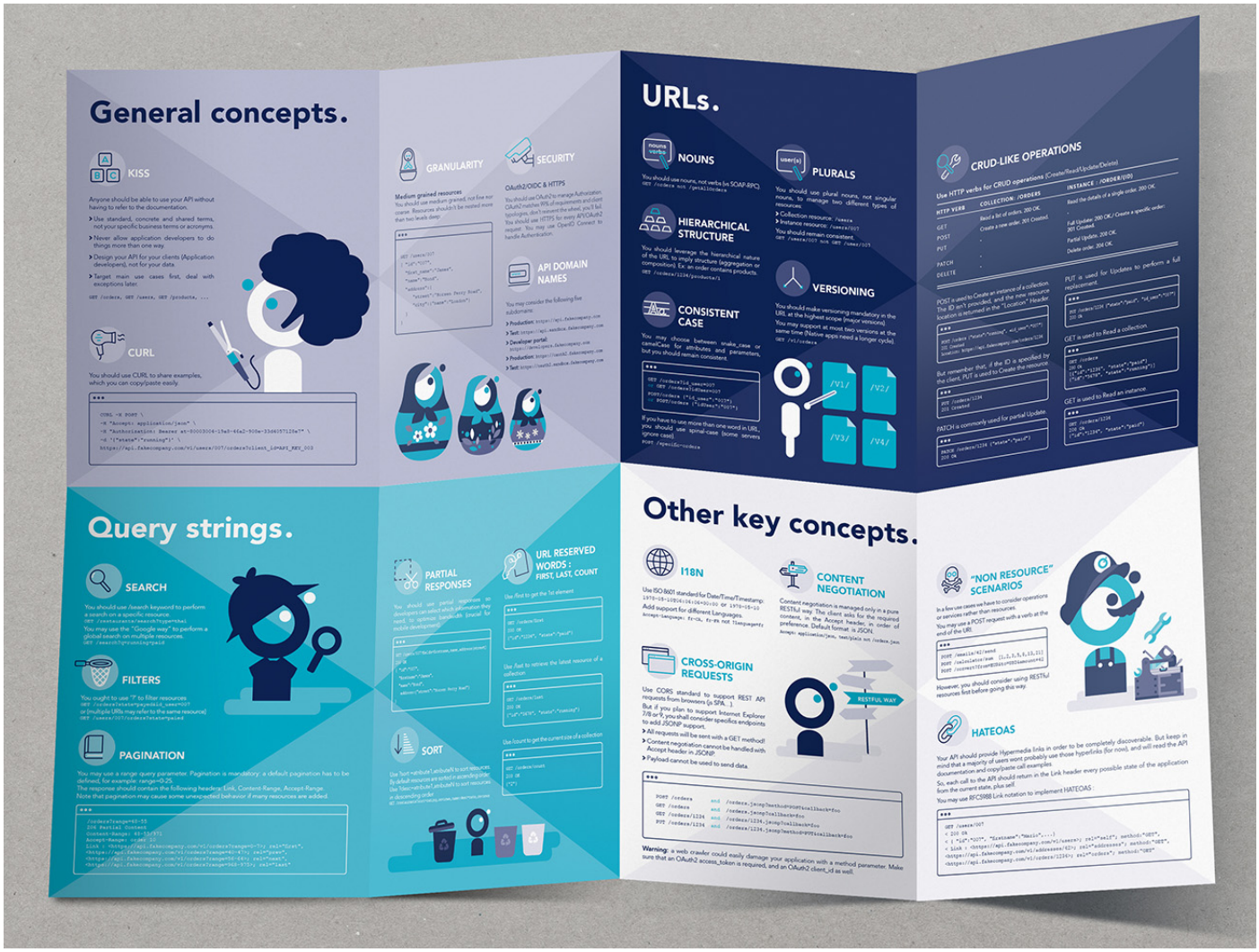
Projetando uma API REST

Postado em 08/12/2015 por *Antoine Chantalou, Augustin Grimprel, Benoit Lafontaine, Florent Jaby, Jérémy Buisson, Mohamed Kissa, Nicolas Laurent, Sergio Fernandes*

Bom, se você está na correria, use logo o nosso "Quick Reference Card", um resumo das boas práticas para APIs REST.

Faça o download do *API Design – Cartão de Referência Rápida*

Com mais tempo, você pode ler o nosso artigo, que mostra ponto a ponto todos as questões do "Guia de Referência". INTERESSADO NO ASSUNTO? CONTACTA-NOS!



- Query strings
  - Paginação
  - Filtros
  - Ordenação
  - Busca
  - Busca nos recursos
  - Busca global

- Outros conceitos importantes
  - Negociação de conteúdo
  - Cross-domain
  - CORS
  - Jsonp
  - HATEOAS
  - Cenários "Sem Recursos"
  - Erros HTTP

Introdução

Logo que se começa a desenvolver uma API, surge a questão: "como se projeta uma API?". Queremos fazer APIs robustas e bem projetadas. Sabemos que APIs mal projetadas dificultam o uso, ou mesmo são rejeitadas pelos seus clientes exigentes: os *desenvolvedores de aplicativos*.

Para que a sua API seja uma obra prima, é preciso levar em conta:

- Os princípios de APIs RESTful descritos na literatura (Roy Fielding, Leonard Richardson, Martin Fowler, HTTP specification, etc.)
- As práticas dos "*Gigantes da Web*"

Geralmente encontramos duas posições opostas: os "*puristas*", que insistem em seguir os princípios REST sem concessões, e os "*pragmáticos*", que preferem uma abordagem mais prática, para dar aos clientes uma API mais fácil de usar. O melhor caminho é o do meio.

Projetar uma API REST gera questões e problemas para os quais não há unanimidade. As boas práticas REST ainda estão sendo debatidas e consolidadas, o que torna esse trabalho mais interessante.

Para facilitar e acelerar o projeto e desenvolvimento das suas APIs, nós compartilhamos nossa visão e nossa experiência em projetos de APIs.

AVISO: Esse artigo é um apanhado de boas práticas com o objetivo de serem discutidas. Você está convidado a *discutir e questionar* no nosso blog.

Conceitos Gerais

KISS – « Keep it simple, stupid »

Abrir uma API na Internet é uma estratégia que tem como objetivo atingir o maior número possível de desenvolvedores. Por isso é crítico que a API seja auto-explicativa, e o mais simples possível, para que os desenvolvedores raramente precisem ler a documentação. Isso é o conceito de *affordance*, a capacidade da API sugerir sua própria utilização.

Na hora de projetar uma API, tenha em mente os seguintes princípios:

- A semântica da API deve ser intuitiva. Seja URI, payload, request ou response, o desenvolvedor deve ser capaz de usá-los sem olhar muito a documentação da API.
- Os termos utilizados devem ser comuns e concretos, evitando termos funcionais ou jargões técnicos. Ex.: *clientes, pedidos, endereços, produtos*.
- Não deve existir mais de uma maneira de se obter um mesmo resultado.

Qual liquidificador é mais fácil de usar?

- **A API é projetada para seus clientes, os desenvolvedores**, e não deve ser um apenas um acesso ao modelo de dados. A API deve prover funcionalidades simples que casam com as necessidades dos desenvolvedores. Um erro comum é projetar a API baseada no modelo de dados, quase sempre complexo.

Qual liquidificador é mais simples?

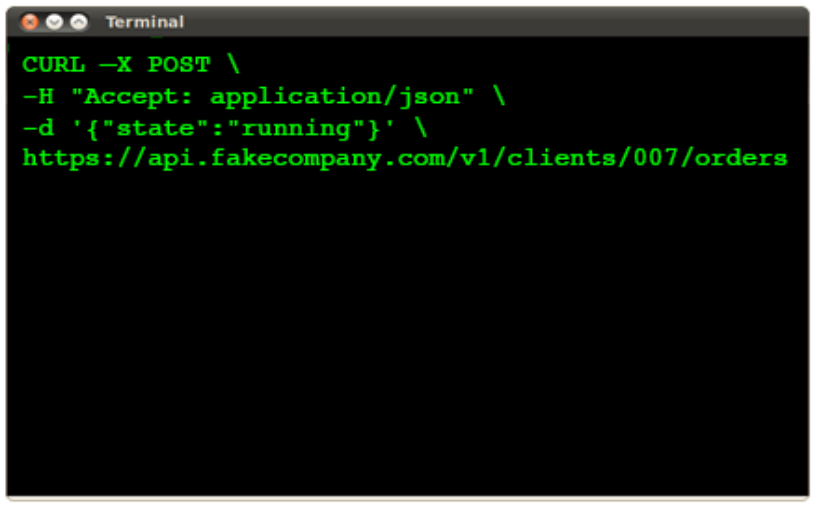
- Durante a fase de concepção, é melhor focar nos principais *casos de uso*, e deixar os casos excepcionais para depois.

Exemplos de cURL

Exemplos de *cURL* são muito usados para ilustrar chamadas à API: os Gigantes da Web fazem isso, assim como a literatura técnica em geral:

- https://developer.github.com/v3/
- https://developers.google.com/youtube/v3/live/authentication#client-side-apps
- https://developer.paypal.com/docs/api/
- https://developers.facebook.com/docs/graph-api/making-multiple-requests
- https://www.dropbox.com/developers/blog/45/using-oauth-20-with-the-core-api
- http://instagram.com/developer/endpoints/likes/
- http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AESDG-chapter-instancedata.html
- etc.

Nós recomendamos que você sempre ilustre a sua documentação de API com exemplos cURL, que permitem ao desenvolvedor *copiar e colar*, e adaptar para acelerar sua utilização.



Exemplo

```
CURL -X POST \
-H "Accept: application/json" \
-d '{"state":"running"}' \
https://api.fakecompany.com/v1/clients/007/orders
```

Granularidade

Apesar da regra "um recurso = uma URL", consideramos importante manter o número de recursos (e de URLs) num limite *razoável*.

Por exemplo: uma pessoa tem um endereço que contém um país.

É importante evitar de fazer 3 chamadas à API:

```
1  CURL https://api.fakecompany.com/v1/users/1234
2  &lt; 200 OK
3  &lt; {"id":"1234", "name":"Antoine Jaby", "address":"https://api.fakecompany.co
4  m/v1/addresses/4567"}CURL https://api.fakecompany.com/addresses/4567
5  &lt; 200 OK
6  &lt; {"id":"4567", "street":"sunset bd", "country": "http://api.fakecompany.co
7  m/v1/countries/98"}CURL https://api.fakecompany.com/v1/countries/98
   &lt; 200 OK
   &lt; {"id":"98", "name":"France"}
```

Principalmente se essas 3 informações forem geralmente usadas juntas. Isso pode levar à problemas de performance.

Por outro lado, se acumularmos muitos dados a *priori*, podemos criar uma verbosidade desnecessária.

Projetar uma API com a granularidade ideal não é tarefa fácil. Depende de uma cultura, e de alguma experiência anterior em APIs. Na dúvida, tente evitar operações muito *grandes* ou muito *específicas*.

Na prática, recomendamos:

- Agrupar somente os recursos que quase sempre serão usados juntos
- Não agrupar coleções que possam ter muitos componentes. Por exemplo, uma lista de empregos atuais de um usuário é limitada (ninguém tem mais que 2 ou 3 empregos ao mesmo tempo), mas uma lista dos empregos anteriores pode ser muito longa.
- Ter no máximo 2 níveis de objetos aninhados. Ex.: /v1/users/addresses/countries

Nomes de Domínio da API

No caso dos nomes de domínio, os Gigantes da Web tem práticas heterogêneas. Alguns, como o *Dropbox,* usam vários domínios ou subdomínios para suas APIs.

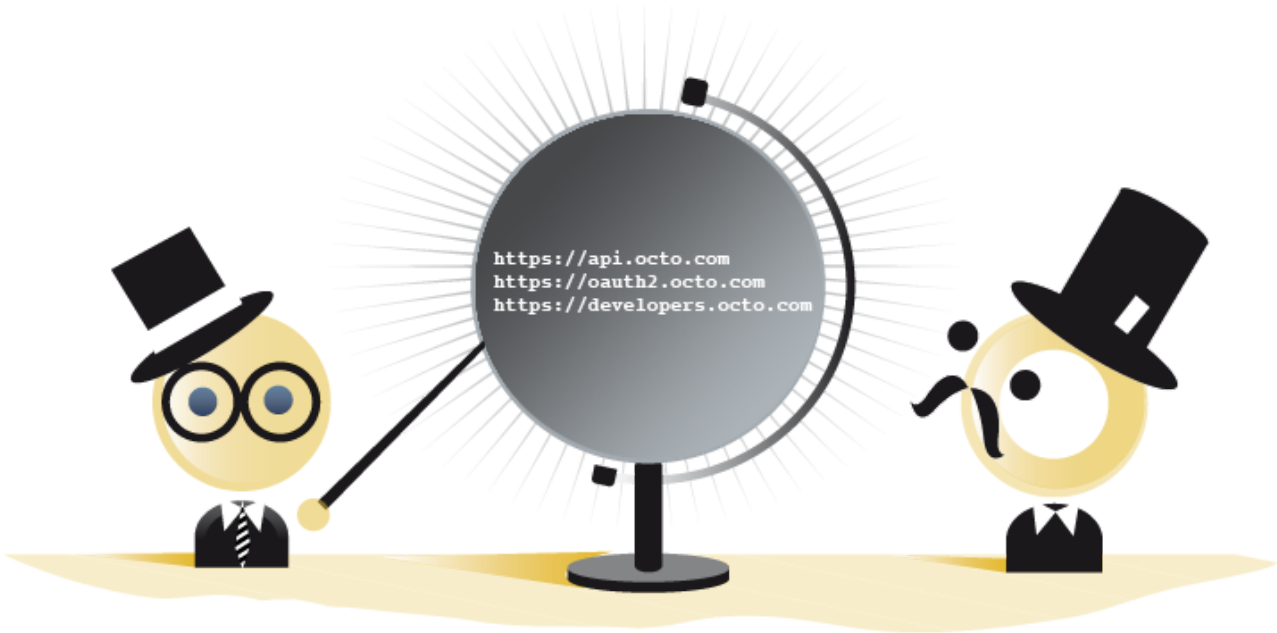
 Exemplos dos Gigantes da Web

API	Domínio / Subdomínio	URI
Google	https://accounts.google.com https://www.googleapis.com https://developers.google.com	https://accounts.google.com/o/oauth2/auth https://www.googleapis.com/oauth2/v1/tokeninfo https://www.googleapis.com/calendar/v3/ https://www.googleapis.com/drive/v2 https://maps.googleapis.com/maps/api/js?v=3.exp https://www.googleapis.com/plus/v1/ https://www.googleapis.com/youtube/v3/ https://developers.google.com
Facebook	https://www.facebook.com https://graph.facebook.com https://developers.facebook.com	https://www.facebook.com/dialog/oauth https://graph.facebook.com/me https://graph.facebook.com/v2.0/{achievement-id} https://graph.facebook.com/v2.0/{comment-id} https://graph.facebook.com/act_{ad_account_id}/adgroups https://developers.facebook.com
Twitter	https://api.twitter.com https://stream.twitter.com https://dev.twitter.com	https://api.twitter.com/oauth/authorize https://api.twitter.com/1.1/statuses/show.json https://stream.twitter.com/1.1/statuses/sample.json https://dev.twitter.com
GitHub	https://github.com https://api.github.com https://developer.github.com	https://github.com/login/oauth/authorize https://api.github.com/repos/octocat/Hello-World/git/commits/7638417db6d59f3c431d3e1f261cc637 https://developer.github.com
Dropbox	https://www.dropbox.com https://api.dropbox.com https://api-content.dropbox.com https://api-notify.dropbox.com	https://www.dropbox.com/1/oauth2/authorize https://api.dropbox.com/1/account/info https://api-content.dropbox.com/1/files/auto/ https://api-notify.dropbox.com/1/longpoll_delta https://api-content.dropbox.com/1/thumbnails/auto/ https://www.dropbox.com/developers
Instagram	https://api.instagram.com http://instagram.com	https://api.instagram.com/oauth/authorize/ https://api.instagram.com/v1/media/popular http://instagram.com/developer/

Foursquare	https://foursquare.com	https://foursquare.com/oauth2/authenticate
	https://api.foursquare.com	https://api.foursquare.com/v2/venues/40a55d80f964a5202
	https://developer.foursquare.com	https://developer.foursquare.com

Para normalizar os nomes de domínio e manter o conceito de affordance, nós recomendamos usar somente 3 subdomínios para seu ambiente de *produção*:

- API – https://api.{fakecompany}.com
- OAuth2 – https://oauth2.{fakecompany}.com
- Portal de desenvolvedores – https://developers.{fakecompany}.com



Essa divisão ajuda os desenvolvedores a entender rápida e intuitivamente como:

1. Chamar a API
2. Conseguir um token OAuth2 para chamar a API
3. Acessar o portal de desenvolvimento

Alguns Gigantes da Web, como *Paypal,* oferecem um ambiente de *sandbox,* para que os desenvolvedores testem a API antes de usá-la em *produção*.

https://developer.paypal.com/docs/api/

Nós recomendamos usar 2 subdomínios para seu ambiente sandbox :

- OAuth2 – https://oauth2.sandbox.{fakecompany}.com
- API – https://api.sandbox.{fakecompany}.com

Segurança

Dois protocolos são muito utilizados para a segurança de APIs REST:

- OAuth1 : http://tools.ietf.org/html/rfc5849
- OAuth2 : http://tools.ietf.org/html/rfc6749

O que os Gigantes da Web e grandes corporações usam?

OAuth1	OAuth2
Twitter, Yahoo, flickr, tumblr, Netflix, myspace, Evernote, etc.	*Google, Facebook, Dropbox, GitHub, Amazon, Instagram, LinkedIn, Foursquare, Salesforce, Viadeo, Deezer, Paypal, Stripe, Huddle, Boc, Basecamp, Bitly, etc.*
MasterCard, CA-Store, OpenBankProject, Intuit, etc.	*AXA Banque, Bouygues telecom, etc.*

Para a segurança de sua API, nós recomendamos o uso de OAuth2.

- Ao contrário do OAuth1, o OAuth2 permite gerar autenticação para todos os tipos de aplicativos (app móvel nativa, nativo de tablet, aplicação javascript, batch/back-office, etc.) com ou sem o consentimento do proprietário dos recursos.
- OAuth2 é o padrão *absoluto* para segurança de APIs. Usar outra tecnologia pode atrasar o desenvolvimento e a adoção da sua API.
- Finalmente, a segurança dos recursos é um assunto complexo, e uma solução feita em casa vai provavelmente ter falhas de segurança.

Nossa recomendação é de implementar a solução da Google para o fluxo de validação de token no OAuth2, o *implicit grant flow*:

- https://developers.google.com/accounts/docs/OAuth2UserAgent#validatetoken
- http://en.wikipedia.org/wiki/Confused_deputy_problem

Nós recomendamos utilizar sempre o protocolo HTTPS quando se comunicar:

- com provedores OAuth2
- com provedores de API

Para validar a sua implementação OAuth2, faça o seguinte teste:

- Desenvolva um cliente para consumir sua implementação OAuth2 e fazer uma chamada à sua API
- Depois troque os nomes de domínio da sua API, pela API do Google

Se funcionar, então está tudo bem :)

URIs

Nomes > verbos

Para descrever seus recursos, nós recomendamos que você use nomes, nunca verbos.

Por décadas os desenvolvedores usaram verbos para expor serviços no modelo RPC, como por exemplo:

- getClient(1)
- createClient(1)
- updateAccountBalance(1)
- addProductToOrder(1)
- deleteAddress(1)

Mas no mundo RESTful, seria assim:

- GET /clients/1

- POST /clients
- PATCH /accounts/1
- PUT /orders/1
- DELETE /addresses/1

Um dos pontos principais de uma API REST é usar HTTP como protocolo da aplicação. Ele dá consistência, e falicita a interação entre os sistemas de informação. Ele também nos salva de tentar inventar a roda com protocolos caseiros "tipo" SOAP/RPC/EJB.

Então devemos sempre usar os verbos HTTP para descrever todas as operações feitas sobre os recursos (veja tópico CRUD)

O uso dos verbos HTTP também torna a API mais intuitiva, e faz com que os desenvolvedores entendam como manipular os recursos sem precisar olhar uma longa documentação, aumentando o "affordance" da sua API.

Na prática, o desenvolvedor vai usar ferramentas que vão gerar requisições HTTP, com os verbos e payloads corretos, a partir de um modelo de dados, desde que esse esteja atualizado.

Plural > singular

A maior parte do tempo, os Gigantes da Web mantém a consistência em relação aos nomes de recursos seja no plural ou no singular. De fato, variar os nomes de recursos entre plural e singular diminui a navegabilidade da API.

Os nomes de recursos parecem mais naturais no plural, de modo a acessar de forma consistente *coleções* e *instâncias* dos recursos.

Por isso, nós recomendamos o plural para 2 tipos de recursos:

- Coleções de recursos: /v1/users
- Instância de recurso: /v1/users/007

Como exemplo, consideramos que *POST /v1/users* é a chamada para criar um usuário dentro da coleção de usuários. Da mesma forma, *GET /v1/users/007* pode ser entendido como "eu quero o usuário 007, da coleção de usuários".

Consistência de caixa

Caixa das URIs

Quando se trata de definir nomes para os recursos em um programa, temos basicamente 3 possibilidades: CamelCase, snake_case e spinal-case. Elas são maneiras de criar nomes que se pareçam com a linguagem natural, evitando espaços e caracteres exóticos. Essa prática é comum nas linguagens de programação, onde existe uma limitação dos caracteres que podem ser usados nomes.

- CamelCase: se popularizou pela linguagem Java. A idéia é destacar o início das palavras colocando a primeira letra de cada uma em caixa alta. Ex.: CamelCase, CurrentUser, AddAttributeToGroup, etc. Além dos debates de sua legibilidade, seu maior problema é que não funciona em contextos que não são sensíveis à caixa. Existem duas variantes:
  - **lowerCamelCase:** quando a primeira letra é minúscula.
  - **UpperCamelCase:** quando a primeira letra é maiúscula.

- snake_case: foi amplamente utilizado por programadores C, e mais recentemente em Ruby. As palavras são separadas por underscores "_", permitindo um compilador ou interpretador entendê-lo como símbolo, mas permitindo também que os leitores separem quase naturalmente as palavras. Ao contrário do CamelCase, existem poucos contextos onde o snake_case não pode ser utilizado. Exemplos: snake_case, current_user, add_attribute_to_group, etc.
- spinal-case: é uma variação do snake_case, que usa o hífem "-" para separar as palavras. As vantagens e desvantagens são similares ao snake_case, com exceção que algumas linguagens não permitem hífens em nomes de símbolos (para nomes de variáveis, classes ou funções). É ainda chamado às vezes de lisp-case, porque é a forma usual de nomear funções em dialetos LISP. Também é a forma usual de nomear pastas e arquivos em sistemas UNIX ou LINUX. Exemplos: spinal-case, current-user, add-attribute-to-group, etc.

Esses 3 tipos de nomenclaturas tem outras variações em função da caixa da primeira letra ou do tratamento dado aos caracteres especiais. É recomendável ficar no inglês básico, sem usar caracteres especiais ou acentuação (*ASCII*).

De acordo com a RCF3986, as URLs são sensíveis à caixa (exceto para o *scheme* e o *host*). Mas na prática, usar URLs sensíveis à caixa pode criar problemas para APIs hospedadas em sistemas Windows.

Segue uma compilação das práticas dos Gigantes da Web:

	Google	Facebook	Twitter	Paypal	Amazon	dropbox	github
snake_case	x	x	x			x	x
spinal-case	x			x			
camelCase	x						

Para URIs, nós recomendamos que seja utilizada, de forma consistente, uma das duas formas a seguir:

- spinal-case (destacada na RFC3986)
- e **snake_case** (mais usada pelos Gigantes da Web)

 Exemplos

```
1  POST /v1/specific-orders
```

ou

```
1  POST /v1/specific_orders
```

Caixa do "body"

Existem dois formatos principais para o corpo (body) dos dados.

Por um lado, o snake_case é muito mais usado pelos Gigantes da Web, e, principalmente, foi adotado na especificação OAuth2. Por outro lado, a popularidade crescente do JavaScript contribuiu muito para a adoção do CamelCase, mesmo que, **teoricamente**, o REST deveria promover a independência de linguagens e expor uma API de última geração sobre XML.

Nós recomendamos usar a caixa de forma consistente para o body, a escolher entre:

- snake_case (mais usada pela comunidade Ruby)
- lowerCamelCase (mais usada pelas comunidades Java e Javascript)

 Exemplos

```
1  GET /orders?id_client=007 ou GET /orders?idClient=007
2  POST /orders {"id_client":"007"} ou POST /orders {"idClient":"007”}
```

Versionamento

Qualquer API precisa evoluir com o tempo. Existem várias maneiras de versionar uma API:

- Por um *timestamp,* pelo *número do release*
- No *path,* no início ou no fim da *URI*
- Como um parâmetro do *request*
- Num *Header HTTP*
- Com um *versioning* obrigatório ou opcional

 Na prática dos Gigantes da Web:

API	Versionamento
Google	*URI path ou parâmetro* https://www.googleapis.com/oauth2/v1/tokeninfo https://www.googleapis.com/calendar/v3/ https://www.googleapis.com/drive/v2 https://maps.googleapis.com/maps/api/js?v=3.exp https://www.googleapis.com/plus/v1/ https://www.googleapis.com/youtube/v3/
Facebook	*URI (optional)* https://graph.facebook.com/v2.0/{achievement-id} https://graph.facebook.com/v2.0/{comment-id}
Twitter	https://api.twitter.com/1.1/statuses/show.json https://stream.twitter.com/1.1/statuses/sample.json
GitHub	*Accept Header (optional)* Accept: application/vnd.github.v3+json

Dropbox	*URI* https://www.dropbox.com/1/oauth2/authorize https://api.dropbox.com/1/account/info https://api-content.dropbox.com/1/files/auto https://api-notify.dropbox.com/1/longpoll_delta https://api-content.dropbox.com/1/thumbnails/auto
Instagram	*URI* https://api.instagram.com/v1/media/popular
Foursquare	*URI* https://api.foursquare.com/v2/venues/40a55d80f964a52020f31ee3
LinkedIn	*URI* http://api.linkedin.com/v1/people/
Netflix	*URI, parâmetro opcional* http://api.netflix.com/catalog/titles/series/70023522?v=1.5
Paypal	*URI* https://api.sandbox.paypal.com/v1/payments/payment

Nós recomendamos incluir um dígito do número da versão, de forma obrigatória, no nível mais alto do *path da URI*.

- O número da versão se refere a um major release da API para um recurso.
- REST e JSON, são muito mais flexíveis que SOAP/XML para desenvolver a API sem causar impacto aos clientes. Por exemplo, adicionar atributos aos recursos existentes não implica em incrementar a versão.
- Não deve haver versão default. Isso porque em caso de mudanças na API, os desenvolvedores não terão nenhum controle sobre os impactos nas suas aplicações.
- O número da versão da API é uma peça chave. Por isso, e pela affordance, é bem melhor que ele apareça na URL do que no header HTTP.
- Recomendamos suportar no máximo 2 versões da API ao mesmo tempo. Isso porque o ciclo de adoção de novas versões pelas aplicações nativas é muito longo.

 Exemplo

```
1  GET /v1/orders
```

CRUD

Como falamos antes, um dos objetivos da abordagem REST é usar o HTTP como protocolo de aplicação, e com isso evitar a criação de protocolos caseiros.

Então devemos usar de forma sistemática os verbos HTTP para descrever quais ações serão feitas nos recursos, e facilitar o trabalho dos desenvolvedores nas tarefas CRUD corriqueiras. A tabela a seguir sintetiza as práticas mais comuns:

Verbo HTTP	Ação CRUD	Coleção: /orders	Instância: /orders/{id}
GET	READ	Lê a lista de orders. 200 OK.	Lê os detalhes de uma order. 200 OK.
POST	CREATE	Cria uma nova order. 201 Criada.	–
PUT	UPDATE/CREATE	–	Full Update. 200 OK. Cria uma order especí Criada.
PATCH	UPDATE	–	Update Parcial. 200 OK.
DELETE	DELETE	–	Deleta a order. 200 OK.

O verbo HTTP POST é usado para criar uma instância dentro de uma coleção. O id do recurso a ser criado não precisa ser fornecido.

```
1  CURL -X POST \
2  -H "Accept: application/json" \
3  -H "Content-Type: application/json" \
4  -d '{"state":"running","id_client":"007"}' \
5  https://api.fakecompany.com/v1/clients/007/orders
6  &lt; 201 Created
7  &lt; Location: https://api.fakecompany.com/orders/1234
```

O código de retorno é 201 ao invés de 200.

A URI e o id são retornados no *header* "Location" da resposta.

Se o id for especificado pelo cliente, então o verto HTTP PUT é usado para a criação da instância na coleção. No entanto essa prática é menos comum.

```
CURL -X PUT \
-H "Content-Type: application/json" \
-d '{"state":"running","id_client":"007"}' \
https://api.fakecompany.com/v1/clients/007/orders/1234
&lt; 201 Created
```

O verbo HTTP PUT é usado sistematicamente para fazer um "full update" de uma instância em uma coleção (todos os atributos são substituídos e os que não forem enviados serão deletados).

No exemplo abaixo, nós atualizamos os atributos state e id_client. Todos os outros campos serão deletados.

```
CURL -X PUT \
-H "Content-Type: application/json" \
-d '{"state":"paid","id_client":"007"}' \
https://api.fakecompany.com/v1/clients/007/orders/1234
&lt; 200 OK
```

O verbo HTTP PATCH (que não existia na especificação original HTTP, sendo adicionado mais tarde) é usado para fazer um update parcial de uma instância em uma coleção.

No exemplo abaixo, nós atualizamos o atributos state, mas os demais atributos continuarão como antes.

```
CURL -X PATCH \
-H "Content-Type: application/json" \
-d '{"state":"paid"}' \
https://api.fakecompany.com/v1/clients/007/orders/1234
&lt; 200 OK
```

O verbo HTTP GET é usado para ler a coleção. Na prática, a API geralmente não retorna todos os itens da coleção (veja paginação).

```
CURL -X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/clients/007/orders
&lt; 200 OK
&lt; [{"id":"1234", "state":"paid"}, {"id":"5678", "state":"running"}]
```

O verbo HTTP GET é usado também para ler uma instância em uma coleção.

```
CURL -X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/clients/007/orders/1234
&lt; 200 OK
&lt; {"id":"1234", "state":"paid"}
```

Respostas parciais

As respostas parciais permitem que o cliente recupere apenas as informações que ele precisa. Isso se torna vital em contextos de apps móveis (3G) onde o uso de banda deve ser otimizado.

 Exemplos dos Gigantes da Web:

API	Respostas parciais
Google	?fields=url,object(content,attachments/url)
Facebook	&fields=likes,checkins,products
LinkedIn	https://api.linkedin.com/v1/people/~:(id,first-name,last-name,industry)

Nós recomendamos que, no mínimo, seja possível selecionar os atributos a serem recebidos em 1 nível de recursos, através da notação Google fields=attribute1,attributeN:

```
GET /clients/007?fields=firstname,name
200 OK
{
"id":"007",
"firstname":"James",
"name":"Bond"
}
```

Nos casos em que a performance é crítica, nós recomendamos usar a notação Google fields=objects(attribute1,attributeN). Como exemplo, se você quiser recuperar apenas o nome e o sobrenome, e a rua do endereço de cliente:

```
GET /clients/007?fields=firstname,name,address(street)
200 OK
{
"id":"007",
"firstname":"James",
"name":"Bond",
"address":{"street":"Horsen Ferry Road"}
}
```

Query strings

Paginação

É necessário planejar com antecedência a paginação dos seus recursos, desde o início do seu projeto de API. É difícil prever com precisão a evolução da quantidade de dados a serem retornados. Por isso recomendamos o uso da paginação default dos seus recursos. Se o seu cliente não especificar a paginação na chamada, use uma faixa de valores default de, por exemplo, [0-25].

A paginação sistemática também dá consistência aos seus recursos, o que é muito bom. Tenha em mente o princípio de affordance: quanto menos documentação para ler, mais fácil a adoção da sua API.

 Nos Gigantes da Web:

API	Paginação

Facebook Parâmetros: before, after, limit, next, previous

```
"paging": {
"cursors": {
"after": "MTAxNTExOTQ1MjAwNzI5NDE=",
"before": "NDMyNzQyODI3OTQw"
},
"previous": "https://graph.facebook.com/me/albums?limit=25&amp;before=NDMyNzQyODI3OTQw"
"next": "https://graph.facebook.com/me/albums?limit=25&amp;after=MTAxNTExOTQ1MjAwNzI5NDE="
}
```

Google Parâmetros: maxResults, pageToken

```
"nextPageToken":"CiAKGjBpNDd2Nmp2Zml2cXRwYjBpOXA",
```

Twitter Parâmetros: since_id, max_id, count

```
"next_results": "?max_id=249279667666817023&amp;q=*freebandnames&amp;count=4&amp;include_entities=1&amp;result_type=mixed",
"count": 4,
"completed_in": 0.035,
"since_id_str": "24012619984051000",
"query": "*freebandnames",
"max_id_str": "250126199840518145"
```

GitHub Parâmetros: page, per_page

```
Link: &lt;https://api.github.com/user/repos?page=3&amp;per_page=100&gt;; rel="next",
&lt;https://api.github.com/user/repos?page=50&amp;per_page=100&gt;; rel="last"
```

Paypal Parâmetros: start_id, count

```
{'count': 1,'next_id': 'PAY-5TU010975T094876HKKDU7MZ',
```

Muitos mecanismos diferentes de paginação são usados pelos Gigantes da Web. Como nenhum padrão parece emergir, nossa proposta é usar:

- o parâmetro de request ?range=0-25
- e os HTTP Headers padrão para o response:
  - Content-Range
  - Accept-Range

 A paginação no request

Do ponto de vista prático, a paginação é geralmente feita via *query-string* na URL. Mas o header HTTP também tem esse mecanismo. Nossa proposta é de paginar somente via query-string, e de não considerar o HTTP Header Range. A paginação é uma peça muito importante, e faz sentido que ela apareça na URL, para clareza e simplicidade (e *affordance*).

Nós propomos a utilização de uma faixa de valores, sobre o índice da sua coleção. Por exemplo, recursos do índice 10 até o 25 inclusive, equivalem a: *?range=10-25*.

 A paginação no response

O código de retorno HTTP de um request paginado será *206 Partial Content*, exceto de os valores requisitados resultarem no retorno da coleção completa, que nesse caso geraria o código de retorno *200 OK*.

A resposta da sua API para uma coleção deve obrigatoriamente conter nos Headers HTTP:

- Content-Range ***offset*** – ***limit*** / ***count***
  - ***offset***: o índice do primeiro elemento retornado
  - ***limit***: o índice do último elemento retornado
  - ***count***: o número total de elementos que a coleção contém

- Accept-Range ***resource max***
  - ***resource***: o tipo da paginação. Deve lembrar o recurso em uso, ex.: *client, order, restaurant, etc.*
  - ***max*** : o número máximo de recursos que podem ser retornado em cada requisição.

Caso a paginação requisitada não se encaixe nos valores permitidos pela API, a resposta HTTP deve ser um error code 400, com a descrição explícita do erro no body.

 Links de navegação

É fortemente recomendado incluir a tag Link no header HTTP das suas respostas. Ela permite adicionar, entre outras, os links de navegação, como próxima página, página anterior, primeira e última páginas, etc.

 Exemplos

Nós temos na nossa API uma coleção de 48 restaurantes, para os quais não é permitido consultar mais que 50 por requisição. A paginação default é 0-50:

```
CURL -X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/restaurants
&lt; 200 Ok
&lt; Content-Range: 0-47/48
&lt; Accept-Range: restaurant 50
&lt; [...]
```

Se 25 recursos são requisitados dos 48 existentes, recebemos um 206 Partial Content como código de retorno:

```
CURL -X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/restaurants?range=0-24
&lt; 206 Partial Content
&lt; Content-Range: 0-24/48
&lt; Accept-Range: restaurant 50
```

Se 50 recursos são requisitados dos 48 disponíveis, o retorno é o código 200 OK:

```
CURL -X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/restaurants?range=0-50
&lt; 200 Ok
&lt; Content-Range: 0-47/48
&lt; Accept-Range: restaurant 50
```

Se a faixa requisitada está acima do número máximo de recursos permitido por requisição (*Header Accept-Range*), o código *400 Bad Request* é retornado:

```
CURL -X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/orders?range=0-50
&lt; 400 Bad Request
&lt; Accept-Range: order 10
&lt; { reason : "Requested range not allowed" }
```

Nós recomendamos usar a seguinte notação para retornar os links para outras faixas. Ela é usada pelo GitHub, e é compatível com a RFC5988. Ela também permite gerenciar clientes que não suportam vários *Link Headers*.

```
CURL -X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/orders?range=48-55
&lt; 206 Partial Content
&lt; Content-Range: 48-55/971
&lt; Accept-Range: order 10
&lt; Link : &lt;https://api.fakecompany.com/v1/orders?range=0-7&gt;; rel="firs
t", &lt;https://api.fakecompany.com/v1/orders?range=40-47&gt;; rel="prev", &lt;
https://api.fakecompany.com/v1/orders?range=56-64&gt;; rel="next", &lt;https://
api.fakecompany.com/v1/orders?range=968-975&gt;; rel="last"
```

Uma outra notação frequentemente encontrada, é a tag Link do *HTTP Header* contendo uma URL seguida pelo tipo do link. Essa tag pode ser repetida quantas vezes forem os links associados à resposta:

```
&lt; Link: &lt;https:<i>//api.fakecompany.com/v1/orders?range=0-7&gt;; rel="fir
st"</i>
&lt; Link: &lt;https://api.fakecompany.com/v1/orders?range=40-47&gt;; rel="pre
v"
&lt; Link: &lt;https://api.fakecompany.com/v1/orders?range=56-64&gt;; rel="nex
t"
&lt; Link: &lt;https://api.fakecompany.com/v1/orders?range=968-975&gt;; rel="la
st"
```

Ou, seguindo a notação no payload, que é usada pela Paypal:

```
[
{"href":"https://api.fakecompany.com/v1/orders?range=0-7", "rel":"first", "meth
od":"GET"},
{"href":"https://api.fakecompany.com/v1/orders?range=40-47", "rel":"prev", "met
hod":"GET"},
{"href":"https://api.fakecompany.com/v1/orders?range=56-64", "rel":"next", "met
hod":"GET"},
{"href":"https://api.fakecompany.com/v1/orders?range=968-975", "rel":"last", "m
ethod":"GET"},
]
```

Filtros

Filtrar consiste em limitar o número de recursos requisitados, especificando alguns atributos e seus valores esperados. É possível filtrar uma coleção por vários atributos ao mesmo tempo, e permitir filtrar vários valores para um atributo.

Por isso nós propomos usar diretamente o nome do atributo com um sinal de igual e os valores esperados, separados por vírgula.

Exemplo: recuperar os restaurantes tailandeses (*thai*)

```
CURL -X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/restaurants?type=thai
```

Exemplo: recuperar os restaurantes com *rating* de 4 ou 5, com cozinha chinesa ou japonesa, abertos aos domingos

```
CURL -X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/restaurants?type=japanese,chinese&amp;rating=4,5
&amp;days=sunday
```

Ordenação

Ordenar os resultados de uma query numa coleção de recursos requer 2 parâmetros:

- ***sort***: contém os nomes dos atributos que serão usados para ordenar a lista, separados por vírgula.
- ***desc***: por default a ordem é ascendente (ou crescente). Para uma ordenação descendente (ou decrescente), é necessário adicionar esse parâmetro (sem nenhum valor). Em alguns casos específicos pode-se querer especificar quais atributos serão ascendentes e quais serão descendentes. Nesse caso, o parâmetro desc deve conter a lista dos atributos que serão descendentes, ficando os demais ascendentes por default.

Exemplo: recuperar a lista de restaurantes ordenados alfabeticamente pelo nome.

```
1  CURL -X GET \
2  -H "Accept: application/json" \
3  https://api.fakecompany.com/v1/restaurants?sort=name
```

Exemplo: recuperar a lista de restaurantes, ordenados por rating decrescente, depois por número de reviews decrescente, e por fim alfabeticamente pelo nome.

```
1  CURL -X GET \
2  -H "Accept: application/json" \
3  https://api.fakecompany.com/v1/restaurants?sort=rating,reviews,name&amp;desc=ra
   ting,reviews
```

 Ordenação, Filtro e Paginação

A paginação provavelmente será afetada pela ordenação e pelos filtros. A combinação desses 3 parâmetros deve ser usada com consistência nas requisições para a sua API.

Exemplo: requisição dos 5 primeiros restaurantes chineses, ordenados por rating descendente.

```
1  CURL -X GET \
2  -H "Accept: application/json" \
3  https://api.fakecompany.com/v1/restaurants?type=chinese&amp;sort=rating,name&am
4  p;desc=rating&amp;range=0-4
5  &lt; 206 Partial Content
6  &lt; Content-Range: 0-4/12
   &lt; Accept-Range: restaurants 50
```

Busca

Busca nos recursos

Quando filtrar já não é suficiente (para fazer uma correspondência parcial ou aproximada, por exemplo), precisamos fazer uma busca nos recursos.

Uma busca é um sub-recurso da sua coleção. Como tal, seus resultados vão ter um formato diferente da coleção em si. Isso abre espaço para adicionar sugestões, correções e informação relacionada com a busca.

Os parâmetros são fornecidos da mesma maneira que no filtro, através da query-string, mas eles não são necessariamente valores exatos, e sua sintaxe permite fazer aproximações.

Sendo um recurso, a busca deve suportar a paginação da mesma forma que os outros recursos da sua API.

Exemplo: busca dos restaurantes cujos nomes começam por *"la"*.

```
CURL -X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/restaurants/search?name=la*
&lt; 206 Partial Content
&lt; { "count" : 5, "query" : "name=la*", "suggestions" : ["las"], results :
[...] }
```

Exemplo: busca dos 10 primeiros restaurantes que tenham "napoli" em seus nomes, que tenham comida chinesa ou japonesa, situados na área do código postal 75 (Paris), ordenados pelo rating e nome descendentes e nome alfabeticamente.

```
CURL -X GET \
-H "Accept: application/json" \
-H "Range 0-9" \
https://api.fakecompany.com/v1/restaurants/search?name=*napoli*&amp;type=chines
e,japanese&amp;zipcode=75*&amp;sort=rating,name&amp;desc=rating&amp;range=0-9
&lt; 206 Partial Content
&lt; Content-Range: 0-9/18
&lt; Accept-Range: search 20
&lt; { "count" : 18, "range": "0-9", "query" : "name=*napoli*&amp;type=chinese,
japanese&amp;zipcode=75*", "suggestions" : ["napolitano", "napolitain"], result
s : [...] }
```

Busca global

A busca global deve ter o mesmo comportamento de uma busca específica em um recurso, exceto que ela é localizada na raíz da API. com isso ela precisa ser bem detalhada na documentação da API.

Nós recomendamos a notação da Google para buscas globais:

```
1  CURL -X GET \
2  -H "Accept: application/json" \
3  https://api.fakecompany.com/v1/search?q=running+paid
4  &lt; [...]
```

Outros conceitos importantes

Negociação de conteúdo

Nós recomendamos que sejam gerados diversos formatos de distribuição do conteúdo na sua API. Podemos usar o "Accept" HTTP Header, que existe para essa finalidade.

Por default, a API vai retornar os recursos no formato JSON, mas se o request começar por "Accept: application/xml", os recursos deverão ser enviados no formato XML.

É recomendável suportar no mínimo 2 formatos: JSON e XML. A ordem dos formatos enviada no header "accept" deve ser considerada para definir o formato da resposta.

Nos casos onde não é possível fornecer o formato requerido, um erro HTTP 406 deve ser enviado (ver Erros – Status Codes)

```
1  GET https://api.fakecompany.com/v1/offers
2  Accept: application/xml; application/json XML préféré à JSON
3  &lt; 200 OK
4  &lt; [XML]GET https://api.fakecompany.com/v1/offers
5  Accept: text/plain; application/json The API cannot provide text
6  &lt; 200 OK
7  &lt; [JSON]
```

Cross-domain

CORS

Quando a aplicação (JavaScript SPA) e a API estão hospedadas em domínios diferentes, por exemplo:

- https://fakeapp.com
- https://api.fakecompany.com

Uma boa prática é usar o protocolo CORS, que é padrão no HTTP.

No servidor, a implementação do CORS geralmente consiste em adicionar alguns parâmetros de configuração no servidor HTTP (Nginx/Apache/NodeJs, etc.).

No lado do cliente, a implementação é transparente: o browser vai enviar um request HTTP com o verbo OPTIONS, antes de cada request GET/POST/PUT/PATCH/DELETE.

Aqui, por exemplo, duas chamadas sucessivas são feitas num browser para recuperar, via GET, informações de um usuário na API do Google+:

```
1  CURL -X OPTIONS \
2  -H 'Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8' \
3  'https://www.googleapis.com/plus/v1/people/105883339188350220174?client_id=API_
   KEY'
```

```
1  CURL -X GET\
2  'https://www.googleapis.com/plus/v1/people/105883339188350220174?client_id=API_
3  KEY' \
4  -H 'Accept: application/json, text/JavaScript, */*; q=0.01'\
   -H 'Authorization: Bearer foo_access_token'
```

Jsonp

Na prática, o CORS é mal suportado ou mesmo não suportado pelos browsers antigos, especialmente IE7, 8 e 9. Se você não tem controle sobre os browsers que usam a sua API (pela Internet, por clientes finais), é necessário expor um Jsonp da sua API como contingência para a implementação do CORS.

Na verdade Jsonp é uma solução de contorno ao uso da tag <script/> para permitir o gerenciamento cross-domain, e possui algumas limitações:

- Não é possível fazer a negociação de conteúdo através do *Accept Header* => um novo *endpoint* tem que ser publicado, por exemplo com extensão .jasonp, para permitir que o controller possa determinar que é se trata de uma requisição jsonp.

- Todos os requests são enviados com o verbo HTTP GET => deve ser usado um parâmetro method=XXX
  - Tenha em mente que um web crawler pode causar sérios danos aos seus dados se não houver mecanismo de autorização para a chamada do method=DELETE...

- O payload da requisição não pode conter dados => todos os dados precisam ser enviados como parâmetros do request

Para ser compatível com CORS & Jsonp, por exemplo, sua API deve expor os seguintes endpoints:

```
POST /orders and /orders.jsonp?method=POST&amp;callback=foo
GET /orders and /orders.jsonp?callback=foo
GET /orders/1234 and /orders/1234.jsonp?callback=foo
PUT /orders/1234 and /orders/1234.jsonp?method=PUT&amp;callback=foo
```

HATEOAS

Conceito

Vamos pegar como exemplo o nome *Angelina Jolie*. *Angelina* é cliente da *Amazon*, e quer ver detalhes sobre seu último pedido. Para isso, ela precisa 2 passos:

1. Listar todos os seus pedidos
2. Selecionar seu último pedido

No *website da Amazon*, *Angelina* não precisa ser expert para consultar seu último pedido: basta se logar no site, clicar no link "meus pedidos", e então selecionar o pedido mais recente.

Imagine que *Angelina* quer fazer o mesmo usando uma API.

Ela deve começar por consultar a documentação da *Amazon*, para achar a URL que retorna a lista de pedidos. Quando ela achar, ela deve fazer uma chamada HTTP para essa URL. Ela vai ver a referência para o seu pedido na lista, mas ela vai precisar fazer uma segunda chamada para outra URL para ver os detalhes. *Angelina* deve então consultar a documentação da *Amazon* para fazer a chamada corretamente.

A diferença entre os dois cenários é que no primeiro *Angelina* não precisa conhecer nada além da URL inicial, "http://www.amazon.com", para depois seguir os links na página. Já no segundo, ela precisa ler a documentação para montar as URLs.

O problema do segundo cenário é que:

- Na vida real, a documentação raramente está atualizada. *Angelina* pode deixar de usar um ou vários serviços apenas por eles não estarem bem documentados.
- *Angelina* é, nesse caso, uma desenvolvedora, e desenvolvedores normalmente **não gostam de documentação**.
- A API é menos acessível.

Supondo que *Angelina* desenvolva um componente batch para automatizar a criação dessas duas URLs. O que acontecerá quando a Amazon modificar as suas URLs?

Implementação

O HATEOAS é como uma lenda urbana: todos falam nisso, mas ninguém nunca viu uma implementação de verdade.

O Paypal propõe a seguinte implementação:

```
[
{
"href": "https://api.sandbox.paypal.com/v1/payments/payment/PAY-6RV70583SB70280
5EKEYSZ6Y",
"rel": "self",
"method": "GET"
},
{
"href": "https://www.sandbox.paypal.com/webscr?cmd=_express-checkout&amp;token=
EC-60U79048BN7719609",
"rel": "approval_url",
"method": "REDIRECT"
},
{
"href": "https://api.sandbox.paypal.com/v1/payments/payment/PAY-6RV70583SB70280
5EKEYSZ6Y/execute",
"rel": "execute",
"method": "POST"
}
]
```

Uma chamada para */customers/007* retornaria os detalhes do *customer*, além dos links para os recursos relacionados:

```
GET /clients/007
&lt; 200 Ok
&lt; { "id":"007", "firstname":"James",...,
"links": [
{"rel":"self","href":"https://api.domain.com/v1/clients/007", "method":"GET"},
{"rel":"addresses","href":"https://api.domain.com/v1/addresses/42", "method":"G
ET"},
{"rel":"orders", "href":"https://api.domain.com/v1/orders/1234",
"method":"GET"},
...
]
}
```

Para implementar HATEOAS, nós recomendamos usar o seguinte método, aplicado pelo GitHub, compatível com a RFC5988, e usado por clientes que não suportam vários *Header "Link"*:

```
GET /clients/007
&lt; 200 Ok
&lt; { "id":"007", "firstname":"James",...}
&lt; Link : &lt;https://api.fakecompany.com/v1/clients&gt;; rel="self"; metho
d:"GET",
&lt; &lt;https://api.fakecompany.com/v1/addresses/42&gt;; rel="addresses"; meth
od:"GET",
&lt; &lt;https://api.fakecompany.com/v1/orders/1234&gt;; rel="orders"; metho
d:"GET"
```

Cenários "Sem Recursos"

Na teoria do RESTful, qualquer requisição precisa ser vista e tratada como um recurso. Mas na prática isso nem sempre é possível, especialmente quando se tem que lidar com ações, como traduções, cálculos, conversões, serviços de negócios complexos ou serviços fortemente integrados.

Nesses casos, sua operação precisa ser representada por um verbo ao invés de um nome. Por exemplo:

```
POST /calculator/sum
[1,2,3,5,8,13,21]
&lt; 200 OK
&lt; {"result" : "53"}
```

Ou ainda:

```
POST /convert?from=EURato=USD&amp;amount=42
&lt; 200 OK
&lt; {"result" : "54"}
```

Então usaremos as ações ao invés de recursos. Nesse caso, usaremos o método HTTP POST.

```
CURL -X POST \
-H "Content-Type: application/json" \
https://api.fakecompany.com/v1/users/42/carts/7/commit
&lt; 200 OK
&lt; { "id_cart": "7",<i> [...]</i> }
```

Para acomodar melhor esses casos excepcionais na sua API, a melhor maneira é considerar que qualquer requisição POST é uma ação, com um verbo implícito ou explícito.

No caso de uma coleção de entidades, por exemplo, a ação default seria a criação:

```
1  POST /users/create POST /users
2  &lt; 201 OK == &lt; 201 OK
3  &lt; { "id_user": 42 } &lt; { "id_user": 42 }
```

Já no caso de um recurso de "email", a ação default seria enviar para seu destinatário:

```
1  POST /emails/42/send POST /emails/42
2  &lt; 200 OK == &lt; 200 OK
3  &lt; { "id_email": 42, "state": "sent" } &lt; { "id_email": 42, "state": "sent"
   }
```

No entanto é importante ter em mente que usar um verbo no projeto da sua API deve ser uma excessão. Na maioria dos casos, isso pode e deve ser evitado. Se vários recursos expõem uma ou mais ações, isso acaba virando uma falha de projeto na sua API: você está usando uma abordagem RPC ao invés de REST, e você vai precisar tomar medidas para corrigir o projeto da sua API.

Para evitar confundir os desenvolvedores com os recursos (sobre os quais você accessa como CRUD) e as ações, é altamente recomendado separar claramente as duas formas na sua documentação.

 Exemplos dos Gigantes da Web:

API	API "Sem Recursos"
Google Translate API	GET https://www.googleapis.com/language/translate/v2?key=INSERT-YOUR-KEY&target=de&q=Hello%20world
Google Calendar API	POST https://www.googleapis.com/calendar/v3/calendars/calendarId/clear
Twitter Authentification	GET https://api.twitter.com/oauth/authenticate?oauth_token=Z6eEdO8MOmk394WozF5oKyuAv855l4Mlqo7hhlSLik

Erros

Estrutura do erro

Nós recomendamos seguir a seguinte estrutura *JSON*:

```
{
"error": "descrição_curta",
"error_description": "descrição longa, legível por humanos",
"error_uri": "URI que leva a uma descrição detalhada do erro no portal do desen
volvedor"
}
```

O atributo error não é necessariamente redundante com o status HTTP: podemos ter dois diferentes valores para o atributo error mantendo o mesmo status HTTP.

- 400 & error=invalid_user
- 400 & error=invalid_cart

Essa representação é da especificação OAuth2. Um uso sistemático dessa sintaxe na API vai evitar que os clientes de tenham que gerenciar 2 estruturas de erro diferentes.

Nota: Em alguns casos é bom usar uma coleção dessa estrutura, para retornar vários erros de uma só vez (isso pode ser útil no caso de uma validação de formulário *server-side*, por exemplo).

Status Codes

Nós recomendamos fortemente usar os códigos de retorno HTTP, já que esses códigos cobrem todos os casos comuns, e todos os desenvolvedores entendem. É claro que não é necessário usar toda a coleção de códigos. Normalmente os 10 códigos mais usados são suficientes.

SUCCESS

200 OK é o código de sucesso clássico, e funciona para a maioria dos casos. É especialmente usado quando o primeiro request GET para um recurso tem sucesso.

HTTP Status	Description
201 Created	Indica que o recurso foi criado. Resposta típica para um request PUT ou POST, incluindo o Header HTTP "Location", que aponta para a URL do novo recurso.
202 Accepted	O request foi aceito, e será processado mais tarde. Sua resposta será assíncrona (para melhor UX ou performance)
204 No Content	Indica que o request foi processado com sucesso, mas não há nada para retornar. É comum em respostas para DELETE requests.
206 Partial Content	A resposta está incompleta. Normalmente retornado em respostas paginadas.

CLIENT ERROR

HTTP Status	Description
400 Bad Request	Geralmente usado para erros de chamada, se não se encaixarem em nenhum outro status. Erro do request, exemplo:

```
1  GET /users?payed=1
2  &lt; 400 Bad Request
3  &lt; {"error": "invalid_request", "error_description": "There is no 'payed' pro
   perty on users."}
```

Condição de erro na aplicação, exemplo:

```
1  POST /users
2  {"name":"John Doe"}
3  &lt; 400 Bad Request
4  &lt; {"error": "invalid_user", "error_description": "A user must have an email
   adress"}
```

401 Unauthorized

Eu não conheço o seu id. Diga-me quem você é, e eu vejo sua autorização.

```
GET /users/42/orders
&lt; 401 Unauthorized
&lt; {"error": "no_credentials", "error_description": "This resource is under permission, you must be authenticated with the right rights to have access to it" }
```

403 Forbidden

Você foi autenticado corretamente, mas não tem privilégios suficientes.

```
GET /users/42/orders
&lt; 403 Forbidden
&lt; {"error": "not_allowed", "error_description": "You're not allowed to perform this request"}
```

404 Not Found

O recurso que você pediu não existe.

```
GET /users/999999/
&lt; 400 Not Found
&lt; {"error": "not_found", "error_description": "The user with the id ‘999999' doesn't exist" }
```

405 Method not allowed

Você chamou um método que não faz sentido nesse recurso, ou o usuário não tem permissão de fazer essa chamada.

```
POST /users/8000
&lt; 405 Method Not Allowed
&lt; {"error":"method_does_not_make_sense", "error_description":"How would you even post a person?"}
```

406 Not Acceptable

Nenhum formato se encaixa no Header Accept-* do seu request. Por exemplo, você pediu o recurso em formato XML, mas ele só está disponível em JSON.

```
GET /usersAccept: text/xmlAccept-Language: fr-fr
&lt; 406 Not Acceptable
&lt; Content-Type: application/json
&lt; {"error": "not_acceptable", "available_languages":["us-en", "de", "kr-ko"]}
```

SERVER ERROR

HTTP Status	Description
500 Server error	A requisição está correta, mas ocorreu um problema de execução. O cliente não tem muito o que fazer a respeito disso. Nós recomendamos sistematicamente retornar Status 500 nesses casos.

```
GET /users
&lt; 500 Internal server error
&lt; Content-Type: application/json
&lt; {"error":”server_error", "error_description":"Oops! Something went wrong..."}
```

Fontes

- Design Beautiful REST + JSON APIs
  - http://www.slideshare.net/stormpath/rest-jsonapis
- Web API Design: Crafting Interfaces that Developers Love
  - https://pages.apigee.com/web-api-design-website-h-ebook-registration.html
- HTTP API Design Guide
  - https://github.com/interagent/http-api-design
- RESTful Web APIs
  - http://shop.oreilly.com/product/0636920028468.do
- How are REST APIs versioned?
  - http://www.lexicalscope.com/blog/2012/03/12/how-are-rest-apis-versioned/
- REST World

- http://nodejsparis.bitbucket.org/20140312/rest_world/#/

INTERESSADO NO ASSUNTO? CONTACTA-NOS!

No related posts.

This entry was posted in Arqui e tecno. Bookmark the permalink.

PARIS | RABAT | LAUSANNE | SÃO PAULO | SYDNEY

Sede: 34 avenue de l'Opéra, 75002 Paris, France | +33 (0)1 58 56 10 00