Na primeira parte sobre melhores práticas para uma API RESTful pragmática, começou-se a ver pontos imprescindíveis para a construção de APIs eficientes no mundo real. Atendendo a pedidos e continuando esta tão aclamada série, confira a parte 2 sobre as melhores práticas para se desenvolver APIs!

> Este artigo é baseado em Best Practices for Designing a Pragmatic RESTful API, do site do Vinay Sahni.

Sumário

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

Usar “pretty print” por padrão e garantir que gzip é suportado

Uma API que fornece saída compactada em espaço em branco (*white-space compressed output*) não é muito divertida de se olhar a partir de um navegador. Embora algum tipo de parâmetro de consulta (como `?pretty=true`) possa ser fornecido para permitir **pretty print** (ou "impressão bonita"), uma API que a imprime por padrão é muito mais acessível. O custo da transferência de dados extra é desprezível, especialmente quando comparado ao custo de não implementar gzip.

Considerem-se alguns casos de uso: se um consumidor de API estiver depurando e imprimindo dados que recebeu da API, estes serão legível por padrão, ou; se o consumidor pegou o URL que seu código estava gerando e o acessou diretamente do navegador, este será legível por padrão. Estas são apenas pequenas coisas e pequenas coisas é que tornam uma API agradável de se usar!

E sobre os dados extras transferidos?

Como exercício "do mundo real", é possível puxar alguns dados da API do GitHub — que usa pretty print por padrão. Fazendo algumas comparações de gzip:

```
$ curl https://api.github.com/users/webfatorial > with-whitespace.txt
$ ruby -r json -e 'puts JSON JSON.parse(STDIN.read)' < with-whitespace.txt
$ gzip -c with-whitespace.txt > with-whitespace.txt.gz
$ gzip -c without-whitespace.txt > without-whitespace.txt.gz
```

Os arquivos gerados têm o seguintes tamanhos:

- `without-whitespace.txt`: 1252 bytes
- `with-whitespace.txt`: 1369 bytes
- `without-whitespace.txt.gz`: 496 bytes
- `with-whitespace.txt.gz`: 509 bytes

Neste exemplo, o espaço em branco aumentou o tamanho de saída em 8,5% quando o gzip não está em reprodução e 2,6% quando o gzip está em jogo. Por outro lado, o ato de "gzipar", em si, forneceu mais de 60% em poupança de largura de banda. Uma vez que o custo da pretty print é relativamente pequeno, é melhor imprimir assim por padrão e garantir que a compressão gzip é suportada.

O pessoal do Twitter descobriu que houve uma economia de 80% em alguns casos ao ativar a compactação gzip em sua Streaming API. Stack Exchange foi além, aconselhando a jamais retornar uma resposta que não seja comprimida!

Não usar "envelope" por padrão, mas tornar isso possível quando necessário

Muitas APIs envolvem/envelopam (*wrap*) suas respostas assim:

```
{
  "data" : {
    "id" : 123,
    "name" : "John"
```

```
  }
}
```

Há algumas justificativas para fazer isso, como facilitar a inclusão de informações adicionais sobre metadados ou paginação, já que alguns clientes REST que não permitem acesso fácil a cabeçalhos HTTP e solicitações JSONP não têm acesso a cabeçalhos HTTP. No entanto, com os padrões que estão sendo rapidamente adotados, como CORS e o Link header from RFC 5988, fazer esse wrapper está começando a se tornar desnecessário — por curiosidade, também dê uma olhada no RESTed NARWHL.

Quer dizer, é possível tornar uma API "*future proof*" ao não envelopar as respostas por padrão, mas deixando isso acontecer em alguns casos excepcionais.

Como deve ser utilizado o envelopamento em casos excepcionais

Existem 2 situações em que envelopar a resposta da API é realmente necessário: se a API precisa suportar requisições de domínio cruzado sobre JSONP ou se o cliente é incapaz de trabalhar com cabeçalhos HTTP.

As solicitações JSONP vêm com um parâmetro de consulta adicional (geralmente chamado `callback` ou `jsonp`) que representa o nome da função de callback. Se esse parâmetro estiver presente, a API deve alternar para um modo de envelopamento completo, no qual ela sempre responde com um código de status HTTP 200 e passa o código de status real no payload JSON.

Todos os cabeçalhos HTTP adicionais que teriam sido passados ao lado da resposta devem ser mapeados para campos JSON, da seguinte forma:

```
callback_function({
  status_code: 200,
  next_page: "https://..",
  response: {
    ... corpo da resposta JSON ...
  }
})
```

Da mesma forma, para dar suporte a clientes HTTP limitados, permitir um parâmetro especial de consulta `?envelope=true` para responder com envelopagem completa (sem a função de callback JSONP).

POST codificado em JSON, corpo de PUT e PATCH

Segundo os conselhos deste post e do anterior, o uso de JSON para toda saída de APIs é aconselhável. Vamos considerar, também, usar JSON para entradas (*inputs*) na API.

Muitas APIs usam codificação de URL (URL encoding) em suas requisições. A codificação de URL é exatamente o que parece: solicitar respostas nas quais os pares de valores-chave são codificados usando as mesmas convenções usadas para codificar dados em parâmetros de consulta de URL. É simples, amplamente suportado e faz o trabalho que tem que fazer.

Entretanto, a codificação URL tem algumas questões que a tornam problemática. Para começar, não existe o conceito de tipos de dados (*data types*). Isso força a API a analisar (*parse*) inteiros e booleanos fora de seqüências de strings. Além disso, não há um conceito real de estrutura hierárquica. Embora existam algumas convenções que podem construir alguma estrutura de pares de valor de chave (como ao usar `[]` a uma chave para representar um array), isso não é comparação com a estrutura hierárquica nativa do JSON.

Se a API for simples, a codificação de URL pode ser suficiente. No entanto, APIs complexas devem preferir usar JSON para o input. De qualquer forma, escolha um e seja consistente em toda a API — uma API que aceite solicitações POST, PUT e PATCH codificadas em JSON também deve exigir que o cabeçalho `Content-Type` seja definido como `application/json` ou mande um código HTTP `415 Unsupported Media Type`.

Paginação

APIs envelopadas geralmente incluem dados de paginação no próprio envelope… E não dá para culpar ninguém, já que, até recentemente, não havia muitas opções melhores. A maneira correta de incluir detalhes de paginação hoje em dia é usando o cabeçalho Link da RFC 5988.

Uma API que usa o cabeçalho Link pode retornar um conjunto de links prontos para que o consumidor da API não tenha que construir links próprios. Isso é especialmente importante quando a paginação é baseada em cursor. Eis um

exemplo de um cabeçalho Link usado corretamente, tirado da documentação do GitHub:

```
Link: <https://api.github.com/user/repos?page=3&per_page=100>; rel="next",
```

Mas esta não é uma solução completa, pois muitas APIs gostam de retornar informações de paginação adicionais, como uma contagem do número total de resultados disponíveis. Uma API que requer o envio de uma contagem pode usar um cabeçalho HTTP personalizado, como `X-Total-Count`.

Auto carregamento de representações de recurso relacionados

Há muitos casos em que um consumidor de API precisa carregar dados relacionados ao (ou referenciado pelo) recurso solicitado. Ao invés de exigir que o consumidor dê vários hits na API para pegar essa informação, haveria um ganho de eficiência significativo ao permitir que dados relacionados fossem retornados e carregados sob demanda ao lado do recurso original.

No entanto, como isso vai contra alguns princípios RESTful, é possível minimizar esse “desvio” fazendo isso com base em um parâmetro `embed` (ou `expand`).

Neste caso, `embed` seria uma lista separada por vírgulas de campos a serem incorporados — sendo possível também usar uma notação de ponto para se referir a sub-campos. Por exemplo:

```
GET /tickets/12?embed=customer.name,assigned_user
```

Que poderia retornar algo como:

```
{
  "id" : 12,
  "subject" : "Tenho uma dúvida!",
  "summary" : "Lorem ipsum...",
  "customer" : {
    "name" : "Bob"
  },
  assigned_user: {
   "id" : 42,
   "name" : "Jim",
  }
}
```

Naturalmente, a capacidade de implementar algo como isso realmente depende da complexidade interna e requisitos de negócio da API. Esse tipo de incorporação pode facilmente resultar em um problema de seleção N+1.

Substituição do método HTTP

Alguns clientes HTTP só podem trabalhar com requisições GET e POST simples. Para aumentar a acessibilidade a esses clientes limitados, a API precisa de uma forma de substituir o método HTTP. Embora não haja nenhum padrão rígido aqui, a convenção popular é aceitar um cabeçalho de solicitação `X-HTTP-Method-Override` com um valor de string contendo um PUT, PATCH ou DELETE.

> ⚠ O cabeçalho de substituição só deve ser aceito em solicitações POST. Requisições GET nunca devem alterar os dados no servidor!

Limitação de taxas

Para evitar abusos, é prática padrão adicionar algum tipo de limitação de taxas a uma API. O RFC 6585 introduziu um código de status HTTP 429 Too Many Requests para lidar com esse tipo de situação.

Contudo, pode ser muito útil notificar o consumidor de seus limites *antes*que eles realmente sejam atingidos. Esta é uma área atualmente não regida por padrões, mas há um número de convenções populares que usam cabeçalhos de resposta HTTP.

No mínimo, inclua os seguintes cabeçalhos (usando as convenções de nomenclatura do Twitter como cabeçalhos):

- `X-Rate-Limit-Limit`: número de solicitações permitidas no período atual
- `X-Rate-Limit-Remaining`: número de pedidos restantes no período atual

- `X-Rate-Limit-Reset`: número de segundos restantes no período atual

Por que usar número de segundos restantes ao invés de um timestamp para X-Rate-Limit-Reset?

Um timestamp contém todos os tipos de informações úteis, mas desnecessárias, como a data e, possivelmente, o fuso horário. Um consumidor de API realmente só quer saber quando ele pode enviar o pedido novamente e o número de segundos responde a esta pergunta com processamento adicional mínimo — e também evita problemas relacionados à clock skew.

> ⚠ Algumas APIs usam um timestamp UNIX (segundos desde epoch) para `X-Rate-Limit-Reset`. **Não faça isso!**

Autenticação

Uma API RESTful deve ser stateless (“sem estado”). Isso significa que a autenticação de solicitação não deve depender de cookies ou sessões; em vez disso, cada solicitação deve vir com algumas credenciais de autenticação.

Ao sempre usar SSL, as credenciais de autenticação podem ser simplificadas para um token de acesso gerado aleatoriamente que é entregue no campo de nome de usuário de HTTP Basic Auth. A grande vantagem é que é completamente explorável pelo navegador — o navegador apenas abre um prompt pedindo credenciais se ele recebe um código `401 Unauthorized` do servidor.

Contudo, este método de autenticação token-ao-invés-de-autenticação-básica só é aceitável nos casos em que é prático que o usuário copie um token de uma interface de administração para o ambiente de consumidor da API. Nos casos em que isso não é possível, OAuth 2 deve ser usado para fornecer transferência segura de token para terceiros. OAuth 2 usa Bearer tokens e também depende de SSL para sua criptografia de transporte subjacente.

Uma API que precisa oferecer suporte a JSONP precisará de um terceiro método de autenticação, uma vez que as solicitações JSONP não podem enviar credenciais HTTP Basic Auth ou Bearer tokens. Neste caso, um parâmetro de consulta especial `access_token` pode ser usado.

> *i* Existe um problema de segurança inerente ao utilizar um parâmetro de consulta para o token, uma vez que a maioria dos servidores web armazena parâmetros de consulta nos registos do servidor (server logs).

Todos os três métodos acima são apenas maneiras de transportar o token através dos limites da API. O próprio token subjacente, em si, poderia ser idêntico.

Caching

Muitos não sabem disso, mas o HTTP fornece uma estrutura de cache integrada! Tudo o que é preciso fazer é incluir alguns cabeçalhos de resposta de saída adicionais e fazer uma pequena validação quando receber alguns

cabeçalhos de solicitação de entrada. Existem 2 abordagens: ETag e Last-Modified.

ETag

Ao gerar uma solicitação, inclua um cabeçalho HTTP ETag contendo um hash ou checksum da representação. Esse valor deve mudar sempre que a representação de saída muda. Agora, se uma solicitação HTTP de entrada contém um cabeçalho `If-None-Match` com um valor ETag correspondente, a API deve retornar um código `304 Not Modified` em vez da representação de saída do recurso.

Last-Modified

Basicamente, funciona como ETag, exceto que há uso de timestamps. O cabeçalho de resposta `Last-Modified` contém um timestamp no formato RFC 1123 que é validado contra `If-Modified-Since`. A especificação HTTP teve 3 diferentes formatos de data aceitável e o servidor deve estar preparado para aceitar qualquer um deles.

Erros

Assim como uma página de erro HTML mostra uma mensagem de erro útil para um visitante, uma API deve fornecer uma mensagem de erro útil em formato consumível. A representação de um erro não deve ser diferente da representação de qualquer recurso — apenas terá seu próprio conjunto de campos, evidentemente.

A API deve sempre retornar códigos de status HTTP sensíveis. Erros de API tipicamente se dividem em 2 tipos: códigos de status da série 400 para problemas de cliente e códigos de status da série 500 para problemas de servidor. No mínimo, a API deve padronizar que todos os erros da série 400 vêm com representação de erro JSON consumível. Se possível — por exemplo, se os balanceadores de carga e os proxies reversos puderem criar corpos de erros personalizados –, isso deve se estender aos códigos da série 500.

Um corpo de erro JSON deve fornecer algumas coisas para o desenvolvedor: uma mensagem de erro útil, um código de erro exclusivo (que pode ser procurado nos documentos para se obter mais detalhes) e, possivelmente, uma descrição detalhada. Uma saída JSON para algo parecido com isso seria:

```
{
  "code" : 1234,
  "message" : "Algo de ruim aconteceu :(",
  "description" : "Mais detalhes sobre o erro"
}
```

Os erros de validação para solicitações PUT, PATCH e POST precisarão de um desdobramento de campo. Isso é melhor modelado usando um código de erro de nível superior fixo para falhas de validação e fornecendo os erros detalhados em um campo de erros adicionais, assim:

```
{
  "code" : 1024,
  "message" : "Validação falhou",
  "errors" : [
```

```
    {
      "code" : 5432,
      "field" : "first_name",
      "message" : "Nome não pode conter caracteres especiais"
    },
    {
       "code" : 5622,
       "field" : "password",
       "message" : "Senha não pode estar vazia"
    }
  ]
}
```

Códigos de status HTTP

O HTTP define um grupo de códigos de status significativos que podem ser retornados em APIs. Estes podem ser usados para ajudar os consumidores da API a encaminhar as suas respostas em conformidade. Aqui está uma pequena lista de códigos HTTP que uma API definitivamente deve usar (no mínimo):

- `200 OK`: Resposta a um bem-sucedido GET, PUT, PATCH ou DELETE. Também pode ser usado para um POST que não resulte em uma criação.
- `201 Created`: Resposta a um POST que resulta em uma criação. Deve ser combinado com um Location header que aponta para a localização do novo recurso.
- `204 No Content`: Resposta a um pedido bem-sucedido que não retornará um corpo (como uma solicitação DELETE)
- `304 Not Modified`: Usado quando cabeçalhos de cache HTTP estão em jogo
- `400 Bad Request`: O pedido é malformado; não foi possível fazer o parse do corpo
- `401 Unauthorized`: Quando não são fornecidos detalhes de autenticação ou estes são inválidos. Também é útil para disparar um popup de autenticação se a API for usada a partir de um navegador.

- `403 Forbidden`: Quando a autenticação foi bem-sucedida, mas o usuário autenticado não tem acesso ao recurso.
- `404 Not Found`: Quando um recurso inexistente é solicitado.
- `405 Method Not Allowed`: Quando um método HTTP que não é permitido para o usuário autenticado está sendo solicitado.
- `410 Gone`: Indica que o recurso nesse endpoint não está mais disponível. Útil como resposta geral para versões antigas da API.
- `415 Unsupported Media Type`: Se o content type incorreto foi fornecido como parte do pedido.
- `422 Unprocessable Entity`: Usado para validação de erros.
- `429 Too Many Requests`: Quando um pedido é rejeitado devido à limitação da taxa.

Conclusão sobre melhores práticas para APIs RESTful pragmáticas

Como pode ser visto na parte 1 e nesta segunda parte sobre o assunto de melhores práticas para APIs RESTful pragmáticas, existem muitas recomendações, técnicas, normas e convenções da comunidade para garantir o desenvolvimento de APIs eficientes no mundo real.

Não raramente não será possível cumprir todas as boas recomendações apresentadas — estamos falando do “mundo real”, afinal de contas –, mas é importante se esforçar ao máximo para que o que foi apresentado seja implementado o máximo possível, garantindo que quem vá “consumir” a API tenha as melhores respostas, dentro de uma gama de acessibilidade indispensável quando o assunto são **APIs RESTful profissionais**.

Uma API é como uma interface para desenvolvedores. Sempre que possível, coloque o esforço necessário para garantir que sua API é não apenas funcional, mas **agradável** de ser usada!