Building APIs You Won't Hate

By Phil Sturgeon

Build APIs You Won't Hate

Everyone and their dog wants an API, so you should probably learn how to build them.

Phil Sturgeon

This book is for sale at http://leanpub.com/build-apis-you-wont-hate

This version was published on 2013-12-30

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 Phil Sturgeon

Tweet This Book!

Please help Phil Sturgeon by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought @philsturgeon's book about APIs because he said if I didn't he would hurt me: http://bit.ly/apisyouwonthate

The suggested hashtag for this book is #apisyouwonthate.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

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

Also By Phil Sturgeon

Desenvolvendo APIs que você não odiará

PHP: The ”Right” Way

Contents

Introduction

I've been building APIs for a long time now and it is becoming ever more common for server-side developer thanks to the rise of front-end JavaScript frameworks, iPhone applications and API-centric architectures. On one hand you're just grabbing stuff from a data source and shoving it out as JSON, but surviving changes in business logic, database schema updates, new features or deprecated endpoints, etc gets super difficult.

I found most resources out there to be horribly lacking or specifically aimed at one single framework. Many tutorials and books use apples and pears examples which are not concrete enough, or talk like listing /users and /users/1 are the only endpoints you'll ever need. I've spent the last year working at a company called Kapture where my primary function has been to inherit, rebuild, maintain and further develop a fairly large API with many different endpoints exposing a lot of different use-cases.

The API in question was v2 when I joined the company and written in FuelPHP, utilizing a now deprecated ORM which had been hacked to death by the original developer. Kapture was in the process of rebuilding it's iPhone application to implement new functionality, so I used this as an opertunity to delete that mess and build v3 in Laravel 4, leveraging it's simple (initially Symfony-based) Routing, Database Migrations, Schema, Seeding, etc. Now we are doing the same for v4 but no rewrite was required this time, even though we have some different functionality the v3 repo was forked to a new one for v4 and both are being actively developed and living side-by-side on the same "API" servers.

By passing on some best practices and general good advice you can hit the ground running if you are new to API development. On the flip side, by recounting some horror stories (and how they were overcome/avoided/averted) you can hopefully avoid a lot of the pitfalls I either fell into, or nearly fell into, or saw others fall into. This book will discuss the theory of designing and building APIs in any language or framework with this theory applied in examples built in PHP. I'm going to try and avoid making it code-heavy to stop you falling asleep and to keep the non-PHP developers happy.

Some of the more advanced topics covered here are endpoint testing, embedding data objects in a consistent and scalable maner, paginating responses (including embedded objects) and HATEOAS links.

Sample Code

Throughout this book I will refer to source code which exists on a GitHub repo, which can be downloaded in a few ways. You can clone it:

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

Browse around it:

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

Download it as a `.zip` file:

```
http://bit.ly/apisyouwonthate-zip
```

This contains a few bits and bobs that will save you having to copy and paste things out of the ebook, which would probably be horrendous.

1 Useful Database Seeding

The first step to creating any sort of application is creating the database. Whether you are using some sort of relational platform, MongoDB, Riak, or whatever, you'll need a vague idea of how your data is going to be stored.

For relational databases it is very likely you will start off your planning with an entity-relationship diagram and for document based databases such as MongoDB, CouchDB or ElasticSearch you will just let your application magically build a schema, but either way you need to create a plan - even if it is on a napkin. This book will assume a traditional relational database is storing your data but the principles are easily adapted for NoSQL systems too.

This chapter assumes you've already got a database designed and built. I'll skip the "planning a database" section because it is incredibly boring, and there are plenty of other books on that. We're going straight onto the next step.

1.1 Introduction to Database Seeding

With a database schema designed and implemented it's time to store some data, but hold off entering your real data. It is far easier to use "dummy data" to test if the schema is appropriate for your API application as you can ditch the database and try again without worrying about keeping your data.

The process of populating a database is known as "seeding".

This data could be test users, content entries with a bunch of comments, fake locations available for check-in, fake notifications to display in an iPhone app (one of each type) or credit-card payments at various stages of processing - with some complete, some half done and some super-fraudulent looking ones.

The process of creating seeding scripts means you don't need to waste time creating this manually over and over again. Ultimately, the more processes you can automate during the development of your API, the more time you've got to consider the intricacies of your applications which need much more consideration.

Dummy data is necessary for realistic acceptance testing, getting freelancers/new hires up to speed with useful content, keeping real customer data private to those outside your company, and avoiding the temptation to copy live data over to your development environments.

Why is using production data in development bad?

Have you ever been writing a script that sends out emails and used some dummy copy while you're building it? Ever used some cheeky words in that content? Ever accidentally sent that email out to 10,000 real customers email addresses? Ever been fired for losing a company north of £200,000?

I haven't, but I know a guy that has. Don't be that guy.

What data should you use?

Garbage! Use absolute nonsense for your development database, but nonsense of the correct data type, size, and format. That can be done with a fun little library called Faker[1] by François Zaninotto[2] which is a wonderful little library that can essentially bullshit for Queen and country.

1.2 Building Seeders

Kapture uses the Laravel framework which has the joys of having Database Seeding[3] baked in. This is essentially a tarted up task which almost any modern PHP framework will have (or bloody well should do) so the principles are applicable to all.

Break your Database Seeders down into logical groupings. This doesn't need to be "one seeder-per-table" but it can be. The reason I don't try to stick to that rule is that sometimes your data needs to be built at the same time as other types of data, so for us Users are created in the same "seeder" as their settings, OAuth tokens, and friendship data is made. Putting that stuff into multiple seeders purely to keep things tidy would be an exercise in futility, and slow everything down for no reason.

So, this is a drastically simplified version of our user seeder all in one go, ignoring the Laravel specific structure. *If you're using Laravel 4, just shove this in your `run()` method.*

Creating a user with Faker and Eloquent ORM

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

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

    $user = User::create([
        'name'               => $faker->name,
        'email'              => $faker->email,
        'active'             => $i === 0 ? true : rand(0, 1),
        'gender'             => rand(0, 1) ? 'male' : 'female',
        'timezone'           => mt_rand(-10, 10),
        'birthday'           => rand(0, 1) ? $faker->dateTimeBetween('-40 years', '-18 yea\
rs') : null,
        'location'           => rand(0, 1) ? "{$faker->city}, {$faker->state}" : null,
        'had_feedback_email' => (bool) rand(0, 1),
        'sync_name_bio'      => (bool) rand(0, 1),
        'bio'                => $faker->sentence(100),
        'picture_url'        => $this->picture_url[rand(0, 19)],
    ]);
}
```

[1]

[2] https://twitter.com/francoisz/

[3]

So what do we have here? Let's go through this section at a time:

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

An instance of Faker, our bullshit artist for-hire.

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

We're going to want a certain number of users, but I'd recommend you have a few less on development than you do on testing or staging, because time.

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

Make a random name and random email. We don't have to define the pool of random data it uses, because IT'S MAGIC!

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

Ok I lied, our garbage is not 100% random. We want user number 1 to be active for tests later on.

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

Gender equality is important.

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

Our original developer decided that saving timezones as an integer was a clever thing to do. Bellend. How you gonna handle countries with +4.45 timezones bro? I still need to refactor this, but it's fine for now.

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

Users of all of our target age demographic.

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

Give us a city name and a state name. This works fine with foreign countries too which is cool.

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

Some user flags we don't care much about. True or false, whatever.

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

Make a sentence with 100 characters in it.

1.3 That's about it

You will end up making a lot of these files, and you'll want to populate pretty much every table you have with data. You'll also want to tell your Database Seeder to wipe all the tables you're going to populate. Do this globally right at the start of the process, don't wipe each table at the top of each seeder or you'll be wiping out content in that table from other seeders in the same process.

Example of an overall system in Laravel 4

```
class DatabaseSeeder extends Seeder
{
    public function run()
    {
        if (App::environment() === 'production') {
            exit('I just stopped you getting fired. Love Phil');
        }

        Eloquent::unguard();

        $tables = [
            'locations',
            'merchants',
            'opps',
            'opps_locations',
            'moments',
            'rewards',
            'users',
            'oauth_sessions',
            'notifications',
            'favorites',
            'settings',
            'friendships',
            'impressions',
        ];
```

```
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  }
```

This wipes everything, then runs other seeder classes to do their thing.

1.4 Secondary Data

As I said it is quite likely that you will need to insert data that relates to each other. To do this you work out which data will be primary (like users) and in the case of a check-in system probably "venues" or "merchants" depending on the nomenclature of your system.

For this example I will show how to create merchants, then attach "opportunities", which are essentially "campaigns".

Primary Seeder for the Merchant Table

```
1   <?php
2
3   class MerchantTableSeeder extends Seeder
4   {
5       /**
6        * Run the database seeds.
7        *
8        * @return void
9        */
10      public function run()
11      {
12          $faker = Faker\Factory::create();
13
14          // Create however many merchants
15          for ($i = 0; $i < Config::get('seeding.merchants'); $i++) {
16              Merchant::create([
17                  'name'        => $faker->company,
18                  'website'     => $faker->url,
```

```
                'phone'       => $faker->phoneNumber,
                'description' => $faker->text(200),
            ]);
        }
    }
}
```

Primary Seeder for the Opp Table

```
<?php

use Carbon\Carbon;
use Kapture\CategoryFinder;

class OppTableSeeder extends Seeder
{
    /**
     * Build it up
     *
     * @param Place
     */
    public function __construct(CategoryFinder $finder, Place $places)
    {
        $this->categoryFinder = $finder;
        $this->places = $places;
    }

    /**
     * Images.
     *
     * @var string
     */
    protected $imageArray = [
        'http://example.com/images/example1.jpg',
        'http://example.com/images/example2.jpg',
        'http://example.com/images/example3.jpg',
        'http://example.com/images/example4.jpg',
        'http://example.com/images/example5.jpg',
    ];

    /**
     * Run the database seeds.
     *
     * @return void
```

```
36      */
37     public function run()
38     {
39         $faker = Faker\Factory::create();
40 
41         foreach (Merchant::all() as $merchant) {
42 
43             // Create however many opps for this merchant
44             foreach (range(1, rand(2, 4)) as $i) {
45 
46                 // There are three types of image to add
47                 $image = Image::create([
48                     'name' => "{$merchant->name} Image #{$i}",
49                     'url' => $faker->randomElement($this->imageArray),
50                 ]);
51 
52                 // Start it immediately and make it last for 2 months
53                 $starts = Carbon::now();
54 
55                 // We need to definitely have at least one we are in control of
56                 if ($i === 1) {
57                     // Have ONE that ends really soon
58                     $ends = Carbon::now()->addDays(2);
59                     $teaser = 'Something about cheese';
60 
61                 } else {
62                     $ends = Carbon::now()->addDays(60);
63                     $teaser = $faker->sentence(rand(3, 5));
64                 }
65 
66                 $category = $this->categoryFinder->setRandom()->getOne();
67 
68                 $opp = Opp::create([
69                     'name'           => $faker->sentence(rand(3, 5)),
70                     'teaser'         => $teaser,
71                     'details'        => $faker->paragraph(3),
72                     'starts'         => $starts->format('Y-m-d H:i:s'),
73                     'ends'           => $ends->format('Y-m-d H:i:s'),
74                     'category_id'    => $category->id,
75                     'merchant_id'    => $merchant->id,
76                     'published'      => true,
77                 ]);
78 
79                 // Attach the location to the opp
80                 $opp->images()->attach($image, [
```

```
                    'published' => true
                ]);
            }

            echo "Created $i Opps for $merchant->name \n";
        }
    }
}
```

This might look a little crazy and it is certainly a mixture of lazy-static ORM usage in the controller and some dependency injection, but these seeders have not received a large amount of love. They definitely do their job, and could always be cleaner, but the basics here are:

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

Loop through all merchants.

```
            // Create however many opps for this merchant
            foreach (range(1, rand(2, 4)) as $i) {
```

Create between 1 and 4 opportunities for a merchant.

```
                // There are three types of image to add
                $image = Image::create([
                    'name' => "{$merchant->name} Image #{$i}",
                    'url' => $faker->randomElement($this->imageArray),
                ]);
```

Add an image from our array of example images on S3 or our website somewhere. The more the merrier.

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

I will talk more about finders in a later chapter, but for now just know this is a convenient way of getting a single random category back.

The rest should all be relatively obvious.

If you're using Laravel 4 you can run the above commands on the command line with: `$ php artisan db:seed`.

1.5 When to run this

This is often run manually, and automatically depending on the instances.

For example, if you've just added a new endpoint with new data, you will want to let your team-mates know to pull the latest code, run the migrations and run the db seed.

This is also great of course when a freelancer comes in to do some work, or a new developer starts up, or your iPhone dev wants to get some data to use. In all these instances that command just needs to be run on the command line.

This is also occasionally run manually on the staging server, and automatically on the Jenkins testing server when we deploy new builds of the API.

2 Planning and Creating Endpoints

With your database planned and full of fake but useful data it is time to plan what your endpoints are going to look like. The first step is to work out the requirements of an API, then we'll move onto some theory and finally see the theory implemented in some examples.

2.1 Functional Requirements

Try thinking of *everything* your API will need to handle. This will initially be a list of CRUD (Create, Read, Update, Delete) endpoints for your resources, Talk to your mobile app developer, your JS frontend people, or just talk to yourself if you are the only developer on the project.

Definitely talk to your customers or "the business" (they are the customers) and get them to help you think of functionality too, but don't expect them to know what an endpoint is.

When you have a relatively extensive list the next step is to make a simple list of "Actions". This is very much like planning a PHP class, you first write up pseudo-code referencing the classes and methods like they exist, right? TDD? If not that is how you should do it, or Chris Hartjes will find you, and he *will* kill you.

So if I have a "Places" resource in mind, I need to list out with just bullet points what it will do:

> Places
> - **Create**
> - **Read**
> - **Update**
> - **Delete**

That is fairly obvious. Who will be able to view these places and who will be able to create and edit them is (for now) irrelevant in our planning stages, as this API will get much smarter with the ideas of user-context and permissions at a later date. For now just list all the things that need to be done.

A paginate-able list of places is also a requirement, so get that down:

> Places
> - Create
> - Read
> - Update
> - Delete
> - **List**

The API will need to offer the ability to search places by location too, but that is not a brand new endpoint. If the API was built with SOAP or XML-RPC you would create a `getPlacesByLatAndLon` method to hit in the URL, but this isn't SOAP - thankfully. The list method will handle that with a few parameters, so why not shove them in as a note for later:

Places
- Create
- Read
- Update
- Delete
- List **(lat, lon, distance or box)**

Adding a few parameters as a reminder at this stage is cool, but lets not worry about adding too much. For example, create and update are complicated so adding every single field would be a mess.

Update is more than just updating the specific "places" fields in the places SQL table. Update can do all sorts of cool stuff. If you need to "favorite" a place, just send `is_favorite` to that endpoint and you've favorited it. More on that later, just remember that not every single action requires its own endpoint.

Places will also need to have an image uploaded via the API. In this example we are only going to accept one image for a place and a new image overrides the old, so add "Image" to the list. Otherwise you'd add "Images" to the list:

Places
- Create
- Read
- Update
- Delete
- List (lat, lon, distance or box)
- **Image**

A complete API "action plan" might look like this:

Categories
- Create
- List

Checkins
- Create
- Read
- Update
- Delete
- List
- Image

Opps
- Create
- Read
- Update
- Delete

- List
- Image
- Checkins

Places
- Create
- Read
- Update
- Delete
- List (lat, lon, distance or box)
- Image

Users
- Create
- Read
- Update
- Delete
- List (active, suspended)
- Image
- Favorites
- Checkins
- Followers

That might not contain everything, but it seems like a fairly solid start to our API. It is certainly going to take long enough to write all that so if somebody thinks of something else they can just make an issue.

Moving on.

2.2 Endpoint Theory

Turning this "Action Plan" into actual endpoints requires knowing a little theory on RESTful APIs and "best practices" for naming conventions. There are no right answers here, but some approaches have fewer cons than others. I will try to push you in the direction I have found to be most useful, and highlight the pros and cons of each.

GET Resources

- `GET /resources` - Some paginated list of stuff, in some logical default order for that specific data.
- `GET /resources/X` - Just entity X. That can be an ID, hash, slug, username, etc as long as it unique to one "resource".
- `GET /resources/X,Y,Z` - The client wants multiple things, so give them multiple things.

It can be hard to pick between sub-resource URLs or embedded data. Embedded data can be rather difficult to pull off so that will be saved for Chapter 7: Embedding Data. For now the answer is "just sub-resources", but eventually the answer will be "both". This is how sub-resources look:

- `GET /places/X/checkins` - Find all the checkins for a specific place.
- `GET /users/X/checkins` - Find all the checkins for a specific user.
- `GET /users/X/checkins/Y` - Find a specific checkin for a specific user.

The latter is questionable, and not something I have ever personally done. At that point I would prefer to simply use `/checkins/X`.

Auto-Increment is the Devil

In these examples X and Y can be an auto-incrementing ID as many developers will assume. One important factor with auto-incrementing ID's is that anyone with access to your API will know exactly how many resources you have, which might not be a statistic you want your competitors to have.

Consumers could also write a script which hits `/users/1`, then `/users/2` and `/users/3`, etc scraping all data as it goes. Sure they could probably do that from the "list" endpoints anyway, but not all resources should have a "get all" approach.

Instead a unique identifier is often a good idea. A universal unique identifier (UUID) seems like a logical thing to do: ramsey\uuid for PHP[1], uuid for Ruby[2], uuid in Python 2.5+[3].

DELETE Resources

Want to delete things? Easy:

- `DELETE /places/X` - Delete a single place.
- `DELETE /places/X,Y,Z` - Delete a bunch of places.
- `DELETE /places` - This is a potentially dangerous endpoint that could be skipped, as it should delete all places.
- `DELETE /places/X/image` - Delete the image for a place, or:
- `DELETE /places/X/images` - If you chose to have multiple images this would remove all of them.

POST v PUT: FIGHT!

What about creating and updating? This is where it gets almost religious. There are lots of people who will try to pair the HTTP POST or HTTP PUT verb to a specific CRUD action and always only ever do that one action with that one verb. That sucks and is not productive or functionally scalable.

Generally speaking, PUT is used if you know the entire URL before hand and the action is idempotent. Idempotent is a fancy word for "can do it over and over again without causing different results".

For example, create *could* be a PUT if you are creating one image for a place. If you were to do this:

[1] https://github.com/ramsey/uuid
[2] https://rubygems.org/gems/uuid
[3] http://docs.python.org/2/library/uuid.html

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

That would be a perfect example of when to use a `PUT` because you already know the entire URL and you can do it time and time again.

At Kapture we use a POST to `/checkins` to create the metadata for that new checkin, then that will return the URL for us to PUT the image to. You could try checking in multiple times and it wouldn't matter because none of those processes would be complete, but POSTing multiple times is not idempotent because each checkin is different. PUT is idempotent because you are uploading that image to the full URL and you can do it over and over again if you like (because the upload failed and it has to try again).

So, if you have multiple images for places maybe you could `POST /places/X/images` and multiple attempts would be different images. If you know you're only going to have one image and a new attempt is an override then `PUT /places/X/image` would be ideal.

Another example could be user settings:

- `POST /me/settings` - I would expect this to allow me to POST specific fields one at a time, not force me to send the entire body of settings.
- `PUT /me/settings` - Send me ALL the settings.

It's a tricky difference, but do not try and tie a HTTP Method to one CRUD action only.

Plural, Singular or Both?

Some developers decide to make all endpoints singular but I take issue with that. Given `/user/1` and `/user`, which user is that last one returning? Is it "me"? What about `/place`? It returns multiple? Meh.

I know it can be tempting to create `/user/1` and `/users` because the two endpoints do different things, right? I started off down this route (#pun) with my original planning, but in my experience this convention grows badly. Sure it works with the example of "users", but what about those fun English words that create exceptions, like `/opportunity/1` which becomes `/opportunities`. Gross.

I pick plural for everything as it is the most obvious:

- `/places` - "If I run a GET on that I will get a collection of places"
- `/places/45` - "Pretty sure I am just talking about places 45"
- `/places/45,28` - "Ahh, places 45 and 28, got it"

Another solid reason for using plural consistently is that it allows for consistently named sub-resources:

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

Consistency is key.

2.3 Planning Endpoints

Controllers

You need to list events, venues, users and categories? Easy. One controller for each type of resource:

- CategoriesController
- EventsController
- UsersController
- VenuesController

Everything in REST is a resource, so each resource needs a controller.

Later on we will look at some things that are not resources. Sub-resources can sometimes just be a method, for example profile and settings are a sub-resource of Users, so maybe they can go in the User controller. These rules are flexible.

Routes

Try to avoid the temptation to screw around with magic routing conventions[4], just make them manually. I will keep going with the previous examples and show the process of turning the action plan into routes, using Laravel 4 syntax because why not:

Action	Endpoint	Route
Create	POST /users	Route::post('users', 'UsersController@create');
Read	GET /users/X	Route::get('users/{id}', 'UsersController@show');
Update	POST /users/X	Route::post('users/{id}', 'UsersController@update');
Delete	DELETE /users/X	Route::delete('users/{id}', 'UsersController@delete');
List	GET /users	Route::get('users', 'UsersController@list');
Image	PUT /users/X/image	Route::put('users/{id}/image', 'UsersController@uploadImage');
Favorites	GET /users/X/favorites	Route::get('users/{id}/favorites', 'UsersController@favorites');
Checkins	GET /users/X/checkins	Route::get('users/{user_id}/checkins', 'CheckinsController@index');

There are a few things in here worth considering.

1. Create and Update are both `POST`. Not because `PUT` is evil, but because when we create a user in this example we do not know what their URL would be because our URLs are auto-generated with their ID. If we knew their username for example we could have `PUT /users/philsturgeon` but then it would be hard to work out if we are accidentally making the same HTTP request a second time, or trying to override an existing user.
2. Favorites go to the `UserController`, because favorites are only ever relevant to the user.

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

3. Checkins go to the `CheckinController`, because we might already have a checkin controller handling `/checkins` and the logic is basically identical. We'll know if there is a `user_id` param in the URL if our router is nice enough to let us know, so we can use that to make it user specific if needs be.

Those last two are kinda complex, but are examples of things you can be thinking about at this point. You don't want to have multiple endpoints doing painfully similar things with copy and paste logic because A) PHP Copy/Paste Detector[5] will be angry, B) your iPhone developer will be mad that different endpoints are acting differently - therefore confusing RestKit and C) it's boring and ain't nobody got time for that.

Methods

When you have listed all of the routes you will need for your application go and make them all in their controllers. Make them all empty and have one of them `return "Oh hai!";` and check the output. `GET /places` for example should `Oh hai!` in the browser. You just wrote an API.

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

3 Input and Output Theory

Now that we have a good idea how endpoints work the next glass of theory to swallow down is input and output. This is the easiest of all really as it's just HTTP "requests" and "responses", which is the same as AJAX or anything else. If you have ever been forced to work with SOAP you will know all about WSDLs. If you know what they are, be happy you don't have to work with them anymore. If you don't know what they are then be happy you never had to learn. SOAP was the worst.

Input is purely a HTTP request and there are multiple parts to this. Here is an example:

3.1 Requests

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

This is a very simple `GET` request. We can see the URL path being requested is `/places` with a query string of `lat=40.759211&lon=-73.984638`. The HTTP version in use is HTTP/1.1, the host name is defined. This is essentially what your browser does when you go to any website. Rather boring I'm sure.

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

{ "user_id" : 2 }
```

Here we make a POST request with a "HTTP Body" The `Content-Type` header points out we are sending JSON and the blank line above the JSON separates the "HTTP Headers" from the "HTTP Body". HTTP really is amazingly simple, this is all you need to do for anything and you can do all of this with a HTTP client in whatever programming language you feel like using this week:

Using PHP and the Guzzle HTTP library to make a HTTP Request

```
use Guzzle\Http\Client;

$headers = [
    'Authorization' => 'vr5HmMkzlxKE70W1y4MibiJUusZwZC25NOVBEx3BD1',
    'Content-Type' => 'application/json',
];
$payload = [
```

```
    'user_id' => 2
];

// Create a client and provide a base URL
$client = new Client('http://api.example.com');

$req = $client->post('/moments/1/gift, $headers, json_encode($payload))
```

Using Python and the Requests HTTP library to make a HTTP Request

```
import requests

headers = {
    'Authorization': 'vr5HmMkzlxKE70W1y4MibiJUusZwZC25NOVBEx3BD1',
    'Content-Type': 'application/json',
}
payload = {
    'user_id': 2
}
req = requests.post('http://api.example.com/moments/1/gift', data=json.dumps(payload), hea\
ders=headers)
```

It's all the same. Define your headers, define the body in an appropriate format and send it on its way. Then you get a response, so let's talk about those.

3.2 Responses

Much the same as a HTTP Request your HTTP Response is going to end up as plain text (unless you're using SSL but shut up we aren't there yet).

Example HTTP response containing a JSON body

```
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":"Occaecati exceptu\
ri magni odio distinctio dolores illum voluptas voluptatem in repellendus eum enim ","gend\
er":"female","picture_url":"https:\/\/si0.twimg.com\/profile_images\/711293289\/hhdl-twitt\
er_normal.png","cover_url":null,"location":null,"timezone":-1,"birthday":"1989-09-17 16:27\
:36","status":"available","created_at":"2013-11-22 16:37:57","redeem_by":"2013-12-22 16:37\
:57"}
```

We can spot some fairly obvious things here. `200 OK` is a standard "no issues here buddy" response. We have a `Content-Type` again, and the API is pointing out that caching this is not ok. The `X-Powered-By` header is also a nice little reminder that I should switch `expose_php = On` to `expose_php = Off` in php.ini. Oops.

This is essentially the majority of how an API works. Just like learning a programming language you will always come across new functions and utilities which will improve the RESTful-ness of your API and I will point out a bunch of them as we go, but just like the levenshtein()[1] there will be HTTP Headers that you had no idea existed popping up that you will think "How the shit did I not notice that?".

3.3 Supporting Formats

Picking what formats to support is hard, but there are a few easy wins to make early on.

No Form Data

PHP developers always try to do something that literally nobody else does, and that is to send data to the API using: `application/x-www-form-urlencoded`.

This mime-type is one of the few ways that browsers send data via a form when you use HTTP POST, and PHP will take that data, slice it up and make it available in `$_POST`. Because of this convenient feature many PHP developers will make their API send data that way, then wonder why sending data with PUT is "different".

Urf.

`$_GET` and `$_POST` have nothing to do with HTTP GET and HTTP POST. `$_GET` just contains query string content *regardless* of the HTTP method. `$_POST` contains the values of the HTTP Body if it was in the right format and the Content-Type header is `application/x-www-form-urlencoded`.

So knowing that PHP just has some silly names for things, we can move on and completely ignore $_POST. Pour one out in the ground, because it is dead to you.

Why? So many reasons, including the fact that once again everything in `application/x-www-form-urlencoded` is a string.

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

Yeah you have to use `1` or `0` because `bar=true` would be `string("true")` on the server-side. Data-types are important, so lets not just throw them out the window for the sake of "easy access to our data". That argument is also moronic as `Input::json('foo')` is possible in most decent PHP frameworks and even without it you just have to `file_get_contents('php://input')` to read the HTTP body yourself.

[1] http://php.net/manual/en/function.levenshtein.php

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

{
    "checkin": {
        "place_id" : 1,
        "message": "This is a bunch of text.",
        "with_frinds": [1, 2, 3, 4, 5]
    }
}
```

This is a perfectly valid HTTP body for a checkin. You know what they are saying, you know who the user is from their auth token, you know who they are with and you get the benefit of having it wrapped up in a single `checkin` key for simple documentation and easy "You sent a checkin object to the user settings page... dumbass." responses.

That same request using form data is a mess.

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

checkin[place_id]=1&checkin[message]=This is a bunch of text&checkin[with_frinds][]=1&chec\
kin[with_frinds][]=2&checkin[with_frinds][]=3&checkin[with_frinds][]=4&checkin[with_frinds\
][]=5
```

This makes me upset *and* angry. Do not do it in your API.

Finally, do not try to be clever by mixing JSON with form data:

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

json="{
    \"checkin\": {
        \"place_id\" : 1,
        \"message\": \"This is a bunch of text.\",
        \"with_frinds\": [1, 2, 3, 4, 5]
    }
}"
```

Who the hell is the developer trying to impress with stuff like that? It is insanity and anyone who tries this needs to have their badge and gun revoked.

JSON and XML

Any modern API you talk to will support JSON unless it is a financial services API or the developer is a moron - probably both to be fair. Sometimes they will support XML too. XML used to be a the popular format for data transfer with both SOAP and XML-RPC (duh). XML is however a nasty-ass disgusting mess of tags and the file-size of an XML file containing the same data as a JSON file is often much larger.

Beyond purely the size of the data being stored, XML is horribly bad at storing type. That might not worry a PHP developer all that much as PHP is not really any better when it comes to type, but look at this:

```
{
    "place": {
        "id" : 1,
        "name": "This is a bunch of text.",
        "is_true": false,
        "maybe": null,
        "empty_string": ""
    }
}
```

That response in XML:

```
<places>
    <place>
        <id>1</id>,
        <name>This is a bunch of text.</name>
        <is_true>0</is_true>
        <maybe />
        <empty_string />
    </place>
</places>
```

Basically in XML *everything* is considered a string, meaning integers, booleans and nulls can be confused. Both `maybe` and `empty_string` have the same value, because there is no way to denote null either. Gross…

Now, the XML-savvy among you will be wondering why I am not using attributes to simplify it? Well, this XML structure is a typical "auto-generated" chunk of XML converted from an array in the same way that JSON is built - but this of course ignores attributes and does not allow for all the specific structure that your average XML consumer will demand.

If you want to start using attributes for some bits of data but not others then your conversion logic becomes INSANELY complicated. How would we build something like this?

```
<places>
    <place id="1" is_true="1">
        <name>This is a bunch of text.</name>
        <empty_string />
    </place>
</places>
```

The answer is unless you seek specific fields and try to guess that an "id" is probably an attribute, etc then there is no programatic way in your API to take the same array and make JSON AND XML. Instead you realistically need to use a "view" (from the MVC pattern) to represent this data just like you would with HTML or work with XML generation in a more OOP way. Either way it is an abomination and I refuse to work in those conditions. Luckily nobody at Kapture wants XML so I don't have to move back to England just yet.

If your team is on the fence about XML and you don't 100% need it, then don't bother using it. I know it is fun to show off your API switching formats and supporting all sorts of stuff, but I would strongly urge you to work out what format(s) you actually need and STICK TO THOSE. Sure Flickr supports lolcat as input and output, but they have a much bigger team so you don't need to worry about it. JSON is fine. If you have a lot of Ruby bros around then you will probably want to output YML too, which is as easy to generate as JSON in most cases.

3.4 Content Structure

This is a tough topic and there is no right answer and whether you use EmberJS, RestKit or any other framework with knowledge of REST you will find somebody annoyed that the data is not in their preferred format. There are a lot of factors and I will simply explain them all and let you know where I landed.

JSON API

There is one recommended format on JSON API[2] which maybe you all just want to use. It suggests that both single resources and resource collections should both be inside a plural key.

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

Pros

- Consistent response, *always* has the same structure

[2] http://jsonapi.org/format/

Cons

- Some RESTful/Data utilities freak about have single responses in an array
- Potentially confusing to humans

EmberJS (EmberData) out of the box will get fairly sad about this and I had trouble hacking it to support the fact that only requesting one item would still return an array that looks like it could contain multiple. It seems (to me) to be a weird rule. Imagine you call `/me` to get the current user, and it gives you a collection? What the hell?

Do not discount JSON API as it is a wonderful resource with a lot of great ideas, but it strikes me as over-complicated in multiple areas.

Twitter-style

Ask for one user get one user:

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

Ask for a collection of things and get a collection of things:

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

Pros

- Minimalistic response
- Almost every framework/utility can comprehend it

Cons

- No space for pagination or other meta data

This is potentially a reasonable solution if you will never use pagination or meta data.

Facebook-style

Ask for one user get one user:

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

Ask for a collection of things and get a collection of things, but namespaced:

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

Pros

- Space for pagination and other meta data in collection
- Simplistic response even with the extra namespace

Cons

- Single items still can only have meta data by embedding it item resource

By placing the collection into the “data” namespace you can easily add other content next to it which relates to the response but is not part of the list of resources at all. Counts, links, etc can all go here (more on this later). It also means when you embed other nested relationships you can include a “data” element for them and even include meta data for those embedded relationships. More on that latr too.

The only potential “con” left with Facebook is that the single resources are not namespaced, meaning that adding any sort of meta data would pollute the global namespace - something which PHP developers are against after a decade of flagrantly doing so.

So the final output example (and the one which I am starting to use at Kapture for v4) is this;

Much Namespace, Nice Output

Namespace the single items.

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

Namespace the multiple items.

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

This is close to the JSON API response, has the benefits of the Facebook approach and is just like Twitter but everything is namespaced. Some folks (including me in the past) will suggest that you should change "data" to "users" but when you start to nest your data you want to keep that special name for the name of the relationship. For example:

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

So here we can see the benefits of keeping the root scope generic. We know that a user is being returned because we are requesting a user, and when comments are being returned we wrap that in a "data" item so

that pagination or links can be added to that nested data too. This is the structure I will be testing against and using for examples, but it is only a simple tweak between any of these structures.

We will get to links, relationships, side-loading, pagination, etc in later chapters, but for now forget about it. All you want to worry about is your response, which consists of this chunk of data or an error.

4 Status Codes, Errors and Messages

If everything goes smoothly you want to show some data. If a valid request comes in for a data which is valid you show data, if creating something on the API with valid data, you show the created object. If something goes wrong, however, you want to let people know what is wrong using two simultaneous approaches:

1. HTTP status codes
2. Custom error codes and messages

4.1 HTTP Status Codes

Status Codes are used in all responses and have a number from 200 to 507 - with plenty of gaps in between - and each has a message and a definition. Most server-side languages, frameworks, etc default to "200 OK".

Status codes are grouped into a few different categories:

2xx is all about success
Whatever the client tried to do was successful up to the point that the response was send. Keep in mind that a status like 202 Accepted doesn't say anything about the actual result, it only indicates that a request was accepted and is being processed asynchronously.

3xx is all about redirection
These are all about sending the calling application somewhere else for the actual resource. The best known of these are the 303 See Other and the 301 Moved Permanently which are used a lot on the web to redirect a browser to another URL.

4xx is all about client errors
With these status codes we indicate that the client has done something invalid and needs to fix the request before resending it.

5xx is all about service errors
With these status codes we indicate that something went wrong in the service. For example a database connection failed. Typically a client application can retry the request. The server can even specify when the client is allowed to retry the command using a Retry-After HTTP header.

Using HTTP status codes in a REST service[1] – *Maurice de Beijer*

For a more complete list of HTTP status codes and their definitions the REST & WOA Wiki[2] has an extensive list of them.

Arguments between developers will continue for the rest of time over the exact appropriate code to use in any given situation, but these are the status codes the API uses at Kapture:

[1] http://www.develop.com/httpstatuscodesrest
[2] http://restpatterns.org/HTTP_Status_Codes

- 200 - Generic everything is OK
- 201 - Created something OK
- 202 - Accepted but is being processed async (video is encoding, image is resizing, etc)
- 400 - Wrong arguments (missing validation)
- 401 - Unauthorized (no current user and there should be)
- 403 - The current user is forbidden from accessing this data
- 404 - That URL is not a valid route, or the item resource does not exist
- 410 - Data has been deleted, deactivated, suspended, etc
- 405 - Method Not Allowed (your framework will probably do this for you)
- 500 - Something unexpected happened and it is the APIs fault
- 503 - API is not here right now, please try again later

It can be tempting to try and squeeze as many error codes in as you can, but I would advise you to try and keep it simple. You won't unlock any achievements for using them all.

Most 5xx issues will most likely happen under odd architecture or server related issues that are nothing to do with your API, like if PHP-FPM segfaults behind nginx (502), if your Amazon Elastic Load Balancer has no health instances (503) or if your hard-drive fills up somehow (507).

4.2 Error Codes and Error Messages

Error codes are usually strings or integers that act as a unique index to a correspond human-readable error message with more information about what is going wrong. That sounds a lot like HTTP status codes, but these errors are about application specific things that may or may not be anything to do with HTTP specific responses.

Some folks will try to use HTTP status codes exclusively and skip using error codes because they do not like the idea of making their own error codes or having to document them, but this is not a scalable approach. There will be some situations where the same endpoint could easily return the same status code for more than one different condition. The status codes are there to merely hint what is going on, relying on the actual error code and error message to provide more information if the client is interested.

For example, an issue with the access token will always result in the user not being recognized. An uninterested client would simply say "User could not get in" while a more interested client would probably prefer to offer suggestions via messages in their own webapp/iPhone app interface.

```
{
  error: {
    type: "OAuthException",
    message: "Session has expired at unix time 1385243766. The current unix time is 138584\
8532."
  },
}
```

Everyone can understand that. Facebook sadly is missing an error code, so sometimes you find yourself doing string checking on the message which is lunacy, so throw in a code too.

Foursquare is not a bad example of using both, but they place an emphasis on tying their errors to a status code.

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

Twitter does a great job of having HTTP status codes documented and having specific error codes for other issues too. Some are tied to HTTP status codes (which is fine) but many are not. Some are also tied to the same status code, highlighting the issues raised above.

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

Code	Text	Description
161	You are unable to follow more people at this time	Corresponds with HTTP 403 - thrown when a user cannot follow another user due to some kind of limit
179	Sorry, you are not authorized to see this status	Corresponds with HTTP 403 - thrown when a Tweet cannot be viewed by the authenticating user, usually due to the tweet's author having protected their tweets.

Programmatically Detecting Error Codes

Here is an example of the sort of ways people can use error codes to make their application respond intelligently to failure of something as basic as a posted Twitter status:

```
    try:
        api.PostUpdates(body['text'])

    except twitter.TwitterError, exc:

        skip_codes = [
            # Page does not exist
            34,

            # You cannot send messages to users who are not following you
            150,

            # Sent too many
            # TODO Make this requeue with a dekay somehow
            151
        ]

        error_code = exc.__getitem__(0)[0]['code']
```

```
19
20          # If the token has expired then lets knock it out so we don't try again
21          if error_code in skip_codes:
22              message.reject()
23
24          else:
25              # Rate limit exceeded? Might be worth taking a nap before we requeue
26              if error_code == 88:
27                  time.sleep(10)
28
29              message.requeue()
```

Compare this sort of logic with Facebook - and their lack of error codes:

```
1       except facebook.GraphAPIError, e:
2
3           phrases = ['expired', 'session has been invalidated']
4
5           for phrase in phrases:
6
7               # If the token has expired then lets knock it out so we dont try again
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)
```

If they change their error messages then this might stop working, which would be shitty.

4.3 Common Pitfalls

200 OK and Error Code

If you return a HTTP status code of 200 with an error code then Chuck Norris will roundhouse your door in, destroy your computer, instantly 35-pass wipe your backups, cancel your Dropbox account and block you from GitHub. HTTP 4xx or 5xx codes alert the client that something bad happened, and error codes provide specifics of the exact issue if the client is interested.

Non-Existent, Gone, or Hiding?

404 is drastically overused in applications. People use it for “never existed”, “no longer exists”, “you can’t view it” and “it is deactivated” which is way too vague. That can be split up into 404, 403 and 410 but this is still vague.

If you get a 403 this could be because the requesting user is in not in the correct group to see the requested content. Should the client suggest you upgrade your account somehow? Are you not friends with the users content you are trying to view? Should the client suggest you add them as a friend?

A 410 on a resource could be due to a user deleting that entire piece of content or it could be down to the user deleting their entire account.

In all of these situations the ideal solution is to compliment the HTTP status code with an error code, which can be whatever you want as long as they are unique within your API and documented somewhere. Do not do what Google does and supply a list of error codes then have other error codes which are not documented anywhere, because if I see that I will come for you.

5 Endpoint Testing

You might be sitting there thinking "This really escalated quickly, I'm not ready for testing!" but this is essentially the point. You have to set up your tests as early as possible so you actually bother using them, otherwise they become the "next thing" that just never gets done. Have no fear, testing an API is not only easy it is actually really quite fun.

5.1 Concepts & Tools

With an API there are a few things to test, but the most basic idea is "when I request this URL, I want to see a 'foo' resource", and "when I throw this JSON at the API, it should accept it or freak out."

This can be done in several ways and a lot of people will instantly try to unit-test it, but that quickly becomes a nightmare as while you might think just writing a bit of code with your favorite HTTP client is simple, if you have over 50 endpoints and want to do multiple checks per endpoint you end up with a MESS of code and it's no fun for anyone.

The more code you have in your tests, the higher the chances of your tests being rubbish and you run the risk of false positives, which are super dangerous.

So, the most simplistic approach will be to use a BDD tool. A very popular BDD tool is Cucumber[1] and this is considered by many to be a Ruby tool. It can in fact be used for Python, PHP and probably a whole bevy of other languages but some of the integrations can be tricky. For the PHP users here, we will be using Behat which is pretty much the same thing, along with Gherkin[2] (the same DSL that Cucumber uses, so all of us are on basically the same page.)

5.2 Setup

If you are a Ruby user you have the ease of simply running `$ gem install cucumber`, or shove it in your `Gemfile`, whatever.

As a PHP developer you simply need to install Behat, and this can be done with Composer[3]. It is fair to assume that if you are using any sort of modern PHP framework you are already familiar with this so I won't bore the non-PHP devs by getting stuck into it, but your `composer.json` file should have these items inside:

[1] http://cukes.info/
[2] http://docs.behat.org/guides/1.gherkin.html
[3] http://getcomposer.org/

Basic composer.json requirements for Behat

```
{
    "require": {
        "behat/behat": "2.4.*@stable"
    },

    "config": {
        "bin-dir": "bin/"
    }
}
```

Run `$ composer install` and you should be good to go.

God knows what to do with Python. Google should help you there. This could get complicated so I'm going to stick with PHP and Behat and you folks can just convert in your head as we go.

5.3 Initialise

Make a folder inside your codebase repo called "tests" and another inside that called "behat" - because one day maybe there will be lots of different types of test.

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

This will have the following output:

```
+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
```

Next.

5.4 Features

Features are a way to group your various tests together. For me I keep things fairly simple and consider each "resource" and "sub-resource" to be its own "feature".

Looking at our users example from Chapter 2:

Action	Endpoint	Feature
Create	POST /users	features/users.feature
Read	GET /users/X	features/users.feature
Update	POST /users/X	features/users.feature
Delete	DELETE /users/X	features/users.feature
List	GET /users	features/users.feature
Image	PUT /users/X/image	features/users-image.feature
Favorites	GET /users/X/favorites	features/users-favorites.feature
Checkins	GET /users/X/checkins	features/users-checkins.feature

So, anything to do with `/places` and `/places/X` would be the same, but as soon as you start looking at `/places/X/checkins` that becomes a new feature because we are talking about something else.

You can use that convention or try something else, but this grows pretty well without having a bazillion files to sift through.

5.5 Scenarios

Gherkin uses "Scenarios" as its core structure and they each contain "steps". In a unit-testing word these would be their own methods, and the "steps" would be "assertions".

These Features and Scenarios line up with the "Action Plan" created in Chapter 2. Each RESTful Resource needs at least one "Feature", and because each "Action" has an "Endpoint" we need *at least* one "Scenario" for each "Action".

Too much jargon? Time for an example:

```
Feature: Places

Scenario: Finding a specific place
    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: Listing all places is not possible
    When I request "GET /places"
    Then I get a "400" response

Scenario: Searching non-existent places
    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: Searching places with filters
    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
```

This uses some custom rules which have been defined in `features/bootstrap/FeatureContext.php` but more on that shortly.

The Feature file is called `places.feature` and has 4 scenarios. One to show that listing all places is not allowed (400 means bad input, your should specify lat lon), another to find a specific place and two more to test how well searching works.

I try to think up the guard clauses that my endpoints will need, then make a "Scenario" for each of those. So, if you don't send a lat/lon to search then it errors. Test that.

Expecting a boolean value but get a string? Test that:

```
Scenario: Wrong Arguments for user follow
    Given I have the payload:
        """
        {"is_following": "foo"}
        """
    When I request "PUT /users/1"
    Then I get a "400" response
```

Want to be sure your controllers can handle weird requests with a 404 instead of freaking out and going all 500 Internal Error? Test that.

```
Scenario: Try to find an invalid moments
    When I request "GET /moments/nope"
    Then I get a "404" response
```

Sure you don't actually have any code yet, but you can write all of these tests based off of nothing but your "Action Plan" and your Routes. You should use what you know about the output content structure from Chapter 3 to plan what output you expect to see.

Then all you need to do is... you know... build your entire API. We'll get there.

5.6 Prepping Behat

You are probably wondering how you actually run these tests, because Behat involves making HTTP requests and you've just been writing text-files. Well, the class in `FeatureContext.php` handles all of that and a lot more, but first we need to configure Behat so we know what the hostname is going to be for these requests.

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

In this file put in something along the lines of:

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

If you have virtual hosts set up on your machine then use those, and if you are running a local web-server on a different port then obviously you can use that too. That value could be `http://localhost:4000` or `http://dev-api.example.com`, it does not matter.

With that ready - and if you wish to use the extra functionality provided in the examples above - you will need to use the file located in the resource ZIP listed in the intro. The file is `build-apis-you-wont-hate/behat/FeatureContext.php` and it replaces the default `tests/behat/bootstrap/FeatureContext.php`.

5.7 Running Behat

This is the easiest bit:

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

Everything is broken. Yaaay!

In the next chapter we will start to turn some of these tests green by putting actual data in there.

6 Outputting Data

In Chapter 3: Input and Output Theory we looked at the theory of the output structure and the pros and cons for various different formats. This book assumes you have picked your favorite, and it assumes that favorite is my favorite. This doesn't matter all that much but doing everything for everyone would be an exercise in futility and boredom.

The aim of this chapter is to help you build out your controller endpoints so that you can fill up a few of those tests with green lights, instead of the omnishambles of errors and fails you are most likely facing.

The examples in the first section will be trying to show off a list of places, and show of one specific place:

```
{
    "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 The Direct Approach

The first thing that every developer tries to do is take their favorite ORM, ODM, DataMapper or Query Builder, pull up a query and wang that result directly into the output.

Dangerously bad example of passing data from the database directly as output

```
<?php
class PlaceController extends ApiController
{
    public function show($id)
    {
        return json_encode([
            'data' => Place::find($id)->toArray(),
        ]);
    }

    public function list()
    {
        return json_encode([
            'data' => Place::all()->toArray(),
        ]);
    }
}
```

This is the absolute worst idea you could have for enough reasons for me to fill up a chapter on its own, but I will try to keep it to just a section.

ORMs in Controllers

Your controller should definitely not have this sort of ORM/Query Builder logic scattered around the methods. This is done to keep the example to one class.

Performance: If you return "all" items then that will be fine during development, but suck when you have a thousand records in that table… or a million.

Display: PHP's popular SQL extensions all type-cast all data coming out of a query as a string, so if you have a MySQL "boolean" field (generally this is a `tinyint(1)` field with a value of `0` or `1`) will display in the JSON output as a string, with a value of `"0"` or `"1"` which is lunacy. If you're using PostgreSQL it is even worse, the value directly output by PHP's PostgreSQL driver is `"f"` or `"t"`. Your mobile developers won't like it one bit, and anyone looking at your public API is going to immediately consider this an amateur API. You want `true` or `false` as an actual JSON boolean, not a numeric string or a `char(1)`.

Security: Outputting all fields can lead to API clients (users of all sorts) being able to view your users passwords, see sensitive information like email addresses for businesses involved (venues, partners, events,

etc), gain access to secret keys and tokens generally not allowed. If you leak your forgotten password tokens for example then you're going to have an EXTREMELY bad time, its as bad as leaking the password itself.

Some ORM's have a "hidden" option to hide specific fields from being output. If you can promise that you and every single other developer on your team (now, next year and for the entire lifetime of this application) will remember about that then congratulations, you could also achieve world peace with a team that focused.

Stability: If you change the name of a database field, or modify your MongoDB document, or change the statuses available for a field between v3 and v4 then your API will continue to behave perfectly, but all of your iPhone users are going to have busted crashing applications and it is your fault. You'll promise yourself that you won't change things, but you absolutely will. Change happens.

So, next our theoretical developer friend will try hard-coding the output.

Laborious example of type-casting and formatting data for output

```
<?php
class PlaceController extends ApiController
{
    public function show($id)
    {
        $place = Place::find($id);

        return json_encode([
            'data' => [
                'id'         => (int) $place->id,
                'name'       => $place->name,
                'lat'        => (float) $place->lat,
                'lon'        => (float) $place->lon,
                'created_at' => (string) $place->created_at,
            ],
        ]);
    }

    public function list()
    {
        $places = array();

        foreach (Place::all() as $place) {
            $places[] = [
                'id'         => (int) $place->id,
                'name'       => $place->name,
                'lat'        => (float) $place->lat,
                'lon'        => (float) $place->lon,
                'created_at' => (string) $place->created_at,
            ];
        }
```

```
32
33          return json_encode([
34              'data' => $places,
35          ]);
36      }
37  }
```

Thanks to specifying exactly what fields to return in the JSON array the security issues are taken care of. The type-casting of various fields turn numeric strings into integers, coordinates into floats, and that pesky `Carbon` (DateTime) object from Laravel into a string, instead of letting the object turn itself into an array.

The only issue this has not taken care of from the above example is performance, but that is a job for pagination which will be covered in a later chapter.

A new issue has however been created, which should be a fairly obvious one: This is icky. Our theoretical developer now tries something else.

Considerably better approach to formatting data for output

```
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          return json_encode([
21              'data' => $places,
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  }
```

Certainly much better, but what if a different controller wants to show a place at any point? You could theoretically move all of these transform methods to a new class or shove them in the `ApiController`, but that would just be odd.

Really you want to make what I have come to call "Transformers", partially because the name is awesome and because that is what they are doing.

These are essentially just classes which have a transform method, which does the same as the `transformPlaceToJson()` above, but to avoid you having to learn how to make your own I have released a PHP package which takes care of it: Fractal[1].

6.2 Transformations with Fractal

With Fractal, Transformers are created as either a callback or an instance of an object implementing `League\Fractal\TransformerAbstract`. They do exactly the job that our `transformPlaceToJson()` method did but they live on their own, are easily unit-testable (if that floats your boat) and remove a lot of presentation clutter from the controller.

Fractal does a lot more than that which will be explored later on, but it covers concerns with transformation perfectly, removes the security, stability and display concerns addressed earlier.

While other languages have great solutions for this already, PHP seemed to be rather lacking for this exact purpose. Some call it "Data Marshalling" or "Nested Serialization", but it is all achieving roughly the same goal: take potentially complicated data from a range of stores and turn it into a consistent output.

- Jbuilder[2] looks fairly slick for the Ruby crowd
- Tweet other suggestions to @philsturgeon

That is the end of theory in this book. We will now be working with code. Open up the Sample Code ZIP file or head to the GitHub repo[3] and extract it somewhere useful.

[1] https://packagist.org/packages/league/fractal
[2] https://github.com/rails/jbuilder
[3] https://github.com/philsturgeon/build-apis-you-wont-hate

```
$ 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.
```

Open your browser and go to `http://localhost:5000/places`, and there is a list of places looking like this:

```
{
  - 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"
        },
```


Fractal default JSON structure

This is a Laravel 4 application but only because it has migrations and seeding and I like it. This is made up of a few bits of PHP that would work in any framework, and the approach works in any language.

- **composer.json** - Added an autoloadable folder using PSR-0 to allow my own code to be loaded
- **app/controllers/ApiController.php** - Insanely simple base controller for wrapping responses
- **app/controllers/PlaceController.php** - Grab some data and pass it to the `ApiController`

Other than defining some basic GET routes in `app/routes.php` that is basically all that is being done.

The `PlaceController` looks like this:

Example of a controller using Fractal to output data

```php
<?php
use App\Transformer\PlaceTransformer;

class PlaceController extends ApiController
{
    public function index()
    {
        $places = Place::take(10)->get();
        return $this->respondWithCollection($places, new PlaceTransformer);
    }

    public function show($id)
    {
        $place = Place::find($id);
        return $this->respondWithItem($place, new PlaceTransformer);
    }
}
```

The "raw data" (happens to be an ORM model but could be anything) is sent back with the appropriate convenience method and a transformer instance is provided too. These `respondWithCollection()` and `respondWithItem()` methods come from ApiController, and their job is just to create Fractal instances without exposing as many classes to interact with.

The `PlaceTransformer` looks like this:

```php
<?php namespace App\Transformer;

use Place;
use League\Fractal\TransformerAbstract;

class PlaceTransformer extends TransformerAbstract
{
    /**
     * Turn this item object into a generic array
     *
```

```
     * @return array
     */
    public function transform(Place $place)
    {
        return [
            'id'           => (int) $place->id,
            'name'         => $place->name,
            'lat'          => (float) $place->lat,
            'lon'          => (float) $place->lon,
            'address1'     => $place->address1,
            'address2'     => $place->address2,
            'city'         => $place->city,
            'state'        => $place->state,
            'zip'          => (float) $place->zip,
            'website'      => $place->website,
            'phone'        => $place->phone,
        ];
    }
}
```

Simple.

The ApiController is kept super simple at this point too:

Simple ApiController for basic responses using Fractal

```
<?php

use League\Fractal\Resource\Collection;
use League\Fractal\Resource\Item;
use League\Fractal\Manager;

class ApiController extends Controller
{
    protected $statusCode = 200;

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

    public function getStatusCode()
    {
        return $this->statusCode;
    }
```

```
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  }
```

The method `respondWithArray()` takes a general array to convert into JSON, which will prove useful with errors. Other than that everything you return will be a Fractal Item, or a Collection.

6.3 Hiding Schema Updates

Schema updates happen, and they can be hard to avoid. If the change in question is simply a renamed field then this is insanely easy to handle:

Before

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

After

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

By changing the right (our internal data structure) and keeping the left the same (the external field name) we maintain control over the stability for the client applications.

Sometimes it is a status change. A new status is added, or the change is fairly drastic and the status all change, but the old API version is still expecting the old one. Maybe someone changed "available" to "active" to be consistent with the other tables, because the original developer was as consistent and logical as a rabid ferrit.

Before

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

After

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

Gross, but useful.

6.4 Outputting Errors

Exactly how to output errors is something I personally am still toying with. The current front-runner is adding convenience methods to the ApiController which handle global routes with a constant as the code and a HTTP error code set, with an optional message in case I want to override the message.

Simple error codes and responses added to ApiController

```
<?php

// ...

class ApiController extends Controller
{
    // ...

    const CODE_WRONG_ARGS = 'GEN-FUBARGS';
    const CODE_NOT_FOUND = 'GEN-LIKETHEWIND';
    const CODE_INTERNAL_ERROR = 'GEN-AAAGGH';
    const CODE_UNAUTHORIZED = 'GEN-MAYBGTFO';
    const CODE_FORBIDDEN = 'GEN-GTFO';

        // ...

    protected function respondWithError($message, $errorCode)
```

```
18      {
19          if ($this->statusCode === 200) {
20              trigger_error(
21                  "You better have a really good reason for erroring on a 200...",
22                  E_USER_WARNING
23              );
24          }
25
26          return $this->respondWithArray([
27              'error' => [
28                  'code' => $errorCode,
29                  'http_code' => $this->statusCode,
30                  'message' => $message,
31              ]
32          ]);
33      }
34
35      /**
36       * Generates a Response with a 403 HTTP header and a given message.
37       *
38       * @return  Response
39       */
40      public function errorForbidden($message = 'Forbidden')
41      {
42          return $this->setStatusCode(403)->responseWithError($message, self::CODE_FORBIDDEN\
43  );
44      }
45
46      /**
47       * Generates a Response with a 500 HTTP header and a given message.
48       *
49       * @return  Response
50       */
51      public function errorInternalError($message = 'Internal Error')
52      {
53          return $this->setStatusCode(500)->responseWithError($message, self::CODE_INTERNAL_\
54  ERROR);
55      }
56
57      /**
58       * Generates a Response with a 404 HTTP header and a given message.
59       *
60       * @return  Response
61       */
62      public function errorNotFound($message = 'Resource Not Found')
```

```
63      {
64          return $this->setStatusCode(404)->responseWithError($message, self::CODE_NOT_FOUND\
65  );
66      }
67
68      /**
69       * Generates a Response with a 401 HTTP header and a given message.
70       *
71       * @return  Response
72       */
73      public function errorUnauthorized($message = 'Unauthorized')
74      {
75          return $this->setStatusCode(401)->responseWithError($message, self::CODE_UNAUTHORI\
76  ZED);
77      }
78
79      /**
80       * Generates a Response with a 400 HTTP header and a given message.
81       *
82       * @return  Response
83       */
84      public function errorWrongArgs($message = 'Wrong Arguments')
85      {
86          return $this->setStatusCode(400)->responseWithError($message, self::CODE_WRONG_ARG\
87  S);
88      }
```

This basically allows for generic error messages to be returned in your controller without having to think too much about the specifics.

Controller using Fractal, combined with a simple error response

```
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)
```

```
    {
        $place = Place::find($id);

        if (! $place) {
                return $this->errorNotFound('Did you just invent an ID and try loading a place? M\
uppet.');
                }

        return $this->respondWithItem($place, new PlaceTransformer);
    }
}
```

Other "Place" specific errors could go directly into the `PlaceController` as methods just like these, with their own constants in the controller, picking a statusCode in the method or relying on one as an argument.

6.5 Testing this Output

You have already seen how to test your endpoints using the Gherkin syntax in Chapter 5: Endpoint Testing, so we can apply that testing logic to this output:

```
Feature: Places

Scenario: Listing places without search criteria is not possible
    When I request "GET /places"
    Then I get a "400" response

Scenario: Finding a specific place
    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: Searching non-existent place
    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: Searching places with filters
    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
```

This is again using the `FeatureContext.php` provided in the sample code, which makes it really easy to test output. We are again assuming that all output is in a `"data"` element, which is either an object (when one resource has been requested) or an array of objects (multiple resources or a collection have been requested).

When you are searching for data you want to ensure that not finding any data doesn't explode. This can be down to your controller processing on output and failing because what should be an array is null, or because some PHP collection class is missing methods, etc. This is why we perform the search with a hardcoded invalid search term, then check that it returns an empty collection:

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

The line `And the "data" property contains 0 items` will cover this. Then we can search for valid terms, knowing that our database seeder has made sure at least one Place has the keyword "cheese" in the name.

Using the line `And scope into the first "data" property` the scope changes to be inside the first data item returned, and the properties can be checked for existence too. If no data, or required fields are missing, this test will fail.

6.6 Homework

Your homework is to take apart the sample application, fit it into your API and try to build valid output for as many of your GET endpoints as possible. Check the data types and make sure the array structure is being output in the way you expect using the test example above.

With valid output covered and basic errors covered, what is next? The most complicated part of API generation, which at some point every developer has to try and work out: embedding/nesting resources, or making “relationships”.

7 Data Relationships

If you've ever worked with relational databases the chances are you understand relationships. Users have comments. Authors have one or many books. Books belong to a Publisher. Southerners have one or more teeth. Whatever the example, relationships are incredibly important to any application and therefore an API too.

RESTful Relationships don't necessarily need to be directly mapped to database relationships. If your database relationships are built properly, RESTful relationships will often be similar, but your RESTful output might have extra dynamic relationships that aren't defined by a JOIN, and might not necessarily include every possible database relationship.

Put more eloquently:

> REST components communicate by transferring a representation of a resource in a format matching one of an evolving set of standard data types, selected dynamically based on the capabilities or desires of the recipient and the nature of the resource. Whether the representation is in the same format as the raw source, or is derived from the source, remains hidden behind the interface. – Roy Fielding[1]

This explanation highlights an important factor: the output has to be based on the "desires of the recipient". There are many popular approaches to designing RESTful relationships, but many of them don't satisfy the "desires of the recipient". Still, I will cover the popular approaches with their pros and cons regardless.

7.1 Sub-Resources

One very simplistic way to approach related data is to offer up new URL's for your API consumers to digest. This was covered lightly in Chapter 2: Planning and Creating Endpoints, and is a perfectly valid approach.

If an API has `places` as a resource and wants to allow access to a place's checkins, an endpoint could be made to handle exactly that:

```
/places/X/checkins
```

The downside here is that it is an extra HTTP request. Imagine an iPhone application that wants to get all `places` in an area and put them on a map, then allow a user to browse through them. If the `place` search happens as one request, then the `/places/X/checkins` is executed each time the user clicks on a place, and they will end up doing a lot of unecessary waiting. This is known as "n+1", meaning your work done is increased by an extra one request for each place you look up.

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

That also assumes the only related data is checkins. At Kapture our API also has `merchant`, `images`, `current_campaign` and `previous_campaigns` to look up. Using "sub-resources" only would mean that 4 extra HTTP requests per place need to happen, which is n+4.

If 50 `places` were returned and each time the related data had to be loaded, assuming the app user looked through all 50 `places` there would be 1 initial request to get 50 results. For each of those results would be 4 more, meaning: `1 + (50 x 4) = 251`. 251 HTTP requests happening (even assuming they are asyncronous) is just unecessary and going over HTTP on a mobile is the slowest things you can do. Even with caching, depending on the data set it could still be 251 requests.

Some API developers try to avoid going over HTTP too many times by shoving as much data as possible into one request, so when you call the `/places` endpoint you automatically get `checkins`, `current_opps`, `merchants` and `images`. Well, if you do not want that information you are waiting for huge file downloads full of irrelevant JSON! Even with GZIP compression enabled on the web-server, downloading something you don't need is obviously not desirable, and can be avoided. This can mean major performance gains on mobile, and minor gains over a slow network or weak Wi-Fi for desktop or tablets.

The trade-off here between "downloading enough data to avoid making the user wait for subsequent loads" and "downloading too much data to make them wait for the initial load". It's hard, but you need the flexibility and making sub-resources your only related data approach does not give the API consumer any flexibility.

7.2 Foreign Key Arrays

Another approach to related data is to provide an array of foreign keys in the output. To use an example from EmberJS[2], if a `post` has multiple `comments`, the `/posts` endpoint could contain the following:

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

This is better. You still end up with n+1 requests, but at least you can take those ID's and make a grouped request like `/comments/1,2` or `/comments?ids=1,2` to reduce how many HTTP requests are being made.

Back to the places example, if you have 50 places returned and need 4 extra pieces of data, you could itterate through the 50, map which items expect which pieces of data, request all unique pieces of data and only end up with `1 + 4 = 5` HTTP requests instead of 251.

The downside is that your API consumer has to stitch all of that data together, which could be a lot of work if it is a large dataset, so it is still not ideal.

[2] http://emberjs.com/guides/models/defining-models/

7.3 Compound Documents (a.k.a Side-Loading)

Instead of just putting the foreign keys into the resource you can optionally side-load the data. I was having a rough time of things trying to word an introduction, so I'll let somebody else do it:

> Compound documents contain multiple collections to allow for side-loading of related objects. Side-loading is desirable when nested representation of related objects would result in potentially expensive repetition. For example, given a list of 50 comments by only 3 authors, a nested representation would include 50 author objects where a side-loaded representation would contain only 3 author objects. **Source: ** https://canvas.instructure.com/doc/api/file.compound_documents.html

I found that by searching for "compound document". I found that term by searching for "REST Side-Loading". I found that after having a horrible time with EmberJS forcing me to use the "side-loading" approach for Ember Data, and they barely explain it themselves.

It looks a little like this:

```
{
  "meta": {"primaryCollection": "comments"},
  "comments": [...],
  "authors": [...]
}
```

The pro suggested in the quote above is: if an embedded piece of data is commonly recuring, you do not have to download the same resource multiple times. The con is that context gets lost in larger data structures and it has the same issue as the "Foreign Key Array": the mapping of data to create an acurate structure is left to the API consumer, and that can be hard work.

7.4 Embedded Documents (a.k.a Nesting)

This is the approach I have been using in the latest two versions of the API at Kapture, and I will continue to use it for the foreseeable future. It offers the most flexibility for the API consumer: meaning it can reduce HTTP requests or reduce download size depending on what the consumer wants.

If a API consumer were to call the URL `/places?embed=checkins,merchant` then they would see checkin and merchant data in the response inside the book resource:

```
{
    "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" : {
                // ...
            }
        }
    ]
}
```

Some systems (like Facebook, or any API using Fractal) will let you nest those embeds with dot notation:

E.g: `/places?embed=checkins,merchant,current_opp.images`

Embedding with Fractal

Picking back up from Chapter 6, your transformer at this point is mainly just giving you a method to handle array conversion from your data source to a simple array. Fractal can however embed resources and collections too. Going back to the book example in Chapter 6, the `BookTransformer` might contain an optional embed for an author.

```php
<?php namespace App\Transformer;

use Book;
use League\Fractal\TransformerAbstract;

class BookTransformer extends TransformerAbstract
{
    /**
     * List of resources possible to embed via this transformer
     *
     * @var array
     */
    protected $availableEmbeds = [
        'author'
    ];

    /**
     * Turn this item object into a generic array
     *
     * @return array
     */
    public function transform(Book $book)
    {
        return [
            'id'    => (int) $book->id,
            'title' => $book->title,
            'year'  => $book->yr,
        ];
    }

    /**
     * Embed Author
     *
     * @return League\Fractal\ItemResource
     */
    public function embedAuthor(Book $book)
    {
        $author = $book->author;

        return $this->item($author, new AuthorTransformer);
    }
}
```

This example happens to be using the lazy-loading functionality of an ORM for `$book->author`, but there is no reason that eager-loading could not also be used by inspecting the `$_GET['embed']` list of requested

scopes. Something like this can easily go in your controller constructor, somewhere in the base controller or... something:

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

// Left is the embed names, right is relationship names.
// avoids exposing relationships and whatnot directly
$possibleRelationships = [
    'publisher' => 'publisher',
    'author' => 'user',
];

// Check for potential ORM relationships, and convert from generic "embed" names
$eagerLoad = array_values(array_intersect($possibleRelationships, $requestedEmbeds));

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

Being a RESTful Rebel

I read a blog article by Ian Bentley[3] suggesting that this approach is not entirely RESTful. It points to a Roy Fielding quote:

> The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface between components (Figure 5-6). By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. Roy Fielding[4]

All of these solutions are - according to somebody - "wrong". There are technical pros and cons and what I refer to as "moral" issues, but those moral issues are just down to how technically RESTful you care about being. The technical benefits that optional embedded relationships provide are so beneficial I do not care about crossing Roy and his RESTful spec to do it.

Make your own choices. Facebook, Twitter and most popular "RESTful API's" fundamentally ignore parts of (or dump all over the entirety of) the RESTful spec, so respecting everything else and popping your toe over the line here a little would not be the biggest travesty for your API.

[3] http://idbentley.com/blog/2013/03/14/should-restful-apis-include-relationships/
[4] http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1

8 Debugging

8.1 Coming Soon

This chapter will hopefully be with you by January 5th, 2014.