Desenvolvendo APIs que você não odiará

Por Phil Sturgeon

Desenvolvendo APIs que você não odiará

Todo mundo e seus cães querem uma API, é melhor você aprender a desenvolvê-las.

Phil Sturgeon e Pedro Borges

Esse livro está à venda em http://leanpub.com/desenvolvendo-apis

Essa versão foi publicada em 2014-05-08

Tweet Sobre Esse Livro!

Por favor ajude Phil Sturgeon e Pedro Borges a divulgar esse livro no Twitter!

A hashtag sugerida para esse livro é #desenvolvendoapis.

Descubra o que as outras pessoas estão falando sobre esse livro clicando nesse link para buscar a hashtag no Twitter:

https://twitter.com/search?q=#desenvolvendoapis

Conteúdo

1 Nota do Autor

Eu já vi muitas tendências irem e virem durante a minha longa e diversificada carreira de "construir" coisas por dinheiro como um empregado, *freelancer*, consultor e agora como CTO. Uma das tendências modernas é o crescimento das APIs como parte diária do trabalho da maioria dos desenvolvedores *server-side*.

Alguns anos atrás, quando eu ainda era um usuário e contribuidor do CodeIgniter, eu publiquei um Servidor Rest para o CodeIgniter e escrevi alguns artigos sobre como utilizá-lo. Naquela época, eu sabia que aquilo não era *tudo* que uma API precisava, mas cobria roteamento RESTful, autenticação HTTP básica/Digest/Chave de API; eu também acrescentei *logging* e *throttling*, e não obriguei o uso de convenções baseadas em CRUD como: `PUT = CREATE OR DIE!`. Essa era de longe uma opção muito melhor do que as disponíveis em outros *frameworks*. A *internet* concordou comigo e atualmente este código é utilizado pela Apple, Nações Unidas e o Governo dos EUA (USA.gov).

Mais tarde, como parte da equipe do FuelPHP, eu implementei esta funcionalidade neste *framework* e, mais uma vez, desenvolvi bastante APIs para terceiros. Então me ofereceram um trabalho em Nova Iorque e eu aceitei liderar o pessoal técnico desta empresa que também tinha uma API desenvolvida no FuelPHP e buscava alguém para aprimorá-la.

Eu tenho desenvolvido APIs há muito tempo e possuo uma longa lista de como desenvolvê-las sem criar um monstro. Eu gostaria de compartilhar esta informação com todos vocês.

Phil Sturgeon

2 Planejando e Criando Pontos de Acesso

Com o seu banco de dados planejado e repleto de dados fictícios, porém úteis, é hora de planejar os pontos de acesso[1]. O primeiro passo será descobrir os requerimentos de uma API, em seguida veremos um pouco de teoria e, finalmente, veremos a teoria implementada em alguns exemplos.

2.1 Requerimentos Funcionais

Tente pensar em *tudo* que sua API precisará fazer. Inicialmente esta será uma lista de pontos de acesso CRUD (Criar, Ler, Atualizar, Excluir) dos seus recursos. Converse com o seu desenvolvedor de aplicativos móveis, o pessoal do JS no *frontend* ou simplesmente converse consigo mesmo, caso você seja o único desenvolvedor no projeto.

Definitivamentes converse com os seus consumidores ou "a empresa" (eles são consumidores) e peça-lhes para te ajudar a pensar em funcionalidades também, mas não espere que eles entendam o que um ponto de acesso é.

Quando você tiver uma lista relativamente comprida, o próximo passo será criar uma lista simples de "Ações". Este passo é muito parecido com o planejamento de uma classe de PHP; primeiramente você escreve um código fictício referindo-se a classes e métodos como se eles existissem, não é mesmo? TDD? Se não for, você estiver fazendo assim o Chris Hartjes *mataria* você.

Assim, se eu tiver um recurso chamado "Local" em mente, precisarei listar o que ele fará:

> **Locais**
>
> - **Criar**
> - **Ler**
> - **Atualizar**
> - **Excluir**

Isto é algo óbvio. Quem poderá visualizar, criar ou editar estes locais é (por enquanto) irrelevante na fase de planejamento, pois esta API ficará mais inteligente com as ideias de contexto de usuário e permissões, que veremos no futuro. Por enquanto, apenas liste tudo aquilo que precisará ser feito.

Uma lista de todos os locais também é um requerimento, anote-o aí:

> **Locais**

[1] *endpoints*, em inglês.

- Criar
- Ler
- Atualizar
- Excluir
- **Listar**

A API também precisará ser capaz de pesquisar locais por sua localização, mas este não é um ponto de acesso completamente novo. Se a API fosse desenvolvida com SOAP ou XML-RPC você precisaria criar um método `getPlacesByLatAndLon` para ser acessado na URL, mas felizmente não estamos usando SOAP. O método "listar" pode cuidar disso com alguns parâmetros, então porque não acrescentar uma nota para o futuro:

Locais

- Criar
- Ler
- Atualizar
- Excluir
- Listar **(lat, lon, distância ou caixa)**

Acrescentar alguns parâmetros como lembrete neste estágio é bacana, mas não vamos nos preocupar em acrescentar demais. Por exemplo, "criar" e "atualizar" já são complicados; acrescentar cada um dos campos criaria uma bagunça.

Atualizar é mais do que apenas atualizar campos específicos na tabela de "locais" em SQL. Na atualização podemos fazer coisas bem legais. Se você precisa "favoritar" um local, é só enviar `is_favorite` para aquele ponto de acesso e você já o *favoritou*. Veremos sobre isso mais à frente, apenas lembre-se que nem toda ação requer um ponto de acesso próprio.

Locais também precisarão de uma imagem enviada via API. Neste exemplo aceitaremos apenas uma imagem por local, e uma imagem nova substitui a antiga. Assim, acrescente "Imagens" à sua lista:

Locais

- Criar
- Ler
- Atualizar
- Excluir
- Listar (lat, lon, distância ou caixa)
- **Imagem**

Nosso "plano de ação" completo da API ficará assim:

Categorias

- Criar

- Listar

Checkins

- Criar
- Ler
- Atualizar
- Excluir
- Listar
- Imagem

Oportunidades

- Criar
- Ler
- Atualizar
- Excluir
- Listar
- Imagem
- Checkins

Locais

- Criar
- Ler
- Atualizar
- Excluir
- Listar (lat, lon, distância ou caixa)
- Imagem

Usuários

- Criar
- Ler
- Atualizar
- Excluir
- Listar (ativo, suspenso)
- Imagem
- Favoritos
- Checkins
- Seguidores

Isso pode não ser tudo, mas me parece bem sólido para começarmos nossa API. Certamente levará um bom tempo para se chegar a esta lista, por isso, se alguém pensar em algo que precisa ser acrescentado, anote.

Seguindo em frente.

2.2 Teoria dos Pontos de Acesso

Transformar este "Plano de Ação" em pontos de acesso reais requer um pouco conhecimento sobre teoria de RESTful APIs e "boas práticas" para convenções de nomes. Não há uma resposta correta aqui, mas algumas abordagens possuem menos desvantagens que outras. Vou tentar te levar em direção à abordagem que considero mais útil e destacarei as vantagens e desvantagens de cada uma.

Obtendo Recursos

- `GET /resources` - Lista de alguma coisa paginada com um pouco de lógica para determinar a sua ordem padrão.
- `GET /resources/X` - Apenas a entidade X. Pode ser uma "chave" ou um ID, mas você não deve usar um ID auto-incrementado, a menos que você queira expor a quantidade de usuários, estabelecimentos e etc de sua aplicação.
- `GET /resources/X,Y,Z` - O cliente deseja mais do que uma "coisa".

Pode ser difícil diferenciar entre URLs com sub-recurso e URLs com dados incorporados. Bem, dados incorporados é um assunto tão difícil que falaremos mais sobre eles no futuro. Por enquanto a reposta é "apenas sub-recursos", mas eventualmente a resposta será "ambos". Sub-recursos são assim:

- `GET /places/X/checkins` - Encontre todos os *checkins* de um local específico.
- `GET /users/X/checkins` - Encontre todos os *checkins* de um usuário específico.
- `GET /users/X/checkins/Y` - Encontre um *checkin* específico de um usuário específico.

O último é questionável e é algo que eu, pessoalmente, não tenho feito. Nesse ponto eu simplesmente prefiro usar `/places`.

Excluíndo Recursos

Deseja excluir algo? É fácil:

- `DELETE /places/X` - Exclui um local.
- `DELETE /places/X,Y,Z` - Exclui vários locais.
- `DELETE /places` - Este é um ponto de acesso muito perigoso e deve ser evitado, pois ele excluiria todos os locais.
- `DELETE /places/X/image` - Exclui a imagem de um local, ou:
- `DELETE /places/X/images` - Exclui todas as imagens, se você permitiu mais de uma imagem por local.

POST x PUT: Lutem!

E para a criação e atualização? É onde as coisas se tornam mais religiosas. Muitas pessoas tentam associar os verbos HTTP POST e PUT a uma ação específica no CRUD e usam somente aquela ação com tal verbo. Isso não é legal, não é produtivo, nem é funcionalmente escalável.

De modo geral, PUT é usado quando conhecemos a URL completa de antemão e a ação é idempotente. Idempotente é uma palavra bonita para algo que "pode se repetir inúmeras vezes sem causar diferença nos resultados".

Por exemplo, o verbo PUT *poderia* ser usado na criação se você está criando uma imagem para um local. Algo assim:

```
PUT /places/1/image HTTP/1.1
Host: example.com
Content-Type: image/jpeg
```

Este é um ótimo exemplo de quando se pode usar `PUT`, porque já conhecíamos toda a URL e esta ação poderia ser repetida quantas vezes quiséssemos. Você poderia tentar o *check-in* várias vezes e não importaria, porque nenhum desses processos seria completo. POSTando múltiplas vezes **não** é idempotente porque cada *check-in* é diferente. Mas PUT é idempotente porque você está enviando aquela imagem para uma URL completa e você pode repetir este processo inúmeras vezes (caso o envio falhe e você precise tentar novamente, por exemplo).

Assim, se você tem mais do que uma imagem para cada local, você poderia usar `POST /places/X/images` e cada tentativa criaria uma imagem diferente. Mas se você sabe que usará apenas uma imagem e cada nova tentativa será uma substituição, então `PUT /places/X/image` seria o ideal.

Outro exemplo seria as configurações de um usuário:

- `POST /me/settings` - Eu esperaria que este ponto me permitisse, enviar um campo de cada vez, sem me forçar a reenviar todas as configurações.
- `PUT /me/settings` - Envie-me **todas** as configurações.

É uma diferença complicada, mas não adote um método HTTP para apenas uma ação CRUD.

Plural, Singular ou Ambos?

Alguns desenvolvedores decidem nomear todos os pontos de acesso no singular, mas eu tenho problemas com isso. Em `/user/1` e `/user`, qual usuário é retornado pelo último ponto? Sou "eu"? Que tal `/place`? Ele retorna vários locais? Bah.

Eu sei que é tentador criar `/user/1` e `/users` porque dois pontos de acesso realizam tarefas diferentes, não é mesmo? Eu já trilhei este caminho no meu plano original, mas em minha experiência isso não funcionou bem. É lógico que funciona em exemplos como *"users"*, mas e aquelas palavras inglesas que fogem à regra como `/opportunity/1` que viraria `/opportunities`?

Eu escolhi o plural para tudo, pois ele é mais óbvio:

- `/places` - “Se eu executar GET neste ponto eu obterei uma coleção de locais”
- `/places/45` - “É claro que estou falando sobre o local 45”
- `/places/45,28` - “Ah sim, locais 45 e 28, entendido”

Outro motivo sólido para o uso consistente do plural é que ele também permite nomear sub-recursos consistentemente:

- `/places`
- `/places/45`
- `/places/45/checkins`
- `/places/45/checkins/91`
- `/checkins/91`

Consistência é a chave.

2.3 Planejando Pontos de Acesso

Controladores

Precisa de uma lista de eventos, locais, usuários e categorias? Fácil. Um controlador para cada tipo de recurso:

- CategoriesController
- EventsController
- UsersController
- VenuesController

Em REST, tudo é um recurso. Sendo assim, cada recurso precisa de um controlador.

À frente veremos algumas coisas que não são recursos. Às vezes, um sub-recurso pode ser apenas um método. Por exemplo, perfil e configurações são sub-recursos de `Users`, por isso eles podem fazer parte do controlador de usuários. Essas regras são flexíveis.

Rotas

Resista à tentação de usar convenções mágicas de roteamento[2], faça-as manualmente. Darei continuidade aos exemplos anteriores e demonstrarei o processo de transformar o plano de ações em rotas. Usarei a sintaxe do Laravel 4, por que não:

[2] http://philsturgeon.co.uk/blog/2013/07/beware-the-route-to-evil

Ação	Ponto de Acesso	Rota
Criar	POST /users	Route::post('users', 'UsersController@create');
Ler	GET /users/X	Route::get('users/{id}', 'UsersController@show');
Atualizar	POST /users/X	Route::post('users/{id}', 'UsersController@update');
Excluir	DELETE /users/X	Route::delete('users/{id}', 'UsersController@delete');
Listar	GET /users	Route::get('users', 'UsersController@list');
Imagem	PUT /users/X/image	Route::put('users/{id}/image', 'UsersController@uploadImage');
Favoritos	GET /users/X/favorites	Route::get('users/{id}/favorites', 'UsersController@favorites');
Checkins	GET /users/X/checkins	Route::get('users/{user_id}/checkins', 'CheckinsController@index');

Algumas precisam ser consideradas:

1. Tanto a criação quanto a atualização usam o método `POST`. Não que o método `PUT` seja mal, mas porque neste exemplo não conhecemos a URL quando criamos um usuário. As URLs são geradas automaticamente baseadas no ID do usuário. Por exemplo, se conhecêssemos o nome de usuário, teríamos usado `PUT /users/philsturgeon`. Mas isso difilmente funcionaria caso fizessemos a mesma *request* HTTP acidentalmente uma segunda vez ou caso tentássemos substituir um usuário existente.
2. Favoritos fazem parte do controlador `UserController`, pois eles são relevantes ao usuário.
3. *Checkins* fazem parte do controlador `CheckinController`, pois ele já cuida da rota `/checkins` e a sua lógica é basicamente idêntica. Nós saberemos se há um parâmetro `user_id` na URL, se o nosso roteador for gentil o bastante para nos informar, e assim poderemos usá-lo para tornar o *check-in* específico àquele usuário.

Esse dois últimos são um tanto complexos, mas são exemplos de coisas que você já deve estar pensando neste estágio. Você não vai querer múltiplos pontos de acesso para coisas similares copiando e colando a lógico porque: A) o Detector de Copia e Cola PHP[3] ficaria zangado; b) seu desenvolvedor para iOS ficaria bravo porque pontos de acesso diferentes estão agindo diferentemente e confundindo o RestKit; e, C) porque isso é chato e ninguém tem tempo pra perder com isso.

Métodos

Quando você terminar de listar todas as rotas necessárias para a sua aplicação e ligá-las aos seus controladores, esvazie-as e faça com que uma rota retorne "Oh hai!". Verifique no navegador. Por exemplo, `GET /places` deve exibir `Oh hai!`. Pronto, você acabou de escrever uma API!

[3] https://github.com/sebastianbergmann/phpcpd

3 Aprenda Mais

Você acabou de ler o **Capítulo 2** de Desenvolvendo APIs que você não odiará[1], há muito mais conteúdo como este na edição completa. Este são os capítulos disponíveis até este momento:

1. Semeando o Banco de Dados
2. Planejando e Criando Pontos de Destino
3. Teoria de Entrada e Saída
4. Códigos de Status, Erros e Mensagens
5. Testando Pontos de Destino
6. Exibindo Dados

Mas não é só isso, alguns dos próximos assuntos incluem:

- Incorporando/Aninhando Dados
- Debugando
- Paginando
- Links
- Criando Classes "Localizadoras" para Controladores e Modelos "Magros"
- Autenticação (Digst, Generic Key ou OAuth 2)
- Logging
- Throttling
- Considerações sobre Performance
- Suportando Multiplas Versões (URL ou Cabeçalhos)

Este livro está disponível no Leanpub[2], vá até lá e adquira a sua cópia.

[1] https://leanpub.com/desenvolvendo-apis
[2] https://leanpub.com/desenvolvendo-apis