blog.caelum.com.br

REST: Princípios e boas práticas - Blog da Caelum: desenvolvimento, web, mobile, UX e Scrum

19-26 minutos

Representational State Transfer, abreviado como REST, não é uma tecnologia, uma biblioteca, e nem tampouco uma arquitetura, mas sim um modelo a ser utilizado para se projetar arquiteturas de software distribuído, baseadas em comunicação via rede.

REST é um dos modelos de arquitetura que foi descrito por Roy Fielding, um dos principais criadores do protocolo HTTP, em sua tese de doutorado e que foi adotado como o modelo a ser utilizado na evolução da arquitetura do protocolo HTTP.

Muitos desenvolvedores perceberam que também poderiam utilizar o modelo REST para a implementação de Web Services, com o objetivo de se integrar aplicações pela Web, e passaram a utilizá-lo como uma alternativa ao SOAP.

REST na verdade pode ser considerado como um conjunto de princípios, que quando aplicados de maneira correta em uma aplicação, a beneficia com a arquitetura e padrões da própria Web.

Vejamos agora esses princípios e como utilizá-los de maneira correta.

Identificação dos Recursos

Toda aplicação gerencia algumas informações. Uma aplicação de um E-commerce, por exemplo, gerencia seus produtos, clientes, vendas, etc. Essas **coisas** que uma aplicação gerencia são chamadas de **Recursos** no modelo REST.

Um recurso nada mais é do que uma abstração sobre um determinado tipo de informação que uma aplicação gerencia, sendo que um dos princípios do REST diz que todo recurso deve possuir uma identificação única. Essa identificação serve para que a aplicação consiga diferenciar qual dos recursos deve ser manipulado em uma determinada solicitação.

Imagine a seguinte situação: Você desenvolveu um Web Service REST que gerencia seis tipos de recursos. Os clientes desse Web Service manipulam esses recursos via requisições HTTP. Ao chegar uma requisição para o Web Service, como ele saberá qual dos recursos deve ser manipulado? É justamente por isso que os recursos devem possuir uma identificação única, que deve ser informada nas requisições.

A identificação do recurso deve ser feita utilizando-se o conceito de URI (Uniform Resource Identifier), que é um dos padrões utilizados pela Web. Alguns exemplos de URI's:

- http://servicorest.com.br/produtos;
- http://servicorest.com.br/clientes;
- http://servicorest.com.br/clientes/57;
- http://servicorest.com.br/vendas.

As URI's são a interface de utilização dos seus serviços e funcionam como um **contrato** que será utilizado pelos clientes para acessá-los. Vejamos agora algumas boas práticas no uso de

URI's.

Utilize URI's legíveis

Ao definir uma URI, utilize nomes legíveis por humanos, que sejam de fácil dedução e que estejam relacionados com o domínio da aplicação. Isso facilita a vida dos clientes que utilizarão o serviço, além de reduzir a necessidade de documentações extensas.

Utilize o mesmo padrão de URI na identificação dos recursos

Mantenha a consistência na definição das URI's. Crie um padrão de nomenclatura para as URI's dos recursos e utilize sempre esse mesmo padrão. Evite situações como:

- http://servicorest.com.br/produto (Singular);
- http://servicorest.com.br/clientes (Plural);
- http://servicorest.com.br/processosAdministrativos (Camel Case);
- http://servicorest.com.br/processos_judidicais (Snake Case).

Evite adicionar na URI a operação a ser realizada no recurso

Os recursos que uma aplicação gerencia podem ser manipulados de diversas maneiras, sendo para isso disponibilizada algumas operações para manipula-los, tais como: criar, listar, excluir, atualizar, etc.

A manipulação dos recursos deve ser feita utilizando-se os métodos do protocolo HTTP, que inclusive é um dos princípios do REST que será discutido mais adiante.

Portanto, evite definir URI's que contenham a operação a ser realizada em um recurso, tais como:

- http://servicorest.com.br/produtos/cadastrar;
- http://servicorest.com.br/clientes/10/excluir;
- http://servicorest.com.br/vendas/34/atualizar.

Evite adicionar na URI o formato desejado da representação do recurso

É comum que um serviço REST suporte múltiplos formatos para representar seus recursos, tais como XML, JSON e HTML. A informação sobre qual o formato desejado por um cliente ao consultar um serviço REST deve ser feita via Content Negotiation, conforme será mostrado mais adiante.

Portanto, evite definir URI's que contenham o formato desejado de um recurso, tais como:

- http://servicorest.com.br/produtos/xml;
- http://servicorest.com.br/clientes/112?formato=json.

Evite alterações nas URI's

A URI é a porta de entrada de um serviço. Se você a altera, isso certamente causará impacto nos clientes que estavam a utilizando, pois você alterou a forma de acesso a ele. Após definir uma URI e disponibilizar a manipulação de um recurso por ela, evite ao máximo sua alteração.

Nos casos mais críticos, no qual realmente uma URI precisará ser alterada, notifique os clientes desse serviço previamente. Verifique também a possibilidade de se manter a URI antiga, fazendo um redirecionamento para a nova URI.

Utilização dos métodos HTTP para manipulação dos recursos

Os recursos gerenciados por uma aplicação, e identificados unicamente por meio de sua URI, geralmente podem ser manipulados de diversas maneiras. É possível criá-los, atualizá-los, excluí-los, dentre outras operações.

Quando um cliente dispara uma requisição HTTP para um serviço, além da URI que identifica quais recursos ele pretende manipular, é necessário que ele também informe o tipo de manipulação que deseja realizar no recurso. É justamente aí que entra um outro conceito da Web, que são os métodos do protocolo HTTP.

O protocolo HTTP possui diversos métodos, sendo que cada um possui uma semântica distinta, e devem ser utilizados para indicar o tipo de manipulação a ser realizada em um determinado recurso.

Vejamos agora os principais métodos do protocolo HTTP e o cenário de utilização de cada um deles:

GET	Obter os dados de um recurso.
POST	Criar um novo recurso.
PUT	Substituir os dados de um determinado recurso.
PATCH	Atualizar parcialmente um determinado recurso.
DELETE	Excluir um determinado recurso.
HEAD	Similar ao GET, mas utilizado apenas para se obter os cabeçalhos de resposta, sem os dados em si.
OPTIONS	Obter quais manipulações podem ser realizadas em um determinado recurso.

Geralmente as aplicações apenas utilizam os métodos GET, POST, PUT e DELETE, mas se fizer sentido em sua aplicação utilizar

algum dos outros métodos, não há nenhum problema nisso.

Veja a seguir o padrão de utilização dos métodos HTTP em um serviço REST, que é utilizado pela maioria das aplicações e pode ser considerado uma boa prática. Como exemplo será utilizado um recurso chamado Cliente.

Método	URI	Utilização
GET	/clientes	Recuperar os dados de todos os clientes.
GET	/clientes/id	Recuperar os dados de um determinado cliente.
POST	/clientes	Criar um novo cliente.
PUT	/clientes/id	Atualizar os dados de um determinado cliente.
DELETE	/clientes/id	Excluir um determinado cliente.

Como boa prática, em relação aos métodos do protocolo HTTP, evite utilizar apenas o método POST nas requisições que alteram o estado no servidor, tais como: cadastro, alteração e exclusão, e principalmente, evite utilizar o método GET nesses tipos de operações, pois é comum os navegadores fazerem cache de requisições GET, as disparando antes mesmo do usuário clicar em botões e links em uma pagina HTML.

Representações dos recursos

Os recursos ficam armazenados pela aplicação que os gerencia. Quando são solicitados pelas aplicações clientes, por exemplo em uma requisição do tipo GET, eles não “abandonam” o servidor, como se tivessem sido transferidos para os clientes. Na verdade, o que é transferido para a aplicação cliente é apenas uma **representação** do recurso.

Um recurso pode ser representado de diversas maneiras,

utilizando-se formatos específicos, tais como XML, JSON, HTML, CSV, dentre outros. Exemplo de representação de um recurso no formato XML.

```
<cliente>
  <nome>Rodrigo</nome>
  <email>rodrigo@email.com.br</email>
  <sexo>Masculino</sexo>
  <endereco>
    <cidade>Brasilia</cidade>
    <uf>DF</uf>
  </endereco>
</cliente>
```

A comunicação entre as aplicações é feita via transferência de representações dos recursos a serem manipulados. Uma representação pode ser também considerada como a indicação do **estado** atual de determinado recurso.

Essa comunicação feita via transferência de representações dos recursos gera um desacoplamento entre o cliente e o servidor, algo que facilita bastante a manutenção das aplicações.

Suporte diferentes representações

É considerada uma boa prática o suporte a múltiplas representações em um serviço REST, pois isso facilita a inclusão de novos clientes. Ao suportar apenas um tipo de formato, um serviço REST limita seus clientes, que deverão se adaptar para

conseguir se comunicar com ele.

Os três principais formatos suportados pela maioria dos serviços REST são:

- HTML;
- XML;
- JSON.

Utilize Content Negotiation para o suporte de múltiplas representações

Quando um serviço REST suporta mais de um formato para as representações de seus recursos, é comum que ele espere que o cliente forneça a informação de qual o formato desejado. No REST, essa negociação do formato da representação dos recursos é chamada de **Content Negotiation** e no mundo Web ela deve ser feita via um cabeçalho HTTP, conhecido como **accept**.

Ao fazer uma chamada ao serviço REST, um cliente pode adicionar na requisição o cabeçalho accept, para indicar ao servidor o formato desejado da representação do recurso. Claro, deve ser um formato que seja suportado pelo serviço REST.

Comunicação Stateless

A Web é o principal sistema que utiliza o modelo REST. Hoje ela suporta bilhões de clientes conectados e trocando informações. Mas, como é possível a Web ter uma escalabilidade e performance tão boas, a ponto de conseguir suportar tamanho número de clientes sem problemas?

A resposta: Comunicação Stateless!

Requisições feitas por um cliente a um serviço REST devem conter todas as informações necessárias para que o servidor as interprete e as execute corretamente. Clientes não devem depender de dados previamente armazenados no servidor para processar uma requisição. Qualquer informação de estado deve ser mantida pelo cliente e não pelo servidor. Isso reduz a necessidade de grandes quantidades de recursos físicos, como memória e disco, e também melhora a escalabilidade de um serviço REST.

É justamente por essa característica que a Web consegue ter uma escalabilidade praticamente infinita, pois ela não precisa manter as informações de estado de cada um dos clientes.

Esse é um dos princípios mais difíceis de ser aplicado em um serviço REST, pois é muito comum que aplicações mantenham estado entre requisições de clientes. Um exemplo dessa situação acontece quando precisamos armazenar os dados dos usuários que estão autenticados na aplicação.

Evite manter dados de autenticação/autorização em sessão

A principal dificuldade em criar um serviço REST totalmente Stateless ocorre quando precisamos lidar com os dados de autenticação/autorização dos clientes. A dificuldade ocorre porque é natural para os desenvolvedores armazenarem tais informações em sessão, pois essa é a solução comum ao se desenvolver uma aplicação Web tradicional.

A principal solução utilizada para resolver esse problema é a utilização de Tokens de acesso, que são gerados pelo serviço REST e devem ser armazenados pelos clientes, via cookies ou HTML 5 Web Storage, devendo também ser enviados pelos

clientes a cada nova requisição ao serviço.

Já existem diversas tecnologias e padrões para se trabalhar com Tokens, dentre elas:

- OAUTH;
- JWT (JSON Web Token);
- Keycloack.

Inclusive tal assunto já foi tratado aqui no blog.

HATEOAS (Hypermedia As The Engine Of Application State)

Para entendermos melhor o conceito de HATEOAS, vamos a um exemplo comum quando utilizamos a Web.

Imagine que você quer comprar um produto pela Web, em algum site de E-commerce. Você entra no site, navega pelas categorias de produtos via menu do site, encontra o seu produto e chega até a tela de detalhes dele, que possui as informações de preço, frete e o botão para realizar a compra.

Mas nem sempre o produto está disponível em estoque, por isso o site costuma fazer um tratamento nessa situação, escondendo o botão de realizar a compra e exibindo um botão para notificação quando o produto voltar a estar disponível.

No exemplo anterior, o conceito de HATEOAS foi aplicado duas vezes. Primeiro, quando o cliente entrou no site de E-commerce, como ele fez para chegar até a página de detalhes do produto? Navegando via **links**.

Normalmente todo site ou aplicação Web possui diversas

funcionalidades, sendo que a navegação entre elas costuma ser feita via links. Um cliente até pode saber a URI de determinada página ou recurso, mas se ele não souber precisamos guiá-lo de alguma maneira para que ele encontre as informações que procura.

O segundo uso do HATEOAS ocorreu quando o cliente acessou a página de detalhes do produto. Se o produto estivesse em estoque, o site apresentaria o botão de comprar, e o cliente poderia finalizar seu pedido. Caso contrário, ele apenas poderia registrar-se para ser notificado. Tudo isso é feito, principalmente, para garantir a consistência das informações.

Perceba que os links foram utilizados como mecanismo para conduzir o cliente quanto à navegação e ao estado dos recursos. Esse é o conceito que foi chamado de HATEOAS, que nada mais é do que a utilização de Hypermedia, com o uso de links, como o motor para guiar os clientes quanto ao estado atual dos recursos, e também quanto as transições de estado que são possíveis no momento.

Veja um exemplo de uma representação de um recurso sem a utilização do conceito de HATEOAS:

```
<pedido>
  <id>1459</id>
  <data>2017-01-20</data>
  <status>PENDENTE</status>
  <cliente>
    <nome>Rodrigo</nome>
```

```
  </cliente>
</pedido>
```

No exemplo anterior foi apresentado a representação do recurso “Pedido” no formato XML, contendo suas informações, mas sem o uso do HATEOAS. Isso certamente pode gerar algumas dúvidas para os clientes desse serviço REST, como por exemplo:

- É possível solicitar o cancelamento do pedido? Como?
- Quais são os outros estados do pedido e como transitar entre eles?
- Como obter mais informações sobre o cliente desse pedido?

Essas dúvidas poderiam ser respondidas facilmente se o conceito HATEOAS fosse utilizado, facilitando assim a vida dos clientes do serviço REST. Vejamos agora a mesma representação, porém com a utilização do HATEOAS:

Perceba como agora ficou muito mais simples **explorar** as informações e descobrir quais caminhos seguir. HATEOAS é um dos princípios que dificilmente vemos sendo aplicados em serviços REST no mercado, quase sempre por falta de conhecimento dos desenvolvedores.

Utilização correta dos códigos HTTP

Na verdade, esse não é um princípio do REST, mas sim uma boa prática.

No protocolo HTTP, toda requisição feita por um cliente a um servidor deve resultar em uma resposta, sendo que nela existe um código HTTP, utilizado para informar o resultado da requisição, tenha ela sido processada com sucesso ou não.

Existem dezenas de códigos HTTP, cada um tendo sua semântica especifica e devendo ser utilizado quando fizer sentido. Os códigos HTTP são agrupados em classes, conforme demonstrado a seguir:

Classe	Semântica
2xx	Indica que a requisição foi processada com sucesso.
3xx	Indica ao cliente uma ação a ser tomada para que a requisição possa ser concluída.
4xx	Indica erro(s) na requisição causado(s) pelo cliente.
5xx	Indica que a requisição não foi concluída devido a erro(s) ocorrido(s) no servidor.

A boa prática consiste em conhecer os principais códigos HTTP e utilizá-los de maneira correta. Veja os principais códigos HTTP e quando os utilizar:

Código	Descrição	Quando utilizar
200	OK	Em requisições GET, PUT e DELETE executadas com sucesso.
201	Created	Em requisições POST, quando um novo recurso é criado com sucesso.
206	Partial Content	Em requisições GET que devolvem apenas uma parte do conteúdo de um recurso.
302	Found	Em requisições feitas à URI's antigas, que foram alteradas.
400	Bad Request	Em requisições cujas informações enviadas pelo cliente sejam invalidas.
401	Unauthorized	Em requisições que exigem autenticação, mas seus dados não foram

		fornecidos.
403	Forbidden	Em requisições que o cliente não tem permissão de acesso ao recurso solicitado.
404	Not Found	Em requisições cuja URI de um determinado recurso seja inválida.
405	Method Not Allowed	Em requisições cujo método HTTP indicado pelo cliente não seja suportado.
406	Not Acceptable	Em requisições cujo formato da representação do recurso requisitado pelo cliente não seja suportado.
415	Unsupported Media Type	Em requisições cujo formato da representação do recurso enviado pelo cliente não seja suportado.
429	Too Many Requests	No caso do serviço ter um limite de requisições que pode ser feita por um cliente, e ele já tiver sido atingido.
500	Internal Server Error	Em requisições onde um erro tenha ocorrido no servidor.
503	Service Unavailable	Em requisições feitas a um serviço que esta fora do ar, para manutenção ou sobrecarga.

Utilize o código correto para cada tipo de situação. Evite a má prática de sempre utilizar um mesmo código genérico para todas as situações, como por exemplo o código 200 para qualquer tipo de requisição bem-sucedida ou o código 500 para qualquer requisição malsucedida.

Conclusão

REST não é um bicho de sete cabeças. Possui apenas alguns poucos princípios e restrições que devem ser utilizados para se garantir algumas características importantes em aplicações e serviços, tais como: portabilidade, escalabilidade e desacoplamento.

A Web é o principal exemplo de implementação do modelo REST, e deveríamos nos espelhar nela ao construir nossos serviços seguindo esse mesmo modelo.

Todos os conceitos do modelo REST descritos nesse post podem também ser utilizados na implementação de aplicações Web, e não somente nos casos de integração de sistemas via Web Services.

Inclusive se você utilizar esses conceitos em uma aplicação Web tradicional, e um dia ela precisar se tornar uma API REST, o impacto das mudanças será mínimo, pois sua aplicação já estará seguindo os princípios REST. O que mudará é que agora ao invés de devolver para os clientes apenas páginas HTML ela também pode devolver as informações dos recursos em algum formato, como o JSON, e o cliente é quem vai se preocupar em como formatá-los.

Quando uma aplicação ou serviço segue os princípios descritos nesse artigo, ela é chamada de RESTful. Alguns puristas consideram apenas como RESTful a aplicação ou serviço que seguir **todos** os princípios do REST, inclusive os menos utilizados, como comunicação Stateless e HATEOAS.

Mas, não foque em ser ou não considerado RESTful, e sim em tentar utilizar o máximo possível dos princípios REST, para que sua aplicação ou serviço obtenha todos os benefícios desse modelo.

E você leitor, utiliza de maneira correta os princípios REST? Conte-

nos sua experiência
🙂