API RESTful - Boas práticas

BY BRUNO BRITO ON 26 DE JANEIRO DE 2020

READ IN 9 MIN

Sua API faz update na chamada de um GET? O verbo DELETE nunca nem foi utilizado? Nesse artigo vamos falar de algumas boas práticas para API's RESTFul

O **objetivo** de uma API RESTFul é o mesmo de um **Design Pattern**.
Um **Design Pattern** fornece uma solução *reutilizável* para problemas comuns. A ideia é acelerar o processo de desenvolvimento através de um paradigma de desenvolvimento bem testado e adotado.

Portanto o padrão REST visa padronizar suas API's. Tornando a utilização e resposta dos recursos homogêneo. Economizando esforço e aprendizado dos desenvolvedores que consomem a API.

HTTP do jeito certo.

Os princípios do REST envolvem **resources** e cada um deve possuir uma URI única.
Esses **resources** são manipulados usando solicitações HTTP. E cada método (GET, POST, PUT, PATCH, DELETE) tem um significado específico.

HTTP Method	Idempotente	Safe
OPTIONS		
GET		
HEAD		
PUT		
POST		
DELETE		
PATCH		

Idempotente significa que o endpoint pode ser chamado várias vezes sem resultados diferentes. Não importa se o método é chamado apenas uma ou dez vezes. O resultado deve sempre o mesmo. Isso se aplica apenas ao resultado, não ao próprio recurso.

Safe são métodos Read-only . Isso significa que são seguros e não importa quantas vezes chamar, ele não irá alterar o estado do **resource**.

HTTP Method	Descrição
OPTIONS	Retorna os verbos http de um resource e outras opções, como CORS, por exemplo.
GET	Busca um resource
HEAD	Busca apenas o header de um resource
PUT	Atualiza um resource
POST	Cria um resource
DELETE	Remove um resource
PATCH	Atualiza parcialmente um resource

GET

Um método GET é Safe e Idempotente. Por definição o GET jamas deveria criar, atualizar ou deletar um recurso. O resultado do GET sempre será o mesmo para um determinado conjunto de informações.

PUT

O PUT pode atualizar um recurso, mas o retorno do Servidor deve ser sempre igual. Independente de quantas veze for feito a mesma chamada. Por exemplo, considere uma classe que possui o atributo nome:

```
{
    "nome": "Maria"
}
```

Ao chamar o **PUT** solicitando que o nome seja alterado para `João`, na primeira vez o nome é alterado. Retorna `200 Ok` do servidor. Nesse caso o recurso foi atualizado. Logo não é uma chamada **Safe**. No entanto a partir da segunda chamada em diante, toda vez que for feito a solicitação para o nome ser `João`, o recurso não é alterado, porém seu retorno deve ser sempre 200 Ok.

DELETE

O **DELETE** segue o mesmo princípio do **PUT**.

POST

Já o **POST** é o verbo mais sensível. Toda vez que é chamado um recurso pode ser criado. E se chamar 1000x o endpoint vai resultar em mil novos recursos similares. Por isso ele não é Safe. Ele também não é idempotente, pois o **POST** pode retornar parâmetros diferente na resposta. Por exemplo o parâmetro **Location** no Header da resposta. Esse parâmetro contém a URI onde pode ser localizado o recurso criado.

PATCH

Se o patch é similar ao PUT, por que ele é idempotente? É uma questão de definição. O **PATCH** não é Idempotente, enquanto o **PUT** é. Nesse caso o patch pode ser utilizado em uma chamada de atualização que tem resultados diferentes.

Por exemplo,

```
 [
    {"operation": "replace", "field": "email", "value":
"bruno@gmail.com"}
 ]
```

Na primeira chamada há uma integração com a Fábrica de crachá e retorna um `200 Ok`. Na segunda chamada, há uma limitação que não pode alterar um recurso duas vezes em menos de 24hr. Tendo como resultado `400 Bad Request`.

Endpoints

Padrão de nomenclatura - Algumas dicas para padrão de nomenclatura.

- Dê preferência para o plural ao disponibilizar o **resource**. Utilize `/users` ao invés de `/user`.
- Se um recurso possui um nome composto utilize o **kebab-case**. Por exemplo Global Configuration seria `GET /global-configuration`.
- Dê preferência para URL's em minúsculo, evite `GET /Users`, use `GET /users`.

A raiz do resource deve retornar uma coleção. Por exemplo `/users` deve retornar um lista de usuários.
Se desejar obter um **resource** especifico utilize o nível seguinte especificando seu identificador único. GET `/users/2`. Não precisa ser o id do banco, poderia ser outro campo, desde que seja identificador único. Um usuário poderia ser o username.

- `GET /users` -> Retorna uma lista de usuários
- `GET /users/bruno` -> Retorna o usuário com username bruno
- `POST /users` -> Cria um usuário
- `PUT /users/bruno` -> Atualiza o usuário bruno
- `PATCH /users/bruno` -> Atualiza parcialmente o usuário bruno
- `DELETE /users/bruno` -> Remove o usuário bruno

Relacionamento filho - Se existir alguma tabela filho do **resource**, eles devem estar mapeados para o mesmo endpoint. Exemplo

- `GET /users/bruno/claims` -> Retorna uma lista de claims do usuário bruno
- `GET /users/bruno/claims/6` -> Retorna o claims com Id 6
- `POST /users/bruno/claims` -> Cria uma claim para usuário bruno
- `PUT /users/bruno/claims/6` -> Atualiza a claim 6 do usuário bruno
- `PATCH /users/bruno/claims/6` -> Atualiza parcialmente a claim 6 do usuário bruno
- `DELETE /users/bruno/claims/6` -> Remove a claim 6 do usuário bruno

Há situações de relacionamento que não seguem essa hierarquia. Por exemplo, **departamento**. Um usuário pertence a um departamento, mas no design pode fazer sentido
tanto **departamento** quanto **usuário** terem URI diferentes.

Ações

Os exemplos acimas, são perfeitos exemplos de CRUD. No mundo real, os **resources** podem ter comportamento.

Considere o endpoint `/candidatos/{id}`. Agora é necessário atualizar o status do candidato para Aprovado ou Recusado. Dentro de um cenário CRUD, duas abordagens.

1. Reestruturar o status para ser um **resource**, por exemplo `PUT /candidato/{id}`.
2. Disponibilizar um PATCH para que apenas o campo Status seja atualizado, `PATCH /candidato/{id}`

Em ambos cenários há um **grande** problema. O backend vai precisar entender a intenção da ação. Em ambos tanto o **Status** quanto o **Nome** do candidato poderia ser **Atualizado**. Porém cada situação pode haver desdobramentos de negócio diferente.

O fluxo de aprovação do candidato poderia envolver outras etapas, como enviar e-mail para gestor da área.
Já uma atualização de grafia no nome, pode ter uma integração com a gráfica para reemitir o crachá do candidato.

Veja que esses endpoints não traduzem a complexidade do negócio.

Uma outra opção seria criar um URI, `PUT /candidato/{id}/status`. Ainda assim quebra certos fundamentos de design. No DDD, por exemplo, a Linguagem Obíqua diz que não deve haver traduções. Logo atualizar um Status (campo do banco de dados) é diferente de "Aprovar" / "Recusar" o candidato. A comunicação deve ser clara.

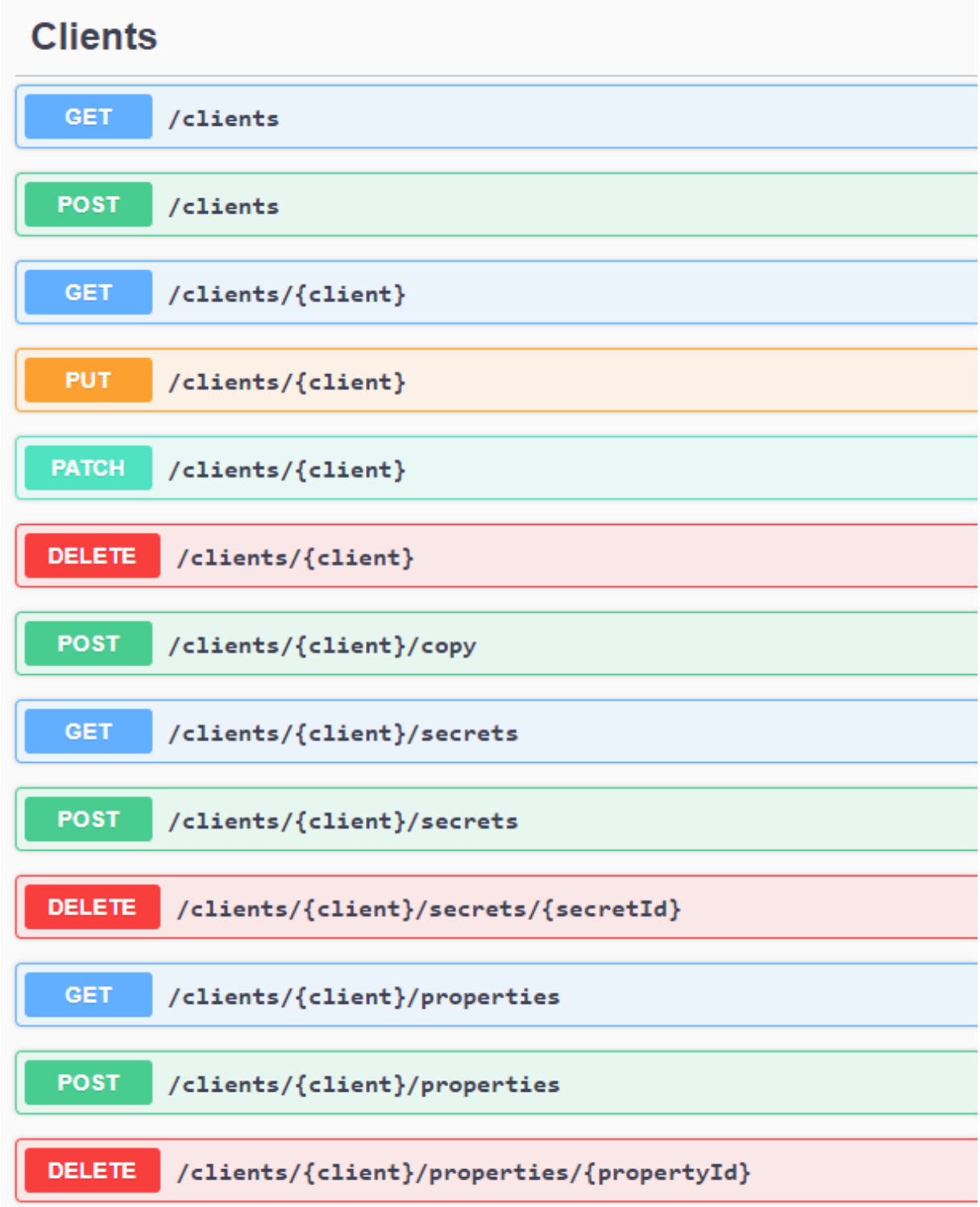
Logo ações se fazem necessários no dia-a-dia. E é importante reforçar que **não fere os padrões REST**.

```
POST /candidato/{id}/candidatar -- Grava a aplicação de um
candidato a uma vaga
POST /candidato/{id}/aprovar -- Aprova um candidato
DELETE /candidato/{id}/aprovar -- Cancela a aprovação
POST /candidato/{id}/reprovar -- Reprova o candidato
DELETE /candidato/{id}/reprovar -- Cancela a reprovação
POST /candidato/{id}/transferir -- Transfere o candidato para
outra vaga
DELETE /candidato/{id}/transferir -- Cancela a transferência
```

Nesse formato a comunicação é clara. No backend o desenvolvedor não vai precisar criar if's ou switchs e traduzir a intenção do método "CRUD". O próprio endpoint comunica sua intenção.

Documentação - Open API

O Open Api, anteriormente conhecido como Swagger, é hoje o padrão mais utilizado para documentar API's. No ASP.NET Core é muito fácil configurar. Ele disponibiliza mecanismos para testar sua API no Browser.



Hoje o Open API é quase um requisito obrigatório para qualquer API.

Pesquisas

É muito comum ver times que criam classes de pesquisas e enviam as opções da pesquisa por POST. Na teoria um POST é idempotente, isso indica que haverá mudança de estado no server. Não é safe, ou seja, múltiplas chamadas resultam em resultados diferente.
Pelas características é o verbo ideal para criar objetos. Por isso, a associação POST -> INSERT.

Use o GET e permita filtros para buscar o recurso.

- `GET /users` -> Busca uma lista de usuários
- `GET /users/{id}` -> Busca um único usuário

Filtrando

Crie parâmetros para que o GET filtre os dados.

- `GET /users?status=active&older_than=30`

Utilize Alias

Em certas situações a pesquisa pode ser muito complexa, por exemplo, `GET /users?status=active&lives_in=blazil&older_than=18&younger_than=35`.

Ao invés de utilizar todos esses parâmetros, faça um alias, `GET /users/young-brazilians`.

Ordenação

Permita ao usuário selecionar os campos que serão ordenados. Utilizando + (mais) ou -(menos) na frente do campo para indicar o tipo de ordenação, ascendente ou descendente. Sendo que o + é **opcional**.

- `GET /users?sort=firstname,-last_login`

Nesse exemplo, **firstname** será ordenado em **asc** e **last_login** em **desc**.

Paginação

Disponibilize a paginação para o usuário. Particularmente gosto da nomenclatura do SQL. **Limit** e **Offset**ou **Skip** e **Take**.

- `GET /users?limit=10&offset=20` -> Pula os 20 primeiros resultados e busca os 10 seguintes.

Respostas

Quando o servidor recebe uma solicitação HTTP, o cliente deve saber o resultado da ação. Se houve sucesso ou falhou. Os códigos de status HTTP são vários códigos padronizados, com várias explicações em vários cenários. O servidor sempre deve retornar o código de status correto.

A seguir estão as categorias dos códigos HTTP:

- 2xx - Status de sucesso
- 3xx - Categoria de redirecionamento
- 4xx - Erro no Cliente
- 5xx - Erro no server

2xx - Sucesso

Abaixo uma tabela de quando utilizar cada um dos Status Code.

- 200 Ok - Padrão que representa sucesso. Resposta padrão para GET, quando há resultados.
- 201 Created - Indica que um recurso foi criado. Utilize para a resposta do POST. E também retorne o **Header** `Location` indicando o GET dessa informação.
- 204 No Content - Representa que a request foi processada com **sucesso**, mas não há conteúdo para ser retornado. Utilize para PUT, PATCH e DELETE. E também para situações em que o GET não retornou resultados.
- 202 Accepted - Indica que o servidor aceitou a request. Esse é um ótimo Status para processos assíncronos. Por exemplo um cenário onde o usuário faz upload de uma planilha que será processada em background.

3xx - Redirect

Como desenvolvedor, não consigo visualizar cenários em que será utilizado o redirect a partir de uma API RESTFul.

Em geral o 302 é bastante utilizado em desenvolvimento Frontend, como Razor. E o **301 Moved Permanently** é utilizado por sysadmins, para indicar que um site foi permanentemente redirecionado para um novo endereço.

4xx - Client error

Indica que o client cometeu uma falha na requisição.

- 400 Bad Request - A solicitação não foi processada, pois o servidor não entendeu o que o cliente está solicitando.
- 401 Unauthorized - Indica que o client não está autenticado e não tem autorização para acessar o recurso.
- 403 Forbidden - Indica que o Client está **autenticado** e a requisição é **válida**. Porém o client não tem permissão de acesso naquele recurso.
- 404 Not Found - indica que o recurso não foi localizado.

5xx - Erros no servidor

Os status 5xx há dois que merecem destaque. É o erro 500, houve uma falha no servidor e não conseguiu processar a requisição. E o erro 503 que vai botar o pessoal de infra para correr. Indica que o serviço está indisponível. Status comum quando o Webserver sobrecarrega de requisições e não consegue mais processar **nenhuma** requisição. O **503** é comum em cenários de ataques **DDoS**.

Utilizar ProblemDetails

Existe a RFC 7807 que define um formato padrão de respostas de erro. Se chama ProblemDetails. O RFC diz que Problem Details é uma maneira de transportar detalhes legíveis em uma resposta HTTP para evitar a necessidade de definir novos formatos de resposta a APIs HTTP.

O ponto central desse POST é consistência na criação de APIs. Ter respostas consistentes em situações semelhantes é crucial para uma API ser fácil de utilizar e previsivel.

> This specification defines simple JSON [RFC7159] and XML [W3C.REC-xml-20081126] document formats to suit this purpose. They are designed to be reused by HTTP APIs, which can identify distinct "problem types" specific to their needs.

JSON

Em API's expostas na internet, dê preferência para o formato JSON de resposta. O grande problema do XML e SOAP está no tamanho da resposta. Ele é verboso, difícil entender. O GRPC com HTTP/2 e protobuf é uma alternativa mais performática. Mas ainda está evoluindo os componentes e a adoção. Fique de olho que em breve ele vai substituir o JSON.

propriedades null

Não envie propriedades null na resposta do JSON, só aumenta a quantidade de dados enviados de maneira desnecessária, poderia dizer que todos componentes que trabalham com JSOn sabem lidar com respostas que não possuem todos os campos, ainda que null.

HATEOAS

A teoria diz que para uma API ser RESTFul deve implementar todos os padrões REST. Que inclui o HATEOAS. A web funciona com HATEOAS, basta entrar em um site e ali estará todos os links que você vai navegar. Porém no contexto de API, ainda parece impraticável. Pode ser que mude minha opinião no futuro.

Conclusão

Aqui neste post descrevo as boas práticas que venho adotando, que aprendi com passar dos anos e experiência. Atualmente não há uma RFC ou um guia Oficial com as melhores práticas.

Assim como eu, deixo abaixo outras referências que também compactuam com as boas práticas descritas aqui.

Deixe seu comentário!

Referências

- REST in Practice
- RESTful API Designing guidelines — The best practices
- RESTful API Design — Step By Step Guide
- Best Practices for Designing a Pragmatic RESTful API
- REST: Good Practices for API Design
- RFC 7807