Melhores práticas para uma API RESTful pragmática (parte 1)

Conheça algumas das melhores práticas para o planejamento e criação de APIs RESTful para o mundo real.

Tárcio Zemel • 12/09/2016 • 25min para ler

publica para o seu aplicativo web, mas percebe que é difícil fazer mudanças significativas à sua API, uma vez que o site/app já está em produção. Uma vez que não existe um padrão amplamente adotado para design de APIs, que funcione em todos os casos, diversas questões começam a aparecer: Quais formatos devem ser aceitos? Como deve ser a autenticação? A API deve ser versionada? etc

Nesta primeira parte, conheça algumas das **melhores práticas para uma API RESTful pragmática** — ficando claro de que não existem práticas "oficiais", mas um esforço de toda a comunidade para a adoção de práticas de APIs RESTful que, averiguadas pela experiência, produzem bons resultados.

Sumário

Eis o sumário com as primeiras dicas de melhores práticas para APIs RESTful:

Requerimentos-chave para APIs RESTful pragmáticas

Muitas das opiniões de design de API encontradas na web são discussões acadêmicas que giram em torno de interpretações subjetivas de padrões difusos, não

em alguns princípios, esforçando-se para:

- Usar padrões web (web standards) *quando fizer sentido*
- Ser amigável ao desenvolvedor e explorável através da barra de endereços do navegador
- Ser simples, intuitiva e consistente, a fim de a adoção ser não somente fácil, mas agradável
- Ser eficiente, ao mesmo tempo que mantém o balanceamento com outros requerimentos

Uma API é praticamente a UI de um desenvolvedor e, assim como qualquer UI, é importante ser pensada com cuidado para garantir a boa experiência de quem a usa.

RESTful URLs e ações

Se há algo que recebeu ampla adoção na comunidade de **desenvolvimento web**, são os princípios RESTful. Estes princípios foram introduzidos pela primeira vez por Roy Fielding no Capítulo 5 da sua dissertação sobre arquiteturas de software baseados em rede.

Os princípios fundamentais do REST envolvem separar sua API em recursos lógicos. Esses recursos são manipulados usando solicitações HTTP nas quais o método (GET, POST, PUT, PATCH ou DELETE) tem um significado específico.

Como deve ser feito um recurso

Recursos devem ser substantivos (não verbos!) que fazem sentido do ponto de vista do quem vai “consumir” a API — termo este que, convenhamos, apesar de amplamente usado, é um tanto que desprovido de sentido. Embora seus modelos internos possam mapear perfeitamente aos recursos, não é necessariamente um mapeamento um-para-um. A chave aqui é não “vazar” detalhes de implementação irrelevantes para fora da API! Alguns dos nomes de Encantamento seria bilhete, usuário e grupo. Exemplos: *ticket*, *user*, *group* etc.

- `GET /tickets` Retorna a lista de bilhetes
- `GET /tickets/12` Retorna um bilhete específico
- `POST /tickets` Cria um novo bilhete
- `PUT /tickets/12` Atualiza o bilhete #12
- `PATCH /tickets/12` Atualiza parcialmente o bilhete #12
- `DELETE /tickets/12` Apaga o bilhete #12

A grande vantagem em REST é que se está aproveitando métodos HTTP existentes para implementar funcionalidades significativas em apenas um único endpoint `/tickets`. Não existem convenções de nomenclatura a seguir e a estrutura de URL é limpa e clara.

O nome do endpoint deve ser no plural

A regra de se manter simples (*keep-it-simple*) se aplica aqui. Embora possa parecer estranho descrever uma única instância de um recurso usando o plural, a resposta pragmática é manter o formato de URL consistente e **sempre usar plural**. Não ter que lidar com pluralizações estranhas (person/people, goose/geese) faz da vida de quem vai consumir a API melhor e é mais fácil para quem a vai prover para implementação — como a maioria dos frameworks modernos, que nativamente conseguem lidar com `/tickets` e `/tickets/12` no âmbito de um controller).

Como lidar com relacionamentos

Se uma relação só pode existir dentro de outro recurso, os princípios RESTful fornecem orientações úteis. Tomando por base um cenário hipotético, um bilhete consiste em um certo número de mensagens. Estas mensagens podem ser logicamente mapeadas para o endpoint `/tickets`, como em:

- `GET /tickets/12/messages` Retorna a lista de mensagens do bilhete #12
- `GET /tickets/12/messages/5` Retorna a mensagem #5 do bilhete #12
- `POST /tickets/12/messages` Cria uma nova mensagem no bilhete #12
- `PUT /tickets/12/messages/5` Atualiza a mensagem #5 do bilhete #12

- `DELETE /tickets/12/messages/5` Apaga a mensagem #5 do bilhete #12

Alternativamente, se uma relação pode existir independentemente do recurso, faz sentido apenas incluir um identificador dentro do retorno; o consumidor da API, então, teria que acessar o endpoint da relação. No entanto, se a relação é comumente solicitada ao lado do recurso, o API poderia oferecer funcionalidade para incorporar automaticamente a representação da relação e evitar um segundo hit na API.

Ações que não se encaixam como operações CRUD

Quando há situações em que ações não necessariamente se encaixam como operações CRUD em APIs RESTful, as coisas podem ficar um pouco confusas. Existem algumas abordagens a respeito:

1. Reestruturar a ação para aparecer um campo de um recurso. Isso funciona se a ação não tem parâmetros. Por exemplo, uma ação “activate” poderia ser mapeada para um campo booleano `activated` e atualizada através de um PATCH para o recurso.
2. Tratá-lo como um sub-recurso com princípios RESTful. Por exemplo, a API do GitHub permite marcar com estrela (favoritar) com `PUT /gists/:id/star` e desmarcar a estrela (“desfavoritar”) com `DELETE /gists/:id/star`.
3. Às vezes, realmente não há uma forma de mapear a ação a uma estrutura RESTful de maneira sensata. Por exemplo, não faz sentido aplicar a um multi-recurso “search” um endpoint de um recurso específico. Neste caso, `/search` faria mais sentido, mesmo que não fosse um recurso. Está tudo bem em trabalhar dessa forma — basta fazer o que é certo do ponto de vista de quem vai consumir a API e verificar se tudo está bem documentado para evitar confusões.

SSL sempre!

Sempre use SSL. Sem exceções. Hoje, APIs web podem se acessadas a partir de qualquer lugar com uma conexão à internet (como bibliotecas, cafés, aeroportos etc) e nem todos estes são seguros. Muitos (ou a maioria) não criptografam as comunicações, dando ensejo a eavesdropping ou impersonation se as credenciais de autenticação forem acedidas indevidamente.

Um ponto a ser observado é o acesso não-SSL às URLs da API: **não os redirecione para seus homólogos SSL!** A melhor prática é lançar um *hard error* em vez disso. A última coisa que se quer é que clientes mal configurados enviem pedidos para um terminal sem criptografia apenas para serem silenciosamente redirecionados para o terminal real criptografado.

Documentação

Uma API é tão boa quanto sua documentação. Os documentos devem ser fáceis de encontrar e de acesso público. A maioria dos desenvolvedores irá verificar os documentos antes de tentar qualquer esforço de integração; quando os documentos estão escondidos dentro de um arquivo PDF ou exigem login, eles não são apenas difíceis de encontrar, mas também não são fáceis de serem pesquisados.

Os documentos devem mostrar exemplos completos de ciclos de requisição/resposta. De preferência, os pedidos devem ser exemplos "coláveis" — quer sejam links que possam ser colados em um navegador ou exemplos curl que podem ser colado em um terminal. GitHub e Stripe fazem um grande trabalho em relação a isso.

Depois de lançar uma API pública, é preciso se comprometer para não quebrar as coisas sem aviso prévio. A documentação deve incluir todos os agendamentos de descontinuação e detalhes que cercam atualizações visíveis da API externamente. As atualizações devem ser entregues através de um blog (ou seja, um *changelog*) ou uma lista de correio (de preferência, ambos).

Versionamento

Sempre versione uma API. Versionamento ajuda iterações mais rápidas e evita pedidos inválidos endpoints atualizados. Versionamento também ajuda a suavizar quaisquer grandes transições entre versões da API e é possível continuar a oferecer versões antigas durante um período de tempo.

Há opiniões mistas sobre se a versão de API deve ser incluída no URL ou em um cabeçalho (*header*). Academicamente falando, provavelmente deve estar em um header.

Um bom exemplo é a abordagem que a Stripe tomou para as versões de sua API — a URL tem o número major da versão (v1), mas a API tem sub-versões baseadas em data que podem ser escolhidas usando um header personalizado na requisição. Neste caso, a versão principal fornece estabilidade estrutural da API como um todo, enquanto as sub-versões são responsável por alterações menores (campos legados, mudanças de endpoint etc).

Uma API nunca será completamente estável; mudanças são inevitáveis. O importante é como essas mudanças são gerenciadas.

Filtragem de resultados, ordenação e busca

É aconselhável manter os URLs de recursos-base tão enxutos (*lean*) quanto possível. Filtros complexos de resultados, requisitos de ordenação e pesquisas avançadas (quando restritas a um único tipo de recurso) podem ser facilmente implementados como parâmetros de consulta a partir do do URL-base.

Filtragem

Para **filtragem**, usa-se um parâmetro de consulta exclusivo para cada campo que implementa a filtragem. Por exemplo, ao solicitar uma lista de bilhetes a partir do endpoint `/tickets`, pode-se querer limitar o resultado para apenas aqueles no estado “aberto” (open). Isto poderia ser conseguido com um pedido `GET /tickets?state=open`, no qual `state` é um parâmetro de consulta que implementa um filtro.

Ordenação

Similarmente à filtragem, um parâmetro genérico `sort` pode ser usado para descrever regras de **ordenação**, permitindo requisitos complexos de ordenação deixando este parâmetro em classificações em que cada campo é separado por vírgula, cada um com um possível unário negativo para indicar ordem descendente. Por exemplo:

- `GET /tickets?sort=-priority` Retorna uma lista de bilhetes em ordem descendente de prioridade

Busca

Às vezes filtros básicos não são suficientes e é necessário o poder de buscar textos completos (*full text search*) — talvez você já esteja usando ElasticSearch ou alguma outra tecnologia de busca baseada em Lucene. Quando pesquisa de texto completo é usado como um mecanismo de recuperação de instâncias de recursos para um tipo específico de recurso, ela pode ser exposta na API como um parâmetro de consulta no nó de extremidade do recurso. Considerando `q`, as consultas de pesquisa devem ser passados diretamente para o motor de busca e o retorno da API deve ser no mesmo formato, como resultado normal em lista.

Combinando tudo isso, é possível construir queries como:

- `GET /tickets?sort=-updated_at` Retorna bilhetes recém-atualizados
- `GET /tickets?state=closed&sort=-updated_at` Retorna bilhetes recém-fechados
- `GET /tickets?q=return&state=open&sort=-priority,created_at` Retorna bilhetes abertos de alta prioridade que contenham o termo “return”

Nomes alternativos (aliases) para queries comuns

Para tornar a experiência API mais agradável para o consumidor médio, é possível considerar “empacotar” conjuntos de condições em caminhos RESTful de fácil acesso. Por exemplo, a consulta por bilhetes recém-fechados acima poderia ser disponibilizada como `GET /tickets/recently_closed`.

Limitando quais campos são retornados pela API

O consumidor da API nem sempre precisa da representação completa de um recurso. A capacidade de selecionar e escolher campos retornados permite que o consumidor minimize o tráfego de rede e acelere sua própria utilização da API.

Por exemplo, é possível usar um parâmetro de consulta campos que leva uma lista de campos a serem incluídos separados por vírgulas, de maneira que o seguinte pedido iria

```
GET /tickets?fields=id,subject,customer_name,updated_at&state=open&sort=-updated_at
```

Retorno da representação do recurso em atualizações e criação

Uma chamada PUT, POST ou PATCH pode fazer modificações em campos do recurso subjacente que não faziam parte dos parâmetros fornecidos (por exemplo: timestamps de `created_at` ou `updated_at`). Para impedir que um consumidor da API tenha que fazer vários hits para obter uma representação atualizada, a API deve retornar a representação atualizada (ou criada) como parte da resposta.

No caso de um POST que resultou em uma criação, por exemplo, usa-se um código de status HTTP 201 e se inclui um Location header que aponta para o URL do novo recurso.

HATEOAS

Há um monte de opiniões mistas como se o consumidor da API deve criar links ou se os links devem ser fornecidas à API. Princípios de design RESTful especificam HATEOAS, que diz mais ou menos que a interação com um endpoint deve ser definida dentro de metadados que vêm com a representação de saída e não com base em informações *out-of-band*.

Embora a web geralmente funcione com base em princípios tipo HATEOAS — do tipo que se vai para a página inicial de um site e se seguem os links que ali estão –, há os que acreditam que ainda não estamos prontos para HATEOAS em APIs ainda. Ao navegar em um site, as decisões sobre quais links serão clicados são feitas em tempo de execução. No entanto, com uma API, as decisões quanto a quais pedidos serão enviados são feitas quando o código de integração da API está escrito, e não em tempo de execução. Poderiam as decisões ser adiadas para o tempo de execução? Claro, no entanto, não há muito a ganhar indo por esse caminho, já que o código ainda não seria capaz de lidar com mudanças significativas da API sem quebrar. Isto posto, fica que HATEOAS é bastante promissor, mas ainda não está pronto para o horário nobre. Mais esforços têm que ser colocado em prática para definir padrões e ferramentas em torno destes princípios para que seu potencial seja plenamente realizado.

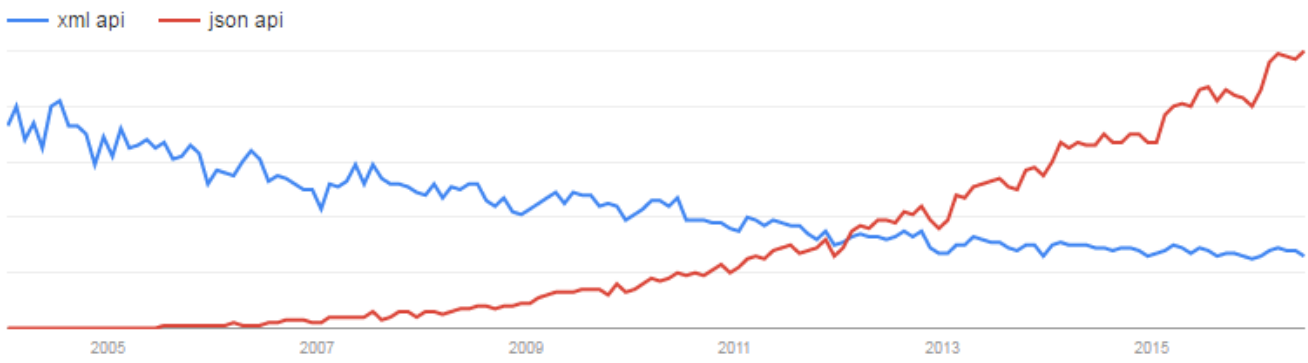
através da rede serem minimizados e os dados armazenados pelos consumidores da API também serem minimizados (como eles estão armazenando pequenos identificadores, ao contrário de URLs que contêm identificadores).

Além disso, dado que entre as dicas deste artigo constam a defesa de usar números de versão no URL, não faz sentido para o consumidor da API a longo prazo armazenar identificadores de recursos ao invés de URLs. Afinal de contas, o identificador é estável em todas as versões, mas o URL que representa não é.

Respostas somente em JSON

Já passou da hora de deixar XML para trás em APIs. XML é verboso, difícil de analisar, difícil de ler, seu modelo de dados não é compatível com os modelos de dados da maioria das linguagens de programação línguas modelo e suas vantagens de extensibilidade são irrelevantes quando as necessidades primárias de sua representação de saída são uma serialização de uma representação interna.

Para reforçar, eis um gráfico do Google Trends num comparativo entre os termos “xml api” e “json api” :



No entanto, para casos em que a base de clientes seja composta por um grande número de clientes corporativos, talvez seja preciso suportar XML de qualquer maneira. Se for o caso, uma nova pergunta se apresenta: o media type deve mudar baseado em Accept headers ou com base em URLs? Para assegurar a explorabilidade através de navegadores, **deve ser com base em URLs**. A opção mais sensata, aqui, seria acrescentar uma extensão “.json” ou “.xml” no URL do endpoint — ou tratar JSON como default e usar “.xml” quando se quer retorno em XML.

JavaScript Object Notation “.

snake_case x camelCase para nomes de campos

Quando se está usando usando JSON como o formato de representação primário, a coisa “certa” a fazer é seguir convenções de nomenclatura de JavaScript, e isso significa **camelCase para nomes de campo**. Se, em seguida, surgir a necessidade da construção de bibliotecas de clientes em várias línguas, é melhor usar convenções de nomenclatura idiomáticas — camelCase para C# e Java, snake_case para Python e Ruby etc.

Como curiosidade, estudos apontam que snake_case é 20% mais fácil de ler do que camelCase — o que também provoca impacto na legibilidade e explorabilidade da API na documentação.

Fim da parte 1 sobre melhores práticas para APIs RESTful pragmáticas

O desenvolvimento/manutenção de APIs RESTful para o mundo real não é tarefa fácil. Entretanto, é possível que todo o processo fique mais fácil ao seguir determinadas convenções e boas práticas. Como já citado algumas vezes, tratam-se de convenções; portanto, não é preciso ficar muito “engessado” a elas — apesar de estas terem sido colocadas depois de muito estudo e análises. Pense da seguinte maneira: são indicações muito bem fundamentadas, a fim de indicar o melhor caminha para a implementação de APIs RESTful.

Na parte 2 , confira o restante das dicas para APIs RESTful pragmáticas. Fique ligado no **desenvolvimento para web**!