EMÍLIO DIAS

DESMISTIFICANDO REST COM JAVA

algaworks

Desmistificando REST com Java
por Emílio Dias

1ª Edição, 11/02/2016

© 2016 AlgaWorks Softwares, Treinamentos e Serviços Ltda. Todos os direitos reservados.

Nenhuma parte deste livreto pode ser reproduzida ou transmitida em qualquer forma, seja por meio eletrônico ou mecânico, sem permissão por escrito da AlgaWorks, exceto para resumos breves em revisões e análises.

AlgaWorks Softwares, Treinamentos e Serviços Ltda
www.algaworks.com
contato@algaworks.com
+55 (11) 2626-9415

Siga-nos nas redes sociais e fique por dentro de tudo!

WORKSHOP ONLINE

Web Services RESTful e Spring

COMPRE AQUI ↓

Sobre o autor

Emílio Dias

Instrutor da AlgaWorks, formado em Ciência da Computação, Especialista em Segurança da Informação e detentor das certificações SCJP e LPIC-1. Palestrante de fóruns internacionais de Software Livre e congressos de Engenharia de Software. Atua também como Professor nos cursos de Ciência da Computação e Sistemas de Informação da Unitri em Uberlândia.

LinkedIn: https://www.linkedin.com/in/emiliodias

Antes de começar...

Antes que você comece a ler esse livro, eu gostaria de combinar algumas coisas com você, para que tenha um excelente aproveitamento do conteúdo. Vamos lá?

Como obter ajuda?

Durante os estudos, é muito comum surgir várias dúvidas. Eu gostaria muito de te ajudar pessoalmente nesses problemas, mas infelizmente não consigo fazer isso com todos os leitores do livreto, afinal, ocupo grande parte do dia ajudando os alunos de cursos online na AlgaWorks.

Então, quando você tiver alguma dúvida e não conseguir encontrar a solução no Google ou com seu próprio conhecimento, minha recomendação é que você poste na nossa Comunidade Java no Facebook. É só acessar:

http://alga.works/comunidadejava/

Como sugerir melhorias ou reportar erros sobre este livreto?

Se você encontrar algum erro no conteúdo desse livreto ou se tiver alguma sugestão para melhorar a próxima edição, vou ficar muito feliz se você puder me dizer.

Envie um e-mail para livros@algaworks.com.

Ajude na continuidade desse trabalho

Escrever um livro (mesmo que pequeno, como esse) dá muito trabalho, por isso, esse projeto só faz sentido se muitas pessoas tiverem acesso a ele.

Ajude a divulgar esse livreto para seus amigos que também se interessam por programação Java. Compartilhe no Facebook e Twitter!

Sumário

6 Conclusão

Capítulo 1

Introdução

Você certamente já deve ter ouvido falar em Web Services RESTful, não é mesmo?

Muito tem se falado a respeito sobre formas de integração de sistemas comerciais, aplicativos móveis e tantos outros. Mas qual a melhor forma de realizar esta integração?

Nesse livreto, vamos apresentar vários conceitos e tirar uma série de dúvidas sobre como podemos implementar um Web Service RESTful e como o Java pode te ajudar nessa caminhada.

Você já se perguntou por que precisamos de Web Services? Essa é uma questão que podemos avaliar sob vários pontos de vista, mas eu quero te apresentar dois.

Primeiramente, muitas empresas adquirem software de vários fornecedores e precisam de alguma forma fazer todos eles se comunicarem. Neste sentido, os Web Services ajudam como uma ponte de ligação entre os mesmos. O segundo aspecto é até um pouco mais interessante, e pra falar sobre ele, vou te contar uma pequena história.

A forma na qual interagimos com meios computacionais vem evoluindo consideravelmente com o passar do tempo. Nos últimos anos, temos acompanhado o surgimento de Smart TVs, Smartphones, Tablets, Internet das Coisas, dentre tantos outros. O aparecimento de tais dispositivos e tecnologias, impactam diretamente o nosso dia a dia.

O mercado de tecnologia passa constantemente por desafios, para de alguma forma, atender as necessidades de todos os consumidores, e nem sempre é simples modelar uma solução adequada para tantos cenários complexos. Atualmente é difícil imaginarmos um mercado sem vendas pela Web e também a execução de atividades corriqueiras a partir de um smartphone.

Consumidores modernos exigem cada vez mais uma experiência de uso avançada, o que permite, por exemplo, a compra de um produto em um e-commerce e a possível troca do mesmo em uma loja física, de forma totalmente transparente para vendedores e consumidores. Esse novo tipo de experiência é o que o mercado vem dando o nome de multicanalidade e omnichannel.

Com a convergência de tantos meios tecnológicos, nós desenvolvedores nos vemos à frente da necessidade de encontrar soluções para integrar todas essas ferramentas. Além disso, APIs (Application Programming Interface) que eram antes "artigos de luxo" discutidos e abordados apenas por nós programadores, passaram a ter muito valor para altos executivos no mundo dos negócios.

Grandes empresas como Twillo, Twitter, Google e Facebook dependem fortemente de um modelo computacional aberto que permita que seus produtos interajam com aplicações de toda a Web. Ou seja, eles precisam de alguma forma (via Web Services), disponibilizar meios para que outros sistemas se integrem com seus serviços.

Dizendo isso, fica mais fácil entender que a escolha de uso por uma tecnologia de Web Services não está associada apenas a um fator técnico, e sim também ao fato que precisamos desenvolver sistemas de forma que o usuário tenha uma experiência de uso cada vez melhor.

Além disso, a quantidade de usuários que utilizam a Web e algum meio computacional aumenta a cada dia, projetos de inclusão são frequentemente noticiados e precisamos de alguma forma desenvolver sistemas que sejam capazes de suportar tudo isso.

Arquiteturas monolíticas e camadas acopladas podem não ser adequadas para a modelagem de um sistema que precisa cada vez mais interagir com tantos outros. Por que então não construirmos APIs baseadas na Web? Ela possui várias

características que nos permitem criar APIs que suportam tudo aquilo que eu disse anteriormente. Apenas para citar algumas, vejamos:

- Simples
- Extensível
- Escalável
- Incremental
- Global
- E muito mais…

Perceba que possuímos uma grande plataforma em mãos. Exato, podemos passar a enxergar a Web não apenas como um lugar onde um conjunto enorme de informações são encontradas ou páginas Web são disponibilizadas, podemos começar a tratá-la como uma plataforma e usar dos seus benefícios para criar aplicações cada vez melhores.

Baseado nisso, nesse livreto nós vamos discutir como o modelo arquitetural REST e o Java podem nos ajudar na implementação de soluções que atendam essa quantidade de requisitos. Vamos falar também sobre a origem do REST, apresentar e tirar algumas dúvidas e confusões que sempre aparecem quando esse assunto é discutido.

Capítulo 2

REST

Apesar de aparentemente ser uma proposta nova, REST surgiu no início dos anos 2000, a partir da tese de Ph.D de um cientista chamado Roy Fielding[1].

O intuito geral era a formalização de um conjunto de melhores práticas denominadas *constraints*. Essas *constraints* tinham como objetivo determinar a forma na qual padrões como HTTP e URI deveriam ser modelados, aproveitando de fato todos os recursos oferecidos pelos mesmos.

Vamos conhecer um pouco melhor essas *contraints*?

2.1. Cliente-servidor

A principal característica dessa *constraint* é separar as responsabilidades de diferentes partes de um sistema. Essa divisão pode se dar de diversas formas, iniciando por exemplo com uma separação entre mecanismos de armazenamento de dados e o *back-end* da aplicação.

Outra divisão muito comum se dá entre a interface do usuário e *back-end*. Isso nos permite a evolução e escalabilidade destas responsabilidades de forma independente. Atualmente várias ferramentas seguem este modelo, frameworks como AngularJS e APIs RESTful permitem o isolamento de forma elegante para cada uma dessas funcionalidades.

---

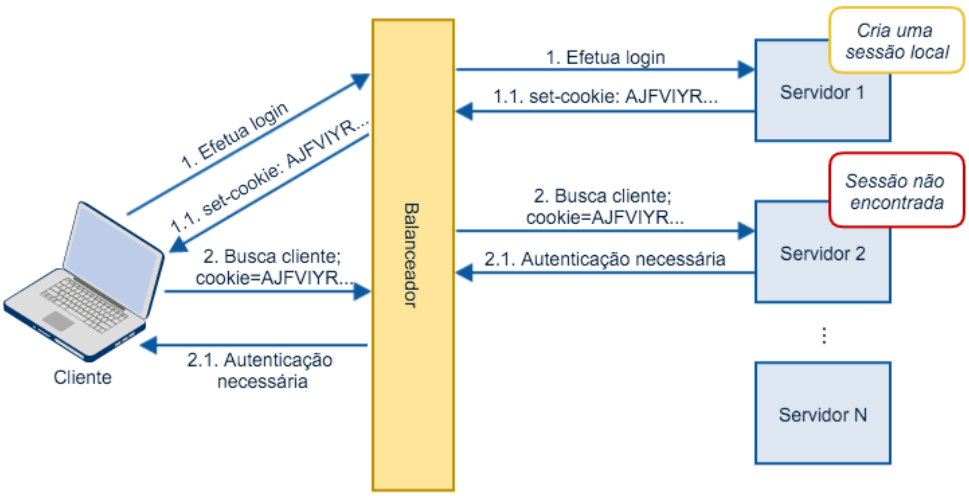
1. http://www.ics.uci.edu/~fielding/

2.2. Stateless

Essa característica propõe que cada requisição ao servidor não deve ter ligação com requisições anteriores ou futuras, ou seja, cada requisição deve conter todas as informações necessárias para que ela seja tratada com sucesso pelo servidor.

O protocolo HTTP segue esse modelo, porém, é muito comum o uso de *cookies* para armazenamento de sessões do lado do servidor. Essa abordagem deve ser usada com cautela, visto os inconvenientes que a mesma carrega.

Um dos principais inconvenientes no uso de *cookies* é que sua utilização diminui de forma drástica a forma na qual podemos escalar nossas aplicações. A figura abaixo ilustra um pouco melhor essa situação.



Perceba que o cliente precisa sempre ser redirecionado para o mesmo servidor, limitando assim questões de alta disponibilidade, escalabilidade e etc. Existem várias maneiras de tratar essa questão, por exemplo, podemos usar o modelo *Sticky Session*[2] ou em um cenário melhor, devemos sempre que possível, transferir a responsabilidade de armazenamento de informações para os clientes.

2. http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html

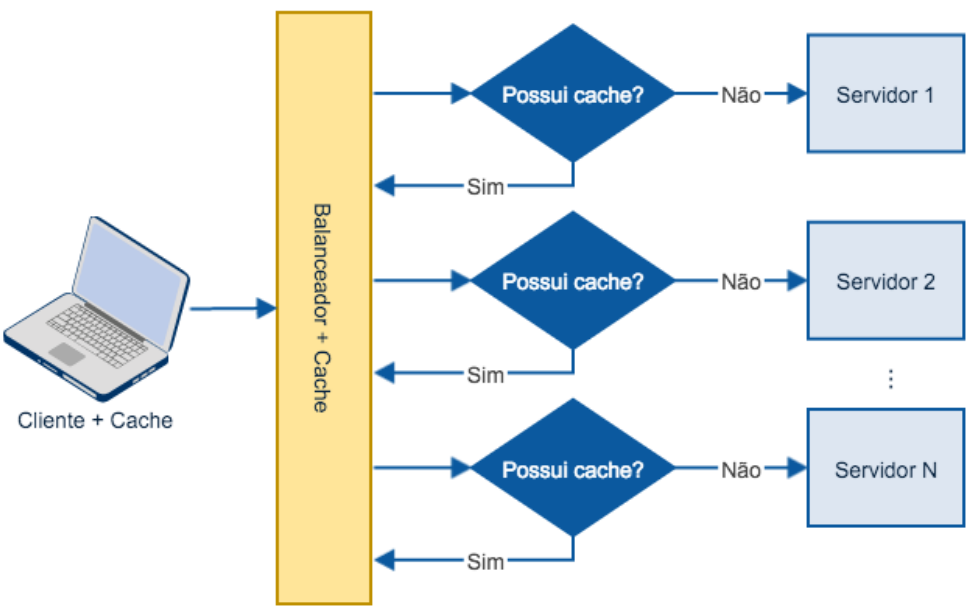
Segundo Fielding, utilizando um modelo de comunicação *stateless*, sua API passa a ter características como visibilidade, confiabilidade e escalabilidade.

Tenha em mente que, sempre que possível, o ideal é o desenvolvimento de sistemas que não dependam da manutenção de estado.

2.3. Cache

Para uma melhor performance, um sistema REST deve permitir que suas respostas sejam passíveis de *cache*. Cada resposta deve de alguma maneira informar para clientes ou elementos intermediários (*proxies*, *gateways* e/ou balanceadores de carga) qual a política de *cache* mais adequada.

O protocolo HTTP permite o funcionamento adequado dessa *constraint* a partir da adição dos *headers* "expires" (versão 1.0 do HTTP) e/ou "cache-control" (versão 1.1 do HTTP).



Veja que em situações onde o cliente local ou o balanceador possui *cache*, não temos a necessidade de requisitar ao servidor o processamento de uma nova

requisição. Esse modelo permite que o servidor execute apenas tarefas que realmente são necessárias.

2.4. Interface uniforme

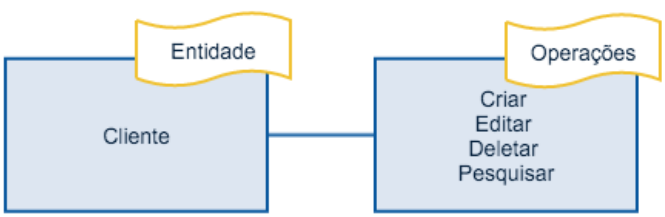
Como temos discutido até agora, podemos ver que REST possui uma série de características, e certamente a interface uniforme é uma das mais importantes.

Bastante esforço deve ser feito para que o sistema possua uma interface modelada seguindo alguns padrões importantes. Quando se fala sobre uma interface, os elementos abaixo devem ser considerados:

- Recursos
- Mensagens autodescritivas
- *Hypermedia*

Quando implementada utilizando HTTP, uma API deve levar em consideração também a correta utilização dos métodos e códigos de retorno. Nós vamos discutir isso melhor em sessões futuras.

Para ficar um pouco mais claro o que de fato vem a ser essa interface, podemos pensar por exemplo como podemos manipular vendas em um e-commerce. Uma loja virtual possui diversas entidades, como por exemplo produto, cliente, pedido e etc. Devemos pensar em criar uma interface que permita a manipulação desses conceitos. Abaixo um exemplo:



Veja que a figura modela o recurso "cliente" e uma série de operações que podem ser executadas sob o mesmo. Futuramente vamos discutir como implementar essa interface e você irá perceber que cada uma das operações reflete sob um método do protocolo HTTP.

2.5. Sistema em camadas

Com o intuito de permitir a escalabilidade necessária para grandes sistemas distribuídos, um sistema REST deve ter a capacidade de adicionar elementos intermediários e que sejam totalmente transparentes para seus clientes.

Tais elementos intermediários são utilizados de forma transparente para o cliente. Isso é algo de bastante valor, imagine se você tivesse que acessar o site do Google sem um mecanismo de DNS, ou se em cada acesso você precisasse informar se deseja ou não fazer um *cache*. Talvez dessa maneira a Web não seria o sucesso que é hoje.

Perceba que nos tópicos que discutimos sobre *stateless* e *cache*, foi utilizado um elemento que demos o nome de balanceador. Esse balanceador é um dos elementos que podemos usar e que permite adicionarmos mais servidores para uma aplicação, sem que o cliente note sua presença.

2.6. Código sob demanda

Você certamente já deve ter estudado uma série de artigos e livros sobre boas práticas de desenvolvimento de software e orientação a objetos. A maioria desses artigos e livros falam sobre a questão de extensibilidade, que é a capacidade que um software possui de evoluir sem a necessidade de quebra do mesmo.

Código sob demanda permite algo semelhante, ele possibilita adaptar o cliente de acordo com novos requisitos e funcionalidades. Exemplos disso são o JavaScript e Applets.

Capítulo 3

É REST ou RESTful?

Você já deve ter se perguntado sobre a diferença dos termos REST e RESTful, não é? Não se preocupe, uma confusão muito comum é o uso intercambiável desses termos. Mas de fato, uma API ou aplicação deve receber o nome REST ou RESTful?

De forma bem direta, o correto é você dizer API RESTful. Além disso, é importante você entender o porquê dessa nomenclatura e qual o momento certo de usar cada uma.

Quando estamos discutindo sobre o modelo e sobre as características que nós vimos anteriormente, você deve utilizar o termo REST, já no momento em que você estiver falando de uma implementação que usa essas mesmas características, você deve usar RESTful.

Apesar de não ser algo tão relevante, achei interessante lhe dizer isso para que você sempre utilize as nomenclaturas corretas. Isso também nos ajuda a deixar claro que REST nada mais é que um conjunto de boas práticas.

3.1. Sendo RESTful com HTTP

Até agora, nós discutimos as características de uma forma bastante conceitual e abstrata. Apesar dessa discussão nos permitir uma visualização adequada dos conceitos, precisamos entender isso de uma forma mais prática.

Vamos discutir agora a forma mais utilizada de implementação de REST: o protocolo HTTP. Vamos falar também sobre os principais conceitos envolvidos na criação de uma API RESTful. O objetivo não é listar todas as funcionalidades possíveis, mas iremos navegar pelas características principais que você deve conhecer para implementar um Web Service RESTful.

3.2. Recursos

Um recurso é utilizado para identificar de forma única um objeto abstrato ou físico. A RFC 3986 especifica detalhadamente todas as características que uma URI deve seguir.

Basicamente, temos a modelagem de um recurso sob um substantivo, ao invés de verbos, como é usualmente utilizado em APIs. Exemplos:

```
/cliente/1
/produto/1
/cliente/1/notificacao
```

Uma URI deve ser visualizada como um recurso, e não como uma ação a ser executada sob um servidor. Alguns autores e literaturas defendem a utilização de verbos em casos específicos, porém, isso não é totalmente aceitável e você deve usar com cautela.

Para que fique mais claro pra você, vamos dar uma olhada em como isso pode ser construído usando o Spring. Não vou entrar em muitos detalhes do framework, minha ideia aqui é lhe mostrar o quão simples pode ser uma implementação inicial.

```
@RestController
@RequestMapping("/cliente")
public class ClienteResource {

  @RequestMapping(value = "/{id}", method = RequestMethod.GET,
      produces = "application/json")
  public ClienteRepresentation buscar(@PathVariable("id") Integer id) {
    System.out.println("Retornando cliente...");
    return new ClienteRepresentation("João da Silva");
```

```
  }

}
```

Apesar de simples, esse código nos permite visualizar algumas características para a criação de um recurso. O principal que devemos analisar é:

A anotação `@RestController` informa que essa classe deve disponibilizar um controlador com características de um Web Service REST.

A anotação `@RequestMapping` na classe `ClienteResource` informa ao Spring qual será o nome do nosso recurso. É nesse momento que devemos conseguir modelar quais os recursos necessários em um sistema.

Já na anotação `@RequestMapping` do método `buscar`, podemos ver algumas coisas interessantes. A primeira é o mapeamento do parametro `{id}`. Se juntarmos o valor do atribuído `value` ao recurso mapado na classe, teremos como possibilidade o acesso ao mesmo a partir da URI "/cliente/{id}". É interessante notar também a presença do atributo `produces`, que informa que a representação retornada ao cliente deve ser em formato JSON.

Executando esse Web Service, podemos ter um resultado semelhante a:

```
{
  "nome": "Joao da Silva"
}
```

Perceba que é uma simples informação em formato JSON (falaremos sobre JSON a seguir), mas que já podemos começar a evoluir.

Você deve ter reparado que usei a palavra "representação" para me referir a mensagem de retorno do Web Service. É importante você sempre usar essa nomenclatura para que fique claro o que você deseja informar.

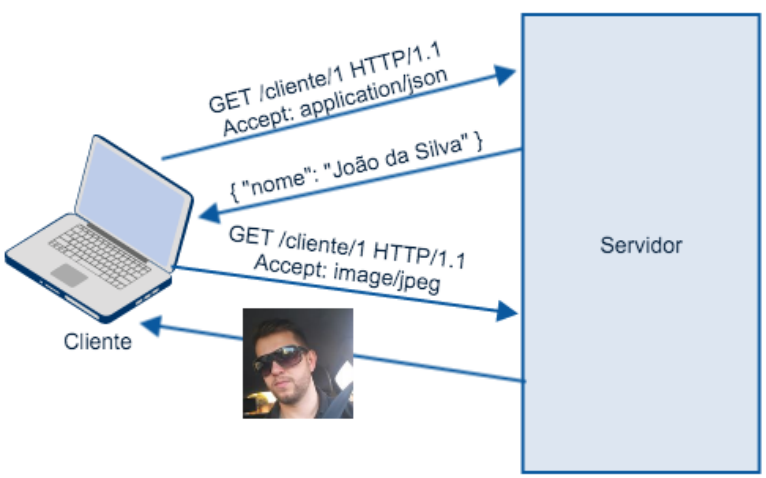
Quando um recurso é solicitado por um cliente (ex: browser), o servidor executa uma série de atividades e retorna uma mensagem ao solicitante. Essa mensagem é de fato uma representação de um determinado recurso. A mesma não necessariamente precisa estar ligada a alguma parte concreta de um sistema, como por exemplo, uma tabela de banco de dados.

Representações

As representações podem ser modeladas em vários formatos, como XML, JSON, HTML e etc. Você deve apenas levar em consideração que esse formato deve suportar a utilização de *hypermedia* (posteriormente nós vamos discutir os seus conceitos).

Para ficar um pouco mais claro, imagine por exemplo o recurso "/cliente/1", que discutimos anteriormente. Naquele caso, nós usamos uma representação em formato JSON, mas nós poderíamos também querer representá-lo a partir de uma imagem.

Isso pode ser feito devido a capacidade de múltiplas representações (mais conhecido como negociação de conteúdo) que o HTTP possui. Para que isso seja possível, você pode usar o *header* "Accept" para especificar qual é o tipo de representação mais adequada. A figura abaixo deixa isso um pouco mais claro.



Note que para cada tipo de formato requisitado, o servidor retornou uma resposta com uma representação adequada.

Agora vamos fazer uma rápida revisão sobre os formatos JSON e XML, que são os mais utilizados em APIs na atualidade, para caso você ainda tenha alguma dúvida sobre eles.

XML

O XML (eXtended Markup Language) é uma linguagem de marcação recomendada pelo W3C (World Web Consortium) e tem como objetivo a descrição de qualquer tipo de dados.

Ele é utilizado como uma forma padrão para troca de informações entre sistemas ou até mesmo para armazenamento dos mesmos. Abaixo, podemos ver a estrutura de um documento XML:

```
<cliente>
  <id>10</id>
  <nome>Alan Turing</nome>
  <nascimento>23/06/1912</nascimento>
  <profissao>Matemático</profissao>
  <endereco>
      <cidade>Manchester</cidade>
      <pais>Inglaterra</pais>
  </endereco>
</cliente>
```

Note que a marcação XML é baseada em um formato hierárquico, o que permite que possamos trabalhar de uma forma muito semelhante a que trabalhamos com objetos em Java.

Um dos inconvenientes encontrados no XML é sua verbosidade. Perceba que para representação da informação, precisamos adicionar várias tags, o que as vezes pode gerar um *overhead* desnecessário.

Baseado nisso, podemos utilizar uma solução mais enxuta, que iremos discutir a seguir.

JSON

O JSON (JavaScript Object Notation) surgiu como uma alternativa para representação de dados de uma forma mais simples e leve.

Como você deve ter percebido, JSON está relacionado à linguagem JavaScript, porém, esse formato de dados pode ser utilizado independente da tecnologia que você estiver usando. Veja abaixo um exemplo:

```
{
    "id": 10,
    "nome": "Alan Turing",
    "nascimento": "23/06/1912",
    "profissao": "Matemático",
    "endereco": {
        "cidade": "Manchester",
        "pais": "Inglaterra"
    }
}
```

Perceba que os dados descritos nesse JSON são os mesmos que nós discutimos anteriormente quando falávamos sobre o XML. Note que a representação é muito mais simples.

Dentre algumas características que o JSON possui, vale apena citar:

- Leitura simplificada
- Analisador mais simples
- Suporte de vários frameworks atuais (principalmente frameworks JavaScript)
- Tamanho reduzido

Por esses e mais vários outros fatores, o JSON está cada dia mais disseminado na comunidade.

De forma resumida, perceba que tanto XML quanto JSON, são formatos que utilizamos para descrever os dados de nossas aplicações, e que para cada cenário devemos verificar qual se encaixa de forma mais adequada. Principalmente pela sua simplicidade, o JSON vem ganhando mais seguidores.

3.3. Métodos

A RFC 7231[3] especifica um conjunto de 8 métodos os quais podemos utilizar para a criação de uma API RESTful. Esse conjunto de métodos possui a semântica de operações possíveis de serem efetuadas sob um determinado recurso.

Dentre esses 8 métodos, abaixo segue um detalhamento dos 4 mais conhecidos.

GET

O método *GET* é utilizado quando existe a necessidade de se obter um recurso. Ele é considerado idempotente, ou seja, independente da quantidade de vezes que é executado sob um recurso, o resultado sempre será o mesmo. Exemplo:

```
GET /cliente/1 HTTP/1.1
```

Essa chamada irá retornar uma representação para o recurso "/cliente/1".

POST

Utilizado para a criação de um recurso a partir do uso de uma representação. Exemplo:

```
POST /cliente HTTP/1.1
<Cliente>
  <Nome>João da Silva</Nome>
  ...
</Cliente>
```

PUT

O método *PUT* é utilizado como forma de atualizar um determinado recurso. Em alguns cenários muitos específicos, ele também pode ser utilizado como forma de criação, por exemplo quando o cliente tem a liberdade de decidir a URI a ser utilizada.

---

3. https://www.ietf.org/rfc/rfc3986.txt

DELETE

O delete tem como finalidade a remoção de um determinado recurso. Exemplo:

```
DELETE /cliente/1 HTTP/1.1
```

Apesar de aparentemente os métodos disponibilizarem uma interface CRUD (*Create, Read, Update* e *Delete*) para manipulação de recursos, alguns autores não concordam muito com esse ponto de vista e defendem a ideia que esses métodos podem possuir também uma outra representação semântica um pouco mais abstrata.

Para tornar nossa vida um pouco mais simples, o Spring nos ajuda a desenvolver essas operações de forma bem tranquila. Vamos dar uma olhada em como isso pode ser feito.

```
@RequestMapping(value = "/{id}", method = RequestMethod.DELETE,
    produces = "application/json")
public void excluir(@PathVariable("id") Integer id) {
  System.out.println("Excluindo o cliente...");
}
```

Se compararmos o código acima com o exemplo demonstrado na sessão sobre recursos, vamos perceber que pouca coisa precisou ser alterada. Basicamente informamos ao Spring, por intermédio do atributo `method`, qual operação HTTP queremos permitir para executar a exclusão de um determinado recurso.

Desta forma, toda requisição *DELETE HTTP* que chegar ao servidor será encaminhada ao método `excluir`.

Além dos métodos, ao se processar uma determinada requisição, o servidor deve retornar uma resposta adequada para cada situação. Veja abaixo um resumo dos tipos de retornos suportados pelo HTTP.

- 1xx - Informações
- 2xx - Sucesso
- 3xx - Redirecionamento
- 4xx - Erro no cliente
- 5xx - Erro no servidor

Para cada um dos tipos acima, existem códigos específicos que podem ser aplicados em cada uma das situações encontradas na manipulação de recursos.

O Spring nos fornece uma série de medidas para trabalharmos de forma adequada o código de retorno. Uma dessas formas é a criação de uma `RuntimeException`, informando que tipo de código ela representa através de uma anotação. Veja um exemplo:

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException {
}
```

Perceba a presença da anotação `ResponseStatus`. Ela nos permite informar qual o tipo de código deve ser enviado caso essa exceção seja lançada. Nesse caso, um código 404 (*Not Found*) será retornado, informando que o recurso solicitado não foi encontrado.

Você deve estar perguntando a forma na qual devemos lançar essa exceção. Então veja um código de exemplo abaixo:

```
@RequestMapping(value = "/{id}", method = RequestMethod.GET,
    produces = "application/json")
public ClienteRepresentation buscar(HttpServletResponse response,
    @PathVariable("id") Integer id) {
  System.out.println("Cliente não encontrado...");
  throw new ResourceNotFoundException();
}
```

Perceba que a utilização da exceção funciona da mesma forma que você já deve estar habituado a trabalhar.

Apesar de termos utilizado códigos bem simples, meu intuito aqui é lhe demonstrar que é necessária pouca codificação para o desenvolvimento de um Web Service RESTful. Obviamente, existem inúmeras outras possibilidades, mas você já deve estar começando a visualizar o caminho a ser seguido.

Além dos itens que discutimos até aqui, o HTTP possui suporte a diversos outros recursos que podem ser muito úteis na modelagem de uma API RESTful. Dentre eles, pode-se citar *web linking*, negociação de conteúdo, *queries*, *caching*,

requisições condicionais e segurança. Com o Spring podemos trabalhar com todas essas características.

Apesar de estarmos focados na implementação do Spring, outros inúmeros frameworks podem ser utilizados. Se você já trabalha com Java EE e gostaria de seguir essa linha, você pode também criar seu Web Service usando a especificação JAX-RS. Ela possui basicamente as mesmas funcionalidades que encontramos no Spring, deixando a desejar apenas em pouquíssimos aspectos.

Capítulo 4

REST versus SOAP

Agora que você já deve estar um pouco familiarizado com REST, eu quero te mostrar algumas discussões que estão sempre presentes no dia a dia de desenvolvedores.

Talvez você já tenha desenvolvido ou pelo menos tenha ouvido falar em Web Services SOAP, e sempre há uma discussão sobre qual é a melhor abordagem (REST ou SOAP).

De um lado os eternos defensores do SOAP (*Simple Object Access Protocol*) e do outro os amantes do REST. Diante disso, como escolher entre esses dois "protocolos"?

A verdade é que essa comparação, muitas vezes, é feita de forma bastante equivocada, visto que REST, como discutido anteriormente, é um modelo arquitetural e apenas SOAP, de fato, é um protocolo.

Por que então existe essa confusão? Muitos sistemas, na verdade, são modelados em um formato conhecido como POX (*Plain Old XML*), que basicamente consiste na passagem de uma mensagem XML utilizando como transporte o protocolo HTTP.

Esse modelo é baseado em uma arquitetura totalmente RPC (*Remote Procedure Call*) e muito pouco tem de ligação com REST, retirando talvez o fato das duas abordagens serem cliente-servidor.

A confusão é ainda maior quando o formato da mensagem utiliza JSON. Nesse caso, temos um mesmo modelo arquitetural (RPC), usando apenas um formato de mensagem diferente.

Talvez agora você esteja com dúvidas sobre o que é esse tal de RPC.

O RPC é um modelo que permite que façamos a disponibilização de métodos para execução de forma remota.

Imagine por exemplo, uma classe Java e seus diversos métodos, o RPC é um modelo que visa a disponibilização desses métodos para execução de forma remota (a partir de outro computador)[4].

O SOAP é baseado nesse formato, ou seja, ele expõe uma série de métodos Java (ou outra linguagem), para que eles possam ser executados sob mensagens em formato XML.

Diante disso, é necessário ficar claro que a escolha entre REST e SOAP vai muito além do formato de mensagens que são trocadas entre sistemas. Essas abordagens devem ser elevadas a um patamar de modelo arquitetural. É necessário a utilização de um modelo com as características providas por REST? O modelo RPC, implementado pelo SOAP e enriquecido com uma série de outras especificações conhecidas como WS-*, é o mais adequado?

O que deve ser levado em consideração são as características do sistema a ser construído. A verdade é que o modelo REST se encaixa adequadamente em muitos cenários e deve ser sempre levado em consideração.

Lembre-se apenas que não existe um modelo *"One size fits all"*, ou seja, REST ou SOAP não é a bala de prata para a resolução de todos os problemas.

Muito dessa confusão se dá pelo fato de APIs ditas como RESTful receberem esse "título" sem nenhuma credibilidade, ou seja, elas tem muito poucas características necessárias para serem realmente RESTful.

---

4. Sistemas Distribuídos, princípios e paradigmas. Andrew S. Tanenbaum

Com o intuito de desmistificar um pouco dessa confusão, vamos discutir sobre o "Modelo de Maturidade Richardson"[5] no próximo capítulo, que tem como propósito mapear em que nível uma API se apresenta e se de fato ela realmente pode ser considerada RESTful.

5. http://martinfowler.com/articles/richardsonMaturityModel.html

Capítulo 5

Modelo de maturidade Richardson

Apesar de Roy Fielding deixar bastante claro que para uma API ser considerada RESTful ela precisa obrigatoriamente seguir todas as *constraints* definidas em seu trabalho[6], e o mais importante, fazer uso de *hypermedia*, na prática, muitas vezes precisamos de uma abordagem um pouco mais simples.

Entusiastas conhecidos como RESTfarians, consideram inapropriado a nomeação de uma API RESTful se ela realmente não segue todas as *constraints* definadas por Fielding.

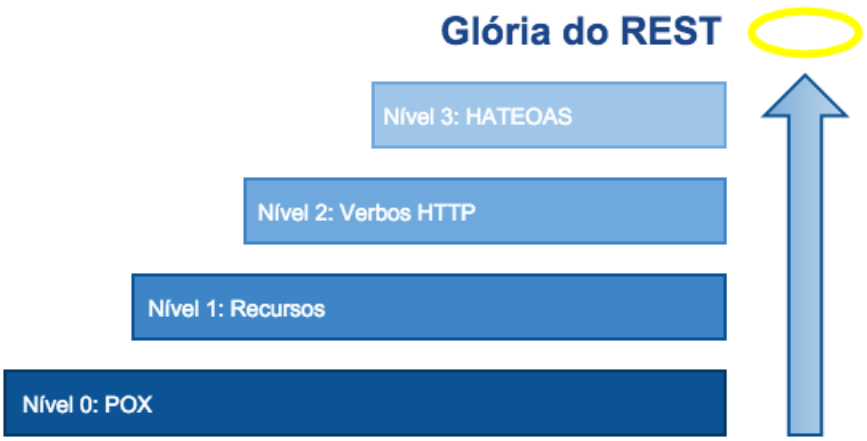
Apesar de tudo isso, o mercado tem uma outra realidade, e com o intuito de mapear e clarear os pensamentos, um modelo de maturidade foi criado.

O modelo proposto por Leonard Richardson[7] possui basicamente 4 níveis e tenta mapear as características de APIs que podem ser encontradas espalhadas hoje em sistemas de todo o mundo.

Os níveis 0, 1 e 2 talvez sejam mais familiares pra você, e de fato são mais fáceis de implementar, porém, deve ficar claro pra você que os mesmos não são considerados RESTful. Na figura abaixo podemos ver quais são esses níveis:

---

6. http://roy.gbiv.com/untangled/2008/restapismustbehypertextdriven
7. http://www.crummy.com/self/



Modelo de maturidade Richardson

Fonte: http://martinfowler.com/articles/richardsonMaturityModel.html

5.1. Nível 0 - POX

Você certamente já deve ter desenvolvido ou visto em algum lugar uma API que segue esse modelo de design. Apesar de ser o nível mais distante do que de fato REST propõe, muitas APIs ditas como RESTful se encontram neste nível de maturidade. Neste cenário, o protocolo HTTP é utilizado apenas como mecanismo de transporte e o que se vê é um modelo arquitetural fortemente baseado em RPC.

Neste nível, as mensagens podem ser serializadas em formatos como XML, JSON ou outros. É importante lembrar, como dito anteriormente, que não é o formato da mensagem que define ou não um sistema REST. Veja abaixo um exemplo de API em nível 0:

```
POST /salvarCliente HTTP/1.1
<Cliente>
  <Nome>João da Silva</Nome>
  ...
</Cliente>
```

Apesar da utilização aparentemente correta do verbo *POST HTTP* na criação de um recurso, a URI não está modelada da forma correta. Como já apresentado,

URIs devem ser modeladas com o uso de substantivos. Veja abaixo uma tabela com URIs mapeadas no formato RPC e no formato REST.

RPC (POX)		
Verbo HTTP	**URI**	**Ação**
GET	/buscarCliente/1	Visualizar
POST	/salvarCliente	Criar
POST	/alterarCliente/1	Alterar
GET/POST	/deletarCliente/1	Remover

REST		
Verbo HTTP	**URI**	**Ação**
GET	/cliente/1	Visualizar
POST	/cliente	Criar
PUT	/cliente/1	Alterar
DELETE	/cliente/1	Remover

Nessas tabelas conseguimos visualizar a diferença na modelagem dos recursos e como os verbos HTTP devem ser utilizados de forma correta.

Um outro problema constantemente encontrado, é a manipulação incorreta dos códigos de resposta do HTTP. Códigos e mensagens de erros são frequentemente manipuladas nas mensagens geradas pela aplicação, o que impede que elementos de *gateway* e *proxy* trabalhem de forma adequada. Veja um exemplo:

```
GET /buscarCliente/1 HTTP/1.1

HTTP/1.1 200 OK
<buscarCliente>
```

```
  <status>CLIENTE NÃO ENCONTRADO</status>
  <codigo>404</codigo>
<buscarCliente>
```

Apesar da mensagem sugerir que o cliente solicitado não foi encontrado, a resposta HTTP apresenta uma informação totalmente diferente (200 OK), ou seja, existe uma diferença semântica entre o procolo HTTP e a representação gerada pela aplicação.

5.2. Nível 1 - Recursos

Modelos como o POX basicamente expõem funcionalidades a partir de métodos remotos (RPC). Esse modelo muitas vezes é considerado um ponto forte de geração de acoplamento entre clientes e servidores, visto que o cliente deve conhecer previamente a funcionalidade de cada um destes métodos e saber corretamente quando invocá-los.

Um primeiro passo em direção a REST é a modelagem adequada de recursos e utilização dos mesmos para interação com uma API.

Um exemplo de chamada adequada nesse nível de maturidade pode ser:

```
POST /cliente HTTP/1.1
<Cliente>
  <Nome>João da Silva</Nome>
  ...
</Cliente>
```

Este código é muito semelhante ao demonstrado na sessão anterior, porem é importante notar a modelagem correta do recurso “Cliente”.

Modelando corretamente os recursos, precisamos usar os métodos HTTP da forma certa, para que a gente possa criar todas as interações necessárias sob um recurso. É o que vamos discutir na próxima sessão.

5.3. Nível 2 - Verbos HTTP

Nesse nível, o HTTP deixa de exercer um papel apenas de transporte e passa a exercer um papel semântico na API, ou seja, seus verbos passam a ser utilizados com o propósito no qual foram criados.

A utilização do métodos mais conhecidos (*GET*, *POST*, *PUT* e *DELETE*), bem como a tratativa correta dos códigos de resposta, permitem a modelagem e interação com os recursos presentes em uma API.

Considerando o recurso "cliente" dos exemplos anteriores, abaixo segue a modelagem de uma API seguindo o nível 3 de maturidade.

Criando um cliente:

```
POST /cliente HTTP/1.1
<Cliente>
  <Nome>João da Silva</Nome>
  ...
</Cliente>
```

Você deve estar pensando: A mensagem acima não é igual a encontrada na sessão anterior? Sendo assim, qual a diferença? Apesar da mensagem de requisição seguir o modelo apresentado anteriormente, o servidor deverá também agora retornar uma mensagem que reflete o uso adequado do protocolo HTTP, como segue abaixo:

```
HTTP/1.1 201 Created
Location: /cliente/1
```

É importante notar 2 aspectos nessa resposta: O primeiro é a utilização correta da resposta "201 Created". Como foi solicitado a criação de um recurso, nada mais adequado que uma resposta que informe que o recuso foi criado com sucesso. Além disso, um importante aspecto é a presença do *header* "Location". Esse *header* informa em qual endereço o recurso criado se encontra disponível.

Com o endereço fornecido no *header* "Location" (/cliente/1), nós podemos agora fazer a busca por esse recurso:

```
GET /cliente/1 HTTP/1.1

HTTP/1.1 200 OK
<Cliente>
  <Id>1</Id>
  <Nome>João da Silva</Nome>
  ...
</Cliente>
```

Com o intuito de buscar o recurso "/cliente/1", foi utilizado o verbo mais adequado (*GET*) e uma resposta "200 OK" informa que o recurso foi encontrado com sucesso e o mesmo pode ser obtido no corpo da mensagem.

Caso seja necessário a remoção desse recurso, você pode utilizar o método *DELETE*, como mostrado abaixo:

```
DELETE /cliente/1 HTTP/1.1
```

Ao receber essa mensagem, o servidor irá prosseguir com o processamento e retornar a mensagem mais adequada.

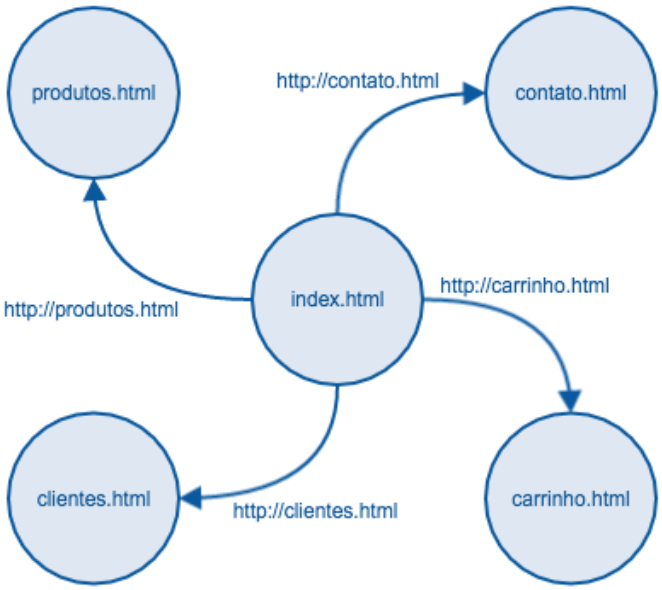
5.4. Nível 3 - HATEOAS

Este nível é certamente a maior dúvida quando se fala sobre REST. Existe uma série de perguntas sobre o que realmente significa HATEOAS (*Hypermedia as the Engine of Application State*) e qual o papel disso em uma API.

Em seu blog pessoal, Fielding deixa muito claro que APIs que não utilizam HATEOAS não podem ser consideradas RESTful, mesmo assim, você vai encontrar muitos conteúdos sobre REST que nem ao menos cita essa característica.

Apesar de aparentemente ser algo não muito familiar, HATEOAS é um conceito presente no dia a dia de todos os usuários da Web. Ele tem como elemento principal uma representação *hypermedia*, que permite um documento descrever seu estado atual e quais os seus relacionamentos com outros futuros estados.

A figura abaixo nos ajuda a entender um pouco melhor o que isso significa.



Perceba que a figura nos mostra uma série de estados e que a ligação entre os mesmos se dá a partir de uma URI. O termo estado é utilizado para denominar cada representação que o servidor responde quando um recurso é solicitado (ex: uma página HTML).

A imagem pode ser traduzida em um código HTML (*Hypermedia Text Markup Language*), para que fique ainda mais claro. Por exemplo, veja como seria o conteúdo de *index.html*:

```
<html>
  <body>
    <a href="produtos.html">Produtos</a>
    <a href="clientes.html">Clientes</a>
    <a href="contato.html">Contato</a>
    <a href="carrinho.html">Carrinho</a>
  </body>
</html>
```

No código HTML acima, podemos notar a presença da tag `<a>`, que diz ao cliente HTML (normalmente um browser) que ele deve executar um *GET* sob cada um dos recursos contidos no atributo `href`.

Como eu posso garantir que sempre que uma tag `<a>` aparecer, o browser irá fazer um *GET*? Isso se dá pelo fato da especificação do HTML nos dizer que toda vez que uma tag `<a>` estiver presente, uma mensagem *GET* deve ser enviada à URI descrita no atributo `href`, ou seja, existe um pré-contrato entre browsers e servidores.

Começamos então a entender um pouco melhor o significado de HATEOAS. Como o próprio nome sugere, a representação *hypermedia* trabalha como um motor de estado, permitindo que clientes naveguem nos mesmos. Cada estado é um documento (uma página HTML) e possui referências para futuros estados (a partir de links).

Dito tudo isso, agora conseguimos entender de forma mais clara a própria origem do nome Web (no português, "teia"). O mesmo se dá pelo fato de links interligarem um conjunto de recursos, formando assim uma enorme teia.

Resumindo, podemos ver que o HTML é uma linguagem que permite que aplicações sejam construídas seguindo o modelo HATEOAS. Porém, como deve ser feita a modelagem de uma API para que ela siga o mesmo formato? E qual o benefício dessa abordagem?

Se analisarmos a forma na qual sistemas Web podem ser construídos ou remodelados, nós podemos perceber que isso se dá sem a necessidade de atualização nos clientes (browsers). O contrato inicialmente conhecido, a linguagem de *hypermedia* HTML, em conjunto com o protocolo HTTP, abstrai qualquer acoplamento direto entre o conteúdo e o que deve ser interpretado pelo navegador, ou seja, todos os dias são adicionadas, removidas e alteradas centenas de páginas Web sem que isso tenha impacto direto em cada um dos elementos envolvidos, como o browser, *proxies*, *gateways*, balanceadores de carga e etc.

Considerando a mesma abordagem para APIs, isso significa que você pode criar APIs totalmente desacopladas e que podem evoluir sem a necessidade de atualizações do lado dos clientes.

Você já se imaginou criando sistemas que podem evoluir sem que haja um impacto dentro de vários sistemas da organização? Já pensou em criar sistemas que possam atender milhares de usuários, assim como serviços que funcionam hoje na Internet?

Criar APIs RESTful significa criar APIs que seguem cada uma das *constraints* descritas nesse livro e suportar HATEOAS. Com isso, você terá a possibilidade de alcançar os benefícios que te mostrei, criando APIs realmente escaláveis, extensíveis e com todas as outras características que a Web possui.

Criar APIs que seguem essa abordagem e programar clientes que usem deste benefício é um enorme desafio, visto que o modelo mental utilizado em aplicações atuais são fortemente baseadas no modelo RPC.

Neste livro, não vou entrar em detalhes sobre como proceder com a criação de clientes de APIs RESTful, mas já deixe em mente que precisamos trabalhar uma forma de criar clientes que usufruam de todos esses benefícios.

Para exemplificar, veja abaixo uma pequena representação de uma API que utiliza *hypermedia*:

```
GET /cliente/1 HTTP/1.1

HTTP/1.1 200 OK
<Cliente>
  <Id>1</Id>
  <Nome>João da Silva</Nome>
  <link rel="deletar" href="/cliente/1" />
  <link rel="notificar" href="/cliente/1/notificacao" />
</Cliente>
```

No exemplo acima, podemos ver a chamada ao recurso "/cliente/1". O retorno evidenciado demonstra a representação do estado e quais as possíveis ações a serem realizadas.

É interessante notar que a URI para deletar e buscar o cliente, aparentemente são iguais (/cliente/1), porém, o atributo "rel" com o valor "deletar", demonstra uma semântica diferente, neste caso, o cliente deve utilizar o método *DELETE* sob a URI.

Como discutimos, o cliente da API deverá entender o significado dos relacionamentos "deletar" e "notificar" para que ele consiga de fato consumir de forma adequada esses links.

A modelagem de uma API baseada em *hypermedia* passa pelo processo de definição sobre qual a melhor linguagem utilizada. Cada API possui características isoladas e precisa ser pensada de forma particular, porém, alguns padrões podem ser utilizados, como o próprio HTML, ATOM PUB ou até mesmo a criação de uma linguagem própria.

Se compararmos a representação do exemplo anterior com um modelo RPC, podemos verificar que conseguimos ir navegando pela API e ela vai nos mostrando quais as possíveis opções, diferente de um sistema RPC como SOAP, que você deve conhecer previamente todas as operações e qual o momento correto de invocar cada uma delas.

De fato, o entendimento e a criação de APIs que seguem esse modelo não é uma tarefa tão trivial, e sugiro que você continue seus estudos a respeito do tema.

Acompanhe a AlgaWorks por e-mail e redes sociais, porque ainda vamos falar muito sobre esse assunto.

Capítulo 6

Conclusão

Chegamos no final do livreto e estou muito feliz por você ter me acompanhado!

Talvez você tenha se surpreendido com uma série de coisas que você aprendeu por aqui. Realmente, existem vários materiais na Web que não condizem ou simplificam muito toda a história por trás do REST. Por isso, é importante você selecionar os materiais confiáveis.

Minha ideia foi realmente abrir um pouco das cortinas e lhe mostrar o que acontece de fato por trás dos palcos deste vasto mundo de APIs RESTful.

De forma resumida, você aprendeu aqui:

- Como o mundo vem mudando e o impacto disso no nosso trabalho
- Como a Web pode nos ajudar a construir sistemas cada vez melhores
- Características de aplicações Web
    - Simples
    - Extensível
    - Escalável
    - Etc…
- Diferenças entre REST e RESTful
- O que de fato difere uma API RESTful de outras APIs
- Diferença entre as abordagens SOAP e REST
- E muito mais…

6.1. Próximos passos

Agora que você já sabe o que é REST, eu recomendo que você aprenda na prática como implementar Web Services RESTful com Spring (Java).

Eu desenvolvi um workshop online com videoaulas sobre esse assunto, e você pode continuar seus estudos agora mesmo. Acesse: http://alga.works/livreto-restspring-cta/.

WORKSHOP ONLINE

Web Services RESTful e Spring

COMPRE AQUI ↓

DESMISTIFICANDO

REST COM JAVA

EMÍLIO DIAS

algaworks