Design da API Web RESTful

Artigo • 28/03/2023

Aplicativos Web mais modernos expõem APIs que os clientes podem usar para interagir com o aplicativo. Uma API da Web bem projetada deve buscar apoiar:

- **Independência de plataforma**. Qualquer cliente deve ser capaz de chamar a API, independentemente de como a API está implementada internamente. Isso requer o uso de protocolos padrão e ter um mecanismo pelo qual o cliente e o serviço Web pode concordar com o formato dos dados a serem trocados.
- **Evolução do serviço**. A API da Web deve ser capaz de evoluir e adicionar funcionalidade independentemente de aplicativos cliente. À medida que a API evolui, aplicativos cliente existentes devem continuar a funcionar sem modificação. Todas as funcionalidades devem ser detectáveis para que os aplicativos cliente possam utilizá-las plenamente.

Essas diretrizes descrevem questões que você deve considerar ao criar uma API da Web.

O que é o REST?

Em 2000, Roy Fielding propôs a Transferência de Estado Representacional (REST) como uma abordagem de arquitetura para criar serviços Web. REST é um estilo arquitetural para a criação de sistemas distribuídos com base em hipermídia. A REST é independente de qualquer protocolo subjacente e não está necessariamente ligada a HTTP. No entanto, as implementações mais comuns da API REST usam HTTP como o protocolo de aplicativo, e este guia foca a criação de APIs REST para HTTP.

Uma vantagem principal do REST sobre HTTP é que ele usa padrões abertos e não vincula a implementação da API ou os aplicativos cliente a nenhuma implementação específica. Por exemplo, um serviço Web REST poderia ser escrito em ASP.NET, e aplicativos cliente podem usar qualquer linguagem ou o conjunto de ferramentas que possa gerar solicitações HTTP e analisar respostas HTTP.

Aqui estão alguns dos princípios de design mais importante de APIs RESTful usando HTTP:

- APIs REST são projetadas para *recursos*, que se tratam de qualquer tipo de objeto, dados ou serviço que possa ser acessado pelo cliente.
- Um recurso tem um *identificador*, o qual se trata de um URI que identifica exclusivamente esse recurso. Por exemplo, o URI para um pedido determinada do

cliente pode ser:

```
HTTP

https://adventure-works.com/orders/1
```

- Os clientes interagem com um serviço por meio da troca de *representações* de recursos. Muitas APIs da Web usam JSON como o formato de troca. Por exemplo, uma solicitação GET para o URI listado acima poderia retornar este corpo de resposta:

```
JSON

{"orderId":1,"orderValue":99.90,"productId":1,"quantity":1}
```

- As APIs REST usam uma interface uniforme, o que ajuda a separar as implementações de cliente e de serviço. Para APIs REST baseadas em HTTP, a interface uniforme inclui o uso de verbos HTTP padrão para executar operações em recursos. As operações mais comuns são GET, POST, PUT, PATCH e DELETE.

- As APIs REST usam um modelo de solicitação sem estado. Solicitações HTTP devem ser independentes e podem ocorrer em qualquer pedido, portanto, não é viável manter informações de estado transitório entre as solicitações. O único local onde as informações são armazenadas é nos próprios recursos, e cada solicitação deve ser uma operação atômica. Essa restrição permite que serviços Web sejam altamente dimensionáveis porque não é necessário manter nenhum tipo de afinidade entre clientes e servidores específicos. Todos os servidores podem tratar de solicitações de qualquer cliente. Dito isso, outros fatores podem limitar a escalabilidade. Por exemplo, muitos serviços Web gravam em um armazenamento de dados de back-end, que pode ser difícil de escalar horizontalmente. Para obter mais informações sobre as estratégias para escalar horizontalmente um armazenamento de dados, confira Particionamento de dados horizontal, vertical e funcional.

- As APIs REST são orientadas por links de hipermídia contidos na representação. Por exemplo, a seguir é mostrada uma representação JSON de um pedido. Ela contém links para obter ou atualizar o cliente associado ao pedido.

```
JSON

{
  "orderID":3,
  "productID":2,
```

```
  "quantity":4,
  "orderValue":16.60,
  "links": [
    {"rel":"product","href":"https://adventure-works.com/customers/3",
"action":"GET" },
    {"rel":"product","href":"https://adventure-works.com/customers/3",
"action":"PUT" }
  ]
}
```

Em 2008, Leonard Richardson propôs seguintes modelo de maturidade para APIs da Web:

- Nível 0: defina um URI, e todas as operações são solicitações POST para esse URI.
- Nível 1: crie URIs separados para recursos individuais.
- Nível 2: use métodos HTTP para definir as operações nos recursos.
- Nível 3: use hipermídia (HATEOAS, descrita abaixo).

O Nível 3 corresponde a uma API RESTful de verdade de acordo com a definição de Fielding. Na prática, muitas APIs da Web publicadas se enquadram em algum lugar do Nível 2.

Organizar o design da API em torno dos recursos

Concentre-se nas entidades comerciais que a API da Web expõe. Por exemplo, em um sistema de comércio eletrônico, as entidades primárias podem ser clientes e pedidos. É possível criar um pedido por meio do envio de uma solicitação HTTP POST que contenha as informações do pedido. A resposta HTTP indica se o pedido foi feito com êxito ou não. Quando possível, os URIs de recursos devem ser baseados em substantivos (o recurso) e não em verbos (as operações no recurso).

HTTP

```
https://adventure-works.com/orders // Good

https://adventure-works.com/create-order // Avoid
```

Um recurso não precisa se basear em um só item de dados físico. Por exemplo, um recurso de pedido pode ser implementado internamente como várias tabelas em um banco de dados relacional, mas apresentado ao cliente como uma única entidade. Evite criar APIs que simplesmente espelhem a estrutura interna de um banco de dados. A

finalidade da REST é modelar entidades e as operações que um aplicativo pode executar nessas entidades. Um cliente não deve ser exposto à implementação interna.

Entidades geralmente são agrupadas em coleções (pedidos, clientes). Uma coleção é um recurso separado do item na coleção e deve ter seu próprio URI. Por exemplo, o seguinte URI pode representar a coleção de pedidos:

```
HTTP

https://adventure-works.com/orders
```

O envio de uma solicitação HTTP GET para o URI da coleção recupera uma lista de itens na coleção. Cada item na coleção também tem seu próprio URI exclusivo. Uma solicitação HTTP GET para o URI do item retorna os detalhes desse item.

Adote uma convenção de nomenclatura consistente nos URIs. Em geral, é bom usar substantivos plurais para URIs que fazem referência a coleções. É uma boa prática organizar URIs de coleções e itens em uma hierarquia. Por exemplo, `/customers` é o caminho para a coleção de clientes, e `/customers/5` é o caminho para o cliente com a ID igual a 5. Essa abordagem ajuda a manter a API da Web intuitiva. Além disso, muitas estruturas de API da Web podem rotear solicitações baseadas em caminhos de URI com parâmetros, de modo que é possível definir uma rota para o caminho `/customers/{id}`.

Também considere as relações entre diferentes tipos de recursos e como você pode expor essas associações. Por exemplo, `/customers/5/orders` pode representar todos os pedidos do cliente 5. Também é possível ir na outra direção e representar a associação de um pedido de volta a um cliente com um URI como `/orders/99/customer`. No entanto, estender esse modelo demasiadamente pode torná-lo difícil de ser implementado. Uma solução melhor é fornecer links navegáveis para recursos associados no corpo da mensagem de resposta HTTP. Esse mecanismo é descrito com mais detalhes na seção Usar HATEOAS para habilitar a navegação para recursos relacionados.

Em sistemas mais complexos, pode ser tentador fornecer URIs que permitam que um cliente navegue entre vários níveis de relações, como `/customers/1/orders/99/products`. No entanto, esse nível de complexidade pode ser difícil de manter e é inflexível no caso de as relações entre os recursos mudarem no futuro. Em vez disso, tente manter os URIs relativamente simples. Uma vez que um aplicativo tem uma referência a um recurso, deve ser possível usar essa referência para localizar itens relacionados a esse recurso. A consulta anterior pode ser substituída com o URI `/customers/1/orders` para localizar todos os pedidos do cliente 1 e depois `/orders/99/products` para localizar os produtos nesse pedido.

> **Dica**
>
> Evite exigir URIs de recurso mais complexos do que *collection/item/collection*.

Outro fator é que todas as solicitações Web impõem uma carga no servidor Web. Quanto mais solicitações, maior a carga. Portanto, tente evitar APIs da Web "verborrágicas" que expõem um grande número de recursos pequenos. Uma API desse tipo pode exigir que um aplicativo cliente envie várias solicitações para localizar todos os dados de que ele precisa. Em vez disso, talvez você deseje desnormalizar os dados e combinar informações relacionadas em recursos maiores, que podem ser recuperados com uma única solicitação. No entanto, você precisa equilibrar essa abordagem em relação à sobrecarga causada pela busca de dados de que o cliente não precisa. Recuperar objetos grandes pode aumentar a latência de uma solicitação e incorrer em custos adicionais de largura de banda. Para obter mais informações sobre esses antipadrões de desempenho, consulte E/S verborrágica e Busca estranha.

Evite introduzir as dependências entre a API da Web e as fontes de dados subjacentes. Por exemplo, se seus dados estão armazenados em um banco de dados relacional, a API da Web não precisa expor cada tabela como uma coleção de recursos. Na verdade, provavelmente esse seja um design ruim. Em vez disso, pense na API da Web como uma abstração do banco de dados. Se necessário, introduza uma camada de mapeamento entre o banco de dados e a API da Web. Dessa forma, os aplicativos cliente ficam isolados das alterações do esquema de banco de dados subjacente.

Por fim, pode não ser possível mapear toda operação implementada por uma API da Web para um recurso específico. Você pode tratar desses cenários *sem recursos* por meio de solicitações HTTP, que invocam uma função e retornam os resultados como uma mensagem de resposta HTTP. Por exemplo, uma API da Web que implementa operações simples de calculadora, como adicionar e subtrair, pode fornecer URIs que exponham essas operações como pseudorrecursos, além de utilizar a cadeia de consulta para especificar os parâmetros necessários. Por exemplo, uma solicitação GET para o URI */add?operand1=99&operand2=1* retornaria uma mensagem de resposta com o corpo contendo o valor 100. No entanto, use essas formas de URIs apenas com moderação.

Definir operações de API em termos de métodos HTTP

O protocolo HTTP define vários métodos que atribuem significado semântico a uma solicitação. Os métodos HTTP comuns usados pelas APIs da Web mais RESTful são:

- **GET**, que recupera uma representação do recurso no URI especificado. O corpo da mensagem de resposta contém os detalhes do recurso solicitado.
- **POST**, que cria um novo recurso no URI especificado. O corpo da mensagem de solicitação fornece os detalhes do novo recurso. Observe que POST também pode ser usado para disparar operações que, na verdade, não criam recursos.
- **PUT**, que cria ou substitui o recurso no URI especificado. O corpo da mensagem de solicitação especifica o recurso a ser criado ou atualizado.
- **PATCH**, que realiza uma atualização parcial de um recurso. O corpo da solicitação especifica o conjunto de alterações a ser aplicado ao recurso.
- **DELETE**, que remove o recurso do URI especificado.

O efeito de uma solicitação específica deve depender de o recurso ser uma coleção ou um item individual. A tabela a seguir resume as convenções comuns adotadas maioria das implementações RESTful usando o exemplo de comércio eletrônico. Nem todas essas solicitações podem ser implementadas; isso depende do cenário específico.

Expandir a tabela

Recurso	POST	GET	PUT	DELETE
/clientes	Criar um novo cliente	Obter todos os clientes	Atualização em massa de clientes	Remover todos os clientes
/clientes/1	Erro	Obter os detalhes do cliente 1	Atualizar os detalhes do cliente 1 se ele existir	Remover cliente 1
/clientes/1/pedidos	Criar um novo pedido para o cliente 1	Obter todos os pedidos do cliente 1	Atualização em massa de pedidos do cliente 1	Remover todos os pedidos do cliente 1

As diferenças entre POST, PUT e PATCH podem ser confusas.

- Uma solicitação POST cria um recurso. O servidor atribui um URI para o novo recurso e retorna esse URI para o cliente. No modelo de REST, você frequentemente aplica solicitações POST para coleções. O novo recurso é adicionado à coleção. Uma solicitação POST também pode ser usada para enviar dados para o processamento de um recurso existente sem que nenhum novo recurso seja criado.

- Uma solicitação PUT cria um recurso *ou* atualiza um recurso existente. O cliente especifica o URI do recurso. O corpo da solicitação contém uma representação completa do recurso. Se já existir um recurso com esse URI, ele será substituído. Caso contrário, um novo recurso é criado, caso o servidor ofereça suporte a isso.

Solicitações PUT são aplicadas mais frequentemente a recursos que sejam itens individuais, como um cliente específico, em vez de coleções. Um servidor pode oferecer suporte a atualizações, mas não a uma criação por meio de PUT. O suporte à criação por meio de PUT depende de se o cliente pode atribuir significativamente um URI a um recurso antes que ele exista. Se não, use o POST para criar recursos e o PUT ou o PATCH para atualizar.

- Uma solicitação PATCH executa uma *atualização parcial* a um recurso existente. O cliente especifica o URI do recurso. O corpo da solicitação especifica um conjunto de *alterações* a ser aplicado ao recurso. Isso pode ser mais eficiente do que usar o PUT, porque o cliente envia apenas as alterações, e não a representação inteira do recurso. Tecnicamente, o PATCH também pode criar um novo recurso (especificando um conjunto de atualizações para um recurso "null"), se o servidor oferecer suporte a isso.

Solicitações PUT devem ser idempotentes. Se um cliente enviar a mesma solicitação PUT várias vezes, os resultados deverão ser sempre os mesmos (o mesmo recurso será modificado com os mesmos valores). Não há garantia de que as solicitações POST e PATCH sejam idempotentes.

Em conformidade com a semântica HTTP

Esta seção descreve algumas considerações comuns para a criação de uma API que esteja em conformidade com a especificação de HTTP. No entanto, não serão abordados todos os detalhes ou cenários possíveis. Em caso de dúvida, consulte as especificações de HTTP.

Tipos de mídia

Como mencionado anteriormente, os clientes e servidores trocam representações de recursos. Por exemplo, em uma solicitação POST, o corpo da solicitação contém uma representação do recurso para criar. Em uma solicitação GET, o corpo da resposta contém uma representação do recurso de busca.

No protocolo HTTP, formatos são especificados por meio do uso de *tipos de mídia*, também chamados de tipos MIME. Para dados não binários, a maioria das APIs da Web oferecem suporte a JSON (tipo de mídia = `application/json`) e, possivelmente, a XML (tipo de mídia = `application/xml`).

O cabeçalho Content-Type em uma solicitação ou resposta especifica o formato da representação. Aqui está um exemplo de uma solicitação POST que inclui dados JSON:

HTTP

```
POST https://adventure-works.com/orders HTTP/1.1
Content-Type: application/json; charset=utf-8
Content-Length: 57

{"Id":1,"Name":"Gizmo","Category":"Widgets","Price":1.99}
```

Se o servidor não oferece suporte para o tipo de mídia, ele deverá retornar um código de status HTTP 415 (Tipo de mídia sem suporte).

Uma solicitação cliente pode incluir um cabeçalho Accept contendo uma lista de tipos de mídia que o cliente aceitará do servidor na mensagem de resposta. Por exemplo:

HTTP

```
GET https://adventure-works.com/orders/2 HTTP/1.1
Accept: application/json
```

Se o servidor não consegue corresponder a nenhum dos tipos de mídia listados, ele deve retornar um código de status HTTP 406 (Não aceitável).

Métodos GET

Geralmente, um método GET bem-sucedido retorna o código de status HTTP 200 (OK). Se o recurso não puder ser encontrado, o método deve retornar 404 (Não encontrado).

Se a solicitação foi atendida, mas não há nenhum corpo de resposta incluído na resposta HTTP, ela deverá retornar o código de status HTTP 204 (Sem Conteúdo); por exemplo, uma operação de pesquisa que não produz correspondências pode ser implementada com esse comportamento.

Métodos POST

Se um método POST cria um novo recurso, ele retornará o código de status HTTP 201 (Criado). O URI do novo recurso está incluído no cabeçalho Location da resposta. O corpo da resposta contém uma representação do recurso.

Se o método executa algum processamento, mas não cria um novo recurso, pode ser que ele retorne o código de status HTTP 200 e inclua o resultado da operação no corpo da resposta. Como alternativa, se não houver nenhum resultado para retornar, o método pode retornar o código de status HTTP 204 (Sem conteúdo) sem o corpo da resposta.

Se o cliente coloca os dados inválidos na solicitação, o servidor deve retornar o código de status HTTP 400 (Solicitação incorreta). O corpo da resposta pode conter informações adicionais sobre o erro ou um link para um URI que forneça mais detalhes.

Métodos PUT

Se um método PUT cria um novo recurso, ele retornará o código de status HTTP 201 (Criado), assim como com um método POST. Se o método atualiza um recurso existente, ele retorna 200 (OK) ou 204 (Sem conteúdo). Em alguns casos, pode não ser possível atualizar um recurso existente. Nesse caso, considere que pode ser retornado o código de status HTTP 409 (Conflito).

Considere a possibilidade de implementar operações HTTP PUT em massa, que podem realizar atualizações em lote para vários recursos em uma coleção. A solicitação PUT deve especificar o URI da coleção, enquanto o corpo da solicitação deve especificar os detalhes dos recursos a serem modificados. Essa abordagem pode ajudar a reduzir a verborragia e a melhorar o desempenho.

Métodos PATCH

Com uma solicitação PATCH, o cliente envia um conjunto de atualizações para um recurso existente na forma de um *documento patch*. O servidor processa o documento patch para executar a atualização. O documento patch não descreve o recurso inteiro, apenas um conjunto de alterações a serem aplicadas. A especificação para o método PATCH (RFC 5789) não define um formato específico para documentos patch. O formato deve ser deduzido a partir do tipo de mídia na solicitação.

Provavelmente JSON seja o formato mais comum de dados para APIs da Web. Há dois formatos principais de patch baseado em JSON, os quais são chamados de *patch JSON* e *patch de mesclagem JSON*.

O patch de mesclagem JSON é um pouco mais simples. O documento patch tem a mesma estrutura que o recurso JSON original, mas inclui apenas o subconjunto de campos que devem ser alterados ou adicionados. Além disso, um campo pode ser excluído especificando `null` como o valor do campo no documento patch. (Isso significa que o patch de mesclagem não é adequado caso o recurso original possa ter valores nulos explícitos.)

Por exemplo, suponha que o recurso original tenha a seguinte representação JSON:

```
JSON
```

```
{
    "name":"gizmo",
    "category":"widgets",
    "color":"blue",
    "price":10
}
```

Aqui está um patch de mesclagem JSON possível para esse recurso:

JSON

```
{
    "price":12,
    "color":null,
    "size":"small"
}
```

Isso instrui o servidor a atualizar `price`, excluir `color` e adicionar `size`, enquanto `name` e `category` não são modificados. Para obter os detalhes exatos do patch de mesclagem JSON, consulte RFC 7396. O tipo de mídia para patch de mesclagem JSON é `application/merge-patch+json`.

Devido a um significado especial de `null` no documento patch, o patch de mesclagem não é adequado se o recurso original puder conter valores nulos explícitos. Além disso, o documento patch não especifica o pedido no qual o servidor deve aplicar as atualizações. Isso pode ou não importar, dependendo dos dados e o domínio. O patch JSON, definido em RFC 6902, é mais flexível. Ele especifica as alterações como uma sequência de operações a ser aplicada. As operações incluem adicionar, remover, substituir, copiar e testar (para validar valores). O tipo de mídia para patch de JSON é `application/json-patch+json`.

Estas são algumas condições comuns de erro que podem ser encontradas ao processar uma solicitação PATCH, juntamente com o código de status HTTP apropriado.

Expandir a tabela

Condição de erro	Código de status de HTTP
Não há suporte para o formato de documento patch.	415 (Tipo de mídia sem suporte)
Documento patch malformado.	400 (Solicitação inválida)

Condição de erro	Código de status de HTTP
O documento patch é válido, mas as alterações não podem ser aplicadas ao recurso em seu estado atual.	409 (Conflito)

Métodos DELETE

Se a operação de exclusão for bem-sucedida, o servidor Web deverá responder com um código de status HTTP 204 (Nenhum Conteúdo), indicando que o processo foi tratado com êxito, mas que o corpo da resposta não contém informações adicionais. Se o recurso não existir, o servidor Web pode retornar HTTP 404 (Não encontrado).

Operações assíncronas

Algumas vezes, uma operação POST, PUT, PATCH ou DELETE pode exigir um processamento que leva algum tempo para ser concluído. Se você aguardar a conclusão antes de enviar uma resposta ao cliente, isso pode causar uma latência inaceitável. Nesse caso, considere fazer a operação assíncrona. O código de status HTTP 202 (Aceito) retorna para indicar que a solicitação foi aceita para processamento, mas não está concluída.

Você deve expor um ponto de extremidade que retorna o status de uma solicitação assíncrona, de modo que o cliente possa monitorar o status por meio da sondagem do ponto de extremidade de status. Inclua o URI do ponto de extremidade de status no cabeçalho Location da resposta 202. Por exemplo:

```
HTTP

HTTP/1.1 202 Accepted
Location: /api/status/12345
```

Se o cliente enviar uma solicitação GET para esse ponto de extremidade, a resposta deve conter o status atual da solicitação. Como opção, ela também pode incluir um tempo estimado para conclusão ou um link para cancelar a operação.

```
HTTP

HTTP/1.1 200 OK
Content-Type: application/json

{
    "status":"In progress",
    "link": { "rel":"cancel", "method":"delete", "href":"/api/status/12345"
```

```
}
}
```

Se a operação assíncrona criar um novo recurso, o ponto de extremidade de status deve retornar o código de status 303 (Ver outros) após a conclusão da operação. A resposta 303 inclui um cabeçalho Location que fornece o URI do novo recurso:

```HTTP
HTTP/1.1 303 See Other
Location: /api/orders/12345
```

Para obter mais informações sobre como implementar essa abordagem, consulte Fornecer suporte assíncrono para solicitações de execução prolongada e o padrão de solicitação-resposta assíncrona.

Conjuntos vazios em corpos de mensagens

Sempre que o corpo de uma resposta bem-sucedida estiver vazio, o código de status deverá ser 204 (Sem Conteúdo). Para conjuntos vazios, como uma resposta a uma solicitação filtrada sem itens, o código de status ainda deve ser 204 (Sem Conteúdo), e não 200 (OK).

Filtrar e paginar dados

Expor uma coleção de recursos por meio de um único URI pode levar a aplicativos que buscam grandes quantidades de dados quando apenas um subconjunto dessas informações é necessário. Por exemplo, suponha que um aplicativo cliente precise localizar todos os pedidos com custo acima de um valor específico. Ele pode recuperar todos os pedidos do URI */order* e, em seguida, filtrar esses pedidos no lado do cliente. Fica claro que esse processo é muito ineficiente. Ele desperdiça tanto energia de processamento quanto de largura de banda de rede no servidor que hospeda a API da Web.

Em vez disso, a API pode permitir a transmissão de um filtro na cadeia de consulta do URI, como */orders?minCost=n*. A API da Web é responsável pela análise e pelo tratamento do parâmetro `minCost` na cadeia de consulta e por retornar os resultados filtrados no lado do servidor.

Solicitações GET sobre os recursos de coleção têm potencial de retornar um grande número de itens. Você deve projetar uma API da Web para limitar a quantidade de dados retornados por qualquer solicitação única. Considere a possibilidade de oferecer

suporte a cadeias de consulta que especifiquem o número máximo de itens para recuperar e um deslocamento de início para a coleção. Por exemplo:

```
HTTP

/orders?limit=25&offset=50
```

Também considere a possibilidade de impor um limite superior na quantidade de itens retornados para ajudar a evitar ataques de negação de serviço. Para ajudar aplicativos cliente, solicitações GET que retornam dados paginados também devem incluir algum tipo de metadados que indiquem o número total de recursos disponíveis na coleção.

Você pode usar uma estratégia semelhante para classificar os dados conforme eles são encontrados por meio do fornecimento de um parâmetro de classificação que use um nome de campo como o valor, tal como */orders?sort=ProductID*. No entanto, essa abordagem pode ter um efeito negativo no cache, pois parâmetros de cadeia de consulta fazem parte do identificador de recurso usado por muitas implementações de cache como a chave para dados armazenados em cache.

Você pode estender esse método para limitar os campos retornados para cada item, se cada um deles contiver uma grande quantidade de dados. Por exemplo, você poderia usar um parâmetro de cadeia de consulta que aceita uma lista de campos delimitada por vírgulas, como */orders?fields=ProductID,Quantity*.

Forneça padrões significativos a todos os parâmetros opcionais nas cadeias de consulta. Por exemplo, defina o parâmetro `limit` como 10 e o parâmetro `offset` como 0 se você implementar a paginação, defina o parâmetro sort para a chave do recurso se você implementar ordenação e, por fim, defina o parâmetro `fields` para todos os campos no recurso se você oferecer suporte a projeções.

Suporte a respostas parciais para recursos binários grandes

Um recurso pode conter campos binários grandes, como arquivos ou imagens. Para superar os problemas causados por conexões não confiáveis e intermitentes e para melhorar os tempos de resposta, considere a possibilidade de habilitar a recuperação desses recursos em partes. Para isso, a API da Web deve oferecer suporte ao cabeçalho Accept-Ranges para solicitações GET para grandes recursos. Esse cabeçalho indica que a operação GET oferece suporte a solicitações parciais. O aplicativo cliente pode enviar solicitações GET que retornam um subconjunto de um recurso, especificado como um intervalo de bytes.

Além disso, considere a possibilidade de implementar solicitações HTTP HEAD para esses recursos. Uma solicitação HEAD é semelhante a uma solicitação GET, porém retorna apenas cabeçalhos HTTP que descrevem o recurso com um corpo de mensagem vazio. Um aplicativo cliente pode emitir uma solicitação HEAD para determinar se deve ou não buscar um recurso pelo uso de solicitações GET parciais. Por exemplo:

```
HTTP

HEAD https://adventure-works.com/products/10?fields=productImage HTTP/1.1
```

Eis um exemplo de corpo da resposta:

```
HTTP

HTTP/1.1 200 OK

Accept-Ranges: bytes
Content-Type: image/jpeg
Content-Length: 4580
```

O cabeçalho Content-Length dá o tamanho total do recurso, e o cabeçalho Accept-Ranges indica que a operação GET correspondente oferece suporte a resultados parciais. O aplicativo cliente pode usar essas informações para recuperar a imagem em partes menores. A primeira solicitação busca os primeiros 2.500 bytes usando o cabeçalho Range:

```
HTTP

GET https://adventure-works.com/products/10?fields=productImage HTTP/1.1
Range: bytes=0-2499
```

A mensagem de resposta indica, ao retornar o código de status HTTP 206, que esta é uma resposta parcial. O cabeçalho Content-Length especifica o número real de bytes retornados no corpo da mensagem (não o tamanho do recurso), enquanto o cabeçalho Content-Range indica que parte do recurso isso é (0-2.499 bytes de 4.580):

```
HTTP

HTTP/1.1 206 Partial Content

Accept-Ranges: bytes
Content-Type: image/jpeg
Content-Length: 2500
Content-Range: bytes 0-2499/4580
```

```
[...]
```

Uma solicitação subsequente do aplicativo cliente pode recuperar o restante do recurso.

Usar o HATEOAS para habilitar a navegação para recursos relacionados

Uma das principais motivações por trás de REST é a que deve ser possível navegar por todo o conjunto de recursos sem exigir conhecimento prévio do esquema de URI. Cada solicitação HTTP GET deve retornar as informações necessárias para localizar os recursos relacionados diretamente ao objeto solicitado por hiperlinks incluídos na resposta, e também deve ser provida de informações descrevendo as operações disponíveis em cada um desses recursos. Esse conceito é conhecido como HATEOAS, ou Hipertexto como o Mecanismo de Estado do Aplicativo. O sistema é efetivamente uma máquina de estado finito, e a resposta para cada solicitação contém as informações necessárias para ir de um estado para outro; nenhuma outra informação deve ser necessária.

> **Observação**
>
> Atualmente, não há padrões de uso geral que definem como modelar o princípio HATEOAS. Os exemplos mostrados nesta seção ilustram uma possível solução proprietária.

Por exemplo, para tratar da relação entre um pedido e um cliente, a representação de um pedido pode incluir links que identificam as operações disponíveis para o cliente do pedido. Eis aqui uma representação possível:

JSON

```
{
  "orderID":3,
  "productID":2,
  "quantity":4,
  "orderValue":16.60,
  "links":[
    {
      "rel":"customer",
      "href":"https://adventure-works.com/customers/3",
      "action":"GET",
      "types":["text/xml","application/json"]
    },
    {
      "rel":"customer",
```

```
        "href":"https://adventure-works.com/customers/3",
        "action":"PUT",
        "types":["application/x-www-form-urlencoded"]
      },
      {
        "rel":"customer",
        "href":"https://adventure-works.com/customers/3",
        "action":"DELETE",
        "types":[]
      },
      {
        "rel":"self",
        "href":"https://adventure-works.com/orders/3",
        "action":"GET",
        "types":["text/xml","application/json"]
      },
      {
        "rel":"self",
        "href":"https://adventure-works.com/orders/3",
        "action":"PUT",
        "types":["application/x-www-form-urlencoded"]
      },
      {
        "rel":"self",
        "href":"https://adventure-works.com/orders/3",
        "action":"DELETE",
        "types":[]
      }]
}
```

Nesse exemplo, a matriz `links` tem um conjunto de links. Cada link representa uma operação em uma entidade relacionada. Os dados para cada link incluem a relação ("cliente"), o URI (`https://adventure-works.com/customers/3`), o método HTTP e os tipos MIME com suporte. Essas são todas as informações de que um aplicativo cliente precisa para ser capaz de invocar a operação.

A matriz `links` também inclui informações referentes a si sobre o próprio recurso que foram recuperadas. Elas apresentam a relação *self*.

O conjunto de links retornados pode mudar, dependendo do estado do recurso. Isso é o que se quer dizer ao se afirmar que o hipertexto é o "mecanismo do estado do aplicativo".

Controlando a versão de uma API da Web RESTful

É muito improvável que uma API da Web permanecerá estática. Conforme os requisitos de negócios mudam, novas coleções de recursos podem ser adicionados, as relações entre os recursos podem mudar e a estrutura dos dados nos recursos pode ser alterada. Enquanto a atualização de uma API da Web para lidar com requisitos novos ou diferentes é um processo relativamente simples, você deve considerar os efeitos que essas mudanças terão em aplicativos cliente que consomem a API da Web. O problema é que, embora o desenvolvedor que projeta e implementa uma API da Web tenha controle total sobre essa API, ele não tem o mesmo grau de controle sobre os aplicativos cliente, que podem ser criados por organizações terceirizadas operando remotamente. A principal prioridade é habilitar aplicativos cliente existentes a continuar funcionando inalterados, permitindo simultaneamente que novos aplicativos cliente tirem proveito das novas funções e recursos.

O controle de versão permite que uma API da Web indique os recursos e as funções que ela expõe, e um aplicativo cliente pode enviar solicitações direcionadas a uma versão específica de uma função ou recurso. As seções a seguir descrevem várias abordagens diferentes, cada qual com suas próprias vantagens e desvantagens.

Sem controle de versão

Essa é a abordagem mais simples e pode ser aceitável para algumas APIs internas. Alterações significativas poderiam ser representadas como novos recursos ou novos links. Adicionar conteúdo aos recursos existentes não deve representar uma alteração significativa, já que aplicativos cliente que não esperavam ver esse conteúdo vão ignorá-lo.

Por exemplo, uma solicitação para o URI `https://adventure-works.com/customers/3` deve retornar os detalhes de um único cliente contendo os campos `id`, `name` e `address` esperados pelo aplicativo cliente:

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Contoso LLC","address":"1 Microsoft Way Redmond WA 98053"}
```

> ⓘ **Observação**
>
> Simplificando, as respostas de exemplo mostradas nesta seção não incluem links HATEOAS.

Se o campo `DateCreated` é adicionado ao esquema do recurso de cliente, a resposta terá essa aparência:

```
HTTP

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-
04T12:11:38.0376089Z","address":"1 Microsoft Way Redmond WA 98053"}
```

Aplicativos cliente existentes podem continuar funcionando corretamente se forem capazes de ignorar campos não reconhecidos, enquanto os novos aplicativos cliente podem ser projetados para lidar com esse novo campo. No entanto, se ocorrerem alterações mais radicais ao esquema de recursos (por exemplo, remover ou renomear campos) ou se as relações entre os recursos mudarem, estas poderão constituir alterações significativas que impedem que aplicativos cliente existentes funcionem corretamente. Nessas situações, considerare uma das abordagens a seguir.

Controle de versão de URI

Cada vez que você modifica a API da Web ou altera o esquema de recursos, você adiciona um número de versão ao URI para cada recurso. Os URIs previamente existentes devem continuar a funcionar como antes, recuperando recursos que estão em conformidade com seu esquema original.

Estendendo o exemplo anterior, se o campo `address` for reestruturado em subcampos contendo cada parte constituinte do endereço (como `streetAddress`, `city`, `state` e `zipCode`), esta versão do recurso poderá ser exposta por meio de um URI contendo um número de versão, como `https://adventure-works.com/v2/customers/3`:

```
HTTP

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-
04T12:11:38.0376089Z","address":{"streetAddress":"1 Microsoft
Way","city":"Redmond","state":"WA","zipCode":98053}}
```

Esse mecanismo de controle de versão é muito simples, mas depende do servidor realizar o roteamento da solicitação para o ponto de extremidade apropriado. No entanto, ele pode se tornar inviável conforme a API da Web amadurece ao passar por

várias iterações e o servidor tem que oferecer suporte a um número de versões diferentes. Além disso, de um ponto de vista purista, em todos os casos os aplicativos cliente estão buscando os mesmos dados (cliente 3), portanto o URI não deve ser diferente, independentemente de qual for a versão. Esse esquema também complica a implementação de HATEOAS, já que todos os links precisarão incluir o número da versão em seus URIs.

Controle de versão de cadeia de consulta

Em vez de fornecer vários URIs, você pode especificar a versão do recurso usando um parâmetro dentro da cadeia de consulta acrescentada à solicitação HTTP, como `https://adventure-works.com/customers/3?version=2`. O parâmetro de versão, caso seja omitido por aplicativos cliente mais antigos, deve passar a usar um valor padrão significativo, como 1.

Essa abordagem tem a vantagem de semântica que o mesmo recurso é sempre recuperado do mesmo URI, mas para isso, é necessário que o código que processa a solicitação analise a cadeia de consulta e envie de volta a resposta HTTP apropriada. Essa abordagem também tem as mesmas complicações para implementar HATEOAS como o mecanismo de controle de versão do URI.

> ⓘ **Observação**
>
> Alguns navegadores e proxies da Web mais antigos não armazenarão respostas em cache para solicitações que incluam, no URI, uma cadeia de consulta. Isso pode degradar o desempenho de aplicativos Web que usam uma API da Web e que são executados de um navegador da Web desse tipo.

Controle de versão de cabeçalho

Em vez de acrescentar o número de versão como um parâmetro de cadeia de consulta, você pode implementar um cabeçalho personalizado que indica a versão do recurso. Essa abordagem requer que o aplicativo cliente adicione o cabeçalho apropriado a quaisquer solicitações, embora o código que processa a solicitação do cliente possa usar um valor padrão (versão 1) se o cabeçalho da versão for omitido. Os exemplos a seguir utilizam um cabeçalho personalizado nomeado *Cabeçalho Personalizado*. O valor desse cabeçalho indica a versão da API da Web.

Versão 1:

```
HTTP
```

```
GET https://adventure-works.com/customers/3 HTTP/1.1
Custom-Header: api-version=1
```

```
HTTP

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Contoso LLC","address":"1 Microsoft Way Redmond WA 98053"}
```

Versão 2:

```
HTTP

GET https://adventure-works.com/customers/3 HTTP/1.1
Custom-Header: api-version=2
```

```
HTTP

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-
04T12:11:38.0376089Z","address":{"streetAddress":"1 Microsoft
Way","city":"Redmond","state":"WA","zipCode":98053}}
```

Assim como nas duas abordagens anteriores, implementar HATEOAS requer a inclusão do cabeçalho personalizado apropriado em quaisquer eventuais links.

Controle de versão do tipo de mídia

Quando um aplicativo cliente envia uma solicitação HTTP GET para um servidor Web, ele deve estipular o formato do conteúdo que pode manipular usando um cabeçalho Accept, conforme descrito anteriormente nestas diretrizes. Frequentemente, a finalidade do cabeçalho *Accept* é permitir que o aplicativo cliente especifique se o corpo da resposta deve ser XML, JSON ou algum outro formato comum que o cliente possa analisar. No entanto, é possível definir tipos de mídia personalizados que incluem informações, permitindo que o aplicativo cliente indique qual versão de um recurso esse aplicativo está esperando.

O exemplo a seguir mostra uma solicitação que especifica um cabeçalho *Accept* com o valor *application/vnd.adventure-works.v1+json*. O elemento *vnd.adventure-works.v1*

indica ao servidor Web que ele deve retornar a versão 1 do recurso, enquanto o elemento *json* especifica que o formato do corpo da resposta deve ser JSON:

```
HTTP

GET https://adventure-works.com/customers/3 HTTP/1.1
Accept: application/vnd.adventure-works.v1+json
```

O código que processa a solicitação é responsável por processar o cabeçalho *Accept* e honrá-lo o máximo possível (o aplicativo cliente pode especificar vários formatos no cabeçalho *Accept*; nesse caso, o servidor Web pode escolher o formato mais apropriado para o corpo de resposta). O servidor Web confirma o formato dos dados no corpo da resposta usando o cabeçalho Content-Type:

```
HTTP

HTTP/1.1 200 OK
Content-Type: application/vnd.adventure-works.v1+json; charset=utf-8

{"id":3,"name":"Contoso LLC","address":"1 Microsoft Way Redmond WA 98053"}
```

Se o cabeçalho Accept não especificar nenhum tipo de mídia conhecido, o servidor Web pode gerar uma mensagem de resposta HTTP 406 (Não Aceitável) ou retornar uma mensagem com um tipo de mídia padrão.

Essa abordagem é possivelmente o mais puro dos mecanismos de controle de versão e aplica-se naturalmente a HATEOAS, que pode incluir o tipo MIME dos dados relacionados em links de recursos.

> **Observação**
>
> Quando você seleciona uma estratégia de controle de versão, você também deve considerar as implicações de desempenho, especialmente no armazenamento em cache no servidor Web. O controle de versão do URI e esquemas de controle de versão de cadeia de consulta facilitam o armazenamento em cache na medida em que, a cada vez, a mesma combinação de URI/cadeia de consulta refere-se aos mesmos dados.
>
> Os mecanismos de controle de versão do cabeçalho e do tipo de mídia, normalmente, exigem lógica adicional para examinar os valores no cabeçalho personalizado ou no cabeçalho Accept. Em um ambiente de grande escala, muitos clientes usando versões diferentes de uma API da Web podem resultar em uma quantidade significativa de dados duplicados em um cache do lado do servidor.

Esse problema pode se tornar importante se um aplicativo cliente se comunica com um servidor Web através de um proxy que implementa caching, e que encaminha uma solicitação ao servidor Web somente se ele não mantém atualmente uma cópia dos dados solicitados em seu cache.

Abrir Iniciativa de API

A Open API Initiative foi criada por um consórcio do setor para padronizar descrições de API REST entre fornecedores. Como parte dessa iniciativa, a especificação de Swagger 2.0 foi renomeada como OAS (OpenAPI Specification) e colocada sob a iniciativa de API aberta.

Talvez você queira adotar a OpenAPI para suas APIs Web. Considere o seguinte:

- A OpenAPI Specification vem com um conjunto de diretrizes insistentes sobre como uma API REST deve ser criada. Isso apresenta vantagens para a interoperabilidade, mas requer mais cuidado durante o design de sua API para ficar de acordo com a especificação.
- A OpenAPI promove uma abordagem de primeiro contrato, em vez de uma primeira abordagem de implementação. Primeiro contato significa que você projeta o contrato da API (a interface) primeiro e então escreve o código que implementa o contrato.
- Ferramentas como o Swagger podem gerar bibliotecas de cliente ou a documentação de contratos de API. Por exemplo, veja as Páginas da ajuda da ASP.NET Web API usando o Swagger.

Próximas etapas

- Diretrizes da API REST do Microsoft Azure. Recomendações detalhadas para a criação de APIs REST no Azure.
- Lista de verificação da API da Web. Uma lista útil de itens a serem considerados ao projetar e implementar uma API da Web.
- Open API Initiative. Documentação e detalhes de implementação sobre a Open API.

Comentários

Esta página foi útil? Yes No