS são:

- **GET** - Usado para recuperar um recurso;
- **POST** - Usado para a criação de um recurso;
- **PUT** - Usado para a atualização de um recurso;
- **DELETE** - Usado para excluir um recurso.

Existem ainda vários outros métodos HTTP. Maiores detalhes nas páginas sobre métodos HTTP da especificação oficial do HTTP 1.1: **https://tools.ietf.org/html/rfc7231#section-4** e **https://tools.ietf.org/html/rfc5789#section-2**.

URL (*Uniform Resource Locator*): Endereço padronizado que contém o IP ou Hostname do servidor onde está a API, a porta TCP (opcional) e uma URI (*Uniform Resource Identifier*), que indica o recurso dentro da API que estamos querendo acessar. Caso a porta seja omitida, significa que a porta usada é a **80**.

Parâmetros: um ou mais dados no formato agrupados na forma de *chave x valor*. São usados para detalhar a operação indicada no **método**. Por exemplo, em uma chamada com GET, os parâmetros podem servir para indicar os campos e valores envolvidos no filtro de uma consulta.

Cabeçalhos: são muito parecidos com parâmetros, também sendo enviados agrupados em *chave x valor*. A diferença é que aqui são enviados dados que não possuem relação direta com a requisição. É muito comum serem usados para enviar as credenciais de acesso à API e o tipo de dado que está sendo enviado no **corpo**. Os cabeçalhos mais comuns são listados e

descritos no *Mozilla Developers Network* (**https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers**).

Corpo: é o conteúdo da requisição. Esse conteúdo pode ser um texto simples, um JSON, um XML ou até mesmo um conteúdo binário (PDF, ZIP etc.). Quando é enviado um corpo, o ideal é que também seja enviado um **cabeçalho** chamado *Content-type*. Alguns dos valores mais comuns para esse cabeçalho são: *application/json* (conteúdo JSON), *text/xml* (conteúdo XML) e *application/octet-stream* (conteúdo binário).

***IMPORTANTE**: nem todos os métodos HTTP permitem o envio de um corpo. Dos citados aqui, apenas POST e PUT permitem.*

Respostas HTTP

As respostas HTTP são usadas pela RESTful API para devolver o resultado de uma requisição HTTP para um cliente. Uma resposta HTTP é composta basicamente de: **Status**, **Cabeçalho**, **Corpo**.

Parte	Breve Descrição
Status	Código padronizado que informa a situação geral da resposta à requisição
Cabeçalhos (ou ***Headers***)	Informações que não vão na URL. Tudo aquilo que não é corpo. Normalmente informações de segurança, detalhes sobre a API e de tipo de dados que serão enviados na resposta.
Corpo	Conteúdo da Resposta

Status: é um código numérico que indica o resultado geral da resposta. Existem vários códigos, divididos em famílias, sendo que as mais usadas em RESTful APIs são:

- **2xx** - Códigos de status que indicam que a requisição foi processada com sucesso. Dentre os comuns estão:
  - **200** - *Ok* (indica que deu tudo certo)
  - **201** - *Created* (indica que um recurso foi criado com sucesso)
  - **204** - *No Content* (indica que uma **consulta** não encontrou resultados)
  - **202** - *Accepted* (indica que a requisição foi aceita mas que seu processamento completo será feito paralelamente)
- **4xx** - Códigos de status que indicam que a requisição foi feita com algo errado, ou seja, que o cliente não fez a requisição como deveria. Dentre os comuns estão:

    - **400** - *Bad request* (indica que alguma coisa nos parâmetros e/ou corpo da requisição está errada)
    - **401** - *Unauthorized* (indica que era necessária alguma credencial de acesso, não fornecida)
    - **403** - *Forbidden* (indica que um recurso não é permitido mesmo com as credenciais de acesso fornecidas)
    - **404** - *Not found* (indica que o recurso solicitado não existe)
    - **409** - *Conflct* (indica que o recurso já foi criado)
- **5xx** - Códigos de status que indicam que a requisição não pode ser processada por problema no serviço. Dentre os comuns estão:
    - **500** - *Internal server error* (indica que alguma ocorreu algum erro não tratado no serviço)
    - **503** - *Service unavailable* (indica que o serviço está praticamente fora, só conseguindo responder para informar que está indisponível)

Existem ainda vários outros status de resposta HTTP. Uma lista completa pode ser obtida no Mozilla Developers Network (**https://developer.mozilla.org/en-US/docs/Web/HTTP/Status**).

Cabeçalhos: Valores agrupados em chave x valor que descrevem detalhes da resposta e do corpo, caso este esteja presente.

Corpo: é o conteúdo da resposta. Esse conteúdo pode ser um texto simples, um JSON, um XML ou até mesmo um conteúdo binário (PDF, ZIP etc.). Quando há um corpo na resposta, o ideal é que também seja enviado um **cabeçalho** chamado **Content-type**. Alguns dos valores mais comuns para esse cabeçalho são: application/json (conteúdo JSON), text/xml (conteúdo XML) e image/jpeg (imagem).

Endpoints

Cada operação disponível em uma API é chamada de **Endpoint**. É para eles que enviamos as requisições HTTP e é deles que esperamos as respostas HTTP. Exemplos simples de Endpoint: uma API para a realização das operações básicas da aritmética deveria ter pelo menos 4 Endpoints: adição, subtração, multiplicação e divisão.

RESTful API da Marvel

Existe uma RESTful API pública que permite consultar dados de personagens e séries da Marvel Comics. Sua URL base é **http://gateway.marvel.com** e algumas de duas **URIs** são:

- **/v1/public/characters** (para a recuperação de todos os personagens)
- **/v1/public/characters/{characterId}** (para a recuperação um personagem)

- **/v1/public/characters/{characterId}/comics** (para a recuperação de todos os quadrinhos onde um personagem está presente)

Todos as URLs dessa API só possuem Endpoint com o método **GET**. Dessa forma, podemos dizer que os Endpoints para as URIs citadas seriam:

- **GET /v1/public/characters**
- **GET /v1/public/characters/{characterId}**
- **GET /v1/public/characters/{characterId}/comics**

Clientes de uma RESTful API

Como o único requisito para consumir uma RESTful API é uma conexão HTTP/HTTPS, clientes desse tipo de serviço podem ser uma simples página HTML, um programa desktop, um aplicativo de smartphone ou até mesmo um processo em um sistema operacional. O cliente não tem a menor ideia de qual plataforma foi usada para desenvolver a API e pode ter sido criado em qualquer linguagem capaz de implementar um cliente HTTP. Esse modelo de comunicação está representado na Figura 2.

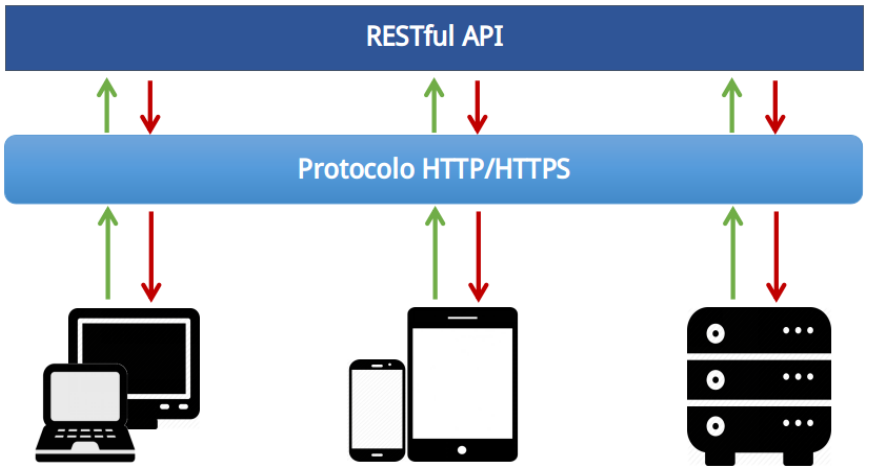

Figura 2: Modelo de comunicação REST.
Fonte: YOSHIRIRO, 2018.

Microsserviços

Segundo **Martin Fowler** e **James Lewis** afirmaram em 2014: “O termo ‘Arquitetura de Microsserviços’ surgiu nos últimos anos para descrever uma maneira específica de desenvolver software como conjuntos de serviços com deploy independente. Embora não exista uma definição precisa desse estilo de arquitetura, há certas características comuns em relação à organização, à capacidade de negócios, ao deploy automatizado, à inteligência nos terminais e ao controle descentralizado de linguagens e de dados.” (Tradução livre. Original em **https://martinfowler.com/articles/microservices.html**).

Usar **Microsserviços** significa criar quantas pequenas e especializadas aplicações forem necessárias a fim de evitar uma **Aplicação Monolítica** (aplicação grande, que faz várias coisas). A ideia é criar aplicações pequenas e que lidam somente com um domínio ou um problema específico. Com isso, ganha-se, entre outros benefícios, uma grande flexibilidade nas atualizações

e facilidade na configuração e manutenção de diferentes funcionalidades de uma solução. Quase sempre, cada aplicação dessa é uma REST API. Na Figura 3 é possível ver as diferenças entre elas.

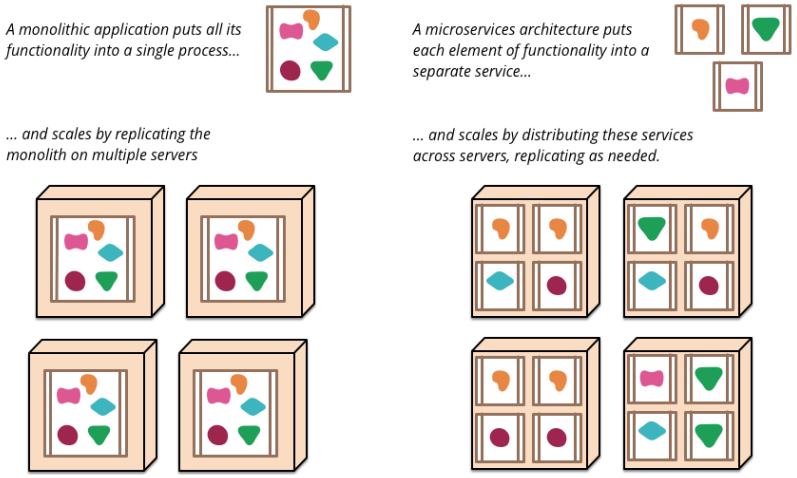

Figura 3: Microsserviços x Aplicação monolítica.
Fonte: https://martinfowler.com/articles/microservices.html

Práticas com REST

A seguir, veremos algumas pequenas práticas que podemos fazer usando APIs gratuitas no protocolo REST. Diferentes tecnologias serão usadas para demonstrar o quão simples é consumir uma REST API.

Prática de REST API: consulta de logradouro por CEP

Existe uma REST API gratuita chamada **Via CEP**. Ela recupera dados de um logradouro a partir de um CEP informado.

Para um teste rápido, acesse, de qualquer navegador, o endereço: **https://viacep.com.br/ws/04301000/json/**. Depois mude o valor de 03216050 para qualquer CEP que você conheça (só os números) e confira o resultado.

Para atestar você mesmo a simplicidade e flexibilidade do protocolo REST, iremos consumir esse serviço de CEP usando 3 tecnologias diferentes: Postman, Javascript e Groovy.

Teste com Postman

Execute o **Postman**. Cole o endereço (URL) de exemplo e clique em "**Send**". O resultado vai aparecer abaixo, em "**Body**" (vide figura 4).

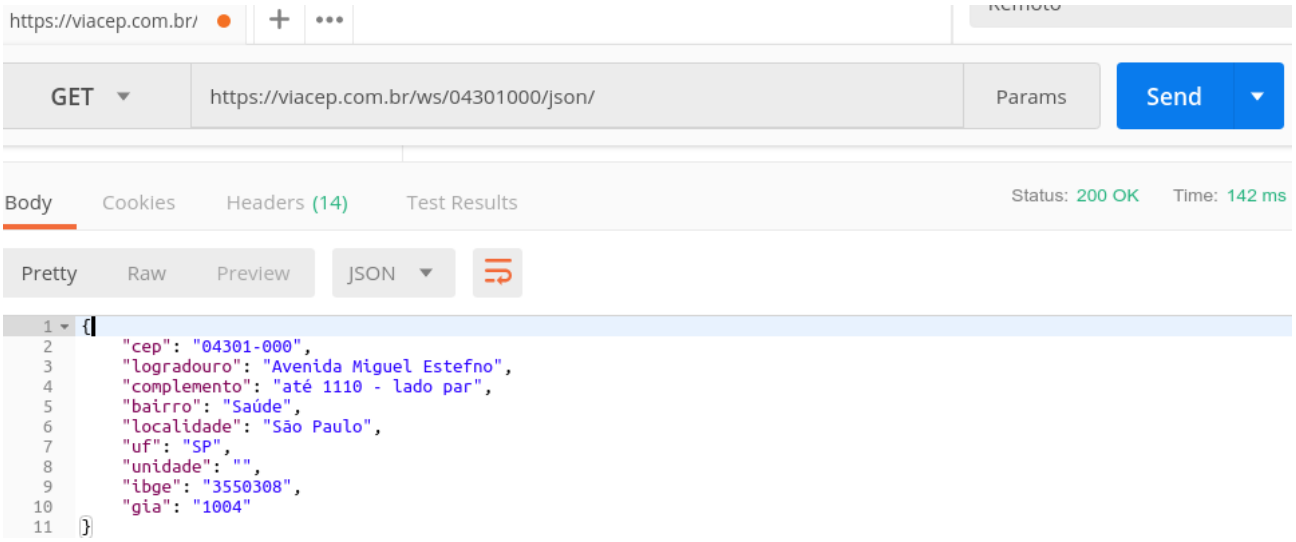

Figura 4: Resultado de chamada a EndPoint da API Via CEP pelo Postman.

Teste com Javascript

Acesse o endereço **https://jsfiddle.net/jyoshiriro/h4g80q6f/**. Use a página que fica no no quadrante inferior direito, preenchendo o CEP no campo indicado e clicando em "**Pesquisar**". O código que consome a REST API está no quadrante inferior esquerdo.

Teste com Groovy

Acesse o endereço **https://groovy-playground.appspot.com/**. Vai abrir um editor de textos online. Copie e cole o código a seguir nele.

```
def cep = '03216050'

def url = "https://viacep.com.br/ws/${cep}/json/".toURL()

try {

    def mapa = new groovy.json.JsonSlurper().parseText(url.text)


    println("Endereço: ${mapa.logradouro}")

    println("Bairro: ${mapa.bairro}")

    println("Cidade: ${mapa.localidade}")

    println("UF: ${mapa.uf}")

} catch (e) {

    println('Cep inválido ou serviço indisponível')

}
```

Depois clique em "**Execute**" na parte inferior da tela. Para testar com outro CEP, apenas coloque o CEP (só números) entre as aspas na primeira linha do código.

Gostou? Entendeu? Você conseguiria adaptar os códigos em Javascript e/ou Groovy para consumir a REST API do **Vagalume**? Se topar esse desafio, seguem alguns endereços de exemplo dela que pode testar de qualquer navegador:

http://api.vagalume.com.br/search.art?apikey=660a4395f992ff67786584e238f501aa&q=adele

http://api.vagalume.com.br/search.excerpt?apikey=660a4395f992ff67786584e238f501aa&q=someone

http://api.vagalume.com.br/search.php?apikey=660a4395f992ff67786584e238f501aa&art=Ladygaga&mus=shallow

Bibliografia

GUTIERREZ, Felipe. Pro Spring Boot. New York (EUA): Apress, 2016.

REMANI, Abdelmonaim. The Art of Metaprogramming in Java. Disponível em: https://cdn.oreillystatic.com/en/assets/1/event/80/The%20Art%20of%20Metaprogramming%20in%20Java%20%20Presentation.pdf. Acesso em: 20 de novembro de 2018.

YOSHIRIRO, José. Spock framework - Testes automatizados para Java, Android e REST. São Paulo: Casa do Código, 2019.