Uma rápida Introdução ao REST

por Stefan Tilkov Stefan Tilkov Seguir 5 Seguidores, traduzido por André Faria Gomes André Faria Gomes Seguir 0 Seguidores em 29 out 2008. Tempo estimado de leitura: 19 minutos 8 Dê sua opinião

Você pode ou não estar ciente de que há um debate acontecendo sobre o modo "certo" de implementar comunicação heterogênea de aplicação-para-aplicação: Embora o mainstream atual claramente incide sobre serviços web baseados em SOAP, WSDL e o universo das especificações WS-* , uma pequena, mas notável minoria reivindica que há uma forma melhor: REST, abreviação de "REpresentational State Transfer". Neste artigo, eu tentarei dar uma introdução pragmática ao REST e à integração de aplicações HTTP RESTful sem entrar neste debate. Eu entrarei em maiores detalhes quando for explicar sobre aqueles aspectos que, na minha experiência, causam mais discussão quando alguém é exposto a essa abordagem pela primeira vez.

Princípios Chave de REST

A maioria das introduções à REST começam com uma definição formal e o seu histórico. Eu vou fugir disso por um instante e darei uma simples e pragmática definição: REST é um conjunto de princípios que definem como Web Standards como HTTP e URIs devem ser usados (o que freqüentemente difere um pouco do que muitas pessoas atualmente fazem). A promessa é que se você aderir a princípios REST enquanto estiver desenhando sua aplicação, você terá um sistema que explora a arquitetura da Web em seu benefício. Em resumo, os cinco princípios fundamentais são os seguintes:

- Dê a todas as coisas um Identificador
- Vincule as coisas
- Utilize métodos padronizados
- Recursos com múltiplas representações
- Comunique sem estado

Vamos ver mais de perto cada um desses princípios.

Dê a todas as "coisas" um ID

Eu estou usando o termo "coisas" ao invés do termo formalmente correto "recurso" porque esse é um principio simples que não deveria ser escondido por trás da terminologia. Se você pensar sobre os sistemas que as pessoas constroem, há usualmente um conjunto de abstrações chave que merecem ser identificadas. Tudo o que deveria ser identificado deveria obviamente ter um ID - Na Web, há um conceito unificado para IDs: A URI. URIs compõe um namespace global,e utilizando URIs para identificar seus recursos chave significa ter um ID único e global.

O principal beneficio de um esquema de nomes consistente para as coisas é que você não tem que criar o seu próprio esquema - você pode confiar em um que já tenha sido definido, trabalha muito bem em escala global e é entendido por praticamente qualquer um. Se você considerar um objeto arbitrário de alto nível com a última aplicação que você construiu (assumindo que não foi construído de forma RESTful), e quase certo que havia muitos casos de uso onde você deve ter lucrado com isso. Se a sua aplicação incluir uma abstração de Cliente, por exemplo, eu estou quase certo de que os usuários gostariam de poder enviar um link para um determinado cliente via e-mail, para um colega de trabalho, criar um favorito para ele no seu navegador, ou mesmo anotá-la em um pedaço de papel. Para esclarecer melhor, imagine que decisão de negócio terrivelmente ruim teria sido se uma loja on-line como a amazon.com não identificasse cada um dos seus produtos com um ID único (uma URI).

Quando confrontados com essa idéia, muitas pessoas se perguntam se isso significa que devem expor suas entradas de banco de dados (ou seus IDs) diretamente - e frequentemente ficam revoltadas pela simples idéia, uma vez que anos de prática de orientação a objetos nos disseram para esconder os aspectos de bancos de dados. Por exemplo, um recurso de Pedido poderia ser composto de itens de pedido, um endereço e muitos outros aspectos que você poderia não querer expor como um recurso identificável individualmente. Tomando a idéia de identificar de tudo o que vale a pena a ser identificado, leva à criação de recursos que você normalmente não vê, em um típico design de aplicação: Um processo ou um passo de um processo, uma venda, uma negociação, uma solicitação de quotação - estes são todos exemplos de "coisas" que precisam ser identificadas. Isto, por sua vez, pode levar à criação de entidades mais persistentes do que em um design não RESTful.

Alguns exemplos exemplos de URIs que poderíamos ter:

```
http://example.com/customers/1234
http://example.com/orders/2007/10/776654
http://example.com/products/4554
http://example.com/processes/salary-increase-234
```

Como eu escolhi criar URIs que possam ser lidas por seres humanos - um conceito útil, mesmo que não seja um pré-requisito para um design RESTful - deve ser bastante fácil de adivinhar seu significado: Elas obviamente identificam "itens" individuais. Mas dê só uma olhada nestas:

```
http://example.com/orders/2007/11
http://example.com/products?color=green
```

No início, isso pode parecer algo diferente - afinal de contas, eles não são a identificação de uma coisa, mas um conjunto de coisas (assumindo que a primeira URI identifica todos os pedidos submetidos em novembro de 2007, e a segunda, um conjunto de produtos verdes). Mas esses conjuntos são realmente coisas - recursos - os por si só, e eles definitivamente precisam ter um identificador.

Note os benefícios de se ter um único esquema de nomes a nível global aplicável tanto para a Web em seu navegador como para comunicação de máquina para máquina.

Para resumir o primeiro princípio: Use URIs para identificar tudo o que precisar ser identificado, especifique todos os recursos de "alto nível" que seu aplicativo oferece, se eles representam itens individuais, conjuntos de itens, objetos virtuais e físicos, ou resultados de computação.

Vincule as coisas

O próximo princípio que veremos tem uma descrição formal que intimida um pouco: "Hipermídia como motor do estado do aplicativo", as vezes abrevido como HATEOAS. (É sério - Eu não estou inventando isso.) No seu núcleo está o conceito de *hipermídia*, ou em outras palavras: *links*. Links são algo que nós estamos familiarizados no HTML, mas eles não são de forma alguma restritos ao consumo de humanos. Considere o seguinte fragmento XML:

```
<order self="http://example.com/customers/1234">
   <amount>23</amount>
   <product ref="http://example.com/products/4554"></product>
   <customer ref="http://example.com/customers/1234"></customer>
</order>
```

Se você observar os links de produto (product) e cliente (customer) nesse documento, você poderá facilmente imaginar como um aplicativo que o tenha obtido pode interpretar os links para obter mais informações. Evidentemente, este seria o caso se houvesse um simples atributo identificador aderindo a algum esquema de nomeclatura específica de um aplicativo, também - *Mas apenas no contexto do aplicativo*. TA beleza da abordagem de links com URIs é que os links podem apontar para recursos que são oferecidos por uma aplicação diferente, um outro servidor, ou até mesmo em uma empresa diferente em outro continente - porque o esquema de nomes é um padrão global, todos os recursos que compõe a Web podem ser ligados uns aos outros.

THá um aspecto ainda mais importante do princípio de hipermídia - a parte de estado do aplicativo. Em suma, o fato de que o servidor (ou provedor de serviços, se você preferir) oferece um conjunto de links para o cliente (o consumidor do serviço), permite ao cliente mudar o aplicativo de um estado para outro, através de um link. Nós vamos observar os efeitos deste aspecto em outro artigo, em breve, por enquanto, tenha em mente que os links são extremamente úteis para fazer uma aplicação dinâmica.

Para resumir esses principios: Use liks para referênciar coisas que possam ser identificadas (recursos) sempre que for possível. Hiperlinks são o que fazem a Web ser a Web.

Utilize os métodos padrão

Havia um pressuposto implícito na discussão dos primeiros dois princípios: a aplicação consumidora pode realmente fazer algo significativo com URIs. Se você ver uma URI escrita na lateral de um ônibus, você pode inserí-la na barra de endereços no seu navegador e presionar enter - mas como é que o seu navegador sabe o que fazer com a URI?

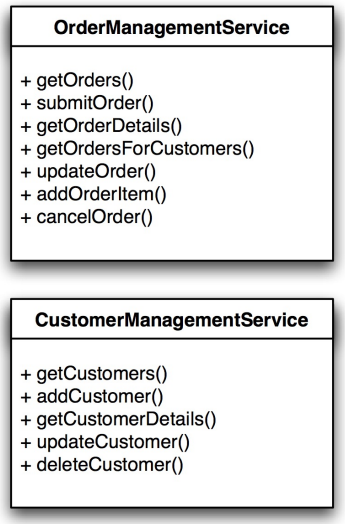
Ele sabe o que fazer com ela, porque todos os recursos possuem a mesma interface, e o mesmo conjunto de métodos (ou operações, se preferir). O HTTP chama esse verbos, e além dos dois que todo mundo conhece (o GET e o POST), o conjunto de métodos padrão inclui PUT, DELETE, HEAD e OPTIONS. O significado de cada um desses métodos é definido na especificação do HTTP, juntamente com algumas garantias sobre o seus comportamentos. Se você é um desenvolvedor OO, você pode imaginar que todo recurso em um cenário HTTP RESTful estende uma classe como essa (em alguma pseudo-sintaxe no estilo Java/C# concentrando-se nos métodos principais):

```
class Resource {
     Resource(URI u);
     Response get();
     Response post(Request r);
     Response put(Request r);
     Response delete();
}
```

Devido ao fato de a mesma interface ser usada para todos os recursos, você pode confiar que é possível obter uma *representação* - ou seja, uma renderização do recurso - utilizando GET. Como a semântica do GET é definida na especificação, você pode ter certeza que tem obrigações quando chamá-lo - é por isso que o método é chamado de "seguro". O GET suporta caching de forma muito eficiente e sofisticada, em muitos casos, você nem sequer precisa enviar uma requisição ao servidor. Você também pode ter certeza de que o GET é *idempotente* - se você emitir uma requisição GET e não obter um resultado, você pode não saber se o seu pedido nunca alcançou o seu destino ou se a resposta foi perdida no caminho de volta para você. A garantia da idempotência significa que você pode simplesmente emitir a requisição novamente. A idempotência também é garantida para o PUT (que basicamente significa "atualizar esse recurso com base nesses dados, ou criá-lo nessa URI se ainda não estiver lá") e para o DELETE (que você pode simplesmente tentar de novo e de novo até obter um resultado - apagar algo que não existe não é um problema). POST, que normalmente significa "criar um novo recurso", pode também ser utilizado para invocar um processamento arbitrário, portanto, não é nem seguro nem idempotente.

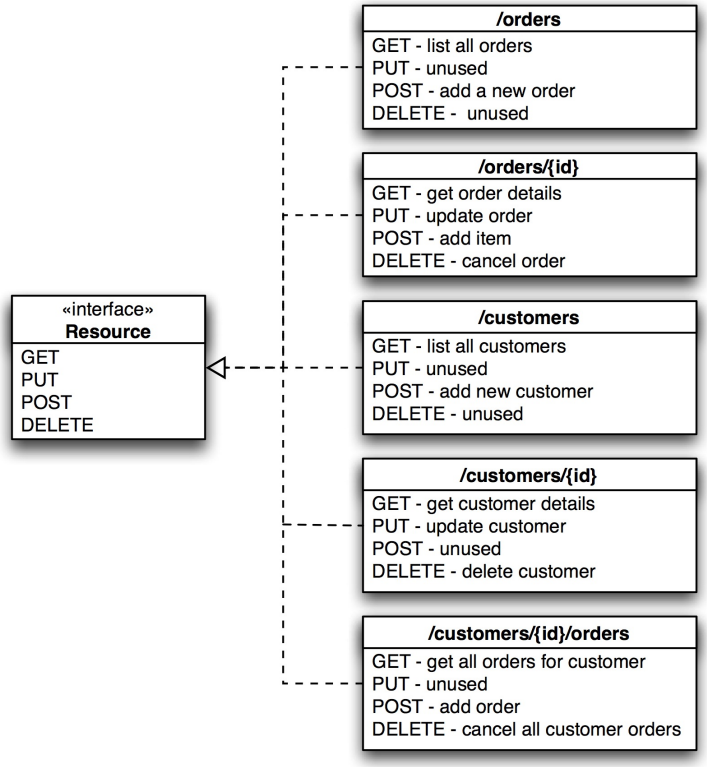
Se você expuser as funcionalidades do seu aplicativo (ou funcionalidades do serviço, ser você preferir) do modo RESTful, *esse princípio e suas restrições se aplicam a você também*. Isso é difícil de aceitar se você estiver acostumado com uma abordagem de design diferente - afinal, você esta provavelmente convencido de que *seu* aplicativo tem muito mais lógica do que é expressado com operações manuais. Permita-me passar algum tempo tentando convencê-lo de que esse não é o caso.

Considere o seguinte exemplo de um simples cenáro de compras:

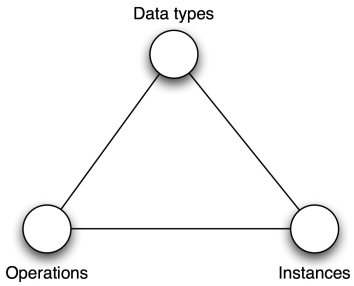


Você pode ver que há dois serviços definidos aqui (sem implicar qualquer implementação tecnológica específica). A interface para esses serviços é específica para a tarefa - são dos serviços OrderManagement (Gerenciador de Pedidos) e CustomerManagement (Gerenciador de Clientes) que estamos falando. Se um cliente (aplicação) quiser consumir esse serviços, será necessário codificar sobre dessa interface específica - não há como usar um cliente que tenha sido construído antes de que essas interfaces tenham sido especificadas para interagir com elas. As interfaces definem o protocolo dos serviços do aplicativo.

Em uma abordagem HTTP RESTful, você teria que começar por uma inteface genérica que compusesse o *protocolo HTTP do aplicativo*. Você poderia fazer algo assim:



Você pode ver que o que tem operações específicas de um serviço que foi mapeado para os métodos padrão do HTTP - e para diferenciar. Eu criei um universo inteiro de novos recursos. "Isso é trapaça!", Eu ouço você chorar. Não - não é. Um GET em uma URI que identifica um cliente é tão significativo como uma operação getCustomerDetails. Algumas pessoas usam um triângulo para visualizar isto:



Imagine as três vértices como maçanetas que você tem que virar. Você pode ver que na primeira abordagem, você tem muitas operações e muitos tipos de dados e um número determinado de "instâncias" (essencialmente, tantas quantos serviços você tiver). Na segunda, você tem um número determinado de operações, muitos tipos de dados e muitos objetos para invocar esse determinado método. O ponto é ilustrar que você pode basicamente expressar qualquer coisa que quiser com ambas as abordagens.

Porque que isto é importante? Essencialmente, isso faz seu aplicativo ser parte da Web - sua contribuição para o que tornou a Web, a aplicação de maior sucesso da Internet é proporcional ao número de recursos que sua app adiciona a ela. Em uma abordagem RESTful, um aplicativo pode adicionar alguns milhões de URIs de clientes na Web; Se for concebido da mesma forma que os aplicativos dos tempos do CORBA, essa contribuição é frequentemente um único "endpoint" - comprando a uma porta muita pequena que ofereça entrada para um universo de recursos apenas para aqueles que têm a chave.

A interface uniforme também permite que qualquer componente que entenda o protocolo de aplicação HTTP interaja com seu aplicativo. Exemplos de componentes que são beneficiados por isso são clientes genéricos como o curl, o wget, proxies, caches, servidores HTTP, gateways e até mesmo o Google, o Yahoo!, o MSN e muitos outros.

Para resumir: Para que clientes possam interagir com seus recursos, eles devem implementar o protocolo de aplicação padrão (HTTP) corretamente, isto é, utilizar os métodos padrão: GET, PUT, POST e DELETE.

Recursos com múltiplas representações

Nós ignoramos uma ligeira complicação até agora: Como é que um cliente saberá como lidar com os dados que obtém, por exemplo, como resultado de uma requisição GET ou POST? A abordagem do HTTP permite uma separação entre as responsabilidades de manipulação de dados e invocação de operações. Em outras palavras, um cliente que sabe como lidar com um formato específico de dados pode interagir com todos os recursos que possam oferecer uma representação nesse formato. Vamos ilustrar isso novamente, com um exemplo. Utilizando a negociação de conteúdo HTTP, um cliente pode solicitar por uma representação em um formato específico:

```
GET /customers/1234 HTTP/1.1
Host: example.com
Accept: application/vnd.mycompany.customer+xml
```

O resultado pode ser um XML em algum formato específico de um empresa que representa os dados de um cliente. Se o cliente (HTTP) enviar uma solicitação diferente, isso é, como essa:

```
GET /customers/1234 HTTP/1.1
Host: example.com
Accept: text/x-vcard
```

O resultado poderia ser o endereço de um cliente no formato vCard. (Eu não estou mostrando as respostas, que conteriam metadados sobre o tipo de dados no cabeçalho de tipo de conteúdo do HTTP.) Isso ilustra porque, idealmente, as representações de um recurso devem ser em formatos padrão - se um cliente "conhece" ambos, os protocolos de aplicação HTTP e um conjunto de formato de dados, *pode interagir com qualquer aplicativo HTTP RESTful do mundo* de forma muito significativa. Infelizmente, nós não temos formatos padronizados para tudo, mas você provavelmente pode imaginar como se poderia criar um ecossistema menor em uma empresa ou em um conjunto de parceiros colaboradores com base nos formatos padrão. Certamente tudo isso não se aplica somente a dados enviados de um servidor para um cliente, mas também na direção oposta - um servidor que pode consumir dados em um formato específico não se preocupa com o tipo específico do cliente, desde que respeite o protocolo do aplicativo.

Há um outro benefício significativo de se ter múltiplos formatos de um recurso na prática: Se você fornecer ambos, um formato HTML e um XML de seus recursos, eles poderão ser consumidos não apenas pelo seu aplicativo, mas também por qualquer navegador web padrão - em outras palavras, a informação do seu aplicativo ficará disponível para qualquer um que sabe como utilizar a Web.

Existe outra forma de explorar isso: Você pode transformar a interface gráfica do seu aplicativo Web em uma API Web - além disso, o Design da API é freqüentemente movido pela ídeia de que tudo que pode ser feito pela interface gráfica deveria também poder ser feito através da API. Combinar as duas tarefas é um boa forma de obter uma melhor interface Web para ambos, seres humanos e aplicativos.

Em resumo: Ofereça diversos formatos dos recursos para diferentes necessidades.

Comunique sem Estado

O último princípio que eu gostaria de abordar é a *comunicação sem estado*. Primeiramente, é importante salientar que embora REST inclua a ídeia de "não manter", isso não significa que o aplicativo que exponha suas funcionalidades não possa ter estado - de fato, isso tornaria a abordagem inútil na maioria dos cenários. REST exige que o estado seja transformado no estado do recurso ou mantido no cliente. Em outras palavras, um servidor não deveria guardar o estado da comunicação de qualquer um dos clientes que se comunique com ele além de uma única requisição. A razão mais óbvia para isso é escalabilidade - o número de clientes que podem interagir com o servidor seria consideravelmente impactado se fosse preciso manter o estado do cliente. (Note que isso geralmente requer um pouco de re-design - você não pode simplesmente manter um URI com algumas sessões com estado e chamar isso de RESTful.)

Mas existem outros aspectos que poderiam ser muito mais importantes: As restrições de "não manter" isolam o cliente de mudanças no servidor, dessa forma, duas solicitações consecutivas não dependem de comunicação com o mesmo servidor. Um cliente poderia receber um documento contendo links do servidor, e então faria algum processamento, e o servidor poderia ser desligado, e o disco rígido poderia ser jogado fora ou substituído, e o software poderia ser atualizado e reiniciado - e se o cliente acessase um dos links recebidos do servidor, ele não teria nem percebido.

REST em Teoria

Eu tenho uma confissão a fazer: O que eu expliquei não é realmente REST, e eu poderia me queimar por simplificar um pouco demais. Mas eu queria começar as coisas de uma forma um pouco diferente do habitual, por isso eu não falei do cenário formal e da história do REST no início. Deixem-me tentar resolver esta questão, com algo breve.

Antes de tudo, eu evitei tomar muito cuidado em separar REST do HTTP em si e do uso do HTTP de forma RESTful. Para compreender a relação entre estes dois diferentes aspectos, nós temos que dar uma olhada na história do REST.

O termo REST foi definido por Roy T. Fielding em sua tese de PhD (talvez você queria ver este link — é bem legível, ao menos para uma dissertação). Roy foi um dos principais desenvolvedores de muitos dos protocolos Web essenciais, incluindo HTTP e URIs, e ele formalizou várias das idéias por trás deles nesse documento. (A dissertação é considerada “A Bíblia do REST”, e justamente por isso — já que o autor inventou o tema, então por definição, tudo que ele escreve deve ser considerado imperativo.) Nesta dissertação, Roy primeiro define uma metodologia de falar sobre *estilos arquiteturais* — alto nível, padrões de abstração que expressam as principais ídeias por trás de uma abordagem arquitetural . Cada estilo arquitetural com um conjunto de *regras* que o define. Exemplos de estilos arquiteturais incluem “o estilo nulo” (que não possue regras), pipe e filter, cliente/servidor, objetos distrubuídos e — adivinhe — REST.

Se tudo isso soa um pouco abstrato pra você. Você está certo - REST em si é um estilo de alto nível que poderá ser implementado utilizando muitas tecnologias diferentes, e instanciado utilizando diferentes valores para suas propriedades abstratas. Por exemplo, REST incluí conceitos de recursos e uma interface uniforme - ou seja, a idéia de que todo recurso deveria responder aos mesmos métodos. Mas REST não diz que métodos devem ser estes, nem quantos eles deveriam ser.

Uma "encarnação" do estilo REST é o HTTP (e um conjunto de conjuntos relacionados de padrões como URIs), ou de forma um pouco mais abstrata: a arquitetura da Web em si. Para continuar com o exemplo acima, o HTTP "instancia" a interface uniforme do REST com uma interface especial, consistindo nos verbos HTTP. Como Fielding definiu,o estilo REST depois da Web - ou pelo menos, da maior parte dela - já estava pronto, alguém poderia argumentar quando fosse 100% relacionado. Mas em qualquer caso, a Web, o HTTP e as URIs são as únicas maiores, certamente as únicas instancias relevantes do estilo REST como um todo. E, como Roy Fielding é tanto o autor da dissertação REST, como tem sido uma forte influência sobre o design da arquitetura da Web, isso não deveria surpreender.

Finalmente, eu usei o termo "HTTP RESTFul" de vez em quando, por uma simples razão: Muitos aplicativos que usam HTTP não seguem os princípios REST - e com alguma justificativa, pode-se dizer que usar HTTP sem seguir os princípios REST é o mesmo que abusar do HTTP. É claro que isso soa um pouco exagerado - e muitas vezes, de fato, existem razões pelas quais pode-se violar uma regra do REST, simplesmente porque toda regra induz alguns trade-offs que podem não ser aceitáveis em uma situação em particular. Mas, muitas vezes, regras REST são violadas devido a uma simples falta de compreensão dos seus benefícios. Para proporcionar um exemplo perverso em particular: o uso de HTTP GET para invocar operações, como apagar um objeto viola uma regra de segurança REST e o bom senso comum (o cliente HTTP não pode ser responsabilizado, o que provavelmente não é o que o devesenvolvedor do servidor queria fazer). Mas mais sobre este e outros abusos notáveis, em um próximo artigo.

Resumo

Neste artigo, eu procurei oferecer um rápida introdução sobre os conceitos por trás do REST, a arquitetura da Web. Uma abordagem HTTP RESTful para expor funcionalidade de forma diferente do RPC, Objetos Distribuídos e Web services; isso leva algum tempo para compreensão e real entendimento da diferença. Estar ciente que os principiios do REST trazem benefícios quando você esta construindo aplicativos que apenas expõe a interface gráfica na Web ou se desejar transformar a API do seu aplicativo em um bom cidadão da Web.

Stefan Tilkov é o editor líder da comunidade de SOA da InfoQ, co-fundador, e consultor principal e lidera a RESTafarian da innoQ da Alemanha e Suiça.