



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



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2013 - 2014 Phil Sturgeon e Pedro Borges

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

Nota do Autor	i
Introdução	ii
Código dos Exemplos	iii
1 Semeando o Banco de Dados	1
1.1 Introdução	1
1.2 Criando Semeadores	2
1.3 É Isso Aí	4
1.4 Dados Secundários	5
1.5 Quando Executar Semeadores?	9
2 Planejando e Criando Pontos de Acesso	10
2.1 Requerimentos Funcionais	10
2.2 Teoria dos Pontos de Acesso	13
2.3 Planejando Pontos de Acesso	15
3 Teoria de Entrada e Saída	17
3.1 Requisições	17
3.2 Respostas	19
3.3 Suportando Formatos	20
3.4 Estrutura do Conteúdo	23
4 Códigos de Status, Erros e Mensagens	28
4.1 Códigos de Status HTTP	28
4.2 Códigos e Mensagens de Erro	29
4.3 Armadilhas Comuns	32
5 Testando Pontos de Acesso	33
5.1 Conceitos & Ferramentas	33
5.2 Preparação	33
5.3 Iniciando	34
5.4 Funcionalidades	34
5.5 Cenários	35
5.6 Preparando o Behat	38
5.7 Executando o Behat	38
6 Exibindo Dados	39

CONTEÚDO

6.1	O Método Direto	40
6.2	Transformações com Fractal	43
6.3	Ocultando Atualizações no Esquema	48
6.4	Exibindo Erros	49
6.5	Testando Esta Saída	52
6.6	Dever de Casa	54
7	Relações Entre Dados	55
7.1	Introdução	55
7.2	Sub-Recursos	55
7.3	Arrays de Chaves Externas	56
7.4	Documentos Compostos (Carregamento Lateral)	57
7.5	Documentos Incorporados (Aninhamento)	58
8	Depurando	65
8.1	Introdução	65
8.2	Depurando na Linha de Comando	65
8.3	Depurando no Navegador	65
8.4	Depurando Através da Rede	71
9	Autenticação	76
9.1	Introdução	76
9.2	Quando a Autenticação É Útil?	76
9.3	Diferentes Métodos de Autenticação	77
9.4	Implementando um Servidor OAuth 2.0	85
9.5	Onde o Servidor OAuth 2.0 Vive?	86
10	Paginação	87
10.1	Introdução	87
10.2	Paginadores	88
10.3	Deslocamentos e Cursorres	91
11	Em Breve	94

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 posso uma longa lista de como desenvolvê-las sem criar um monstro. Eu gostaria de compartilhar esta informação com todos vocês.

Phil Sturgeon

Introdução

Eu desenvolvo APIs há vários anos e elas têm se tornado cada vez mais comum na vida dos desenvolvedores *server-side* graças a ascensão dos *frameworks* JavaScript, aplicativos móveis e arquiteturas centradas em APIs. Por um lado você apenas captura os dados de uma fonte e os repassa como JSON, mas sobreviver às mudanças na lógica de programação, atualizações no esquema do banco de dados, novidades e depreciações pode rapidamente complicar as coisas.

Descobri que a maioria dos recursos disponíveis são incompletos ou voltados unicamente para um *framework*. Muitos livros e tutoriais usam maçãs e peras como exemplos, que não são concretos o suficiente, ou falam como se listar `/users` e `users/1` fossem os únicos pontos de acesso que você precisará acessar em uma API. Neste último ano eu trabalhei em uma empresa chamada Kapture, onde minha principal função foi herdar, refazer, manter e desenvolver uma API com diversos pontos de acesso expondo uma grande quantidade de casos de uso.

A API em questão estava em sua versão 2 e usava o *framework* FuelPHP quando entrei na empresa. Ela utilizava um ORM que já foi descontinuado por seu desenvolvedor original. A Kapture estava no processo de refazer sua aplicação para iPhone para implementar novas funções. Eu aproveitei a oportunidade para acabar com essa bagunça e desenvolver a versão 3 usando o Laravel 4, tirando vantagem do seu Roteador, Migração de Banco de Dados, Esquema, Semeador, etc. Agora estamos fazendo o mesmo com a versão 4, mas desta vez não foi preciso reescrever nada; embora algumas funcionalidades sejam diferentes, o repositório da versão 3 foi forcado para a versão 4 e ambas estão sendo mantidas e rodam lado-a-lado nos mesmos servidores de API.

Com as boas práticas e conselhos gerais compartilhados neste livro, você que é novo no desenvolvimento de API terá uma boa base para começar. Do outro lado, ao relatar algumas histórias horripilantes (e como elas foram superadas ou evitadas) espero ajudá-lo a evitar as mesmas armadilhas em que eu caí, quase caí ou presenciei outros caírem. Este livro falará sobre a teoria do planejamento e desenvolvimento de APIs em qualquer linguagem de programação ou *framework*. Estas teorias serão aplicadas em exemplos em PHP. Evitarei código em excesso para que você não durma e também para manter os programadores de outras linguagens contentes.

Alguns dos assuntos mais avançados cobertos neste livro incluem: testando pontos de acesso, incorporando dados de objetos de um modo consistente e escalável, paginando respostas (incluindo objetos incorporados) e *links* HATEOAS.

Código dos Exemplos

Ao longo deste livro irei me referir ao código-fonte existente em um repositório no GitHub. Você pode cloná-lo:

```
git clone git@github.com:philsturgeon/build-apis-you-wont-hate.git
```

Navegar *online*:

<https://github.com/philsturgeon/build-apis-you-wont-hate>

Ou baixar o arquivo .zip:

<http://bit.ly/apisyouwonthate-zip>

Este repositório contém alguns exemplos que te pouparão de copiar e colar os exemplos deste *ebook*; o que seria uma experiência horrível.

1 Semeando o Banco de Dados

O primeiro passo no desenvolvimento de qualquer aplicação é a criação de um banco de dados. Se você optar por algum tipo de banco de dados relacional, MongoDB, Riak, ou um outro qualquer, você precisará ter uma ideia de como os seus dados serão armazenados.

Para os bancos de dados relacionais, é bem provável que você começará planejando um diagrama com as relações de entidades. Para sistemas baseados em documentos, tais como MongoDB, CouchDB ou ElasticSearch, você deixará com que a aplicação magicamente cuide disso; mas de qualquer forma, você precisará traçar um plano, nem que seja em um guardanapo. Este livro assume que os seus dados serão armazenados no tradicional banco de dados relacional, mas os princípios poderão ser facilmente adaptados para os sistemas NoSQL.

Este capítulo assume que você já possui um banco de dados planejado e pronto. Eu vou pular a seção “planejando um banco de dados” pois ela é terrivelmente entediante, e já existem dezenas de outros bons livros sobre o assunto. Vamos direto para o próximo passo.

1.1 Introdução

Em posse daquele plano para o seu banco de dados, você precisará preenchê-lo com dados fictícios. Isto trás consigo alguns benefícios, pois além de possuir alguns dados para “brincar”, isso te ajudará a aprimorar a estrutura planejada caso ela apresente alguma falha.

Com um banco de dados planejado e implementado, é hora de armazenar alguns dados, mas ainda não é hora dos dados reais. É muito mais fácil usar “dados de mentira” para testar se o esquema é apropriado para a API da sua aplicação, assim você pode se livrar dele e tentar novamente sem o risco de perder dados importantes.

O processo de povoar um banco de dados é conhecido como [seeding](#)¹.

Estes dados podem ser usuários de teste, conteúdos com vários comentários, locais fictícios disponíveis para *checkin*, notificações falsas a serem exibidas numa aplicação para iPhone (uma de cada tipo) ou pagamentos realizados com cartão de crédito em vários estágios do processamento (alguns completos, alguns em andamento e outros que aparentam ser fraudulentos).

Estes dados serão muito úteis e você não precisará perder tempo criando-os manualmente vez após vez; porque quando você digitar esses dados manualmente na correria, quase sempre você se esquecerá de algo que já deveria ter sido considerado.

O processo de criar um *script* semeador ajudará você a não perder tempo digitando dados manualmente vez após vez. Além do mais, quanto mais automatizado for o processo de desenvolvimento da API, mais tempo você terá para considerar as peculiaridades das suas aplicações.

¹semeadura, em português.

Dados fictícios são necessários para testes de aceitação mais realísticos, para acelerar a adaptação de novos membros e *freelancers* na equipe, fornecendo-lhes conteúdo útil, para evitar que informações dos seus clientes caiam na mão de terceiros e para evitar a tentação de copiar dados reais para os ambientes de desenvolvimento.

Por que usar dados da produção no desenvolvimento é ruim?

Alguma vez você escreveu um *script* que envia e-mails usando um texto fictício durante o seu desenvolvimento? Já usou palavras indevidas neste texto? Você já enviou accidentalmente esse e-mail para 10 mil clientes de verdade? Você já foi despedido por dar um prejuízo de £ 200 mil a uma empresa?

Bem... eu também não! Mas eu conheço alguém que já passou por isso. Não seja como ele.

Quais dados você deve usar?

Lixo! Use coisas sem noção para o banco de dados do desenvolvimento. Mas use o tipo de dado, tamanho e formato corretos. Isso pode ser alcançado com uma biblioteca muito divertida chamada [Faker](#)² por [François Zaninotto](#)³.

1.2 Criando Semeadores

Na Kapture, nós usamos o *framework* Laravel, que já possui um [semeador de banco de dados](#)⁴ embutido. Ele é essencialmente uma tarefa de linha de comando (CLI) que quase todos os *frameworks* PHP modernos oferecem (ou pelo menos deveriam oferecer); portanto estes princípios são aplicáveis a todos eles.

Desmembre seu semeador de banco de dados em agrupamentos lógicos. Não precisa ser uma classe semeadora para cada tabela, mas pode ser. Eu não me atendo a essa regra porque às vezes os dados precisarão ser construídos ao mesmo tempo que outros dados relacionados. Por isso, nossos usuários são “semeados” no banco de dados ao mesmo tempo que suas configurações, códigos OAuth e amizades. Colocar esses dados em múltiplos semeadores simplesmente para mantê-los bonitinhos seria pura futilidade e nos desaceleraria sem necessidade.

Assim, esta é uma versão drasticamente simplificada do nosso semeador de usuários, ignorando a estrutura específica do Laravel.

Se você está usando o Laravel 4, apenas coloque isso dentro do método `run()`.

²<https://github.com/fzaninotto/Faker>

³<https://twitter.com/francoisz>

⁴<http://laravel.com/docs/migrations#database-seeding>

Criando um usuário com o Faker e o ORM Eloquent

```

1 $faker = Faker\Factory::create();
2
3 for ($i = 0; $i < Config::get('seeding.users'); $i++) {
4
5     $user = User::create([
6         'name'          => $faker->name,
7         'email'         => $faker->email,
8         'active'        => $i === 0 ? true : rand(0, 1),
9         'gender'        => rand(0, 1) ? 'male' : 'female',
10        'timezone'      => mt_rand(-10, 10),
11        'birthday'      => rand(0, 1) ?
12                           $faker->dateTimeBetween('-40 years', '-18 years') : null,
13        'location'      => rand(0, 1) ?
14                           "{$faker->city}, {$faker->state}" : null,
15        'had_feedback_email' => (bool) rand(0, 1),
16        'sync_name_bio'   => (bool) rand(0, 1),
17        'bio'            => $faker->sentence(100),
18        'picture_url'    => $this->picture_url[rand(0, 19)],
19    ]);
20 }
```

O que temos aqui? Vamos entender melhor cada seção:

```
$faker = Faker\Factory::create();
```

Criamos uma instância do Faker, o artista contratado para realizar este serviço sujo.

```
for ($i = 0; $i < Config::get('seeding.users'); $i++) {
```

Nós precisaremos de uma certa quantidade de usuários, mas por causa do tempo, eu recomendo que você tenha um pouco menos no ambiente de desenvolvimento do que nos ambientes de teste ou *staging*.

```
$user = User::create([
    'name'  => $faker->name,
    'email' => $faker->email,
```

Aqui um nome e um endereço de e-mail é gerado aleatoriamente. Não precisamos definir a fonte dos dados utilizados pelo Faker; ele é pura MÁGICA!

```
'active' => $i === 0 ? true : rand(0, 1),
```

Certo, eu menti. Nossa lixo não é 100% aleatório. Nós queremos que o usuário número 1 seja ativo para testes futuros.

```
'gender' => $faker->randomElement(['male', 'female']),
```

Igualdade dos sexos é importante.

```
'timezone' => mt_rand(-10, 10),
```

O desenvolvedor original decidiu que armazenar fusos horários usando números inteiros seria inteligente. O que vamos fazer com os países com fuso horário +4,45, cara? Eu ainda preciso refazer esta parte, mas tudo bem por enquanto.

```
'birthday' => rand(0, 1) ?
    $faker->dateTimeBetween('-40 years', '-18 years') : null,
```

Usuários dentro da nossa faixa etária de atuação.

```
'location' => rand(0, 1) ? "{$faker->city}, {$faker->state}" : null,
```

Isso nos dá o nome de uma cidade e de um estado. O legal é que ele funciona bem com países estrangeiros também.

```
'had_feedback_email' => $faker->boolean,
'sync_name_bio'       => $faker->boolean,
```

Alguns campos não são tão importantes para nos preocuparmos agora. Verdadeiro ou falso, tanto faz.

```
'bio' => $faker->sentence(100),
```

Cria uma sentença com 100 com caracteres.

1.3 É Isso Aí

No fim, você terá muitos desses arquivos e vai querer preencher cada uma das suas tabelas com dados. Você também vai querer que o semeador do banco de dados apague todos os dados antes de preenchê-lo com novos dados. Faça isso globalmente bem no início do processo.

Exemplo de um sistema global no Laravel 4

```
1 class DatabaseSeeder extends Seeder
2 {
3     public function run()
4     {
5         if (App::environment() === 'production') {
6             exit('Acabei de evitar sua demissão. Te amo Phil!');
7         }
8
9         Eloquent::unguard();
10
11     $tables = [
12         'locations',
13         'merchants',
14         'opps',
15         'opps_locations',
16         'moments',
17         'rewards',
18         'users',
19         'oauth_sessions',
20         'notifications',
21         'favorites',
22         'settings',
23         'friendships',
24         'impressions',
25     ];
26
27     foreach ($tables as $table) {
28         DB::table($table)->truncate();
29     }
30
31     $this->call('MerchantTableSeeder');
32     $this->call('PlaceTableSeeder');
33     $this->call('UserTableSeeder');
34     $this->call('OppTableSeeder');
35     $this->call('MomentTableSeeder');
36 }
37 }
```

Isso limpa tudo antes de executar as demais classes semeadoras.

1.4 Dados Secundários

Como eu já disse, é bem provável que você precisará inserir dados que relacionam entre si. Para fazer isto, você precisará elaborar qual dado será o primário (“usuários” por exemplo) ou, no caso

de um sistema de *checkin*: “locais” ou “estabelecimentos”; dependendo da nomeclatura do seu sistema.

Para este exemplo, vou demonstrar como criar *merchants*⁵ e anexar *opportunities*⁶, que basicamente são “campanhas”.

Semeador Primário para a Tabela Merchant

```

1 <?php
2
3 class MerchantTableSeeder extends Seeder
4 {
5     /**
6      * Executa as sementes do banco de dados.
7      *
8      * @return void
9      */
10    public function run()
11    {
12        $faker = Faker\Factory::create();
13
14        // Cria qualquer tanto de comerciantes
15        for ($i = 0; $i < Config::get('seeding.merchants'); $i++) {
16            Merchant::create([
17                'name'          => $faker->company,
18                'website'       => $faker->url,
19                'phone'         => $faker->phoneNumber,
20                'description'   => $faker->text(200),
21            ]);
22        }
23    }
24 }
```

Semeador Primário para a Tabela Opp

```

1 <?php
2
3 use Carbon\Carbon;
4 use Kapture\CategoryFinder;
5
6 class OppTableSeeder extends Seeder
7 {
8     /**
9      * Construa-o.
```

⁵Comerciantes.

⁶Oportunidades.

```
10  *
11  * @param Place
12  */
13  public function __construct(CategoryFinder $finder, Place $places)
14  {
15      $this->categoryFinder = $finder;
16      $this->places = $places;
17  }
18
19 /**
20  * Imagens.
21  *
22  * @var string
23  */
24 protected $imageArray = [
25     'http://example.com/images/example1.jpg',
26     'http://example.com/images/example2.jpg',
27     'http://example.com/images/example3.jpg',
28     'http://example.com/images/example4.jpg',
29     'http://example.com/images/example5.jpg',
30 ];
31
32 /**
33  * Executa as sementes do banco de dados.
34  *
35  * @return void
36  */
37 public function run()
38 {
39     $faker = Faker\Factory::create();
40
41     foreach (Merchant::all() as $merchant) {
42
43         // Cria uma quantidade qualquer de
44         // oportunidades para este comerciante
45         foreach (range(1, rand(2, 4)) as $i) {
46
47             // Há 3 tipos de imagens para inserir
48             $image = Image::create([
49                 'name' => "{$merchant->name} Imagem #{$i}",
50                 'url' => $faker->randomElement($this->imageArray),
51             ]);
52
53             // Começa imediatamente e dura 2 meses
54             $starts = Carbon::now();
55 }
```

```

56     // Precisamos inserir pelo menos um que conheçamos
57     if ($i === 1) {
58         // Crie UM que se encerra em breve
59         $ends = Carbon::now()->addDays(2);
60         $teaser = 'Algo sobre queijo';
61
62     } else {
63         $ends = Carbon::now()->addDays(60);
64         $teaser = $faker->sentence(rand(3, 5));
65     }
66
67     $category = $this->categoryFinder->setRandom()->getOne();
68
69     $opp = Opp::create([
70         'name'          => $faker->sentence(rand(3, 5)),
71         'teaser'         => $teaser,
72         'details'        => $faker->paragraph(3),
73         'starts'         => $starts->format('Y-m-d H:i:s'),
74         'ends'           => $ends->format('Y-m-d H:i:s'),
75         'category_id'    => $category->id,
76         'merchant_id'   => $merchant->id,
77         'published'      => true,
78     ]);
79
80     // Anexe um local a esta oportunidade
81     $opp->images()->attach($image, [
82         'published' => true
83     ]);
84 }
85
86     echo "$i oportunidades criadas para $merchant->name \n";
87 }
88 }
89 }
```

Isso pode parecer um pouco maluco, pois é uma mistura de métodos estáticos do ORM usados no controlador e um pouco de injeção de dependência; é porque estes semeadores não receberam muito amor. É claro que eles fazem o que esperamos e sempre poderão melhorar, mas o básico é isso:

```
foreach (Merchant::all() as $merchant) {
```

Percorre todos os “comerciantes”.

```
// Cria uma quantidade qualquer de
// oportunidades para este comerciante
foreach (range(1, rand(2, 4)) as $i) {
```

Cria de 1 a 4 oportunidades para um comerciante.

```
// Há 3 tipos de imagens para inserir
$image = Image::create([
    'name' => "{$merchant->name} Image #{$i}",
    'url' => $faker->randomElement($this->imageArray),
]);
```

Inseri uma imagem do *array* de imagens armazenadas no S3 ou em algum lugar em nosso *website*. Quanto mais melhor.

```
$category = $this->categoryFinder->setRandom()->getOne();
```

Falarei mais sobre *finders*⁷ em outro capítulo, apenas entenda que é uma forma conveniente de se obter uma categoria aleatória.

O restante deve ser relativamente óbvio.

Se você está usando o Laravel 4, basta executar o código acima com o comando: `$ php artisan db:seed`.

1.5 Quando Executar Semeadores?

Geralmente eles são executados manualmente, ou automaticamente dependendo da instância.

Por exemplo, se você acabou de acrescentar um novo ponto de acesso⁸ com dados novos, você obviamente vai quer que os demais membros da sua equipe baixem o último código, executem a migração e as sementes do banco de dados.

Isso também será muito útil quando você contratar um *freelancer*, quando um novo desenvolvedor for admitido ou o seu desenvolvedor para iOS precisar de alguns dados. Em todas estas instâncias, o comando acima precisará ser executado apenas uma vez.

Ocasionalmente este comando também é executado manualmente no servidor *staging* e automaticamente no servidor de testes Jenkins quando “lançamos” novas atualizações da API.

⁷Localizadores.

⁸*endpoint*, em inglês.

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 acesso¹. 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.

Definitivamente 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)
- **Imagen**

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-recursos e URLs com dados incorporados. Bem, dados incorporados é um assunto tão difícil que falaremos mais sobre eles no futuro. Por enquanto a resposta 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.

Excluindo 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 roteamento²](#), 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');
Imagen	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 dificilmente funcionaria caso fizessemos a mesma *request* HTTP accidentalmente 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 hail!”. Verifique no navegador. Por exemplo, GET /places deve exibir Oh hail!. Pronto, você acabou de escrever uma API!

³<https://github.com/sebastianbergmann/phpcpd>

3 Teoria de Entrada e Saída

Agora que você já possui uma boa ideia de como um ponto de acesso funciona, o próximo copo de teoria que tomaremos será sobre entrada¹ e saída². Esta é a parte mais fácil, pois estamos falando apenas de “requisições” e “respostas” HTTP, que é o mesmo que AJAX ou qualquer outra coisa. Se você já foi forçado a trabalhar com SOAP, você deve saber tudo sobre WSDLs. Se você sabe o que eles são, alegre-se pois você não precisará mais usá-los. Se não sabe o que é isso, alegre-se por nunca ter sido obrigado a aprendê-lo. SOAP é horrível.

Entrada é puramente uma requisição³ HTTP composta por muitas partes. Vejamos alguns exemplos:

3.1 Requisições

```
GET /places?lat=40.759211&lon=-73.984638 HTTP/1.1
Host: api.example.com
```

Esta é uma requisição GET bem simples. Podemos ver que a URL requerida é `/places` com a *string* de consulta `lat=40.759211&lon=-73.984638`. A versão do HTTP usada foi HTTP/1.1 e o nome do *host* também foi definido. Isto é basicamente o que o seu navegador faz sempre que você visita um site. É um pouco entediante, eu sei.

```
POST /moments/1/gift HTTP/1.1
Host: api.example.com
Authorization: vr5HmMkzlxKE70W1y4MibiJUusZwZC25NOVBEx3BD1
Content-Type: application/json

{ "user_id" : 2 }
```

Aqui estamos fazendo uma requisição POST com um HTTP Body. O cabeçalho Content-Type indica que estamos enviando JSON e a linha em branco acima do JSON separa o “cabeçalho” do “corpo” do HTTP. HTTP é supreendentemente simple, isso é tudo o que você precisa fazer para qualquer coisa. Você ainda pode usar qualquer cliente HTTP em qualquer linguagem que você queira para fazer isso:

¹*input*, em inglês.

²*output*, em inglês.

³*request*, em inglês.

Usando PHP e a biblioteca Guzzle HTTP para fazer uma requisição HTTP

```
1 use Guzzle\Http\Client;
2
3 $headers = [
4     'Authorization' => 'vr5HmMkzIxKE70W1y4MibiJUusZwZC25NOVBEx3BD1',
5     'Content-Type' => 'application/json',
6 ];
7 $payload = [
8     'user_id' => 2
9 ];
10
11 // Cria um cliente com uma URL base
12 $client = new Client('http://api.example.com');
13
14 $req = $client->post('/moments/1/gift', $headers, json_encode($payload))
```

Usando Python e a biblioteca Requests HTTP para fazer uma requisição HTTP

```
1 import requests
2
3 headers = {
4     'Authorization': 'vr5HmMkzIxKE70W1y4MibiJUusZwZC25NOVBEx3BD1',
5     'Content-Type': 'application/json',
6 }
7 payload = {
8     'user_id': 2
9 }
10 req = requests.post('http://api.example.com/moments/1/gift',
11                      data=json.dumps(payload), headers=headers)
```

É tudo a mesma coisa. Defina os cabeçalhos, defina o corpo em um formato apropriado e mande-o embora. Em seguida você receberá uma resposta, então vamos falar sobre elas.

3.2 Respostas

Assim como uma requisição HTTP, sua resposta HTTP também será texto puro (a menos que você esteja usando SSL, ainda vamos chegar lá).

Exemplo de uma resposta HTTP contendo JSON em seu corpo

```
HTTP/1.1 200 OK
Server: nginx
Content-Type: application/json
Connection: close
X-Powered-By: PHP/5.5.5-1+debphp.org~quantal+2
Cache-Control: no-cache, private
Date: Fri, 22 Nov 2013 16:37:57 GMT
Transfer-Encoding: Identity

{"id":1690,"is_gift":true,"user":{"id":1,"name":"Theron Weissnat","bio":"Occae\
cati excepturi magni odio distinctio dolores illum voluptas voluptatem in repe\
llendus eum enim ","gender":"female","picture_url":"https://si0.twimg.com/p\
rofile_images/711293289/hhd1-twitter_normal.png","cover_url":null,"location"\
:null,"timezone":-1,"birthday":"1989-09-17 16:27:36","status":"available","cre\
ated_at":"2013-11-22 16:37:57","redeem_by":"2013-12-22 16:37:57"}
```

Podemos notar algumas coisas bem óbvias aqui. 200 OK é uma resposta padrão que quer dizer: “tudo certo aqui”. Nós temos Content-Type novamente e a API está declarando que esta resposta não deve ser guardada no *cache*. O cabeçalho X-Powered-By é um lembrete de que devo alterar expose_php = On para expose_php = Off no arquivo php.ini. Opsss.

Basicamente, é assim que a maioria das APIs funcionam. Assim como quando estamos aprendendo uma linguagem de programação, você sempre encontrará novas funções e utilitários que tornarão a sua API mais RESTful. Vou destacar muitas delas conforme avançamos, mas assim como a função `levenshtein()`⁴, você sempre descobrirá novos cabeçalhos HTTP que você nunca imaginou que existissem.

⁴<http://php.net/manual/en/function.levenshtein.php>

3.3 Suportando Formatos

Escolher quais formatos suportar é difícil, mas é possível facilitar esta tarefa no início.

Nada de Dados de Formulário

Desenvolvedores PHP tendem a fazer algo que quase ninguém mais faz, enviar dados para a API usando: `application/x-www-form-urlencoded`.

Este *mime-type* é um dos poucos usados pelos navegadores para enviar dados via formulários quando você usa HTTP POST. O PHP pega este dado, separa-o e o disponibiliza em `$_POST`. Porque este é um recurso conveniente, muitos desenvolvedores enviam os dados de sua API desta forma e depois se perguntam porque enviar dados usando PUT é “diferente”. Afff.

`$_GET` e `$_POST` não tem nada haver com HTTP GET e HTTP POST. `$_GET` contém apenas o conteúdo de uma *string* de consulta **independente** do método HTTP utilizado. `$_POST` contém os valores do corpo HTTP se ele estiver no formato correto e se o cabeçalho Content-Type for `application/x-www-form-urlencoded`.

Por isso, sabendo que algumas coisas em PHP possuem nomes idiotas, vamos avançar e ignorar a variável `$_POST` completamente. Jogue uma no chão, pois ela está morta para você.

Mas por quê? Por muitas razões, incluindo o fato de que tudo em `application/x-www-form-urlencoded` é apenas uma *string*.

```
foo=something&bar=1&baz=0
```

Isso mesmo, você precisa usar 1 ou 0 porque `bar=true` seria `string("true")` no servidor. O tipo do dado é algo importante, por isso não vamos ignorá-los com a desculpa de “fácil acesso aos nossos dados.” Este argumento também é imbecil porque é possível usar `Input::json('foo')` na maioria dos *frameworks* decentes, porém mesmo sem esta conveniência, você só precisa de `file_get_contents('php://input')` para ler o conteúdo do corpo HTTP por conta própria.

```
POST /checkins HTTP/1.1
Host: api.example.com
Authorization: vr5HmMkzlxKE70W1y4MibiJUusZwZC25NOVBEx3BD1
Content-Type: application/json

{
    "checkin": {
        "place_id" : 1,
        "message": "Um monte de texto.",
        "with_frinds": [1, 2, 3, 4, 5]
    }
}
```

Este é um corpo HTTP perfeitamente válido para um *checkin*. Você sabe o que ele está dizendo, você sabe quem é o usuário baseado no código de autorização, você sabe com quem ele está e você ainda tem o benefício de ter tudo isso agrupado em uma chave `checkin` para simplificar a documentação e facilitar as respostas: “Você enviou um objeto `checkin` para a página de configuração do usuário...”

Esta mesma requisição usando dados de formulário é uma bagunça.

```
POST /checkins HTTP/1.1
Host: api.example.com
Authorization: vr5HmMkzlxKE70W1y4MibiJUusZwZC25NOVBEx3BD1
Content-Type: application/x-www-form-urlencoded

checkin[place_id]=1&checkin[message]=Um monte de texto.&checkin[with_frinds][]=
=1&checkin[with_frinds][]=2&checkin[with_frinds][]=3&checkin[with_frinds][]=4&
checkin[with_frinds][]=5
```

Isso me deixa chateado e furioso. Não faça isso na sua API.

Para finalizar, não banque o esperto tentando misturar JSON com dados de formulários:

```
POST /checkins HTTP/1.1
Host: api.example.com
Authorization: vr5HmMkzlxKE70W1y4MibiJUusZwZC25NOVBEx3BD1
Content-Type: application/x-www-form-urlencoded

json={
  "checkin": {
    "place_id": 1,
    "message": "Um monte de texto.",
    "with_frinds": [1, 2, 3, 4, 5]
  }
}"
```

Quem o desenvolvedor está tentando impressionar com uma coisa dessas? Isso é loucura e quem insistir nisso deve ter o distintivo e arma confiscados.

JSON e XML

Qualquer API moderna suportará JSON, a menos que a API seja de um serviço financeiro ou o desenvolvedor seja um idiota - ou talvez os dois. Alguns suportam XML também. XML era um formato popular para transferência de dados tanto no universo SOAP quanto no XML-RPC (duh). Porém XML é uma bagunça de *tags* e o tamanho de um arquivo XML contendo os mesmos dados que um arquivo JSON é muito maior.

Além do tamanho dos dados armazenados, XML é terrível para se armazenar o “tipo” desses dados. Isso pode não preocupar os desenvolvedores de PHP, porque o próprio PHP não é melhor em nada neste assunto; mas dê uma olhada nisso:

```
{
  "place": {
    "id" : 1,
    "name": "Um monte de texto.",
    "is_true": false,
    "maybe": null,
    "empty_string": ""
  }
}
```

A mesma resposta em XML:

```
<places>
  <place>
    <id>1</id>,
    <name>Um monte de texto.</name>
    <is_true>0</is_true>
    <maybe />
    <empty_string />
  </place>
</places>
```

Em XML, basicamente *tudo* é considerado uma *string*. Isso quer dizer que podemos confundir *integers*, *booleans* e *nulls*. Tanto *maybe* quanto *empty_string* possuem o mesmo valor, pois não há uma forma de representar *null*. Grosseiro...

Os mais sabidos em XML devem estar se perguntando por que eu não usei atributos. Bem, esta estrutura XML foi gerada automaticamente na conversão de um *array*, da mesma forma que a *string* JSON foi gerada, mas é claro que os atributos foram ignorados e não é possível especificar a estrutura que a maioria dos consumidores de XML exigirão.

E se você começar a usar atributos para alguns dados mas não para outros, o código de conversão se tornará INSANAMENTE complicado. Como produziremos algo assim?

```
<places>
  <place id="1" is_true="1">
    <name>Um monte de texto.</name>
    <empty_string />
  </place>
</places>
```

A resposta é: a menos que você procure por campos específicos e tente adivinhar que um “*id*” provavelmente é um atributo, etc; não há uma forma prática de sua API pegar um mesmo *array* e transformá-lo em JSON e XML. Pelo contrário, você realisticamente precisa usar um “*view*” (do padrão MVC) para representar este dado assim como você faria com HTML ou usar um gerador

de XML de uma forma mais orientada a objeto. Felizmente, ninguém na Kapture quer saber de XML, assim eu ainda não preciso voltar para a Inglaterra.

Se a sua equipe ainda está em cima do murro quanto ao formato XML e vocês não sabem se precisarão usá-lo, então não se preocupe com ele. Eu sei que é legal mostrar que sua API pode alternar formatos e suportar diversas coisas, mas eu recomendo veementemente que você use apenas o(s) formato(s) que você usará. É claro que o Flickr suporta “lolcat” com entrada e saída, mas eles possuem uma equipe muito maior, por isso deixe o “lolcat” pra lá. JSON já está de bom tamanho. Se há muitos irmão do Ruby por perto, você provavelmente vai querer suportar YML também, que na maioria dos casos é tão fácil de produzir quanto JSON.

3.4 Estrutura do Conteúdo

Este é um assunto difícil e não há uma resposta correta. Se você usa EmberJS, RestKit ou um outro *framework* com conhecimento de REST, você encontrará alguém para reclamar que os dados não estão em seu formato predileto. Há muitos fatores e eu vou apenas explicar todos eles e deixar você saber qual eu escolhi.

JSON API

Este é um formato recomendado em [JSON API⁵](#) que talvez todos queiram usar. Ela sugere que, tanto um recurso, quanto uma coleção de recursos devem estar em uma chave no plural.

```
{
  "posts": [
    {
      "id": "1",
      "title": "Rails é Omakase"
    }
  ]
}
```

Vantagens:

- Resposta consistente, *sempre* possui a mesma estrutura

Desvantagens:

- Alguns utilitários de dados RESTful ficam perdidos quando há apenas uma resposta em um *array*
- Potencialmente confusa para humanos

Por padrão, o EmberJS (EmberData) ficaria muito chateado com isso. Eu tive problemas tentando mudar o comportamento de retornar um *array* quando você requisitou apenas um item. Dá a impressão de que pode haver mais de um item. (Me) parece uma regra muito estranha. Imagine solicitar o usuário atual em /me e receber uma coleção?

Não descarte a JSON API completamente pois ela é um ótimo recurso para obtermos ideias legais, mas em diversas áreas ela me parece muito complicada.

⁵<http://jsonapi.org/format/>

Estilo Twitter

Peça um usuário, obtenha um usuário:

```
{  
  "name": "Phil Sturgeon",  
  "id": "511501255"  
}
```

Peça uma coleção de coisas e obtenha uma coleção:

```
[  
  {  
    "name": "Hulk Hogan",  
    "id": "100002"  
  },  
  {  
    "name": "Mick Foley",  
    "id": "100003"  
  }  
]
```

Vantagens:

- Resposta minimalista
- Quase todos os *frameworks/utilitários* conseguem compreendê-lo

Desvantagens:

- Não há espaço para paginação ou outros metadados

Esta é uma solução razoável se você não pretende paginar os resultados ou usar metadados.

Estilo Facebook

Peça um usuário, obtenha um usuário:

```
{  
  "name": "Phil Sturgeon",  
  "id": "511501255"  
}
```

Peça uma coleção de coisas e obtenha uma coleção, mas em um *namespace*:

```
{  
    "data": [  
        {  
            "name": "Hulk Hogan",  
            "id": "100002"  
        },  
        {  
            "name": "Mick Foley",  
            "id": "100003"  
        }  
    ]  
}
```

Vantagens:

- Há espaço para paginação e outros metadados em uma coleção
- Resposta simples mesmo com o *namespace* extra

Desvantagens:

- Um item sozinho só pode conter metadados se eles forem incorporados aos recursos do item

Colocando a coleção no *namespace* “data”, poderemos facilmente acrescentar outro contexto relacionado à resposta, mas que não é parte da lista de recursos. Contagens, *links* e etc poderão fazer parte da resposta (veremos mais sobre isso à frente). Assim também é possível incorporar outras relações aninhadas em seu próprio elemento “data” e até mesmo incluir metadados para essas relações incorporadas. Também veremos sobre isso em outra oportunidade.

A única desvantagem com o Facebook é que um recurso sozinho não possui *namespace*. Se adicionarmos qualquer tipo de metadados estaremos poluindo o *namespace* global; algo que os desenvolvedores PHP são totalmente contra após uma década desta prática.

O exemplo final (e aquele que comecei a usar na Kapture para a versão 4) é este:

Muito Namespace, Resultado Agradável

Use *namespace* em itens sozinhos.

```
{
  "data": {
    "name": "Phil Sturgeon",
    "id": "511501255"
  }
}
```

Use *namespace* para vários itens.

```
{
  "data": [
    {
      "name": "Hulk Hogan",
      "id": "100002"
    },
    {
      "name": "Mick Foley",
      "id": "100003"
    }
  ]
}
```

Este método é muito próximo da resposta da JSON API, mas possui os benefícios do método adotado pelo Facebook e é como o do Twitter, exceto que tudo possui um *namespace*. Algumas pessoas (incluindo eu no passado) vão sugerir que você altere “data” para “users”, mas quando você começar a aninhar dados você perceberá que é melhor dar este nome especial à relação. Por exemplo:

```
{
  "data": {
    "name": "Phil Sturgeon",
    "id": "511501255"
    "comments": {
      "data": [
        {
          "id": 123423
          "text": "MongoDB is web-scale!"
        }
      ]
    }
  }
}
```

Aqui você pode ver os benefícios de manter o escopo da raiz genérico. Nós sabemos que um usuário foi retornado porque foi isso que nós solicitamos. Quando a resposta inclui comentários,

nós o envolvemos em um item chamado “data” para que a paginação ou *links* possam ser acrescentados a estes dados aninhados também. Esta é a estrutura que estarei testando e usando como exemplo.

Ainda veremos sobre *links*, relações, *side-loading*, paginação e etc nos próximos capítulos. Mas esqueça-os por enquanto. Agora você só precisa se preocupar com a resposta, que consiste em um bloco de dados ou um erro.

4 Códigos de Status, Erros e Mensagens

Caso tudo vá bem, alguns dados serão exibidos. Se uma requisição válida for feita para um dado que existe, você enviará uma resposta contendo o dado solicitado. Quando uma informação válida for criada na API, você retornará o objeto recém-criado. Mas, e quando algo der errado? Precisaremos informar o erro usando dois métodos simultâneos:

1. Códigos de *status* HTTP
2. Códigos e mensagens de erro personalizados

4.1 Códigos de Status HTTP

Códigos de *status* são usados em todas as respostas e possuem um número que varia entre 200 e 507, com vários vãos no meio. Cada código possui uma mensagem e uma definição. O código padrão da maioria das linguagens *server-side*, *frameworks* e etc é 200 OK.

Esses códigos estão agrupados em algumas categorias:

2xx engloba tudo relacionado a sucesso

É qualquer coisa que o cliente tente fazer e concretize ao ponto de receber uma resposta de volta. Tenha em mente que *status* como o 202 Accepted não menciona nada sobre o resultado atual, ele apenas informa que a requisição foi aceita e está sendo processada de forma assíncrona.

3xx engloba tudo relacionado a redirecionamentos

Estes códigos enviam a aplicação que realizou a requisição para um local diferente do recurso atual. Os mais conhecidos são 303 See Other e 301 Moved Permanently que são muito usados na web para redirecionar o navegador para outra URL.

4xx engloba tudo relacionado a erros do cliente

Com estes códigos de *status* indicamos que o cliente fez algo inválido que precisa ser resolvido antes de reenviar a requisição.

5xx engloba tudo relacionado a erros do serviço

Com estes códigos indicamos que algum erro ocorreu no serviço. Por exemplo, quando não é possível conectar ao banco de dados. Tipicamente, o cliente pode refazer a requisição. O servidor pode até especificar quando o cliente está autorizado a tentar novamente usando o cabeçalho HTTP Retry-After.

Usando códigos de *status* HTTP em serviços REST¹, por Maurice de Beijer.

Para uma lista completa dos códigos de *status* HTTP e suas respectivas definições, consulte REST & WOA Wiki².

Desenvolvedores argumentarão pelo resto de suas vidas sobre qual é o código mais apropriado para uma determinada situação. Estes são os códigos de *status* que usamos na API da Kapture:

- 200 - Genérico para tudo que está ok
- 201 - Algo foi criado
- 202 - Aceito e sendo processado assincronamente (codificação de vídeos, redimensionamento de imagens e etc)
- 400 - Argumentos errados (validação ausente)
- 401 - Não autorizado (deve existir um usuário atual)
- 403 - Usuário atual não autorizado a acessar esta informação
- 404 - A URL não é uma rota válida ou o recurso solicitado não existe
- 410 - A informação foi excluída, desativada, suspensa e etc
- 405 - Método não permitido (seu *framework* provavelmente cuidará disso pra você)
- 500 - Algo inesperado aconteceu e é culpa da API
- 503 - API indisponível no momento, por favor tente mais tarde

É tentador tentar encaixar o maior número possível de códigos de erros, mas eu te aconselho a simplificar. Você não alcançará nada usando todos eles.

A maioria dos problemas de código 5xx acontecerão devido a falhas na arquitetura ou servidor, e não por falhas na sua API. Pode ser por falta de segmentação do PHP-FPM atrás do nginx (502), ou porque o Amazon Elastic Load Balancer não possui instâncias saudáveis (503) ou porque de alguma forma o disco rígido está cheio (507).

4.2 Códigos e Mensagens de Erro

Códigos de erro geralmente são *strings* ou números que agem como identificadores únicos correspondentes a uma mensagem de erro legível. Essas mensagens fornecem mais informação sobre o erro ocorrido. Pode parecer que já existem muitos códigos de *status* HTTP, mas estes erros são específicos à aplicação e podem ou não ter relação com as respostas HTTP.

Algumas pessoas tentam usar apenas códigos de *status* HTTP e evitam usar códigos de erro porque elas não gostam de criar os seus próprios códigos de erro e ter que documentá-los, porém esta não é uma abordagem escalável. Há situações em que o mesmo ponto de acesso facilmente retornaria o mesmo código de *status* para mais do que uma condição diferente. Códigos de *status* existem meramente para dar uma dica do que aconteceu, é interessante utilizar um código e uma mensagem de erro para oferecer mais informações ao cliente.

Por exemplo, um problema com o *token* de acesso sempre impedirá o reconhecimento do usuário. Um cliente desinteressado simplesmente diria: “Usuário não pode entrar”; enquanto um cliente mais interessado preferiria oferecer alguma sugestão através de uma mensagem na interface do aplicativo.

¹<http://www.develop.com/httpstatuscodesrest>

²http://restpatterns.org/HTTP_Status_Codes

```
{
  error: {
    type: "OAuthException",
    message: "A sessão expirou há 6 horas."
  },
}
```

Qualquer um consegue entender isso. Infelizmente faltam códigos de erro no Facebook e às vezes precisamos verificar a *string* da mensagem (o que é loucura). Por isso, use um código também.

O Foursquare não é um mau exemplo de usar ambos, mas eles enfatizam a ligação dos erros aos códigos de *status*.

<https://developer.foursquare.com/overview/responses>

O Twitter faz um ótimo trabalho documentando todos os seus códigos de *status* HTTP e fornecendo códigos de erro específicos para outros problemas, também. Alguns estão ligados a um código de *status* HTTP (o que é válido), porém outros não. Alguns estão ligados ao mesmo código de *status* HTTP, destacando os problemas levantados acima.

<https://dev.twitter.com/docs/error-codes-responses>

Cód	Texto	Descrição
161	Você não pode seguir mais pessoas neste momento	Corresponde ao código HTTP 403 - exibido quando um usuário não pode seguir outro devido a algum tipo de limitação
179	Desculpe, você não está autorizado a ver este <i>status</i>	Corresponde ao código HTTP 403 - exibido quando um <i>tweet</i> não pode ser visto pelo usuário autenticado, geralmente porque o autor protegeu seus <i>tweets</i> .

Detectando Códigos de Erro Programaticamente

Este é um exemplo de como podemos usar códigos de erro para fazer com que nossa aplicação responda inteligentemente a falhas básicas:

```

1 try:
2     api.PostUpdates(body['text'])
3
4 except twitter.TwitterError, exc:
5
6     skip_codes = [
7         # Página não existe
8         34,
9
10        # Você não pode enviar mensagens para usuários que não seguem você
11        150,
12
13        # Excedeu o limite
14        151
15    ]
16
17    error_code = exc.__getitem__(0)[0]['code']
18
19    # Se o token expirou vamos tirá-lo do caminho para não tentarmos novamente
20    if error_code in skip_codes:
21        message.reject()
22
23    else:
24        # Limite excedido? Vamos tirar uma soneca antes de reenfileirar
25        if error_code == 88:
26            time.sleep(10)
27
28    message.requeue()

```

Compare este tipo de lógica com a do Facebook e sua falta de códigos de erro:

```

1 except facebook.GraphAPIError, e:
2
3     phrases = ['expired', 'session has been invalidated']
4
5     for phrase in phrases:
6
7         # Se o código expirou vamos tirá-lo do caminho para não tentarmos novamente
8         if e.message.find(phrase) > 0:
9             log.info("Deactivating Token %s", user['token_id'])
10            self._deactivate_token(user['token_id'])
11
12     log.error("--- Unknown Facebook Error", exec_info=True)

```

Se o Facebook alterar suas mensagens de erro, o código acima poderá não funcionar mais.

4.3 Armadilhas Comuns

200 OK e Códigos de Erro

Se você retornar um código de *status* HTTP 200 com uma mensagem de erro, o Chuck Norris arrombará sua casa, destruirá seu computador, apagará instantaneamente todos os seus *backups*, cancelará sua conta no Dropbox e bloqueará você do GitHub. Os códigos HTTP 4xx e 5xx alertam o cliente de que algo errado aconteceu e os códigos de erro especificam exatamente qual foi o erro, caso o cliente esteja interessado.

Não Existente, Desaparecido ou Escondido?

404 é usado em demasia em muitas aplicações. Pessoas usam este código para “nunca existiu”, “não existe mais”, “você não podevê-lo” e “está desativado”; o que é muito vago. Tudo isso pode ser dividido entre 404, 403 e 410, e ainda assim continuaria vago.

Se um usuário recebe uma resposta 403 porque ele não está no grupo autorizado a ver o conteúdo requerido, o cliente não deveria sugerir ao usuário mudar de plano? Se o usuário não é amigo do autor do conteúdo, o cliente não deveria sugerir adicionar aquela pessoa como amigo?

Um erro 401 em um recurso pode indicar que o usuário excluiu aquele conteúdo ou que ele excluiu a sua conta por completo.

Em todas estas situações, a solução ideal é complementar o código de *status* HTTP com um código de erro específico. Pode ser o código que você quiser, desde que ele seja único em sua API e esteja documentado em algum lugar. Não faça como o Google, que fornece uma lista de códigos de erro e possui outros erros que não estão documentados em nenhum lugar. Se eu ficar sabendo disso eu vou atrás de você.

5 Testando Pontos de Acesso

Você deve estar aí pensando: “Isso avançou muito rápido e eu não estou pronto para testar!”. Bem, esse é basicamente o ponto. Você precisa preparar seus testes o quanto antes para que você se importe em usá-los, do contrário eles seriam apenas a “próxima coisa” que nunca seria feita. Não tenha medo, testar uma API é fácil e até divertido.

5.1 Conceitos & Ferramentas

Há algumas coisas para se testar em uma API, mas a ideia mais básica é: “quando eu acesso esta URL, eu quero ver um recurso ‘foo’”; e, “quando eu enviei este JSON para a API, ela deve aceitá-lo ou surtar.”

Isso pode ser feito de várias formas e muitos tentam imediatamente realizar testes de unidade, mas isso logo se torna um pesadelo. Enquanto pode parecer simples escolher o um cliente HTTP e começar a escrever um monte de código, quando você tiver mais de 50 pontos de acesso e precisar verificar cada um deles mais de uma vez, você acabará com um código BAGUNÇADO; o que não é nem um pouco divertido.

Quanto mais código em seus testes, maior será a chance deles retornarem falsos positivos, o que é super perigoso.

Assim, a forma mais simplista será utilizar uma ferramenta BDD¹. Uma ferramenta muito popular é o [Cucumber](#)², que muitos acreditam ser uma ferramenta para o Ruby. Na realidade ele pode ser usado com Python, PHP e provavelmente com muitas outras linguagens, porém algumas integrações podem ser complicadas. Para os desenvolvedores PHP, usaremos a ferramenta Behat, que é praticamente a mesma coisa, junto com [Gherkin](#)³ (o mesmo DSL⁴ usado pelo Cucumber, assim todos navegaremos no mesmo barco.)

5.2 Preparação

Se você utiliza o Ruby, basta executar `$ gem install cucumber` ou incluí-lo no seu arquivo `Gemfile`.

Como um desenvolvedor PHP, você só precisa instalar o Behat; isso pode ser feito através do [Composer](#)⁵. É razoável assumir que, se você está usando algum tipo de *framework* moderno, você já está familiarizado com este processo. Por isso, sem mais delongas, o seu arquivo `composer.json` deverá conter estes itens:

¹Desenvolvimento Guiado por Comportamento.

²<http://cukes.info>

³<http://docs.behat.org/guides/1.gherkin.html>

⁴Linguagem de Domínio Específico.

⁵<http://getcomposer.org>

Requerimentos básicos para o Behat no arquivo composer.json

```
{  
    "require": {  
        "behat/behat": "2.4.*@stable"  
    },  
  
    "config": {  
        "bin-dir": "bin/"  
    }  
}
```

Execute `$ composer install` e você estará pronto para começar.

Eu não sei o fazer com o Python, mas o Google pode te ajudar nisso. Para não complicar as coisas, eu ficarei com PHP e Behat, mas você pode converter as ideias em sua mente conforme avançamos.

5.3 Iniciando

Crie uma pasta chamada `tests` na raiz do seu projeto. Dentro da pasta `tests`, crie outra chamada `behat`. Pois é bem provável que um dia teremos outros tipos de testes.

```
$ cd /path/to/my/app  
$ mkdir tests && mkdir tests/behat  
$ cd tests/behat  
$ ../../bin/behat --init
```

O comando acima exibirá:

```
+d features - place your *.feature files here  
+d features/bootstrap - place bootstrap scripts and static files here  
+f features/bootstrap/FeatureContext.php - place your feature related code here
```

Próximo.

5.4 Funcionalidades

Funcionalidades⁶ é a forma como você agrupa vários testes juntos. Eu gosto de simplificar considerando cada “recurso” e “sub-recurso” como uma “funcionalidade”.

Vejamos o exemplo de usuários do [capítulo 2](#):

⁶*features*, em inglês.

Ação	Ponto de Acesso	Funcionalidades
Criar	POST /users	features/users.feature
Ler	GET /users/X	features/users.feature
Atualizar	POST /users/X	features/users.feature
Excluir	DELETE /users/X	features/users.feature
Listar	GET /users	features/users.feature
Imagen	PUT /users/X/image	features/users-image.feature
Favoritos	GET /users/X/favorites	features/users-favorites.feature
Checkins	GET /users/X/checkins	features/users-checkins.feature

Assim, tudo o que tem haver com `/places` e `/places/X` faz parte da mesma funcionalidade. Mas quando olhamos para `/places/X/checkins`, note que usaremos uma nova funcionalidade, pois estamos falando de outra coisa.

Você pode usar esta convenção ou tentar uma outra. Eu recomendo esta convenção pois ela não produz zilhões de arquivos para peneirarmos depois.

5.5 Cenários

O Gherkin usa “cenários” como sua estrutura principal, onde cada cenário possui vários “passos”. Na linguagem do teste de unidade, “cenários” seriam os “métodos” e os “passos” seriam as “asserções”.

Estas **Funcionalidades** e **Cenários** alinharam-se com o “Plano de Ações” criado no capítulo 2. Cada Recurso RESTful precisa de pelo menos uma “Funcionalidade” e, porque cada “Ação” possui um “Ponto de Acesso”, precisaremos de *pelo menos* um “Cenário” para cada “Ação”.

Muitos jargões? É hora de um exemplo:

Feature: Places

```
Scenario: Encontrando um local específico
  When I request "GET /places/1"
  Then I get a "200" response
  And scope into the "data" property
  And the properties exist:
    """
    id
    name
    lat
    lon
    address1
    address2
    city
    state
    zip
    website
```

```
phone
created_at
"""
And the "id" property is an integer
```

Scenario: Não é possível listar todos os locais

```
When I request "GET /places"
Then I get a "400" response
```

Scenario: Buscando um local não existente

```
When I request "GET /places?q=c800e42c377881f8202e7dae509cf9a516d4eb59&lat=1 \
&lon=1"
Then I get a "200" response
And the "data" property contains 0 items
```

Scenario: Buscando um local usando filtros

```
When I request "GET /places?lat=40.76855&lon=-73.9945&q=cheese"
Then I get a "200" response
And the "pagination" property is an object
And the "data" property is an array
And scope into the first "data" property
```

```
And the properties exist:
```

```
"""
id
name
lat
lon
address1
address2
city
state
zip
website
phone
created_at
"""

```

```
And reset scope
```

É possível descrever “funcionalidades” em português acrescentando a linha `# language: pt` no início do arquivo. Para aprender mais sobre a sintaxe em português, execute o comando abaixo:

```
bin/behat --story-syntax --lang
```

Mais informações na [documentação do Behat^a](#).

^a<http://docs.behat.org/guides/1.gherkin.html#gherkin-in-many-languages>

Eu usei algumas regras personalizadas que foram definidas em `features/bootstrap/FeatureContext.php`, veremos sobre isso mais à frente.

O arquivo de funcionalidades é chamado de `places.feature` e possui quatro cenários. Um para mostrar que não é possível listar todos os locais (400 significa entrada incompleta, você deveria especificar latitude e longitude), outro para buscar um local específico e mais dois para mostrar o quanto a busca funciona bem.

Eu tentei pensar nas cláusulas de proteção que meus pontos de acesso precisarão, assim eu faço um “Cenário” para cada um deles. Desta forma, se você não enviar uma latitude e longitude na busca, a requisição falhará. Teste isso.

Esperava um valor `boolean` mas recebeu uma `string`? Teste isso também:

```
Scenario: Argumento errado para seguir usuário
```

```
  Given I have the payload:
```

```
  """

```

```
    {"is_following": "foo"}
```

```
  """

```

```
  When I request "PUT /users/1"
```

```
  Then I get a "400" response
```

Quer ter certeza de que seus controladores sabem lidar com requisições estranhas com um erro 404 ao invés de surtar respondendo tudo com 500 Internal Error? Crie um teste.

```
Scenario: Tentar encontrar um momento inválido
```

```
  When I request "GET /moments/nope"
```

```
  Then I get a "404" response
```

É verdade que você ainda não escreveu nenhuma linha de código, mas é possível escrever todos estes testes tendo como base o seu “Plano de Ações” e rotas. Use aquilo que você já sabe sobre estrutura de conteúdo, aprendido no [capítulo 3](#), para planejar qual “saída” você espera ver.

Agora, tudo o que você precisa fazer é... você sabe... desenvolver toda a sua API. Vamos chegar lá.

5.6 Preparando o Behat

Você deve estar se perguntando como é possível executar os testes que acabamos de criar, pois o Behat envolve realizar uma requisição HTTP e até agora só escrevemos arquivos de texto. Bem, a classe no arquivo FeatureContext.php cuidará de tudo isso e muito mais. Mas, antes disso, precisamos configurar o Behat para que ele saiba qual será o *hostname* para estas requisições.

```
$ vim tests/behat/behat-dev.yml
```

Neste arquivo, coloque algo assim:

```
default:  
  context:  
    parameters:  
      base_url: http://localhost
```

Se sua máquina está configurada para usar *virtual hosts*, use-a acima. Ou, se você possui um *web-server* local rodando em uma porta diferente, acrescente a porta também. O valor pode ser `http://localhost:4000` ou `http://dev-api.example.com`, tanto faz.

Tendo finalizado esta parte, se você quiser as funcionalidades extras dos exemplos acima, você precisará do arquivo ZIP disponibilizado em [Códigos dos Exemplos](#). O arquivo que você precisa é `build-apis-you-wont-hate/behat/FeatureContext.php` e ele substitui o padrão `tests/behat/bootstrap/FeatureContext.php`.

5.7 Executando o Behat

Esta é a parte mais fácil:

```
$ ./vendor/bin/behat -c tests/behat/behat-dev.yml
```

Deu tudo errado. Ébaaa!

No próximo capítulo começaremos a esverdear alguns destes testes acrescentando dados de verdade.

6 Exibindo Dados

No [Capítulo 3: Teoria de Entrada e Saída](#), nós vimos sobre a teoria da estrutura de saída e as vantagens e desvantagens de vários formatos diferentes. Este livro assume que você já escolheu o seu, e assume que o seu favorito é o meu. Isso não é tão importante assim, pois fazer tudo para agradar a todos seria um exercício de futilidade e tédio.

O objetivo deste capítulo é ajudar você a desenvolver controladores para os seus pontos de acesso para esverdiar alguns daqueles testes do capítulo anterior.

Os exemplos na primeira seção tentarão exibir uma lista de todos os locais e também um local específico:

```
{
  "data": [
    {
      "id": 2,
      "name": "Videology",
      "lat": 40.713857,
      "lon": -73.961936,
      "created_at": "2013-04-02"
    },
    {
      "id": 1,
      "name": "Barcade",
      "lat": 40.712017,
      "lon": -73.950995,
      "created_at": "2012-09-23"
    }
  ]
}

{
  "data": [
    "id": 2,
    "name": "Videology",
    "lat": 40.713857,
    "lon": -73.961936,
    "created_at": "2013-04-02"
  ]
}
```

6.1 O Método Direto

A primeira coisa que todo desenvolvedor tenta fazer é pegar o seu ORM, ODM, DataMapper ou *Query Builder* favorito, realizar uma consulta e repassá-la diretamente para a saída.

O mal e perigoso exemplo de repassar dados do banco de dados diretamente para a saída

```
1 <?php
2 class PlaceController extends ApiController
3 {
4     public function show($id)
5     {
6         return json_encode([
7             'data' => Place::find($id)->toArray(),
8         ]);
9     }
10
11    public function list()
12    {
13        return json_encode([
14            'data' => Place::all()->toArray(),
15        ]);
16    }
17 }
```

Esta é absolutamente a pior ideia que você poderia ter por razões suficientes para encher um capítulo, mas tentarei descrevê-las em apenas uma seção.



ORMs nos Controladores

Seu controlador definitivamente não deve ter lógica envolvendo ORM/*QueryBuilder* espalhada pelos métodos. Eu fiz isso apenas para manter os exemplos em uma única classe.

Performance:

Retornar “todos” os itens de uma vez funcionará sem problemas durante o desenvolvimento, mas você terá problemas quando existirem milhares (ou milhões) de registros naquela tabela...

Exibição:

Extensões SQL populares para PHP transformam todos os dados de uma consulta em uma *string*. Assim, se você tiver um campo *boolean* no MySQL (que geralmente é um campo *tinyint(1)* com um valor 0 ou 1), ele será exibido na saída JSON como uma *string* com valor "0" ou "1", o que seria loucura. Se você estiver usando PostgreSQL, o resultado será ainda pior, pois o valor exibido pelo *driver PostgreSQL* do PHP é "f" ou "t". Seu

desenvolvedor móvel não ficará nem um pouco satisfeito e todos que olharem para a sua API pensarão que você é um amador. Você precisa de `true` ou `false` para ter um *boolean* de verdade no JSON, uma *string* numérica ou um `char(1)` não servem.

Segurança:

Exibir todos os campos pode permitir que os clientes da API (usuários de todos os tipos) vejam as senhas dos usuários ou outras informações confidenciais e obtenham acesso à chaves-secretas e *tokens*. Por exemplo, se os *tokens* de recuperação de senhas vazarem, você terá TANTA dor-de-cabeça quanto se as próprias senhas tivessem vazado.

Alguns ORMs possuem uma opção `hidden`, onde é possível especificar os campos que não serão exibidos. Se você prometer que você mesmo e os demais desenvolvedores da sua equipe (agora, no próximo ano e por toda a vida de sua aplicação) se lembrarão disso, meus parabéns. Com uma equipe tão focada assim você conseguiriam alcançar a paz mundial.

Estabilidade:

Se você alterar o nome de um campo no banco de dados, ou modificar um documento no MongoDB ou alterar os *status* disponíveis para um determinado campo entre a versão 3 e 4, sua aplicação continuará funcionando perfeitamente. Porém todos os usuários da sua aplicação no iPhone terão um aplicativo que não funciona, e a culpa terá sido sua. Mesmo que você prometa que não mudará nada, mudanças sempre acontecem.

Agora, o nosso amigo desenvolvedor teórico tentará consertar a saída manualmente.

O exemplo trabalhoso de converter o tipo e formatar os dados para a saída

```
1 <?php
2 class PlaceController extends ApiController
3 {
4     public function show($id)
5     {
6         $place = Place::find($id);
7
8         return json_encode([
9             'data' => [
10                 'id'          => (int) $place->id,
11                 'name'        => $place->name,
12                 'lat'          => (float) $place->lat,
13                 'lon'          => (float) $place->lon,
14                 'created_at'  => (string) $place->created_at,
15             ],
16         ]);
17     }
18
19     public function list()
20     {
21         $places = array();
```

```

23     foreach (Place::all() as $place) {
24         $places[] = [
25             'id'          => (int) $place->id,
26             'name'        => $place->name,
27             'lat'          => (float) $place->lat,
28             'lon'          => (float) $place->lon,
29             'created_at'   => (string) $place->created_at,
30         ];
31     }
32
33     return json_encode([
34         'data' => $places,
35     ]);
36 }
37 }
```

Porque especificamos exatamente quais campos são retornados no *array* JSON, a questão de segurança foi resolvida. A conversão de tipo de vários campos transformou *strings* numéricas em *integers*, coordenadas em *floats* e aquele objeto Carbon (DateTime) do Laravel em uma *string*, ao invés de deixá-lo se transformar em um *array* por conta própria.

O único problema não resolvido no exemplo acima foi a performance, mas este é um trabalho para a paginação, que cobriremos em um próximo capítulo.

Porém um novo problema bem óbvio foi criado: isso ficou nojento. Nosso desenvolvedor teórico tentará algo diferente desta vez.

Um método consideravelmente melhor para formatar os dados para a saída

```

1 <?php
2 class PlaceController extends ApiController
3 {
4     public function show($id)
5     {
6         $place = Place::find($id);
7
8         return json_encode([
9             'data' => $this->transformPlaceToJson($place),
10        ]);
11    }
12
13    public function list()
14    {
15        $places = array();
16        foreach (Place::all() as $place) {
17            $places[] = $this->transformPlaceToJson($place);
18        }
19    }
20}
```

```
19     return json_encode([
20         'data' => $places,
21     ]);
22 }
23
24
25 private function transformPlaceToJson(Place $place)
26 {
27     return [
28         'id'      => (int) $place->id,
29         'name'    => $place->name,
30         'lat'     => (float) $place->lat,
31         'lon'     => (float) $place->lon,
32         'created_at' => (string) $place->created_at,
33     ];
34 }
35 }
```

Com certeza ficou muito melhor, mas e se um outro controlador precisar exibir um local futuramente? Teoricamente você poderia mover todos estes métodos de transformação para uma nova classe ou colocá-los em `ApiController`, mas isso também seria estranho.

O que você realmente deseja fazer é o que tenho chamado de “Transformadores”, particularmente porque o nome é maravilhoso e porque ele descreve bem o que estamos fazendo.

Basicamente, eles são classes que possuem um método chamado `transform`, que faz a mesma coisa que o método `transformPlaceToJson()` acima. Mas para que você não tenha que aprender a criar o seu próprio transformador, eu publiquei um pacote PHP que cuida exatamente disso: [Fractal¹](#).

6.2 Transformações com Fractal

Com o Fractal, transformadores são criados como um *callback* ou como uma instância de um objeto que implementa `League\Fractal\TransformerAbstract`. Eles fazem exatamente o mesmo trabalho que o nosso método `transformPlaceToJson()`, mas eles vivem sozinhos, suas unidades são facilmente testadas (se isso é importante pra você) e eles removem muita da poluição causada pela “apresentação” no controlador.

O Fractal faz muito mais do que isso, vamos explorá-lo melhor mais à frente, mas ele resolve perfeitamente nossa preocupação com a transformação, além de remover os problemas de segurança, estabilidade e exibição abordados anteriormente.

Enquanto outras linguagens já possuem grandes soluções, não havia nada com este propósito no mundo do PHP. Alguns chamam isso de “Triagem de Dados” ou “Serialização Aninhada”, mas o problema resolvido é o mesmo: extrair dados pontencialmente complicados de uma série de armazenamentos e torná-los consistentes para a saída.

¹<https://packagist.org/packages/league/fractal>

- [Jbuilder](#)² parece bem legal para o pessoal do Ruby
- Tuíte outras sugestões para [@philsturgeon](#)

Este é o fim da teoria neste livro. Agora vamos trabalhar com código. Abra o arquivo ZIP ou vá até o [repositório no GitHub](#)³ e clone-o em algum lugar útil.

```
$ cd chapter6
$ ./run-demo.sh
PHP 5.5.6 Development Server started at Tue Dec 10 23:30:32 2013
Listening on http://localhost:5000
Document root is /some/place/chapter6/public
Press Ctrl-C to quit.
```

Abra o seu navegador e vá até <http://localhost:5000/places>. Você encontrará uma lista de locais parecida com esta:

²<https://github.com/rails/jbuilder>

³<https://github.com/philsturgeon/build-apis-you-wont-hate>

```
{
  - embeds: [
    "checkins"
  ],
  - data: [
    - {
      id: 1,
      name: "Mireille Rodriguez",
      lat: -84.147236,
      lon: 49.254065,
      address1: "12106 Omari Wells Apt. 801",
      address2: "",
      city: "East Romanberg",
      state: "VT",
      zip: 20129,
      website: "http://www.torpdibbert.com/",
      phone: "(029)331-0729x4259"
    },
    - {
      id: 2,
      name: "Dr. Judd Goodwin",
      lat: -5.56932,
      lon: -50.95633,
      address1: "9060 Harvey Lodge Suite 527",
      address2: "",
      city: "New Lea",
      state: "AK",
      zip: 18211,
      website: "http://emard.com/",
      phone: "(193)893-3463x099"
    }
  ]
}
```

Estrutura JSON padrão do Fractal

Esta só é uma aplicação feita no Laravel 4 porque ele oferece migrações e semeadores de banco de dados, além de me agradar muito. Ela contém “pedaços” de PHP que funcionariam em qualquer *framework* e os métodos usados funcionam em qualquer linguagem de programação.

- **composer.json** - Acrescentei uma pasta auto-carregável usando PSR-0 para permitir que o meu código seja carregado
- **app/controllers ApiController.php** - Um controlador base insanamente simples para agrupar as respostas

- `app/controllers/PlaceController.php` - Pega alguns dados que são repassados para `ApiController`

Fora isso, eu defini algumas rotas GET básicas em `app/routes.php`. Isso é praticamente tudo o que estamos fazendo.

Este é o `PlaceController`:

Exemplo de um controlador usando o Fractal para exibir dados

```

1 <?php
2 use App\Transformer\PlaceTransformer;
3
4 class PlaceController extends ApiController
5 {
6     public function index()
7     {
8         $places = Place::take(10)->get();
9         return $this->respondWithCollection($places, new PlaceTransformer);
10    }
11
12    public function show($id)
13    {
14        $place = Place::find($id);
15        return $this->respondWithItem($place, new PlaceTransformer);
16    }
17 }
```

O “*dado cru*” (que por acaso é um modelo ORM, mas poderia ser qualquer coisa) é enviado de volta com um método conveniente e uma instância de *transformer* também é fornecida. Estes métodos `respondWithCollection()` e `respondWithItem()` vêm do controlador `ApiController` e o seu trabalho é apenas criar uma instância do Fractal sem expôr um monte de classe para você interagir.

E este é o `PlaceTransformer`:

```

1 <?php namespace App\Transformer;
2
3 use Place;
4 use League\Fractal\TransformerAbstract;
5
6 class PlaceTransformer extends TransformerAbstract
7 {
8     /**
9      * Transforma este objeto em um array genérico
10     *
11     * @return array
12     */
13 }
```

```
13 public function transform(Place $place)
14 {
15     return [
16         'id'          => (int) $place->id,
17         'name'        => $place->name,
18         'lat'          => (float) $place->lat,
19         'lon'          => (float) $place->lon,
20         'address1'    => $place->address1,
21         'address2'    => $place->address2,
22         'city'         => $place->city,
23         'state'        => $place->state,
24         'zip'          => (float) $place->zip,
25         'website'      => $place->website,
26         'phone'        => $place->phone,
27     ];
28 }
29 }
```

Simples.

Até este momento o controlador ApiController foi mantido bem simples também:

ApiController simples para respostas básicas usando o Fractal

```
1 <?php
2
3 use League\Fractal\Resource\Collection;
4 use League\Fractal\Resource\Item;
5 use League\Fractal\Manager;
6
7 class ApiController extends Controller
8 {
9     protected $statusCode = 200;
10
11     public function __construct(Manager $fractal)
12     {
13         $this->fractal = $fractal;
14     }
15
16     public function getStatusCode()
17     {
18         return $this->statusCode;
19     }
20
21     public function setStatusCode($statusCode)
22     {
23         $this->statusCode = $statusCode;
```

```
24     return $this;
25 }
26
27     protected function respondWithItem($item, $callback)
28 {
29     $resource = new Item($item, $callback);
30
31     $rootScope = $this->fractal->createData($resource);
32
33     return $this->respondWithArray($rootScope->toArray());
34 }
35
36     protected function respondWithCollection($collection, $callback)
37 {
38     $resource = new Collection($collection, $callback);
39
40     $rootScope = $this->fractal->createData($resource);
41
42     return $this->respondWithArray($rootScope->toArray());
43 }
44
45     protected function respondWithArray(array $array, array $headers = [])
46 {
47     return Response::json($array, $this->statusCode, $headers);
48 }
49
50 }
```

O método `respondWithArray()` recebe um *array* geral e o converte para JSON, que será muito útil com erros. No mais, tudo que você retornar ou será um `Item` ou uma `Collection` do Fractal.

6.3 Ocultando Atualizações no Esquema

Esquemas são atualizados frequentemente e isso é inevitável. É muito fácil lidar com uma mudança do tipo: um campo no banco de dados foi renomeado.

Antes:

```
'website' => $place->website,
```

Depois:

```
'website' => $place->url,
```

É possível controlar a estabilidade das aplicações clientes alterando apenas a parte da direita (estrutura interna do dado) e mantendo a parte esquerda intacta (nome externo do campo).

Poderemos criar um novo *status* ou, em uma mudança drástica, mudar todos os *status*, mas versões antigas da API ainda precisarão do *status* antigo. Suponha que alteramos “available” para “active”, para melhor consistência com outras tabelas.

Antes:

```
'status' => $place->status,
```

Depois:

```
'status' => $place->status === 'available' ? 'active' : $place->status,
```

Bruto, mas útil.

6.4 Exibindo Erros

Eu ainda estou tentando descobrir a melhor forma de exibir erros na saída. A melhor opção até aqui tem sido adicionar métodos de conveniência ao controlador ApiController para manipular rotas globais com uma constante como o código e um conjunto de códigos de erro HTTP, com uma mensagem opcional para quando eu quiser substituir a mensagem.

Códigos de erro e respostas simples adicionadas ao ApiController

```
1 <?php
2
3 // ...
4
5 class ApiController extends Controller
6 {
7     // ...
8
9     const CODE_WRONG_ARGS = 'GEN-FUBARGS';
10    const CODE_NOT_FOUND = 'GEN-LIKETHEWIND';
11    const CODE_INTERNAL_ERROR = 'GEN-AAAGGH';
12    const CODE_UNAUTHORIZED = 'GEN-MAYBGTFO';
13    const CODE_FORBIDDEN = 'GEN-GTFO';
14
15    // ...
16
17    protected function respondWithError($message, $errorCode)
18    {
19        if ($this->statusCode === 200) {
20            trigger_error(
```


Este código basicamente nos permite retornar mensagens de erro genéricas em nosso controlador sem precisarmos pensar muito sobre as especificidades.

Controlador usando Fractal combinado com uma resposta de erro simples

```
1 <?php
2 use App\Transformer\PlaceTransformer;
3
4 class PlaceController extends ApiController
5 {
6     public function index()
7     {
8         $places = Place::take(10)->get();
9         return $this->respondWithCollection($places, new PlaceTransformer());
10    }
11
12    public function show($id)
13    {
14        $place = Place::find($id);
15
16        if (! $place) {
17            return $this->errorNotFound('Você inventou um ID e
18                                         tentou carregar um local? Idiota.');
19        }
20    }
21}
```

```
19     }
20
21     return $this->respondWithItem($place, new PlaceTransformer);
22 }
23 }
```

Outros erros específicos dos “Locais” poderiam ir diretamente no PlaceController em métodos como os demonstrados acima. Forneça suas próprias constantes no controlador, escolhendo um *statusCode* no método ou passando um como argumento.

6.5 Testando Esta Saída

Você já aprendeu a testar os seus pontos de acesso usando a sintaxe do Gherkin no [Capítulo 5: Testando Pontos de Acesso](#), agora poderemos aplicar aquele teste de lógica a esta saída:

Feature: Places

Scenario: Não é possível listar os locais sem um critério de busca
When I request "GET /places"
Then I get a "400" response

Scenario: Encontrando um local específico
When I request "GET /places/1"
Then I get a "200" response
And scope into the "data" property
And the properties exist:

```
"""
id
name
lat
lon
address1
address2
city
state
zip
website
phone
created_at
"""
```

And the "id" property is an integer

Scenario: Buscando um local não existente
When I request "GET /places?q=c800e42c377881f8202e7dae509cf9a516d4eb59&lat=1\

```
&lon=1"
Then I get a "200" response
And the "data" property contains 0 items
```

Scenario: Buscando um local usando filtros

```
When I request "GET /places?lat=40.76855&lon=-73.9945&q=cheese"
Then I get a "200" response
And the "data" property is an array
And scope into the first "data" property
```

And the properties exist:

```
"""
id
name
lat
lon
address1
address2
city
state
zip
website
phone
created_at
"""


```

And reset scope

Usamos mais uma vez o arquivo FeatureContext.php (fornecido nos exemplos), que facilita bastante o teste das saídas. Assumimos novamente que todas as saídas estão em um elemento "data" que, ou é um objeto (quando um recurso foi solicitado) ou é um *array* de objetos (quando múltiplos recursos ou uma coleção foram solicitados).

Quando você estiver pesquisando seus dados, certifique-se que sua aplicação não explodirá quando nenhum dado for encontrado. Isso pode acontecer no processamento da saída no controlador porque o que deveria ser um *array* é *null*, ou porque está faltando algum método à classe de coleção do PHP. É por isso que fizemos uma busca com um termo inválido para verificar se uma coleção vazia foi retornada:

```
{
  "data": []
}
```

A linha And the "data" property contains 0 items⁴ cuidará disso. Em seguida, podemos pesquisar termos válidos, porque o nosso semeador de banco de dados criou pelo menos um

⁴E a propriedade "data" contém 0 itens.

“Local” com a palavra-chave “*cheese*”. Usamos a linha `And scope into the first "data"` property⁵ para que o escopo utilizado seja dentro do primeiro item retornado, onde podemos verificar a existência de suas propriedades também. Se não houverem dados ou faltarem campos obrigatórios, o teste falhará.

6.6 Dever de Casa

Sua tarefa é dissecar a aplicação de exemplo, encaixá-la em sua API e tentar construir uma saída válida para o maior número possível de pontos de acesso. Usando o teste exemplificado acima, verifique se os tipos dos dados e a estrutura do *array* estão sendo geradas como você planejou.

Depois de falarmos sobre saída válida e o básico sobre erros, o que aprenderemos em seguida? A parte mais complicada da geração API, aquilo que mais cedo ou mais tarde todo desenvolvedor tentará e precisa aprender a fazer: incorporar/aninhar recursos, ou estabelecer “relações”.

⁵E o escopo seja dentro da primeira propriedade de “data”.

7 Relações Entre Dados

7.1 Introdução

Se você já trabalhou com banco de dados relacionais é bem provável que você já entenda sobre relacionamentos. Usuários possuem comentários. Autores possuem um ou muitos livros. Livros pertencem a uma Editora. Qualquer que seja o exemplo, relacionamentos são extremamente importantes para qualquer aplicação e também para sua API.

Os relacionamentos RESTful não precisam necessariamente corresponder diretamente aos relacionamentos no banco de dados. Se você estabeleceu os relacionamentos no banco de dados corretamente, os relacionamentos RESTful serão parecidos, mas o resultado RESTful poderá conter relacionamentos dinâmicos extras que não foram definidos em um JOIN e também poderá não incluir um ou outro relacionamento existente no banco de dados.

De uma forma mais eloquente:

Componentes REST comunicam transferindo uma representação de um recurso em um formato correspondente a um conjunto evolutivo de tipos de dados padrões, selecionada dinamicamente baseada nas capacidades ou desejos do destinatário e na natureza do recurso. Se a representação está no mesmo formato que sua fonte, ou derivou dela, permanece oculto atrás da interface. – [Roy Fielding](#)¹

Esta explicação destaca um fator importante: o resultado precisa estar baseado nos “desejos do destinatário”. Existem muitas abordagens populares sobre o desenvolvimento de relacionamentos RESTful, mas a maioria delas não satisfazem os “desejos do destinatário”. Mesmo assim, falarei sobre esses métodos populares e sobre suas vantagens e desvantagens.

7.2 Sub-Recursos

Uma forma bem simples de abordar dados relacionados é oferecer novas URLs para os consumidores da sua API. Falamos superficialmente sobre isso no [Capítulo 2: Planejando e Criando Pontos de Acesso](#) e esta é uma abordagem perfeitamente válida.

Se `places` é um recurso em uma API e deseja permitir acesso aos `checkins`, poderíamos criar um ponto de acesso para lidar com isso:

`/places/X/checkins`

¹http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_2

As desvantagem aqui é a necessidade de se fazer uma requisição HTTP extra. Imagine um aplicativo para iPhone que deseja exibir todos os locais de uma região em um mapa, para então permitir que o usuário navegue por eles. Se a busca por `places` acontecer em uma requisição, então a requisição `/places/X/checkins` será executada cada vez que o usuário clicar em um local, o que aumentará desnecessariamente o tempo de espera. Isto é conhecido como $1 + n$, e significa que o trabalho a ser feito será aumentado pelas requisições extra.

Isso também assume que o único dado relacionado é `checkins`. Na Kapture, nossa API também precisa obter `merchant`, `images`, `current_campaign` e `previous_campaigns`. Se usássemos “sub-recursos”, seriam necessárias mais quatro requisições HTTP extras por local, ou seja, $1 + 4n$.

Se retornarmos 50 locais e precisarmos carregar os dados relacionados para cada um deles, supondo que o usuário clicou em todos os 50 locais, uma requisição inicial para obter 50 resultados será feita. E para cada resultado, mais 4 requisições, ou seja: $1 + (50 \times 4) = 251$. Seriam 251 requisições HTTP desnecessárias (mesmo que elas sejam assíncronas)! Isso seria a coisa mais lenta que você poderia fazer em uma rede móvel. Mesmo usando `cache`, dependendo do conjunto de dados ainda assim seriam feitas 251 requisições.

Alguns desenvolvedores de APIs “enfiam” o máximo de dados possível em uma única requisição na tentativa de reduzir o número de requisições HTTP. Assim, quando solicitamos `/places`, obtemos automaticamente todos os `checkins`, `current_opps`, `merchants` e `images`. Bem, mesmo que você não queira toda essa informações, você precisaria esperar o *download* de um arquivo enorme cheio de JSON irrelevante! Mesmo que o servidor esteja compactando essa informação com GZIP, não queremos baixar algo que não precisaremos. Isso pode ser evitado e traria um grande ganho de performance à nossa aplicação móvel, além de um ganho menor para computadores ou *tablets* com conexões lentas ou sinais sem-fio fracos.

O dilema aqui é entre “baixar somente a quantidade de dados necessária para evitar que o usuário espere” e “baixar dados em excesso fazendo com que o usuário espere a resposta completa”. É difícil, mas você precisa de flexibilidade e usar apenas sub-recursos para lidar com dados relacionados não oferece nenhuma flexibilidade para o consumidor da API.

7.3 Arrays de Chaves Externas

Outro método para dados relacionados é fornecer um *array* de chaves externas no resultado. Pegando [um exemplo do EmberJS²](#), se um `post` possui muitos `comments`, o ponto de destino `/posts` contém a seguinte resposta:

²<http://emberjs.com/guides/models/defining-models/>

```
{
  "post": {
    "id": 1
    "title": "Progressive Enhancement is Dead",
    "comments": ["1", "2"],
    "_links": {
      "user": "/people/tomdale"
    }
  }
}
```

Isso é melhor. Você continua com $n + 1$ requisições, mas pelo menos você pode pegar esses IDs e agrupá-los em um requisição do tipo `/comments/1,2` ou `/comments?ids=1,2` para reduzir o número de requisições HTTP a serem realizadas.

De volta ao nosso exemplo, se você retornou 50 locais e precisa de 4 “pedaços” de dados extras, você poderia iterar os 50 para mapear quais itens precisam de quais dados. Então bastaria solicitar todos os pedaços de dados únicos e você terminaria com $1 + 4 = 5$ requisições HTTP ao invés de 251.

A desvantagem é que o consumidor da API precisará unir toda essa informação, o que seria muito trabalhoso no caso de uma grande quantidade de dados.

7.4 Documentos Compostos (Carregamento Lateral)

Ao invés de incluir apenas as chaves externas nos recursos, você pode carregar esses dados lateralmente. Eu tive dificuldade para escrever uma introdução, por isso deixarei outra pessoa escrevê-la para mim:

Documentos compostos possuem múltiplas coleções para permitir o carregamento lateral de objetos relacionados. O carregamento lateral é deseável quando representações aninhadas de objetos relacionados resultariam em uma repetição potencialmente cara. Por exemplo, dada uma lista de 50 comentários de apenas 3 autores, uma representação aninhada conteria 50 objetos de autores enquanto uma representação carregada lateralmente incluiria apenas 3 objetos de autores.

Fonte: https://canvas.instructure.com/doc/api/file.compound_documents.html

Eu descobri isso ao pesquisar o termo “documento composto”³. E esse termo foi descoberto enquanto eu pesquisava sobre “Carregamento Lateral REST”⁴. Que por sua vez foi descoberto após uma experiência horrível com o EmberJS me forçando a usar o método de carregamento lateral para o Ember Data, mesmo sendo algo que eles não explicam muito bem.

É algo mais ou menos assim:

³Compound Document, em inglês.

⁴REST Side-Loading, em inglês.

```
{
  "meta": {"primaryCollection": "comments"},  

  "comments": [...],  

  "authors": [...]
}
```

O que foi sugerido na citação acima é: se uma parte dos dados incorporados normalmente é recorrente, você não precisa baixar o mesmo recurso mais de uma vez. A desvantagem é que em grandes estruturas de dados o contexto é perdido, além do mesmo problema que o “Array de Chaves Externas”: mapear os dados para criar uma estrutura correta é deixado por conta do consumidor da API e pode ser algo trabalhoso.

7.5 Documentos Incorporados (Aninhamento)

Este é o método que tenho usado nas duas últimas versões da API da Kapture e é aquele que pretendo continuar usando. Ele é o que oferece maior flexibilidade ao consumidor da API: reduzindo o número de requisições HTTP ou reduzindo o tamanho a ser baixado dependendo das necessidades do consumidor.

Se o consumidor da API consultar a URL `/places?embed=checkins,merchant`, ele verá os dados de `checkins` e `merchant` dentro do recurso `place` na resposta:

```
{
  "data": [
    {
      "id": 2,
      "name": "Videology",
      "lat": 40.713857,
      "lon": -73.961936,
      "created_at": "2013-04-02",
      "checkins": [
        // ...
      ],
      "merchant": {
        // ...
      }
    },
    {
      "id": 1,
      "name": "Barcade",
      "lat": 40.712017,
      "lon": -73.950995,
      "created_at": "2012-09-23",
      "checkins": [
        // ...
      ]
    }
  ]
}
```

```

        ],
        "merchant" : {
            /**
            */
        }
    ]
}

```

Alguns sistemas (como o Facebook ou qualquer API que utilize o Fractal) permitirão que você aninhe esses dados com a notação de ponto, também:

Ex: /places?embed=checkins,merchant,current_opp.images

Incorporando com o Fractal

Continuando do Capítulo 6, até aqui o seu transformador possui apenas um método para manipular a conversão de um *array* de dados da sua fonte para um *array* simples. Mas com o Fractal é possível incorporar recursos e coleções, também. Continuando com o tema de usuários, locais e *checkins*, a classe `UserTransformer` pode conter uma lista de *checkins*. Veja o histórico de *checkins* de um usuário:

UserTransformer com o Fractal

```

1 <?php namespace App\Transformer;
2
3 use User;
4
5 use League\Fractal\TransformerAbstract;
6
7 class UserTransformer extends TransformerAbstract
8 {
9     protected $availableEmbeds = [
10         'checkins'
11     ];
12
13     /**
14      * Turn this item object into a generic array
15      *
16      * @return array
17      */
18     public function transform(User $user)
19     {
20         return [
21             'id'          => (int) $user->id,
22             'name'        => $user->name,
23             'bio'         => $user->bio,
24             'gender'      => $user->gender,

```

```

25      'location'      => $user->location,
26      'birthday'     => $user->birthday,
27      'joined'        => (string) $user->created_at,
28  ];
29 }
30
31 /**
32 * Embed Checkins
33 *
34 * @return League\Fractal\Resource\Collection
35 */
36 public function embedCheckins(User $user)
37 {
38     $checkins = $user->checkins;
39
40     return $this->collection($checkins, new CheckinTransformer);
41 }
42 }
```

O transformador CheckinTransformer pode ter um usuário (user) e um local (place). Neste contexto, não há nenhuma vantagem em solicitar o usuário, porque ele já está lá. Mas solicitando o local, podemos obter informações úteis sobre o local o checkin foi realizado.

CheckinTransformer com o Fractal

```

1 <?php namespace App\Transformer;
2
3 use Checkin;
4 use League\Fractal\TransformerAbstract;
5
6 class CheckinTransformer extends TransformerAbstract
7 {
8     /**
9      * List of resources possible to embed via this processor
10     *
11     * @var array
12     */
13     protected $availableEmbeds = [
14         'place',
15         'user',
16     ];
17
18     /**
19      * Turn this item object into a generic array
20      *
21      * @return array
22 }
```

```

22     */
23     public function transform(Checkin $checkin)
24     {
25         return [
26             'id'          => (int) $checkin->id,
27             'created_at'  => (string) $checkin->created_at,
28         ];
29     }
30
31 /**
32 * Embed Place
33 *
34 * @return League\Fractal\Resource\Item
35 */
36 public function embedPlace(Checkin $checkin)
37 {
38     $place = $checkin->place;
39
40     return $this->item($place, new PlaceTransformer());
41 }
42
43 /**
44 * Embed User
45 *
46 * @return League\Fractal\Resource\Item
47 */
48 public function embedUser(Checkin $checkin)
49 {
50     $user = $checkin->user;
51
52     return $this->item($user, new UserTransformer());
53 }
54 }
```

Estes exemplos usaram o recurso de carregamento lento⁵ de um ORM em `$user->checkins` e em `$checkin->place`, mas também poderíamos ter inspecionado a lista de escopos solicitados `$_GET['embed']`. Isso poderia facilmente ser colocado no construtor do controlador, em algum lugar no controlador base ou... em algum lugar:

⁵lazy-loading, em inglês.

```
$requestedEmbeds = Input::get('embed');

// Nomes incorporados à esquerda, nomes das relação à direita
// evite expor relacionamentos diretamente
$possibleRelationships = [
    'checkins' => 'checkins',
    'place' => 'venue',
];

// Verifica possíveis relações ORM e
// converte os nomes dos "incorporados" genéricos
$eagerLoad = array_values(
    array_intersect($possibleRelationships, $requestedEmbeds)
);

$books = Book::with($eagerLoad)->get();

// o código do Fractal aqui...
```

O código a seguir fará tudo isso funcionar. Coloque-o em algum lugar no ApiController ou no registro da sua aplicação.

```
class ApiController
{
    // ...

    public function __construct(Manager $fractal)
    {
        $this->fractal = $fractal;

        // Vamos tentar incluir dados incorporados?
        $this->fractal->setRequestedScopes(explode(',', Input::get('embed')));
    }

    // ...
}
```

É assim que fazemos isso no Laravel.

Incorporando com o Rails

O pessoal do Rails é fã do pacote ActiveRecord e muitos sugerem utilizá-lo para incorporar dados. Isso é encontrado especificamente na Documentação de Serializaton::to_json⁶.

Para incluir as associações, use blog.to_json(:include => :posts).

⁶http://apidock.com/rails/ActiveRecord/Serialization/to_json

```
{
  "id": 1, "name": "Konata Izumi", "age": 16,
  "created_at": "2006/08/01", "awesome": true,
  "posts": [
    {
      "id": 1,
      "author_id": 1,
      "title": "Welcome to the weblog"
    },
    {
      "id": 2,
      "author_id": 1,
      "title": "So I was thinking"
    }
  ]
}
```

Associações de 2º nível e de ordem superior funcionam também.

```
blog.to_json(:include => {
  :posts => {
    :include => {
      :comments => {
        :only => :body
      }
    },
    :only => :title
  }
})
```

Um pouco mais complicado, mas você está no controle do que é retornado.

```
{
  "id": 1,
  "name": "Konata Izumi",
  "age": 16,
  "created_at": "2006/08/01",
  "awesome": true,
  "posts": [
    {
      "comments": [
        {
          "body": "1st post!"
        },
        {
          "body": "Second!"
        }
      ],
      "title": "Welcome to the weblog"
    },
    {
      "comments": [
        {
          "body": "1st post!"
        },
        {
          "body": "Second!"
        }
      ],
      "title": "Welcome to the weblog"
    }
  ]
}
```

```
        "body": "Don't think too hard"
    },
    "title": "So I was thinking"
]
}
```

Isso funcionará bem desde que tudo esteja representado como o ActiveRecord.

Sendo um Rebelde RESTful

Eu li um artigo escrito por [Ian Bentley](#)⁷ sugerindo que esta abordagem não é inteiramente RESTful. Ele cita Roy Fielding:

O aspecto central que diferencia o estilo de arquitetura REST dos demais estilos baseados em redes é sua ênfase em uma interface uniforme entre componentes. Ao aplicar o princípio da generalidade (da engenharia de *software*) ao componente da interface, a arquitetura global do sistema é simplificada e a visibilidade das interações é melhorada. – [Roy Fielding](#)⁸

Todas estas soluções estão “erradas” de acordo com alguém. Estas são vantagens e desvantagens técnicas, que eu costumo chamar de questões “morais”. Estas questões “morais” tem a ver apenas com o quanto você deseja ser RESTful ao pé da letra. Os benefícios técnicos oferecidos pelo método de relacionamentos incorporados são tão grandes, que não me importo em “misturar” a especificação RESTful definida por Roy para utilizá-lo.

Faça usas próprias escolhas. Facebook, Twitter e a maioria das “APIs RESTful” ignoram partes da especificação RESTful (quando não, toda ela). Sua API não estará cometendo um crime hediondo se você seguir este caminho também.

⁷<http://idbentley.com/blog/2013/03/14/should-restful-apis-include-relationships/>

⁸http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1

8 Depurando

8.1 Introdução

Depurar¹ é a arte e descobrir por que algo não funciona. Isso pode ser bem difícil em se tratando de uma API. Durante a maior parte do desenvolvimento para a web, você precisa verificar o resultado no navegador constantemente, usar e abusar do `var_dump()` ou verificar se há erros do JavaScript no console do navegador. Durante o desenvolvimento de uma API, você trabalha basicamente com requisições e repostas HTTP, mas mesmo assim, você precisa realizar estas requisições de forma reproduzível e, geralmente, podendo controlar os cabeçalhos HTTP, conteúdo do corpo, etc.

Existem alguns métodos que você pode utilizar para depurar sua API:

- Depurando na Linha de Comando
- Depurando no Navegador
- Depurando Através da Rede

8.2 Depurando na Linha de Comando

Depurar através da linha de comando usando ferramentas como o Curl é uma ótima opção para alguns desenvolvedores. Um dos seus benefícios é poder usá-lo dentro de um *firewall* de rede. Certamente esta também é uma opção para depurar servidores “no ar”. Mas para fins de desenvolvimento (que é o nosso caso), precisaríamos nos lembrar de um monte de comando sem necessidade para usá-lo.

```
curl -X POST http://localhost/places/fg345d/checkins --data @payload.json
```

Esta não é a forma mais complicada de realizar uma requisição, mas também não é a mais fácil. Você precisará abrir o arquivo `payload.json` a cada requisição ou tentar ler um monte de JSON no console. Será muito desgastante quando você precisar testar vários pontos de destino com muitas possibilidades de valores.

8.3 Depurando no Navegador

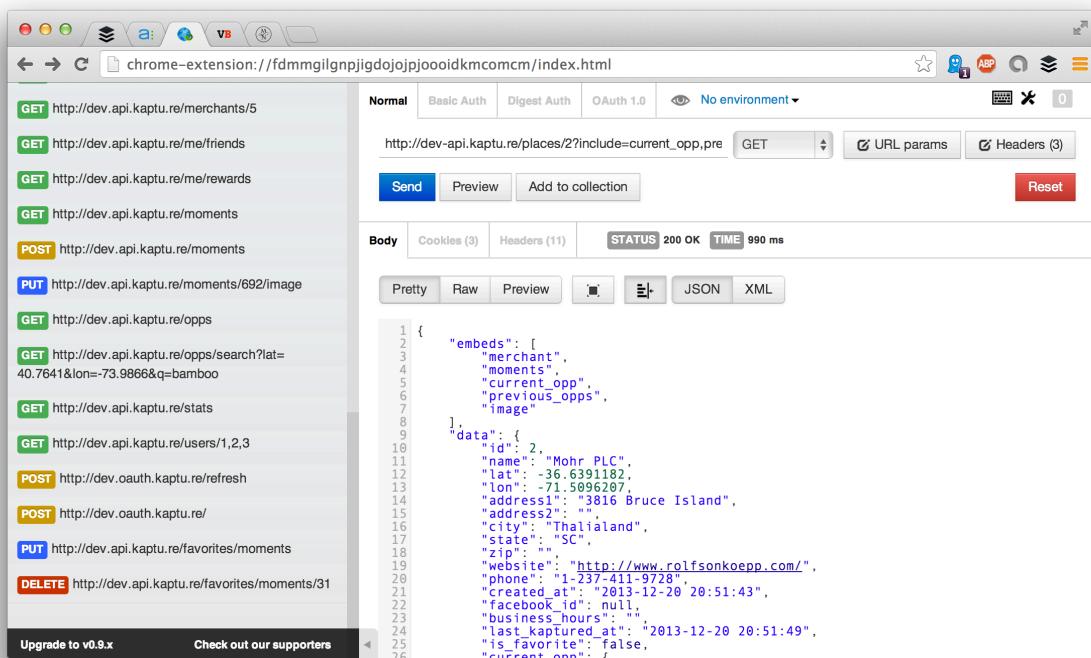
Trabalhar com um navegador é uma ótima opção e é algo que os desenvolvedores já estão acostumados. Infelizmente a maioria dos navegadores só conseguem realizar requisições GET e POST por padrão, enquanto APIs RESTful também exigem PUT, DELETE, PATCH, etc. Uma API RESTful bem desenvolvida também fará uso de cabeçalhos HTTP, que são difíceis de serem manipulados no navegador.

¹debugging, em inglês.

Cientes HTTP

Chamados de “Cientes HTTP” ou “Cientes REST”, esses programas são perfeitos para o trabalho que este livro propõe realizar: desenvolver APIs não-triviais. Eles permitem que você configure sua requisição HTTP através de uma interface conveniente, escolhendo o verbo HTTP, incluindo cabeçalhos, especificando um corpo, etc. Eles também exibem a resposta HTTP formatada ou crua, se você preferir assim. A maioria desses programas permitem que você salve uma “coleção” com as requisições mais comuns. Uma espécie favoritos, mas para os seus pontos de destino e com todos os cabeçalhos e seus respectivos valores.

Esses cientes estão disponíveis para Windows, OS X e Linux; mas o que mais chamou minha atenção foi uma extensão para o Google Chrome chamada [Postman](#)².



Cliente HTTP Postman exibindo uma coleção e uma resposta JSON bem sucedida

Minha coleção é praticamente idêntica aos meus testes no Behat. Eu tenho pelo menos uma para cada ponto de destino (alguns com mais de uma).

Usando o Postman, eu posso desenvolver “no navegador” e visualizar os erros facilmente. Eu continuo alterando coisas e clicando em “Send”. Quando eu acho que tudo está funcionando conforme planejado, executo os cenários do Behat que cobrem aquele ponto de destino para descobrir se os testes passaram ou não. Se o Behat falhar e as mensagens de erros não forem suficientes para resolver o problema, volto para o Postman e continuo tentando.

Reita até que o ponto de destino “funcione” e seus teste passem.

²<http://getpostman.com>

Painel de Depuração

O método acima funciona bem quando o problema é algo que pode ser visto. Mas problemas como respostas lentas, falhas silenciosas, resultados inesperados e etc, exigem informações mais detalhadas. Para isso você precisará de outra extensão.

- [RailsPanel³](#) - Painel para o Chrome com *logging* e *profiling* para Ruby on Rails. ([RailsCasts Video⁴](#)).
- [Clockwork⁵](#) - Painel para o Chrome e *web app* com *logging* e *profiling* para PHP.
- [Chrome Logger⁶](#) - *Logger* para o Chrome exclusivo para Python, PHP, Ruby, Node, .NET, CF e Go.

Os dois primeiros são bem parecidos e possuem mais recursos, mas a última opção oferece *logging* básico para uma seleção maior de linguagens.

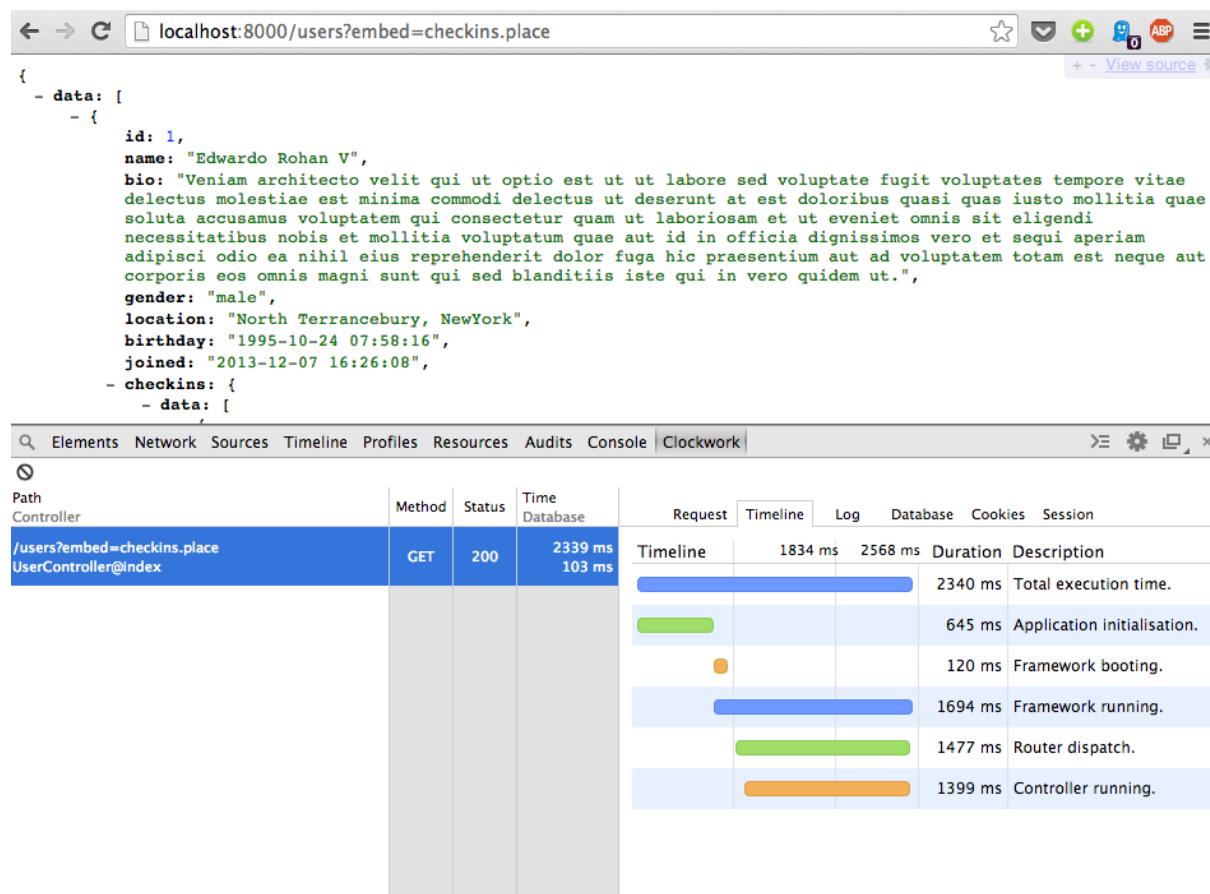
Todos estes exemplos são para o Chrome. É bem provável que existam outras alternativas; mas usar o Google Chrome como o seu navegador de desenvolvimento não lhe fará nenhum mal. Além do mais, você pode continuar usando o seu navegador favorito para navegação em geral.

³https://github.com/dejan/rails_panel

⁴<http://railscasts.com/episodes/402-better-errors-railspanel?view=asciicast>

⁵<https://github.com/itsgoingd/clockwork-chrome>

⁶<http://craig.is/writing/chrome-logger>



Clockwork exibindo a linha do tempo de execução do Laravel no navegador Chromium

Esta linha do tempo é útil para resolver qualquer problema relacionado à velocidade da API. Defina seus próprios eventos para ver onde ela está lenta.

Outra vantagem é poder visualizar os registros da sua API neste painel, para que você não precise recorrer ao terminal para “pegar” o resultado dos registros usando `tail -f`. É verdade que você estará usando a linha de comando em algum momento, mas alternar de programas com Alt+Tab constantemente é algo que distrai e deixa você mais lento.

Para aqueles que normalmente depuram com `var_dump()` ou `breakpoints`, é possível usar o Clockwork/RailsPanel/Chrome Logger para realizar esta mesma tarefa e ao mesmo tempo manter a resposta da API intacta.

CheckinTransformer com o Fractal mais logging

```

1 <?php namespace App\Transformer;
2
3 use Checkin;
4 use Log;
5
6 use League\Fractal\TransformerAbstract;
7
8 class CheckinTransformer extends TransformerAbstract
9 {

```

```
10  /**
11   * List of resources possible to embed via this processor
12   *
13   * @var array
14   */
15  protected $availableEmbeds = [
16      'place',
17      'user',
18  ];
19
20 /**
21  * Turn this item object into a generic array
22  *
23  * @return array
24  */
25  public function transform(Checkin $checkin)
26  {
27      return [
28          'id'          => (int) $checkin->id,
29          'created_at'  => (string) $checkin->created_at,
30      ];
31  }
32
33 /**
34  * Embed Place
35  *
36  * @return League\Fractal\Resource\Item
37  */
38  public function embedPlace(Checkin $checkin)
39  {
40      $place = $checkin->place;
41
42      Log::info("Embedding place-{$place->id} into checkin-{$checkin->id}");
43
44      return $this->item($place, new PlaceTransformer);
45  }
46
47 /**
48  * Embed User
49  *
50  * @return League\Fractal\Resource\Item
51  */
52  public function embedUser(Checkin $checkin)
53  {
54      $user = $checkin->user;
```

```

56     Log::info("Embedding user-{$user->id} into checkin-{$checkin->id}");
57
58     return $this->item($user, new UserTransformer);
59 }
60 }
```

O resultado disso será algo assim:

Path Controller	Method	Status	Time Database	Request Timeline			Log	Database	Cookies	Session
				Time	Level	Message				
/users?embed=checkins.place UserController@index	GET	200	563 ms 6 ms	15:51:31	info	Embedding place-1 into checkin-1				
				15:51:31	info	Embedding place-3 into checkin-69				
				15:51:31	info	Embedding place-4 into checkin-111				
				15:51:31	info	Embedding place-5 into checkin-156				
				15:51:31	info	Embedding place-9 into checkin-279				
				15:51:31	info	Embedding place-10 into checkin-306				
				15:51:31	info	Embedding place-12 into checkin-370				
				15:51:31	info	Embedding place-21 into checkin-676				
				15:51:31	info	Embedding place-24 into checkin-780				

Clockwork exibindo registros no navegador Chromium

Você pode registrar *arrays* e objetos também:

Request Timeline			Log	Database	Cookies	Session	
Time	Level	Message					
15:54:05	info	▼ Object {0: Object, 1: Object, 2: Object, ...} ▼ 0: Object id: "1" name: "Edwardo Rohan V" email: "niko22@yahoo.com" active: "1" gender: "male" birthday: "1995-10-24 07:58:16" location: "North Terrancebury, NewYork" bio: "Veniam architecto velit qui ut optio est ut ut lab molestiae est minima commodi delectus ut deserunt at est d voluptatem qui consectetur quam ut laboriosam et ut evenie voluptatum quae aut id in officia dignissimos vero et sequ fuga hic praesentium aut ad voluptatem totam est neque aut in vero quidem ut." created_at: "2013-12-07 16:26:08" ... "2013-12-07 16:26:08"					

Clockwork exibindo registros no navegador Chromium

Se registrar algo não te ajudar a resolver um problema, provavelmente você precisará registrar mais coisas. Eventualmente o problema será solucionado.

8.4 Depurando Através da Rede

Os métodos de depuração mencionados anteriormente oferecem controle total: você cria uma requisição e em seguida vê o que acontece com a resposta. Mas, às vezes, você precisará depurar o que acontece com sua API sem poder controlar as requisições. Se o seu desenvolvedor para iOS disser que “a API está quebrada”, será difícil descobrir a causa do erro.

Quando você sabe exatamente qual ponto de destino está sendo acessado e qual é o erro (ex: quando o desenvolvedor para iOS mostrou essas informações para você na tela do XCode), fica mais fácil corrigir o problema. Mas geralmente você precisará de mais informações para recriar o problema. Talvez o problema esteja em uma requisição que você não consegue recriar facilmente, como enviar imagens com o método `PUT` após obtê-la da câmera; ou quando várias requisições são executadas em ordem pelo aplicativos móvel usando informações da requisição anterior.

Qualquer que seja o motivo, muitas vezes você precisará depurar a atividade de rede espiando as requisições e respostas para descobrir o que está *realmente* acontecendo.

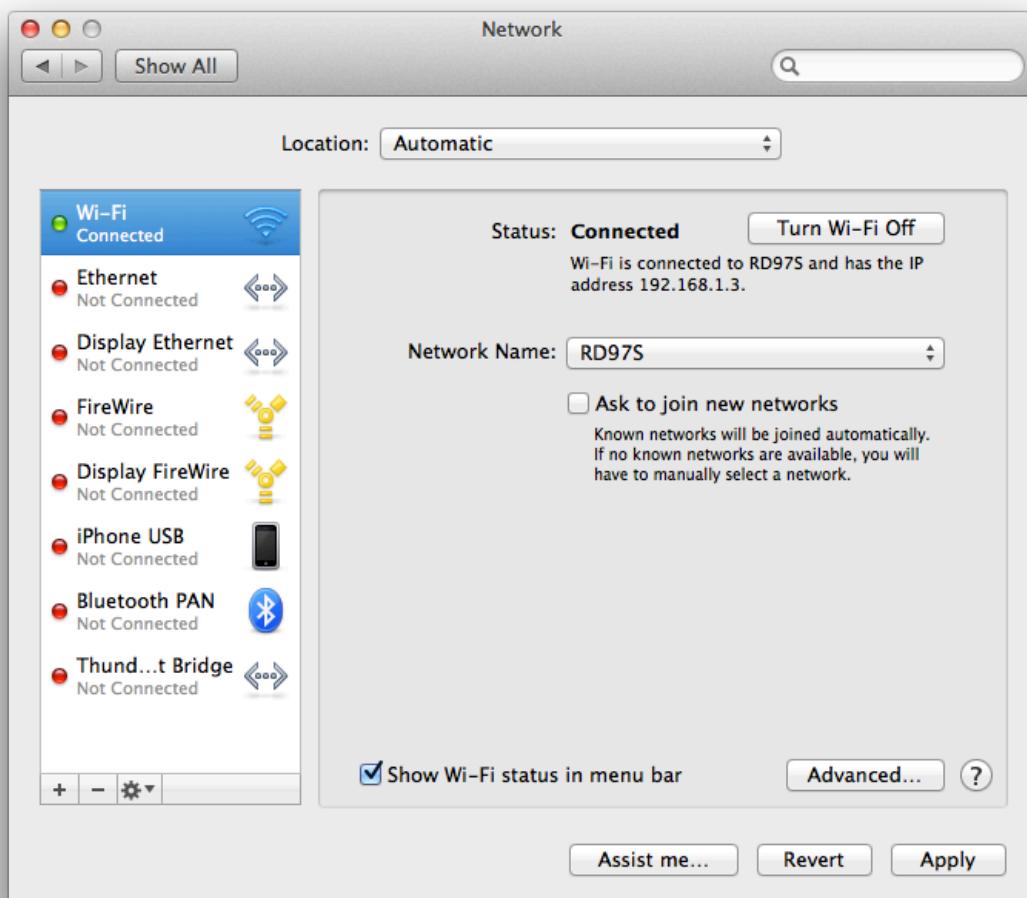
Charles

Caso existam erros que você queira depurar através da sua rede local usando seu dispositivo iOS de desenvolvimento (como um iPhone 4S que você ainda não vendeu no Mercado Livre, por exemplo), então um excelente aplicativo é o [Charles](#)⁷.

O Charles funciona com um *proxy* HTTP exibindo tudo aquilo que entra e sai através da rede. Ele vai além, podendo reescrever cabeçalhos e até mesmo modificar o conteúdo das requisições ou respostas, se você precisar.

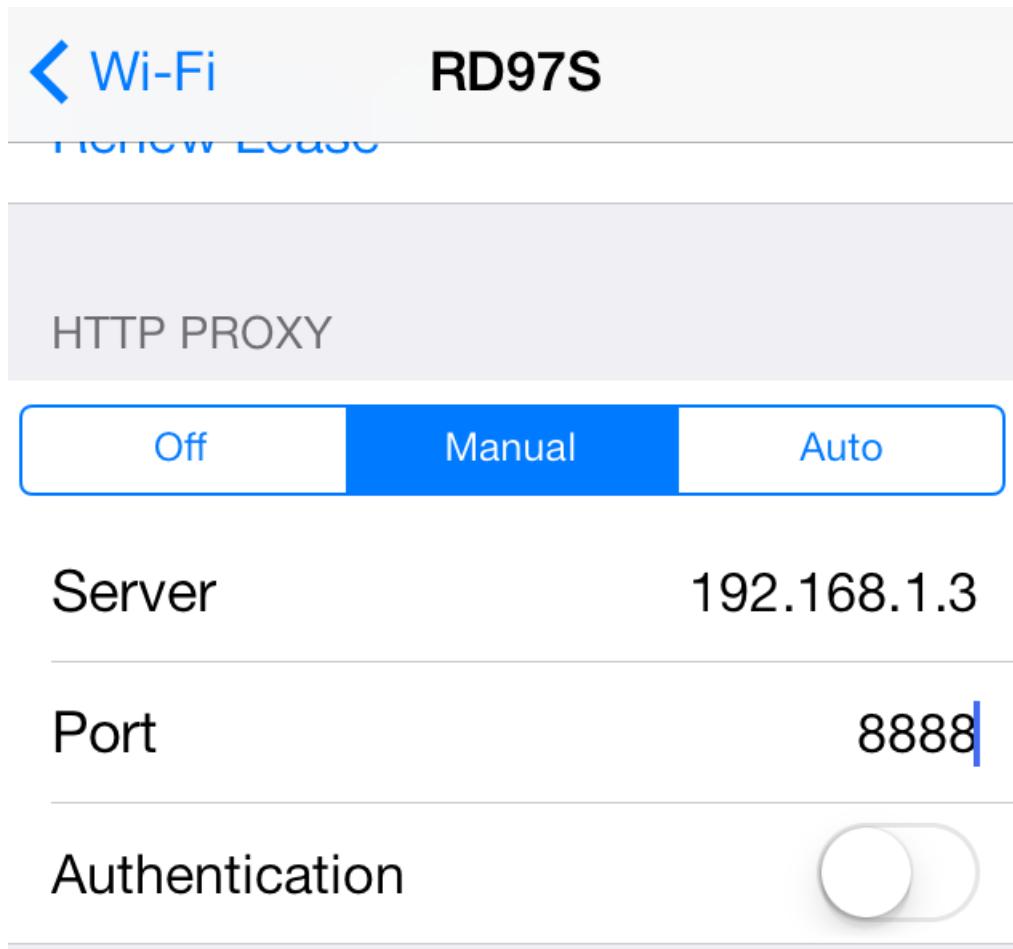
Para configurar o básico, você precisa saber qual é a rede interna da sua máquina.

⁷<http://www.charlesproxy.com>



Configurações de Rede no Mac OS X exibindo o IP local

Já em seu dispositivo móvel, você precisará habilitar um *proxy HTTP*. Digite o “IP local” do seu computador no campo “Proxy Server Address” e selecione a porta 8888 (porta padrão do Charles).



Esta configuração redirecionará todo o tráfego da web para o Charles, que por sua vez irá redirecioná-lo para o seu computador.

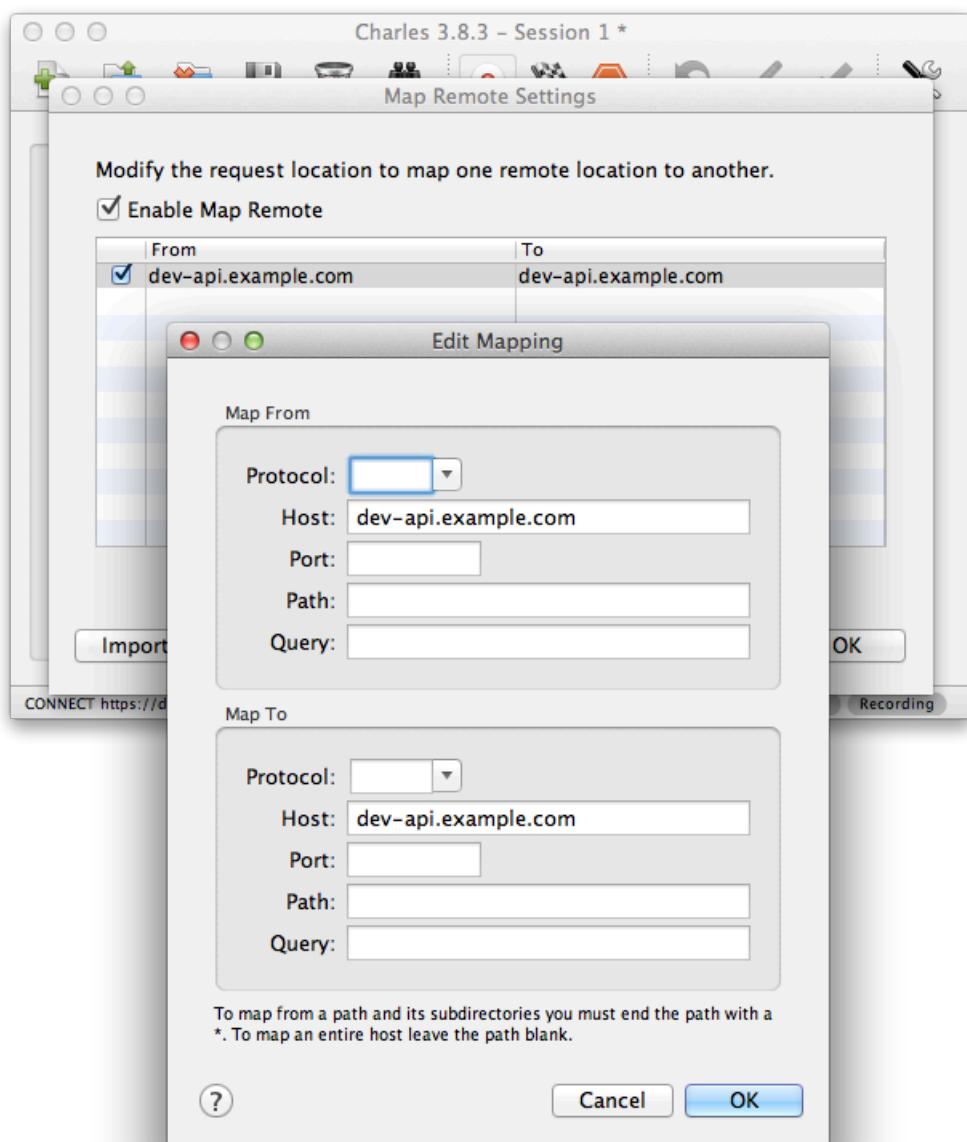
Por mais inútil que possa parecer, o segredo está nas opções que o aplicativo Charles oferece. Mas, se quisermos permitir tráfego web do nosso dispositivo móvel para a API no ambiente de desenvolvimento, ainda falta metade do caminho.



Local x “Remoto”

O Charles tem dificuldade de redirecionar o tráfego cujo resultado final é tráfego local. Por isso, o servidor de desenvolvimento do Laravel que usamos nestes exemplos (`localhost:8000`) não funcionará. Eu particularmente prefiro “apontar” o Charles para uma VM no Vagrant com um endereço de IP próprio e com um *host* virtual habilitado. Isso não é algo que explicarei neste livro, mas é algo você deve buscar fazer também.

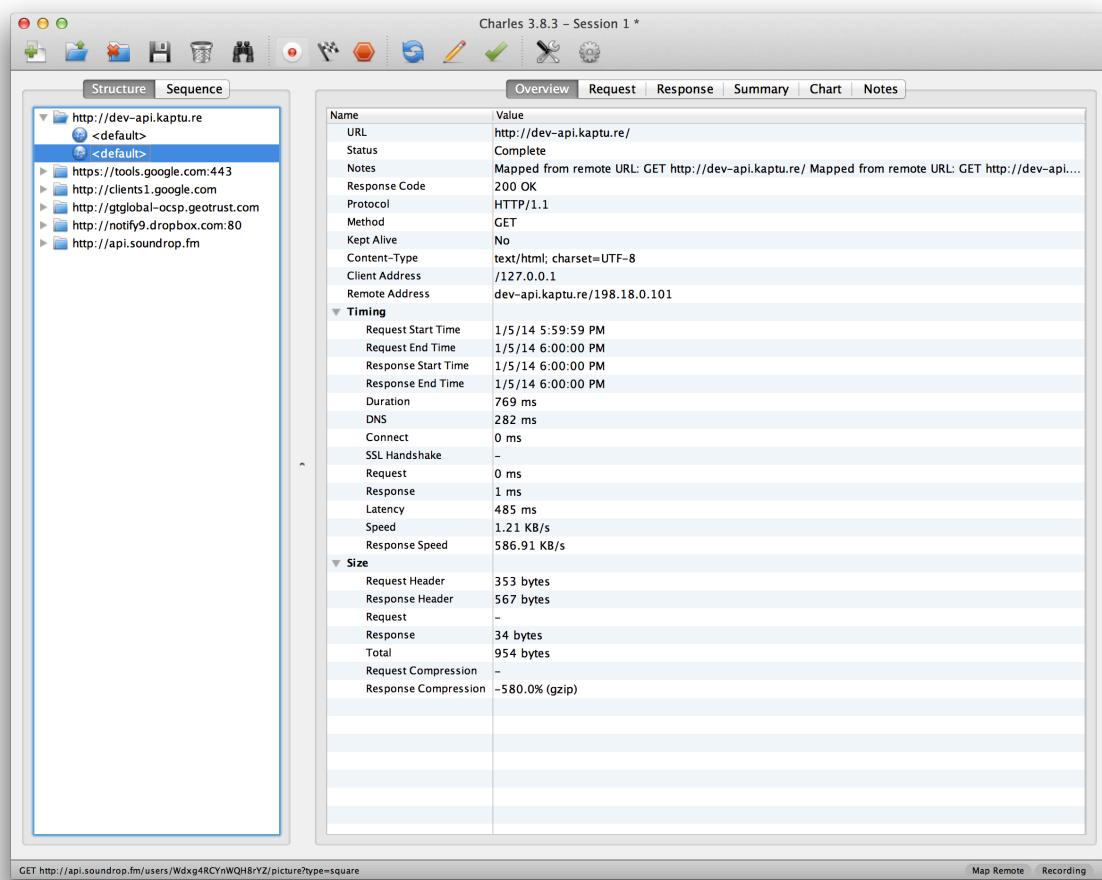
Para que `dev-api.example.com` tenha significado no dispositivo móvel, crie uma regra “Map Remote” no Charles.



Screenshot do Charles no OS X mapeando dev-api.example.com

Conforme eu já expliquei, o Charles age como um “homem-no-meio”, redirecionando o tráfego conforme as regras criadas por você. Ao dizer que dev-api.example.com deve ser direcionado para dev-api.example.com em sua máquina, você ativou esse *hostname* em seu dispositivo móvel (ou qualquer coisa que se comunique com o Charles nessa porta).

Agora, desde que você tenha uma versão do seu aplicativo móvel apontando para dev-api.example.com, será possível usar o aplicativo e visualizar as requisições e as respostas como todos os cabeçalhos e seus devidos valores.



Charles exibindo resultados para Kapture

É bem provável que você não usará o Charles todos os dias ou por muito tempo no início do desenvolvimento, quando clientes HTTP serão suficientes para depurar a maioria dos problemas, mas tê-lo disponível com certeza será útil em algum momento. Tenha isso em mente!

O [Wireshark](#)⁸ é divertido também.

⁸<https://www.wireshark.org>

9 Autenticação

9.1 Introdução

Entender como a autenticação funciona em uma API pode ser grande obstáculos para muitos desenvolvedores, em parte porque existem vários métodos diferentes, mas principalmente porque nenhum deles é parecido com a autenticação em um “aplicação para web”.

Quando desenvolvemos um painel de administração, CMS, blog e etc, o uso de sessões com dados armazenados em *cookies*, Memcache, Redis, Mongo ou qualquer plataforma SQL, é o padrão usado pela maioria de nós. Independente do local onde os dados são armazenados, usamos sessões para que, uma vez autenticado, o navegador se lembre quem é o usuário. Para “entrar”, o usuário é apresentado a um formulário HTML contendo dois campos: um para o nome de usuário ou e-mail e outro para a senha. Quando o usuário fecha o navegador ou fica inativo por um certo período, ele é “esquecido” automaticamente.

Esta é a forma padrão de lidar com *logins* na vasta maioria dos sites desenvolvidos com uma linguagem *server-side*, mas não é assim que lidamos com autenticação em APIs.

Neste capítulo, analizaremos os métodos de autenticação mais populares e eu explicarei as vantagens e desvantagens de cada um deles.

9.2 Quando a Autenticação É Útil?

A autenticação permite que você rastreie seus usuários, oferecendo pontos de destinos contextualizados (“encontre todas as minhas postagens”), limite o acesso dos usuários a vários pontos de destinos, filtre dados ou até mesmo restrinja e desative contas. Tudo isso será muito útil para a maioria das APIs, mas alguns nunca precisarão implementar autenticação.

APIs Apenas de Leitura

Se sua API é somente para leitura e as informações não são sigilosas, você pode disponibilizá-la sem se preocupar com autenticação. Isso é perfeitamente aceitável.

Porém existe a preocupação de que alguém ataque sua API com ataques de negação de serviço (DDoS); neste caso, exigir alguma forma de autenticação reduzirá a chance desses ataques acontecerem. Assim, antes de obter uma resposta da API, eles precisariam ser usuários válidos e a conta desses usuários seriam restringidas ou desativadas caso alguma atividade maliciosa fosse detectada.

Isso não é o que vai evitar todo tipo de ataque DDoS, mas com certeza é algo que ajudará sua API a trabalhar menos, pois as requisições terminarão bem antes quando o usuário for inválido. E se os ataques DDoS ainda forem uma preocupação com ou sem autenticação, a solução será

usar um bom *firewall* ou implementar outras barreiras de segurança. Ter um servidor atacado por alguém nunca é o ideal, por isso é melhor adotar outras medidas de segurança ao invés de apenas acrescentar autenticação à sua API na tentativa de evitar esses ataques.

De qualquer forma, você pode facilmente inaugurar sua API sem um mecanismo de autenticação e implementar um posteriormente.

APIs Internas

Se sua API é executada em uma rede privada ou está protegida atrás de um *firewall* e você não precisa contextualizar o conteúdo para cada usuário, provavelmente você também poderá pular a parte de autenticação da sua API.

Uma preocupação quanto a deixar toda a segurança por conta da rede é que, uma vez invadida, os *hackers* poderão causar um estrago muito maior. Mas se eles conseguiram entrar em sua rede, provavelmente você já tem problemas de segurança mais graves para se preocupar.

Tenha isso em mente.

9.3 Diferentes Métodos de Autenticação

Método #1: Autenticação Básica

O primeiro método adotado por muitos desenvolvedores é a autenticação HTTP básica, que é o método de usuário/senha que todos nós conhecemos e amamos, mas implementada ao nível da requisição HTTP e respeitada pelo navegador.

“Ouçamos” o que a Wikipédia tem a nos dizer:

A implementação da autenticação HTTP básica é a técnica mais simples de controlar o acesso a recursos na web, porque ela não requer cookies, sessões e nem mesmo páginas de login. A autenticação HTTP básica usa cabeçalhos HTTP padrões, assim não é necessário nenhum acordo prévio entre as partes envolvidas (cliente e servidor). – Fonte: [Wikipédia](#)¹

Vantagens

- Fácil implementação
- Fácil compreensão
- Funciona em qualquer navegador ou cliente HTTP

Desvantagens

- É ridiculamente insegura através de HTTP

¹http://en.wikipedia.org/wiki/Basic_access_authentication

- É um tanto insegura através de HTTPS
- As senhas podem ser armazenadas pelo navegador, podendo serem descobertas facilmente

Navegadores que Armazenam Senhas

Com o Chrome se recusando a usar uma senha-mestra para proteger as senhas salvas como texto puro, você estaria facilitando um ataque aos seus usuários ao permitir o acesso à sua API através da autenticação HTTP básica.

Elliott Kember revelou esta [situação do Chrome²](#). O The Guardian se importou³. Sir Tim Berners-Lee se importou⁴. Mas o Google não⁵.

Mais Dores com Texto Puro

Outro questão de segurança é que a autenticação básica é extremamente insegura através de HTTP.

No exemplo fornecido pela Wikipédia, um cabeçalho como este será incluso na requisição HTTP:

```
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

Se uma requisição como esta for realizada por um usuário em um local público, por exemplo, ela poderia ser facilmente interceptada. Tomando este cabeçalho como exemplo, seria insanamente fácil descobrir o nome do usuário e sua senha.

```
$ php -a
php > echo base64_decode('QWxhZGRpbjpvcGVuIHNlc2FtZQ==');
Aladdin:open sesame
```

Isto não é mais nem menos seguro que um formulário de *login* HTML, mas certamente não seguro o bastante para uma API com informações confidenciais.

O uso de SSL melhora a segurança consideravelmente, mas como a senha é enviada em todas as requisições HTTP, ainda há um risco pontencial dela ser descoberta. Mas a esta altura, a pessoa precisaria se esforçar *muito* para conseguir acesso à sua API.

A autenticação HTTP básica pode ser útil para APIs internas menos importantes, que precisam de alguma proteção e de uma implementação rápida. Mas certamente não é o método recomendado para qualquer coisa que lide com dinheiro, tráfego aéreo ou armas nucleares.

²<http://blog.elliottkember.com/chromes-insane-password-security-strategy>

³<http://www.theguardian.com/technology/2013/aug/07/google-chrome-password-security-flaw?INTCMP=SRCH>

⁴https://twitter.com/timberners_lee/status/364839351651274752

⁵<https://news.ycombinator.com/item?id=6166886>

Método #2: Autenticação Condensada

Autenticação condensada⁶ é um método parecido com a autenticação básica, mas ela foi desenvolvida para resolver algumas preocupações na área da segurança.

Ao invés de transmitir a senha como texto puro, este método calcula um *hash* MD5 e o envia. Diferente das senhas baseadas no Base64 usado pela autenticação básica, o MD5 é um *hash* unidirecional. Você não pode simplesmente pegar o *hash* e calcular a senha original, é necessário tentar diferentes combinações.

```
HA1 = MD5(A1) = MD5(username:realm:password)
HA2 = MD5(A2) = MD5(method:digestURI)
response = MD5(HA1:nonce:HA2)
```

O nonce é um número único, que contém (mas não deve ser apenas isso) uma *timestamp*. Isso ajuda a evitar ataques repetidos, visto que a mesma *hash* não poderá ser usada novamente.

Vantagens

- A senha não é transmitida como texto puro
- O uso do *nonce* ajuda a evitar ataques do tipo *rainbow table*
- De forma geral, é mais segura que a autenticação básica
- É mais fácil de implementar do que outros métodos

Desvantagens

- É mais difícil de ser **bem** implementada do que a autenticação básica
- Fácil de ser mal implementada
- Ainda é insegura através de HTTP
- Como acontece com a autenticação básica, as senhas podem ser armazenadas pelo navegador
- Usa MD5

MD5... 4... 3... 2... 1... HACKEADO

MD5 é amplamente conhecido em nossos dias como extremamente fácil de ser “quebrado” na maioria dos cenários. A autenticação condensada não tem avançado desde sua criação em 1993 e, mesmo que o tempo de cálculo ajude a evitar muitos problemas, uma porcaria de implementação da autenticação condensada estaria aberta para ataques esquisitos que só seriam descobertos após a consumação do ataque.

Com certeza a autenticação condensada é mais segura que a básica. Ela é ótima através de SSL e seria uma boa escolha para uma API interna, caso você tenha um pouco mais de tempo para implementá-la. Mas ela ainda exige que o usuário e senha sejam enviados repetidamente, ou seja, ela também pode ser “quebrada” caso o *hacker* tenha uma quantidade de requisições suficientes para analisar.

⁶*digest authentication*, em inglês.

Método #3: OAuth 1.0a

Embora não seja tão popular atualmente, OAuth 1.0a teve um papel importante no cenário da autenticação na *web*, sendo utilizado por serviços como Dropbox, Flickr, Twitter, Google, LinkedIn e Tumblr. Desde então, a maioria desses serviços adotou o OAuth 2, que discutiremos a seguir. Os dois são bem diferentes e não devem ser confundidos.

OAuth oferece um método para clientes acessarem recursos em servidores em nome do dono do recurso (como um cliente diferente ou um usuário final). Ele também oferece uma forma para o usuário final autorizar terceiros a acessarem seus recursos em servidores sem a necessidade de compartilhar suas credenciais (tipicamente usuário e senha), usando redirecionamentos de agente do usuário. –

Fonte: [Wikipedia⁷](http://en.wikipedia.org/wiki/OAuth)

Anteriormente, nós analisamos tecnologias de autenticação “embutidas nos navegadores” e que não eram flexíveis quanto ao seu uso. O OAuth 1.0 foi excelente para serviços como redes sociais, que puderam implementar formulários de *login* HTML parecidos com qualquer outro formulário de *login* (com logomarcas, esquemas de cores, etc), mas que podiam enviar você de volta para um outro *website* para realizar qualquer tipo de integração necessária.

Por exemplo, quando o Twitter mudou sua integração de HTTP básico para OAuth 1.0, ao invés de terceiros (aplicações p/ iPhone, outros sites, CMSs, etc) solicitarem o nome do usuário e sua senha (que poderiam ser armazenadas como texto puro), eles redirecionavam o usuário para o site do Twitter, onde o usuário podia *logar* para então ser redirecionado para o serviço com um código de acesso especial que podia ser salvo (no lugar da senha). No OAuth 1.0, esses códigos de acessos eram chamados “*OAuth Token*” e “*OAuth Token Secret*”.

O OAuth 1.0a foi planejado para ser seguro mesmo em conexões sem SSL. Isso significa que era extremamente complicado configurar assinaturas (para as quais existiam algoritmos diferentes, incluindo HMAC-SHA1 e RSA-SHA1 ou texto puro). Era complicado tentar escrever código no cliente, pois você precisava certificar que as assinaturas corretas eram suportadas. A maioria das implementações disponíveis para PHP (incluindo o meu velho CodeIgniter Spark) não suportavam todas elas.

Geralmente, uma requisição HTTP assinada com OAuth 1.0a era parecida com esta:

```
POST /moments/1/gift HTTP/1.1
Host: api.example.com
Authorization: OAuth realm="http://sp.example.com/",
oauth_consumer_key="0685bd9184jfhdq22",
oauth_token="ad180jjd733k1ru7",
oauth_signature_method="HMAC-SHA1",
oauth_signature="wOJI09A2W5mFwDgiDvZbTSMK%2FPY%3D",
oauth_timestamp="137131200",
oauth_nonce="4572616e48616d6d65724c61686176",
oauth_version="1.0"
```

⁷<http://en.wikipedia.org/wiki/OAuth>

Content-Type: application/json

```
{ "user_id" : 2 }
```

Uau.

Outra complicaçao é que existiam implementações diferentes. Duas pernas (adequada e não adequada) e três pernas. Algo extremamente confuso, por isso deixarei a explicação por conta do Mashape em [OAuth Bible: OAuth Flows⁸](#).

Também existia o xAuth, que ainda era OAuth 1.a, porém voltado para aplicações móveis e *desktops* que não tinham acesso fácil a um navegador. É muito mais fácil para uma aplicação *web* abrir uma janela com JavaScript ou redirecionar um usuário do que para um aplicativo móvel. Essa era uma implementação mais conveniente para se obter o código de acesso OAuth.

No fim das contas, de posse do código de acesso e do segredo, bastava colocar o “OAuth Token” no cabeçalho da requisição e usar o segredo como assinatura – responsável por encriptar a requisição e cuidar da segurança. E, se acrescentássemos SSL a tudo isso, teríamos uma configuração bem segura... Exceto pelo fato dos códigos de acesso permanecerem os mesmos após sua criação; por isso, com o passar do tempo sua segurança era comprometida. Alguém poderia recuperar dados de um computador vendido no Mercado Livre ou um *hacker* poderia bisbilhotar uma quantidade suficiente de tráfego com sua assinatura para eventualmente adivinhar o código de acesso e o secreto.

Vantagens

- Super seguro, até mesmo sem SSL
- Não envia o usuário/senha em todas as requisições (texto puro ou *hashed*)
- Impede que aplicações de terceiros solicitem e armazenem seu usuário e senha
- Mesmo que alguém consiga um código de acesso OAuth ou até mesmo um segredo, ele não poderia alterar sua senha, livrando você de ter sua conta roubada

Desvantagens

- Interação complicada, mesmo com uma boa biblioteca. O PHP nunca teve uma boa biblioteca para OAuth 1.0a, mas o [The League of Extraordinary Packages⁹](#) recentemente desenvolveu um [pacote decente¹⁰](#) para PHP.
- Número limitado de formas de concessão de acesso. xAuth e os fluxos de duas/três pernas eram restritivos
- Os códigos de acesso eram imutáveis, oferecendo risco potencial de segurança

OAuth 1.0a seria uma tecnologia fantástica se você estivesse desenvolvendo um site ou uma API pública baseada em usuários... e se estivéssemos em 2009-2010. Hoje, provavelmente não.

⁸<https://github.com/Mashape/mashape-oauth/blob/master/FLOWS.md#oauth-10a-one-legged>

⁹<http://thephpleague.com/>

¹⁰<https://github.com/thephpleague/oauth1-client>

Método #4: OAuth 2.0

O OAuth 2 abandonou o “código secreto”, agora os usuários obtem apenas um “Código de Acesso”. A assinatura de encriptação também foi abandonada. Muitos viram isso como um grande passo para trás quanto à segurança, mas na realidade esta foi uma escolha acertada. Na especificação do OAuth 1.0a, o uso de SSL era opcional, porém o OAuth 2.0 passou a exigí-lo. Confiar no SSL para lidar com a encriptação das requisições é algo lógico e melhora a implementação drasticamente.

Mesmo as requisições GET mais básicas no OAuth 1.0a eram horrendas, pois sempre era necessário configurar seus consumidores, assinaturas, etc. Mas com o OAuth 2.0 você pode fazer algo simples assim:

```
file_get_contents(  
    'https://graph.facebook.com/me?access_token=DFGJKHDFGHDIIFHGFKDJGHIU'  
)
```

Ou, como nós vimos lá atrás no [Capítulo 3](#), você também pode enviar o código de acesso para o servidor no cabeçalho da requisição:

```
1 POST /moments/1/gift HTTP/1.1  
2 Host: api.example.com  
3 Authorization: Bearer vr5HmMkzlxKE70W1y4MibiJUusZwZC25NOVBEx3BD1  
4 Content-Type: application/json  
5  
6 { "user_id" : 2 }
```

Isso parece um pouco mais fácil de trabalhar do que o OAuth 1.0a, não é mesmo?



Cabeçalhos X URL

Sempre que possível use um cabeçalho `Authorization` para enviar seus códigos de acesso. O método `query-string` é seguro quando se usa SSL, mas a menos que eles sejam bloqueados intencionalmente, os códigos de acesso podem ir parar nos *logs* do servidor e em vários outros locais. Também, os navegadores armazenam a URL completa (incluindo `query-strings`) no histórico. Isso poderia facilmente comprometer a integridade do usuário caso o computador dele seja roubado ou caso um parente resolva fazer uma graça.

Códigos de Curta Duração

Conforme já discutido, o OAuth 1.0a também usava o mesmo código praticamente para sempre. Os códigos de acesso do OAuth 2.0 podem expirar após um período de tempo definido pelo servidor OAuth. Ao solicitar um código de acesso, provavelmente você receberá um “Novo Código de Acesso” e um prazo de validade, que é o número de segundos até que o código expire; alguns servidores enviam um *unix time*.

Usando o prazo de validade, você saberá quando o código de acesso não será mais válido, assim você pode proativamente criar um *cron job* que atualizará os códigos de acessos ou você pode envolver suas requisições HTTP em um manipulador de exceções que atualizará os códigos de acessos quando esbarrar em um erro “Not Authorized”, conforme recomendado pela especificação do OAuth 2.0.

Inicialmente, este passo extra “códigos de acesso expiram e você precisa atualizá-los” parece confuso e irritante, especialmente quando você está acostumado com “o meu código de acesso funcionará para sempre”, mas isso o torna muito mais seguro. O OAuth 1.0a acabou com a necessidade de você fornecer o seu usuário e senha dando-lhe um outro usuário e senha (o código de acesso e o segredo) que funcionam para um cliente específico. Qualquer bom administrador de rede recomendará que você troque sua senha regularmente, e com o OAuth não é diferente. Quanto mais você usa os mesmos código de acesso e senha, maior é a probabilidade de alguém descobrir-los.

Tipos de Permissões

Outro grande benefício foi a possibilidade de se ter múltiplos (até personalizados) tipos de permissões, significando que agora você pode criar algumas implementações fantásticas.

O tipo de permissão mais comum no OAuth 2.0 é o `authorization_code`, que é bem parecido com o OAuth 1.0a.

Uma aplicação *web* cria um *link* para o servidor OAuth do serviço no qual ele deseja autenticar (ex: Facebook) e o usuário entra no Facebook. O Facebook redireciona o usuário de volta para a “URL *callback*” da aplicação *web* com uma variável `?code=FOO`. Então a aplicação *web* pega este código e faz uma segunda requisição (geralmente uma POST, mas às vezes uma GET...) para o Facebook, que por sua vez oferece um código de acesso na resposta HTTP (alguns serviços oferecem um código de expiração e atualização).

Outros tipos de permissões incluem `password` (você fornece o seu usuário/senha e o servidor retorna um código de acesso) e `client_credentials` (você fornece o seu `client_id` e `client_secret` e recebe um `access_token`). Geralmente este `access_token` genérico é limitado e não possui um contexto de usuário, assim ele não pode ser usado para publicar mensagens em um perfil no Twitter, por exemplo.

Bule de Chocolate do Twitter

Até este momento, a implementação OAuth 2 do Twitter disponibiliza apenas `client_credentials`. A [documentação^a](https://dev.twitter.com/docs/auth/application-only-auth) deles explica isso melhor. Ele é útil para consulta de *tags* ou tuítes públicos, mas fora isso não dá para se fazer muito coisa.

^a<https://dev.twitter.com/docs/auth/application-only-auth>

Na verdade, as credenciais do cliente podem usar HTTP básico ao invés de colocar o `client_id` e o `client_secret` no corpo da requisição.

Erin Hammer

Geralmente sou questionado por que alguém deveria usar o OAuth 2 mesmo após o Erin Hammer (autor-chefe e editor do padrão OAuth 2.0) ter retirado o seu nome da especificação. Com certeza isso causou uma certa agitação na internet, mas sinceramente eu discordo das questões levantadas por ele.

1. O OAuth 2.0 é menos seguro se você não usar SSL/TSL. Certo. Então use-os!
2. O OAuth 2.0 não foi bem implementado por muitos desenvolvedores (estou de olho no Facebook/Google/TODOS), mas ele é excelente quando implementado corretamente. Use uma implementação compatível com o padrão, como [esta para o PHP^a](https://github.com/thephpleague/oauth2-server/).
3. Ele acha que códigos atualizáveis são incovenientes, mas eu acho que eles são excelentes. Falando francamente, eu acho que o afastamento dele do projeto não é uma perda tão grande assim. Tenho certeza que o IETF tem encontrado dificuldades, mas após usar ambos por vários anos, estou muito mais contente com o OAuth 2.0. Eu realmente gostaria que o Twitter atualizasse completamente para o OAuth2.0 para que eu não precise mais usar o OAuth 1.0a.

^a<https://github.com/thephpleague/oauth2-server/>

O OAuth 2.0 é bom para praticamente qualquer cenário na autenticação de uma API. A flexibilidade dos tipos de permissão personalizados nos dão a oportunidade de fazer praticamente qualquer coisa. Na Kapture, por exemplo, nós criamos uma permissão chamada `social`, onde um usuário fornece uma `string` correspondente a “facebook” ou “twitter” e um `access_token` (às vezes com um `secret` também, porque o Twitter parou no tempo) e depois fazemos o seguinte:

1. Pegamos os dados do usuário
2. Descobrimos se ele já é um usuário da Kapture, caso contrário criamos uma conta para ele
3. Criamos um código de acesso, um código de atualização, etc para oferecer-lhe acesso à API

Desta forma, os usuários da nossa aplicação para iPhone tem uma experiência de “inscrição” ou “login” praticamente instantânea. Assim, nosso painel de administração e a área do comerciante usam o mesmo servidor OAuth 2.0 para lidar com *logins* para todos os usuários. Muito prático pra todo mundo.

Métodos alternativos

- OpenID - <https://openid.net/>
- Hawk - <https://github.com/hueniverse/hawk>
- Oz - <https://github.com/hueniverse/oz>

9.4 Implementando um Servidor OAuth 2.0

Implementar um servidor OAuth 2.0 (ou qualquer outro método de autenticação) manualmente pode ser bem difícil. Este capítulo buscou explicar as vantagens, desvantagens e usos para cada método, mas infelizmente a implementação é algo fora do seu propósito.

Como acontece com qualquer assunto complicado, instalar um pacote especificamente criado para realizar uma tarefa difícil no seu lugar, quase sempre é uma ótima ideia. Python, Ruby e PHP possuem pacotes excelentes, assim é mais fácil apontá-los para você ao invés de explicar como se fazer em cada uma dessas linguagens.

Implementações para PHP

Uma implementação se destaca das demais no universo do PHP, e não é apenas porque ela foi escrita por um amigo meu: [Alex Bilbie¹¹](#). O motivo principal de sermos amigos é que, além de ser um cara legal, o Alex estudou as especificações das duas versões do OAuth religiosamente e desenvolveu ótimas ferramentas para elas ao longo dos anos. Em seu último emprego na Universidade de Lincoln, ele usou o OAuth para fazer diversas coisas bem interessantes.

Depois ele ganhou uma bolsa de pesquisa para criar um projeto de código-aberto que aprimorasse a autenticação e sua interoperabilidade. Este projeto resultou em alguns bons pacotes, incluindo o [PHP OAuth 2.0 Server¹²](#), que agora faz parte de [The League of Extraordinary Packages¹³](#). Este é o único pacote para PHP que implementa toda a especificação do OAuth 2.0, então vale a pena conferí-lo.

OAuthello

Alex está escrevendo [um livro^a](#) cobrindo a implementação de servidores OAuth detalhadamente além de muita informação sobre o OAuth em geral, eu acho que você deveria comprá-lo. Essa URL contém um cupom de desconto: `apisyouwonthate`.

^a<http://leanpub.com/oauthello-a-book-about-oauth/c/apisyouwonthate>

Existe outra [implementação para servidor PHP OAuth 2.0¹⁴](#) que deve funcionar também.

Implementações para Python

Existem duas boas implementações para Python. Uma é o [oauth2lib¹⁵](#), que é um *fork* de [pyoauth2¹⁶](#). O autor original abandonou o projeto, então o novo precisou renomeá-lo.

¹¹<http://alexbilbie.com>

¹²<https://github.com/theleague/oauth2-server>

¹³<http://theleague.com/>

¹⁴<http://bschaffer.github.io/oauth2-server-php-docs>

¹⁵<https://github.com/NateFerrero/oauth2lib>

¹⁶<https://github.com/StartTheShift/pyoauth2>

Outra opção é o [python-oauth2¹⁷](#), desenvolvido pela SimpleGeo, que foi uma grande Saas de localização geográfica, mas que foi fechada após ser adquirida por outra empresa. A última atividade nesse repositório aconteceu há 2 anos... então... talvez este projeto esteja precisando de um novo dono, também.

Implementações para Ruby

O único servidor OAuth 2.0 para Ruby ativo e documentado que eu consegui encontrar foi um módulo Rack chamado [Rack::OAuth2::Server¹⁸](#). Ele é bem documentado e possui muitos exemplos de implementações em Rails, Sinatra e Padrino.

9.5 Onde o Servidor OAuth 2.0 Vive?

Muitos assumem que o servidor OAuth 2.0 também deve fazer parte do servidor da API. Embora isso seja possível, não é algo obrigatório.

Um servidor OAuth geralmente possui uma interface *web*, com um formulários HTML, validação e todo tipo de recursos estáticos como imagens, CSS, JavaScript, etc. Isso o torna mais parecido com um *site* comum, assim, se sua API e *site* estão em servidores diferentes, o servidor OAuth se encaixaria melhor no mesmo servidor que o *site*.

De modo geral, é melhor mantê-los independentes. Assim, caso você decida desenvolver uma nova versão do seu *site* usando o AngularJS ao invés de um código *server-side*, você terá a dor de cabeça de mudar a implementação do servidor OAuth, também. Se o servidor OAuth está isolado, ou pelo menos possui o seu próprio código, você não terá este tipo de preocupação.

A única coisa que sua API precisará fazer é encontrar um Código de Acesso (no cabeçalho ou variável *get*) para depois acessar o armazenamento de dados (banco de dados SQL, Mongo, etc) que contém o código de acesso. Então sua API verificará se o código é valido (existente no banco de dados e não expirou) para obter o usuário correspondente e os registros necessários para o funcionamento da API.

Nada disso é complicado, por isso não é necessário colocar os servidores da API e OAuth no mesmo servidor que o código da sua aplicação.

¹⁷<https://github.com/simplegeo/python-oauth2>

¹⁸<https://github.com/assaf/rack-oauth2-server>

10 Paginação

10.1 Introdução

Paginação é uma daquelas palavras que significam algo bem específico para muitos desenvolvedores, mas de um modo geral ela significa:

uma sequência de números atribuídos às páginas de um livro ou periódico.

Existem algumas formas de paginar algo, mas no contexto de uma API, queremos dizer:

qualquer forma de se dividir dados em múltiplas requisições HTTP com o intuito de limitar o tamanho da resposta HTTP.

Existem algumas razões para desejarmos fazer isso:

1. Baixar uma grande quantidade de dados leva mais tempo
2. Seu banco de dados não ficará nada contente ao tentar retornar 100 mil registro de uma vez
3. A lógica de apresentação não achará graça em iterar 100 mil registros

Bem, 100 mil é um número arbitrário. Uma API poderia ter um ponto de acesso /places com mais de 1 milhão de registros ou até mesmo *checkins* ilimitados. Durante o desenvolvimento de uma API, muitos se esquecem disso. Enquanto 10 ou 100 registros podem ser exibidos rapidamente, um número infinito seria algo consideravelmente lento. E a quantidade de dados só aumenta!

Uma boa API permite que o cliente solicite a quantidade de itens que ele deseja receber por requisição HTTP. Alguns desenvolvedores se acham espertos e usam um cabeçalho HTTP personalizado para esta função, mas as famosas *query strings* existem exatamente para isso.

/places?number=12

Alguns usam `number`, `limit`, `per_page` ou qualquer outra coisa. Eu acho que `limit` só faz sentido porque é o termo usado pelos usuários do SQL, mas REST não é SQL e eu prefiro usar o termo `number`.



Defina uma quantidade máxima

Ao receber do cliente o parâmetro limite/quantidade, você deve definitivamente estabelecer um limite para este número. Certifique-se que ele é maior que 0 e, dependendo da fonte dos dados, certifique-se também que ele é um número inteiro, pois números decimais causariam alguns efeitos interessantes.

10.2 Paginadores

Eu roubei a palavra “Paginador” do Laravel, que utiliza a classe Paginator para um tipo bem específico de paginação. Este não é, de forma alguma, o método mais eficiente de paginação, mas é bem fácil de se entender e funciona muito bem para quantidades de dados menores.

Como a paginação funciona?

Um dos métodos de paginação é contar a quantidade de registros de um determinado item. Assim, para contarmos a quantidade de places, usariamos uma query SQL como esta:

```
SELECT count(*) as `total` FROM `places`
```

Quando esta consulta retornar um valor de 1000, por exemplo, o código a seguir será executado:

```
1 <?php
2
3 $total = count_all_the_places();
4 $page = isset($_GET['page']) ? (int) $_GET['page'] : 1;
5 $per_page = isset($_GET['number']) ? (int) $_GET['number'] : 20;
6 $page_count = ceil($total / $per_page);
```

Esta matemática básica nos diz qual é o total de páginas existentes, cujo valor será arredondado para cima com `ceil()`. Esta é uma função do PHP equivalente a `Math.round()`, que arredonda um valor decimal para o número inteiro mais próximo. Assim, se o `$total` for 1000, então a variável `$page_count` será 83,333. Obviamente ninguém gostaria de visitar a página 83,333, por isso nos arredondamos este valor para 84.

Usando tais variáveis, uma API pode exibir alguns meta-dados simples que acompanham o *namespace* principal data:

```
{
  "data": [
    ...
  ],
  "pagination": {
    "total": 1000,
    "count": 12,
    "per_page": 12,
    "current_page": 1,
    "total_pages": 84,
    "next_url": "https://api.example.com/places?page=2&number=12",
  }
}
```

Os nomes dos itens neste exemplo de paginação são baseados puramente no que o desenvolvedor para iPhone da Kapture sugeriu na época.

Basicamente, você fornece informações suficientes ao cliente para ele fazer a matemática por conta própria, caso ele prefira, além de oferecer a opção de usar um *link* HTTP também.

Contar muitos dados é difícil

O problema principal com este método é a necessidade de se usar a consulta `SELECT count(*)` para se descobrir um total, algo que pode “encarecer” a requisição.

A primeira coisa que nos vem à cabeça é usar *cache*. Certamente você pode armazenar a contagem em *cache* ou até mesmo pre-popular a requisição. Isso seria possível em muitos casos, mas precisamos considerar que muitos pontos de destinos terão múltiplos parâmetros *query string* para personalizar os dados retornados.

```
/places?merchant=X
```

Isso significa que agora você terá um *cache* para o total de cada local de um determinado comerciante. Isso também é algo que poderia ser armazenado no *cache* ou pre-populado, mas não teríamos escolha no caso dos dados geográficos:

```
/places?lat=42.2345&lon=1.234
```

Infelizmente, as chances de se ter mais de uma pessoa solicitando as mesmas coordenadas com frequência são pequenas e não justificam a criação de um *cache*, especialmente quando estas coordenadas apontam para uma região remota e montanhosa da Espanha.

Pre-popular este tipo de resultado é algo muito improvável. Se você literalmente possui milhões de locais, tentar fazer a contagem de todos eles para alguém na Espanha seria tolice. Índices¹ podem nos ajudar. Separar seus dados em porções geográficas também pode ser algo útil. Porém de modo geral, usar este tipo de paginação (criado para quantidades de dados menores) para grandes quantidades de dados pode causar problemas, especialmente quando se usa a opção de filtragem.

Isso não é ruim (e eu já usei este método várias vezes em APIs), mas é necessário ter em mente as considerações acima.

Listas em movimento

Outra dificuldade com o método de “contar tudo para então pegar o número da página” é que, quando um novo item é adicionado entre requisição HTTP e outra, o mesmo conteúdo poderá ser exibido duas vezes.

Imagine a cena onde o `number` por página é 2, `places` são ordenados por nome e os valores são bares *hip* no Brooklyn, em Nova Iorque:

¹*indexes*, em inglês.

- Página 1
 - Barcade
 - Pickle Shack
- Página 2
 - Videology

Quando o cliente solicita a Página 1, ele vê os dois primeiros resultados. Enquanto os resultados da Página 1 são exibidos para um usuário, um novo bar *hip* foi aberto com o nome “Lucky Dog” e filiou-se à plataforma.

Agora, nosso conjunto de dados ficou assim:

- Página 1
 - Barcade
 - Lucky Dog
- Página 2
 - Pickle Shack
 - Videology

Se o cliente não atualizar a Página 1 (algo que a maioria não faria), “Pickle Shack” seria exibido duas vezes e “Lucky Dog” não apareceria de forma alguma.

Usando paginadores com o Fractal

Este é um exemplo específico que requer os pacotes Eloquent e Pagination do Laravel; além do pacote [Fractal](#)². Se você não está usando nenhum desses pacotes, não se preocupe e apenas faça algumas contas básicas como as do exemplo em JSON abaixo. Para os demais, sigam-me:

```
1 <?php
2
3 use Acme\Model\Place;
4 use Acme\Transformer\PlaceTransformer;
5 use League\Fractal\Resource\Collection;
6 use League\Fractal\Pagination\IlluminatePaginatorAdapter;
7
8 $paginator = Place::findNearbyPlaces($lat, $lon)->paginate();
9 $places = $paginator->getCollection();
10
11 $resource = new Collection($places, new PlaceTransformer());
12 $resource->setPaginator(new IlluminatePaginatorAdapter($paginator));
```

²<http://fractal.thephpleague.com>

10.3 Deslocamentos e Cursos

Outro método comum de paginação é o uso de “cursos” (também conhecidos como “marcadores”). Normalmente, um cursor é um identificador único, ou um deslocador³, para que a API possa solicitar mais dados.

Se existirem mais dados para serem encontrados, a API poderá retorná-los. Caso não haja mais dados, então ela poderá retornar um erro (404) ou uma coleção vazia.

Vazio não é o mesmo que “Em Falta”

Pessoalmente eu não recomendo o uso do erro 404, pois tecnicamente a URL não está errada, simplesmente não existem mais dados para serem retornados naquela coleção. Por isso, uma coleção vazia faria mais sentido.

Dê uma olhada neste exemplo:

```
{  
  "data": [  
    ...  
  ],  
  "pagination": {  
    "cursors": {  
      "after": 12,  
      "next_url": "https://api.example.com/places?cursor=12&number=12"  
    }  
  }  
}
```

Neste exemplo, especificamos em `after` que 12 registros já foram obtidos. Os registros de 1 a 12 foram disponibilizados e todos eram auto-incrementados. Na próxima requisição, solicitaremos conteúdos *após* o número 12, ou seja, os registros de 13 a 24 serão disponibilizados na página seguinte.

Embora este exemplo forneça uma explicação incrivelmente simples, o uso de IDs é uma ideia complicada. Um determinado registro por id de uma categoria para outra ou ter sido desativado, ou qualquer outra coisa. Você *pode* usar IDs, mas é recomendado que se use um deslocador.

É fácil usar um deslocador. Independente dos seus IDs, *hashed*, etc, você simplesmente coloca um 12 lá e diz: “Eu quero 12 registros, com um deslocamento de 12.” Ao invés de dizer: “Eu gostaria dos registros após `id=12`.”

³*offset*, em inglês.

Ofuscando cursores

De vez em quando o Facebook usa cursores para ofuscar as IDs de verdade, mas às vezes eles são usados para “deslocamentos baseados em cursores”. Independente do que o cursor realmente seja, o usuário não precisa se preocupar com ele, por isso é uma boa ideia escondê-los.



Graph API do Facebook usando cursores

Como o Facebook chegou aos valores "MTA=" e "MQ=="? Bem, eles foram criados assim intencionalmente para que você não saiba o que eles são. Um cursor é um valor obscuro que pode ser repassado para o sistema de paginação para se obter mais informação, não importa se ele é 1, 6, 10, 120332435 ou “Terça”.

[Don Gilbert](#)⁴ esclarece que, no exemplo do Facebook, eles apenas usaram codificação Base64 para os cursores deles:

```

php > var_dump(base64_decode('MTA='));
string(2) "10"

php > var_dump(base64_decode('MQ=='));
string(1) "1"

```

Os valores não foram ofuscados por questão de segurança, mas - na minha opinião - para evitar que as pessoas façam contas com estes valores. Ignorância é benéfica neste cenário, pois se alguém fizesse as contas com o resultado de uma paginação baseada em deslocamento, essa mesma pessoa acabaria fazendo as mesmas contas com o número da chave primária. Mas se tudo está em um cursor ofuscado, ou marcador, então ninguém poderá fazer isso.

Requisições extras = Tristeza

Este não é um método apreciado por alguns desenvolvedores porque eles não gostam da ideia de precisar fazer uma requisição extra para descobrir que não existe nenhum dado, mas esta parece ser a única forma de se alcançar um bom desempenho quando o sistema de paginação lida com grandes quantidades de dados. Mesmo em um sistema de “páginas”, se a última página tiver apenas 1 item e este item for removido, de qualquer forma esta página ficará vazia; ou seja, todo e qualquer sistema de paginação precisa responder a uma coleção vazia.

Usando cursores com o Fractal

Novamente, este é um exemplo bem específico, mas que retrata o conceito.

⁴<http://dongilbert.net>

```
1 <?php
2
3 use Acme\Model\Place;
4 use Acme\Transformer\PlaceTransformer;
5 use League\Fractal\Cursor\Cursor;
6 use League\Fractal\Resource\Collection;
7
8 $current = isset($_GET['cursor']) ? (int) base64_decode($_GET['cursor']) : 0;
9 $per_page = isset($_GET['number']) ? (int) $_GET['number'] : 20;
10
11 $places = Place::findNearbyPlaces($lat, $lon)
12     ->limit($per_page)
13     ->skip($current)
14     ->get();
15
16 $next = base64_encode((string) ($current + $per_page));
17
18 $cursor = new Cursor($current, $next, $places->count());
19
20 $resource = new Collection($places, new PlaceTransformer());
21 $resource->setCursor($cursor);
```

O código acima “pega” o cursor atual e armazena a versão decodificada dele na variável \$current para então usá-lo como deslocador no método skip(). Como a classe Cursor é intencionalmente vaga, fizemos um pouco mais de trabalho. Ao invés de usar um deslocador, poderíamos ter usado um ID específico na consulta SQL WHERE id > X, mas é melhor não.

11 Em Breve

Este livro ainda não acabou, você será notificado sempre que houverem novos capítulos disponíveis. Alguns dos próximos assuntos incluem:

- Documentação
- HATEOAS
- Versionamento de APIs

Entre em contato caso você tenha alguma sugestão de conteúdo para os próximos capítulos.

Enquanto isso, que tal compartilhar com alguém aquilo que você tem aprendido?

Caso você tenha alguma sugestão em relação a esta tradução, escreva para: oi@pedroborg.es

Até breve!