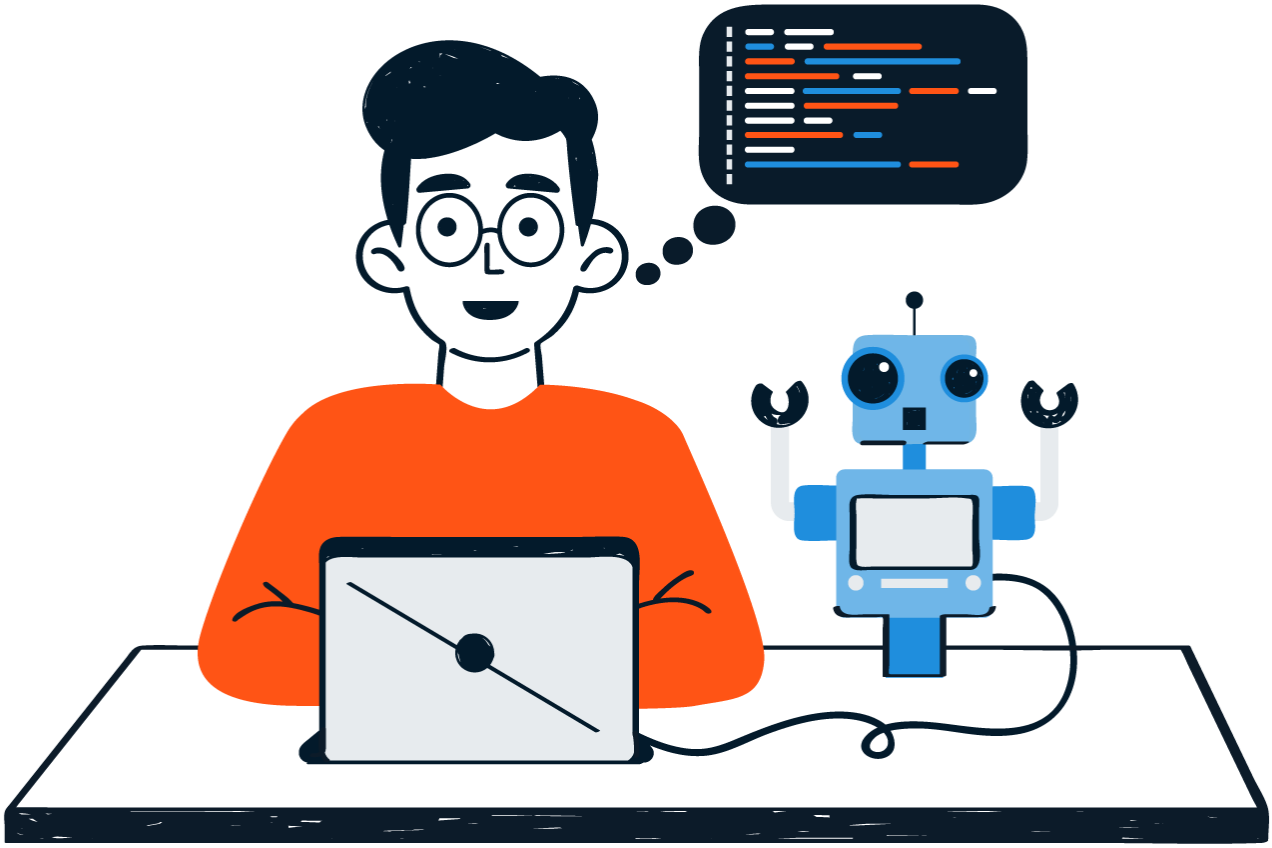
Práticas recomendadas para projetar uma API RESTful pragmática



Seu modelo de dados começou a se estabilizar e você pode criar uma API pública para seu aplicativo da web. Você percebe que é difícil fazer alterações significativas em sua API depois que ela é lançada e deseja acertar o máximo possível desde o início. Agora, a internet não tem escassez de opiniões sobre design de API. Mas, como não há um padrão amplamente adotado que funcione em todos os casos, você fica com várias opções: quais formatos você deve aceitar? Como você deve autenticar? Sua API deve ser versionada?

Ao projetar uma API para o Enchant (uma alternativa ao Zendesk), tentei encontrar respostas pragmáticas para essas perguntas. Meu objetivo é que a API Enchant seja fácil de usar, fácil de adotar e flexível o suficiente para dogfood para nossas próprias interfaces de usuário.

TL;DR

- Uma API é uma interface de usuário para um desenvolvedor - então se esforce para torná-la agradável
- Use URLs e ações RESTful
- Use SSL em todos os lugares, sem exceções
- Uma API é tão boa quanto sua documentação - portanto, tenha uma ótima documentação
- Versão via URL, não via cabeçalhos
- Use parâmetros de consulta para filtragem avançada, classificação e pesquisa
- Fornecer uma maneira de limitar quais campos são retornados da API
- Retornar algo útil de solicitações POST, PATCH e PUT
- HATEOAS ainda não é prático
- Use JSON sempre que possível, XML somente se for necessário
- Você deve usar camelCase com JSON, mas snake_case é 20% mais fácil de ler
- Impressão bonita por padrão e certifique-se de que o gzip seja suportado
- Não use envelopes de resposta por padrão
- Considere usar JSON para corpos de solicitação POST, PUT e PATCH
- Paginar usando cabeçalhos de link
- Fornecer uma maneira de carregar automaticamente representações de recursos relacionadas
- Fornecer uma maneira de substituir o método HTTP
- Forneça cabeçalhos de resposta úteis para limitação de taxa
- Use autenticação baseada em token, transportada por OAuth2 onde a delegação é necessária
- Incluir cabeçalhos de resposta que facilitam o armazenamento em cache

- Definir uma carga útil de erro consumível
- Use efetivamente os códigos de status HTTP

... ou simplesmente pule para o final e **inscreva-se para atualizações**

Principais requisitos para a API

Muitas das opiniões de design de API encontradas na web são discussões acadêmicas que giram em torno de interpretações subjetivas de padrões difusos em oposição ao que faz sentido no mundo real. Meu objetivo com este post é descrever as melhores práticas para uma API pragmática projetada para os aplicativos da web atuais. Não faço nenhuma tentativa de satisfazer um padrão se ele não parecer certo. Para ajudar a orientar o processo de tomada de decisão, escrevi alguns requisitos pelos quais a API deve se esforçar:

- Deve usar padrões da web onde eles *fazem sentido*
- Deve ser amigável para o desenvolvedor e ser explorável por meio de uma barra de endereço do navegador
- Deve ser simples, intuitivo e consistente para tornar a adoção não apenas fácil, mas agradável
- Ele deve fornecer flexibilidade suficiente para alimentar a maioria da interface do usuário do Enchant
- Deve ser eficiente, mantendo o equilíbrio com os outros requisitos

Uma API é a interface do usuário do desenvolvedor - assim como qualquer interface do usuário, é importante garantir que a experiência do usuário seja pensada com cuidado!

Use URLs e ações RESTful

Se há uma coisa que ganhou ampla adoção, são os princípios RESTful. Estes foram introduzidos pela primeira vez por Roy Fielding no Capítulo 5 de sua dissertação sobre arquiteturas de software baseadas em rede .

Os princípios-chave do REST envolvem separar sua API em recursos lógicos. Esses recursos são manipulados por meio de solicitações HTTP onde o método (GET, POST, PUT, PATCH, DELETE) possui significado específico.

Mas o que posso fazer um recurso? Bem, esses devem ser substantivos que façam sentido do ponto de vista do consumidor da API, não verbos. Para ser claro: um substantivo é uma coisa, um verbo é o que você faz com ele. Alguns dos substantivos de Enchant seriam *ticket* , *user* e *customer* .

No entanto, tenha cuidado : embora seus modelos internos possam ser mapeados perfeitamente para recursos, não é necessariamente um mapeamento de um para um. A chave aqui é não vazar detalhes de implementação irrelevantes para sua API! Seus recursos de API precisam fazer sentido do ponto de vista do consumidor de API.

Depois de definir seus recursos, você precisa identificar quais ações se aplicam a eles e como elas seriam mapeadas para sua API. Os princípios RESTful fornecem estratégias para lidar com ações CRUD usando métodos HTTP mapeados da seguinte forma:

- `GET /tickets` - Recupera uma lista de tickets
- `GET /tickets/12` - Recupera um ticket específico
- `POST /tickets` - Cria um novo ticket
- `PUT /tickets/12` - Atualiza o ticket #12
- `PATCH /tickets/12` - Atualiza parcialmente o ticket #12
- `DELETE /tickets/12` - Exclui o ticket #12

O melhor do REST é que você está aproveitando os métodos HTTP existentes para implementar funcionalidades significativas em apenas um único ponto de extremidade `/tickets .` Não há convenções de nomenclatura de métodos a serem seguidas e a estrutura de URL é limpa e clara. *DESCANSE FTW!*

O nome do endpoint deve ser singular ou plural? A regra de manter a simplicidade se aplica aqui. Embora seu gramático interno lhe diga que é errado descrever uma única instância de um recurso usando um plural, a resposta pragmática é manter o formato de URL consistente e sempre usar um plural. Não ter que lidar com pluralização estranha (pessoa/pessoa, ganso/ganso) torna a vida do consumidor de API melhor e é mais fácil para o provedor de API implementar (já que a maioria dos frameworks modernos lidará nativamente com `/tickets` e `/tickets/12` sob um controlador comum).

Mas como você lida com as relações? Se uma relação só pode existir dentro de outro recurso, os princípios RESTful fornecem orientação útil. Vejamos isso

com um exemplo. Um ticket no Enchant consiste em várias mensagens. Essas mensagens podem ser mapeadas logicamente para o terminal `/tickets da seguinte forma:`

- `GET /tickets/12/messages` - Recupera a lista de mensagens para o ticket #12
- `GET /tickets/12/messages/5` - Recupera a mensagem #5 para o ticket #12
- `POST /tickets/12/messages` - Cria uma nova mensagem no ticket #12
- `PUT /tickets/12/messages/5` - Atualiza a mensagem #5 para o ticket #12
- `PATCH /tickets/12/messages/5` - Atualiza parcialmente a mensagem #5 para o ticket #12
- `DELETE /tickets/12/messages/5` - Exclui a mensagem #5 do ticket #12

Alternativa 1 : Se uma relação pode existir independentemente do recurso, faz sentido incluir apenas um identificador para ela na representação de saída do recurso. O consumidor da API teria então que atingir o ponto final da relação.

Alternativa 2 : Se uma relação existente independentemente é comumente solicitada ao lado do recurso, a API pode oferecer funcionalidade para incorporar automaticamente a representação da relação e evitar o segundo hit na API. API limpa e um hit para o servidor. Eu gosto dessa abordagem.

E as ações que não se encaixam no mundo das operações CRUD?

É aqui que as coisas podem ficar confusas. Existem várias abordagens:

1. Reestruturar a ação para aparecer como um campo de um recurso. Isso funciona se a ação não receber parâmetros. Por exemplo, uma ação de *ativação* pode ser mapeada para um campo booleano `ativado` e atualizada por meio de um PATCH para o recurso.
2. Trate-o como um sub-recurso com princípios RESTful. Por exemplo, a API do GitHub permite marcar uma essência com `PUT /gists/:id/star` e desmarcar com `DELETE /gists/:id/ star` .
3. Às vezes você realmente não tem como mapear a ação para uma estrutura RESTful sensata. Por exemplo, uma pesquisa de vários recursos realmente não faz sentido para ser aplicada a um ponto de extremidade de um recurso específico. Nesse caso, `/search` faria mais sentido mesmo não sendo um recurso. Tudo bem - apenas

faça o que é certo da perspectiva do consumidor da API e certifique-se de que está documentado claramente para evitar confusão.

SSL em todos os lugares - o tempo todo

Sempre use SSL. Sem exceções. Hoje, suas APIs web podem ser acessadas de qualquer lugar onde haja internet (como bibliotecas, cafeterias, aeroportos entre outros). Nem todos são seguros. Muitos não criptografam as comunicações, permitindo uma fácil espionagem ou representação se as credenciais de autenticação forem invadidas.

Outra vantagem de sempre usar SSL é que as comunicações criptografadas garantidas simplificam os esforços de autenticação - você pode se safar com tokens de acesso simples em vez de ter que assinar cada solicitação de API.

Uma coisa a ser observada é o acesso não SSL aos URLs da API. Não os **redirecione** para suas contrapartes SSL. Jogue um erro difícil em vez disso! Quando um redirecionamento automático está em vigor, um cliente mal configurado pode vazar parâmetros de solicitação inadvertidamente sobre o ponto de extremidade não criptografado. Um erro grave garante que esse erro seja detectado antecipadamente e que o cliente seja configurado corretamente.

Documentação

Uma API é tão boa quanto sua documentação. Os documentos devem ser fáceis de encontrar e acessíveis publicamente. A maioria dos desenvolvedores verificará os documentos antes de tentar qualquer esforço de integração. Quando os documentos estão ocultos em um arquivo PDF ou exigem login, eles não são apenas difíceis de encontrar, mas também não são fáceis de pesquisar.

Os documentos devem mostrar exemplos de ciclos completos de solicitação/resposta. De preferência, as solicitações devem ser exemplos que podem ser colados - links que podem ser colados em um navegador ou exemplos

de curl que podem ser colados em um terminal. GitHub e Stripe fazem um ótimo trabalho com isso.

Depois de liberar uma API pública, você se comprometeu a não quebrar as coisas sem aviso prévio. A documentação deve incluir quaisquer cronogramas de descontinuação e detalhes sobre atualizações de API visíveis externamente. As atualizações devem ser entregues através de um blog (ou seja, um changelog) ou uma lista de discussão (de preferência ambos!).

Controle de versão

Sempre versione sua API. O controle de versão ajuda você a iterar mais rapidamente e evita que solicitações inválidas atinjam endpoints atualizados. Também ajuda a suavizar as principais transições de versão da API, pois você pode continuar oferecendo versões antigas da API por um período de tempo.

Existem opiniões divergentes sobre se uma versão da API deve ser incluída na URL ou em um cabeçalho . Academicamente falando, provavelmente deveria estar em um cabeçalho. No entanto, a versão precisa estar na URL para garantir a exploração dos recursos do navegador entre as versões (lembra-se dos requisitos da API especificados no início desta postagem?) e para ter uma experiência de desenvolvedor mais simples.

Sou um grande fã da abordagem que o Stripe adotou para o versionamento da API - a URL tem um número de versão principal (v1), mas a API tem subversões baseadas em data que podem ser escolhidas usando um cabeçalho de solicitação HTTP personalizado. Nesse caso, a versão principal fornece estabilidade estrutural da API como um todo, enquanto as subversões são responsáveis por alterações menores (suspensões de campo, alterações de endpoint etc.).

Uma API nunca será completamente estável. A mudança é inevitável. O importante é como essa mudança é gerenciada. Programações de

descontinuação de vários meses bem documentadas e anunciadas podem ser uma prática aceitável para muitas APIs. Tudo se resume ao que é razoável, considerando o setor e os possíveis consumidores da API.

Filtragem, classificação e pesquisa de resultados

É melhor manter os URLs de recursos básicos o mais simples possível. Filtros de resultados complexos, requisitos de classificação e pesquisa avançada (quando restritos a um único tipo de recurso) podem ser facilmente implementados como parâmetros de consulta no topo da URL base. Vejamos estes com mais detalhes:

Filtragem : Use um parâmetro de consulta exclusivo para cada campo que implementa a filtragem. Por exemplo, ao solicitar uma lista de tickets do ponto de extremidade `/tickets` , você pode querer limitá-los apenas aos que estão no estado aberto. Isso pode ser feito com uma solicitação como `GET /tickets?state=open` . Aqui, `o estado` é um parâmetro de consulta que implementa um filtro.

Classificação : Semelhante à filtragem, uma `classificação` de parâmetro genérico pode ser usada para descrever regras de classificação. Acomode requisitos de classificação complexos, permitindo que o parâmetro de classificação inclua uma lista de campos separados por vírgulas, cada um com um possível negativo unário para implicar uma ordem de classificação decrescente. Vejamos alguns exemplos:

- `GET /tickets?sort=-priority` - Recupera uma lista de tickets em ordem decrescente de prioridade
- `GET /tickets?sort=-priority,created_at` - Recupera uma lista de tickets em ordem decrescente de prioridade. Dentro de uma prioridade específica, os bilhetes mais antigos são pedidos primeiro

Pesquisa : Às vezes, os filtros básicos não são suficientes e você precisa do poder da pesquisa de texto completo. Talvez você já esteja usando o ElasticSearch ou outra tecnologia de pesquisa baseada em Lucene . Quando a pesquisa de texto completo é usada 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 endpoint do recurso. Digamos `q` . As consultas de pesquisa devem ser passadas diretamente para o mecanismo de pesquisa e a saída da API deve estar no mesmo formato de um resultado de lista normal.

Combinando-os, podemos criar consultas como:

- `GET /tickets?sort=-updated_at` - Recuperar ingressos atualizados recentemente
- `GET /tickets?state=closed&sort=-updated_at` - Recuperar tickets fechados recentemente
- `GET /tickets?q=return&state=open&sort=-priority,created_at` - Recupera os tickets abertos de prioridade mais alta que mencionam a palavra 'return'

Alias para consultas comuns

Para tornar a experiência da API mais agradável para o consumidor médio, considere empacotar conjuntos de condições em caminhos RESTful facilmente acessíveis. Por exemplo, a consulta de tickets recentemente fechados acima pode ser empacotada 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 da API minimize o tráfego de rede e acelere seu próprio uso da API.

Use um parâmetro de consulta de `campos` que usa uma lista de campos separados por vírgulas para incluir. Por exemplo, a solicitação a seguir recuperaria informações suficientes para exibir uma lista ordenada de tickets abertos:

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

Nota : Esta abordagem também pode ser combinada com o carregamento automático de recursos relacionados :

```
GET /tickets?embed=customer&fields=id,customer.id,customer.name
```

Atualizações e criação devem retornar uma representação de recurso

Uma chamada PUT, POST ou PATCH pode fazer modificações nos campos do recurso subjacente que não faziam parte dos parâmetros fornecidos (por exemplo: created_at ou updated_at timestamps). Para evitar que um consumidor de API precise acessar a API novamente para obter uma representação atualizada, faça com que a API retorne a representação atualizada (ou criada) como parte da resposta.

No caso de um POST que resultou em uma criação, use um código de status HTTP 201 e inclua um cabeçalho Location que aponte para a URL do novo recurso. Ambos devem incluir a representação de recurso recém-criada como o corpo da resposta.

Você deve HATEOAS?

Há muitas opiniões divergentes sobre se o consumidor da API deve criar links ou se os links devem ser fornecidos à API. Os princípios de design RESTful especificam o HATEOAS , que afirma aproximadamente que a interação com um endpoint deve ser definida dentro de metadados que acompanham a representação de saída e não com base em informações fora da banda.

Embora a web geralmente funcione nos princípios do tipo HATEOAS (onde vamos para a página inicial de um site e seguimos os links com base no que vemos na página), acho que ainda não estamos prontos para HATEOAS em APIs. 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 sobre quais solicitações serão enviadas são feitas quando o código de integração da API é

escrito, não em tempo de execução. As decisões poderiam ser adiadas para o tempo de execução? Claro, no entanto, não há muito a ganhar nesse caminho, pois o código ainda não seria capaz de lidar com alterações significativas da API sem quebrar. Dito isso, acho que o HATEOAS é promissor, mas ainda não está pronto para o horário nobre. Um pouco mais de esforço deve ser feito para definir padrões e ferramentas em torno desses princípios para que seu potencial seja plenamente realizado.

Por enquanto, é melhor supor que o usuário tenha acesso à documentação e inclua identificadores de recursos na representação de saída que o consumidor da API usará ao criar links. Existem algumas vantagens em manter identificadores - os dados que fluem pela rede são minimizados e os dados armazenados pelos consumidores da API também são minimizados (já que eles armazenam pequenos identificadores em oposição a URLs que contêm identificadores).

Além disso, dado que este post defende números de versão na URL, faz mais sentido a longo prazo para o consumidor da API armazenar identificadores de recursos em vez de URLs. Afinal, o identificador é estável entre as versões, mas a URL que o representa não é!

Respostas somente JSON

XML não é uma ótima opção para uma API. É detalhado, é difícil de analisar, é difícil de ler, seu modelo de dados não é compatível com a forma como a maioria das linguagens de programação modelam dados e suas vantagens de extensibilidade são irrelevantes quando as principais necessidades da sua representação de saída são a serialização de uma representação interna. Eu poderia continuar...

Não vou me esforçar muito para explicar isso. A chave a notar é que hoje você terá dificuldade em encontrar qualquer API importante que ainda suporte XML. Você também não deveria.

Dito isso, se sua base de clientes consiste em um grande número de clientes corporativos, você pode ter que suportar XML de qualquer maneira. Se você precisar fazer isso, você se encontrará com uma nova pergunta:

O tipo de mídia deve ser alterado com base nos cabeçalhos Aceitar ou com base na URL? Para garantir a explorabilidade do navegador, ele deve estar na URL. A opção mais sensata aqui seria anexar uma extensão `.json` ou `.xml` ao URL do endpoint.

snake_case vs camelCase para nomes de campo

Se você estiver usando JSON (*JavaScript* Object Notation) como seu principal formato de representação, a coisa "certa" a fazer é seguir as convenções de nomenclatura JavaScript - e isso significa camelCase para nomes de campo! Se você seguir o caminho de construir bibliotecas de cliente em várias linguagens, é melhor usar convenções de nomenclatura idiomática nelas - camelCase para C# e Java, snake_case para python e ruby.

Comida para reflexão: sempre achei que snake_case é mais fácil de ler do que a convenção de camelCase do JavaScript . Eu simplesmente não tinha nenhuma evidência para apoiar meus sentimentos, até agora. Com base em um estudo de rastreamento ocular em camelCase e snake_case (PDF) de 2010, **snake_case é 20% mais fácil de ler do que camelCase** ! Esse impacto na legibilidade afetaria a explorabilidade da API e os exemplos na documentação.

Muitas APIs JSON populares usam snake_case. Suspeito que isso se deva às bibliotecas de serialização do lado do servidor que seguem as convenções de nomenclatura da linguagem subjacente em que estão incorporadas. Talvez precisemos ter bibliotecas de serialização JSON para lidar com as transformações de convenções de nomenclatura.

Impressão bonita por padrão e certifique-se de que o gzip seja suportado

Uma API que fornece saída compactada em espaço em branco não é muito divertida de se ver em um navegador. Embora algum tipo de parâmetro de consulta (como `?pretty=true`) possa ser fornecido para habilitar a impressão bonita, uma API que imprime bonita por padrão é muito mais acessível. O custo da transferência de dados extra é insignificante, especialmente quando comparado ao custo de não implementar o gzip.

Considere alguns casos de uso: E se um consumidor de API estiver depurando e seu código imprimir os dados recebidos da API - será legível por padrão. Ou se o consumidor pegar a URL que seu código estava gerando e acessá-la diretamente do navegador - ela será legível por padrão. Estas são pequenas coisas. Pequenas coisas que tornam uma API agradável de usar!

Mas e toda a transferência de dados extra?

Vejamos isso com um exemplo do mundo real. Eu puxei alguns [dados da API do GitHub] , que usa impressão bonita por padrão. Também farei algumas comparações com gzip:

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

Os arquivos de saída têm os seguintes tamanhos:

- `sem-espaço em branco.txt` - 1221 bytes
- `with-whitespace.txt` - 1290 bytes
- `without-whitespace.txt.gz` - 477 bytes
- `with-whitespace.txt.gz` - 480 bytes

Neste exemplo, o espaço em branco aumentou o tamanho da saída em 5,7% quando o gzip não está em execução e 0,6% quando o gzip está em execução. Por outro lado, o ato de **gzipar por si só proporcionou mais de 60% de economia de largura de banda** . Como o custo da impressão bonita é

relativamente pequeno, é melhor imprimir por padrão e garantir que a compactação gzip seja suportada!

Não use um envelope por padrão, mas torne-o possível quando necessário

Muitas APIs envolvem suas respostas em envelopes como este:

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

Existem algumas justificativas para fazer isso - facilita a inclusão de metadados adicionais ou informações de paginação, alguns clientes REST não permitem acesso fácil a cabeçalhos HTTP e solicitações JSONP não têm acesso a cabeçalhos HTTP. No entanto, com padrões que estão sendo rapidamente adotados como CORS e o cabeçalho Link da RFC 5988 , o envelopamento está começando a se tornar desnecessário.

Podemos testar a API no futuro, ficando livre de envelopes por padrão e envolvendo apenas em casos excepcionais.

Como usar um envelope em casos excepcionais?

Existem 2 situações em que um envelope é realmente necessário - se a API precisar oferecer suporte a solicitações de domínio cruzado sobre JSONP ou se o cliente for incapaz de trabalhar com cabeçalhos HTTP.

Para oferecer suporte a JSONP entre domínios: essas solicitações vêm com um parâmetro de consulta adicional (geralmente chamado `callback` ou `jsonp`) representando o nome da função de retorno de chamada. Se esse parâmetro estiver presente, a API deve alternar para um modo de envelope completo, onde sempre responde com um código de status HTTP 200 e passa o código de status

real na carga JSON. Quaisquer cabeçalhos HTTP adicionais que seriam passados junto com a resposta devem ser mapeados para campos JSON, assim:

```
callback_function({
  status_code: 200,
  next_page: "https://..",
  response: {
    ... actual JSON response body ...
  }
})
```

Para oferecer suporte a clientes HTTP limitados: Permita um parâmetro de consulta especial `?envelope=true` que acionaria o envelope sem a função de retorno de chamada JSONP.

Corpos POST, PUT e PATCH codificados em JSON

Se você está seguindo a abordagem deste post, então você adotou o JSON para todas as saídas da API. Vamos considerar JSON para entrada de API.

Muitas APIs usam codificação de URL em seus corpos de solicitação de API. A codificação de URL é exatamente o que parece - corpos de solicitação em que os pares de valores-chave são codificados usando as mesmas convenções usadas para codificar dados em parâmetros de consulta de URL. Isso é simples, amplamente suportado e faz o trabalho.

No entanto, a codificação de URL tem alguns problemas que a tornam problemática. Não tem nenhum conceito de tipos de dados. Isso força a API a analisar inteiros e booleanos de strings. Além disso, não tem um conceito real de estrutura hierárquica. Embora existam algumas convenções que podem construir alguma estrutura a partir de pares de valores-chave (como anexar [ ] a uma chave para representar uma matriz), isso não é uma comparação com a estrutura hierárquica nativa do JSON.

Se a API for simples, suponho que a codificação de URL seja suficiente. Mas eu diria que é inconsistente com o formato de saída.

Para uma API baseada em JSON, você também deve usar JSON para entrada de API.

Uma API que aceita solicitações POST, PUT e PATCH codificadas em JSON também deve exigir que o cabeçalho `Content-Type seja definido como application/json` ou gere um código de status HTTP 415 Unsupported Media Type.

Paginação

As APIs que adoram envelopes normalmente incluem dados de paginação no próprio envelope. E eu não os culpo - até recentemente, não havia muitas opções melhores. A maneira correta de incluir detalhes de paginação hoje é usar o cabeçalho Link introduzido pela RFC 8288 .

Uma API que usa o cabeçalho Link pode retornar um conjunto de links prontos para que o consumidor da API não precise construir links por conta própria. Isso é especialmente importante quando a paginação é baseada em cursor . Aqui está um exemplo de um cabeçalho Link usado corretamente, retirado da documentação do GitHub :

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

Mas essa não é uma solução completa, pois muitas APIs gostam de retornar as 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` .

Carregamento automático de representações de recursos relacionados

Há muitos casos em que um consumidor de API precisa carregar dados relacionados (ou referenciados) ao recurso que está sendo solicitado. Em vez de exigir que o consumidor acesse a API repetidamente para obter essas

informações, haveria um ganho de eficiência significativo ao permitir que os dados relacionados fossem retornados e carregados junto com o recurso original sob demanda.

No entanto, como isso vai contra alguns princípios RESTful, podemos minimizar nosso desvio apenas fazendo isso com base em um parâmetro de consulta `embed` (ou `expand`).

Nesse caso, `embed` seria uma lista separada por vírgulas de campos a serem incorporados. A notação de ponto pode ser usada para se referir a subcampos. Por exemplo:

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

Isso retornaria um ticket com detalhes adicionais incorporados, como:

```
{
  "id" : 12,
  "subject" : "I have a question!",
  "summary" : "Hi, ....",
  "customer" : {
    "name" : "Bob"
  },
  assigned_user: {
   "id" : 42,
   "name" : "Jim",
  }
}
```

Claro, a capacidade de implementar algo assim realmente depende da complexidade interna. Esse tipo de incorporação pode resultar facilmente em um problema de seleção N+1 .

Substituindo o método HTTP

Alguns clientes HTTP só podem funcionar com solicitações GET e POST simples. Para aumentar a acessibilidade a esses clientes limitados, a API precisa de uma maneira de substituir o método HTTP. Embora não existam padrões rígidos aqui, a convenção popular é aceitar um cabeçalho de solicitação `X-HTTP-Method-Override` com um valor de string contendo PUT, PATCH ou DELETE.

Observe que o cabeçalho de substituição deve ser aceito **apenas** em solicitações POST. As solicitações GET nunca devem alterar os dados no servidor !

Limitação de taxa

Para evitar abusos, é prática padrão adicionar algum tipo de limitação de taxa a uma API. A RFC 6585 introduziu um código de status HTTP 429 Too Many Requests para acomodar isso.

No entanto, pode ser muito útil notificar o consumidor sobre seus limites antes que eles realmente o atinjam. Esta é uma área que atualmente carece de padrões, mas tem várias convenções populares usando cabeçalhos de resposta HTTP .

No mínimo, inclua os seguintes cabeçalhos:

- `X-Rate-Limit-Limit` - O número de solicitações permitidas no período atual
- `X-Rate-Limit-Remaining` - O número de solicitações restantes no período atual
- `X-Rate-Limit-Reset` - O número de segundos restantes no período atual

Por que o número de segundos restantes está sendo usado em vez de um carimbo de hora para o X-Rate-Limit-Reset?

Um carimbo de data/hora 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 pode enviar a solicitação novamente e o número de segundos responde a essa pergunta com o mínimo de processamento adicional. Também evita problemas relacionados à distorção do relógio .

Algumas APIs usam um timestamp UNIX (segundos desde a época) para X-Rate-Limit-Reset. Não faça isso!

Por que é uma prática ruim usar um timestamp UNIX para X-Rate-Limit-Reset?

A especificação HTTP já especifica o uso de formatos de data RFC 1123 (atualmente sendo usados nos cabeçalhos HTTP Date , If-Modified-Since e Last-Modified). Se formos especificar um novo cabeçalho HTTP que receba algum tipo de carimbo de data/hora, ele deverá seguir as convenções RFC 1123 em vez de usar carimbos de data/hora UNIX.

Autenticação

Uma API RESTful deve ser 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 de classificação.

Sempre usando 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. O melhor disso é que é completamente explorável pelo navegador - o navegador apenas exibirá um prompt solicitando credenciais se receber um código de status `401 não autorizado` do servidor.

No entanto, esse método de autenticação token-over-basic-auth só é aceitável nos casos em que é prático fazer com que o usuário copie um token de uma interface de administração para o ambiente do consumidor da API. Nos casos em que isso não for possível, o OAuth 2 deve ser usado para fornecer transferência segura de token para terceiros. OAuth 2 usa tokens de portador 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, pois as solicitações JSONP não podem enviar credenciais de autenticação básica HTTP ou tokens de portador. Nesse caso, um parâmetro de consulta especial `access_token` pode ser usado. Observação: há um problema de segurança inerente ao usar um parâmetro de consulta para o token, pois a maioria dos servidores da Web armazena parâmetros de consulta nos logs do servidor.

Vale a pena, todos os três métodos acima são apenas maneiras de transportar o token através do limite da API. O próprio token subjacente real pode ser idêntico.

Cache

O HTTP fornece uma estrutura de cache integrada! Tudo o que você precisa fazer é incluir alguns cabeçalhos de resposta de saída adicionais e fazer uma pequena validação ao receber alguns cabeçalhos de solicitação de entrada.

Existem 2 abordagens: ETag e Last-Modified

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

Last-Modified : Isso basicamente funciona como ETag, exceto que usa timestamps. O cabeçalho de resposta `Last-Modified` contém um carimbo de data/hora no formato RFC 1123 que é validado em relação `a If-Modified-Since` . Observe que a especificação HTTP teve 3 formatos de data aceitáveis diferentes 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 um formato consumível conhecido. A representação de um erro não deve ser diferente da representação de qualquer recurso, apenas com seu próprio conjunto de campos.

A API deve sempre retornar códigos de status HTTP sensíveis. Os erros de API geralmente se dividem em 2 tipos: códigos de status da série 400 para problemas do cliente e códigos de status da série 500 para problemas do servidor. No

mínimo, a API deve padronizar que todos os erros da série 400 venham com representação de erro JSON consumível. Se possível (ou seja, se balanceadores de carga e proxies reversos podem criar corpos de erro personalizados), isso deve se estender aos códigos de status 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 consultado para obter mais detalhes nos documentos) e possivelmente uma descrição detalhada. A representação de saída JSON para algo assim seria:

```
{
  "code" : 1234,
  "message" : "Something bad happened :(",
  "description" : "More details about the error here"
}
```

Erros de validação para solicitações PUT, PATCH e POST precisarão de um detalhamento 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` adicional , assim:

```
{
  "code" : 1024,
  "message" : "Validation Failed",
  "errors" : [
    {
      "code" : 5432,
      "field" : "first_name",
      "message" : "First name cannot have fancy characters"
    },
    {
       "code" : 5622,
       "field" : "password",
       "message" : "Password cannot be blank"
    }
  ]
}
```

Códigos de status HTTP

O HTTP define vários [códigos de status significativos] que podem ser retornados de sua API. Eles podem ser aproveitados para ajudar os consumidores da API a

direcionar suas respostas de acordo. Separei uma pequena lista dos que você definitivamente deveria usar:

- `200 OK` - Resposta a um GET, PUT, PATCH ou DELETE bem-sucedido. Também pode ser usado para um POST que não resulta em criação.
- `201 Created` - Resposta a um POST que resulta em uma criação. Deve ser combinado com um cabeçalho de localização apontando para a localização do novo recurso
- `204 No Content` - Resposta a uma solicitação bem-sucedida que não retornará um corpo (como uma solicitação DELETE)
- `304 Not Modified` - Usado quando os cabeçalhos de cache HTTP estão em jogo
- `400 Bad Request` - A solicitação está malformada, por exemplo, se o corpo não for analisado
- `401 Não autorizado` - Quando não são fornecidos detalhes de autenticação inválidos ou nenhum. Também é útil para acionar um pop-up de autenticação se a API for usada em um navegador
- `403 Proibido` - Quando a autenticação foi bem-sucedida, mas o usuário autenticado não tem acesso ao recurso
- `404 Not Found` - Quando um recurso inexistente é solicitado
- `Método 405 não permitido` - Quando um método HTTP está sendo solicitado que não é permitido para o usuário autenticado
- `410 Gone` - Indica que o recurso neste ponto final não está mais disponível. Útil como resposta geral para versões antigas da API
- `415 Tipo de mídia não suportado` - Se o tipo de conteúdo incorreto foi fornecido como parte da solicitação
- `422 Entidade Não Processável` - Usado para erros de validação
- `429 Demasiados Pedidos` - Quando um pedido é rejeitado devido à limitação da taxa

Em suma

Uma API é uma interface de usuário para desenvolvedores. Esforce-se para garantir que não seja apenas funcional, mas também agradável de usar.