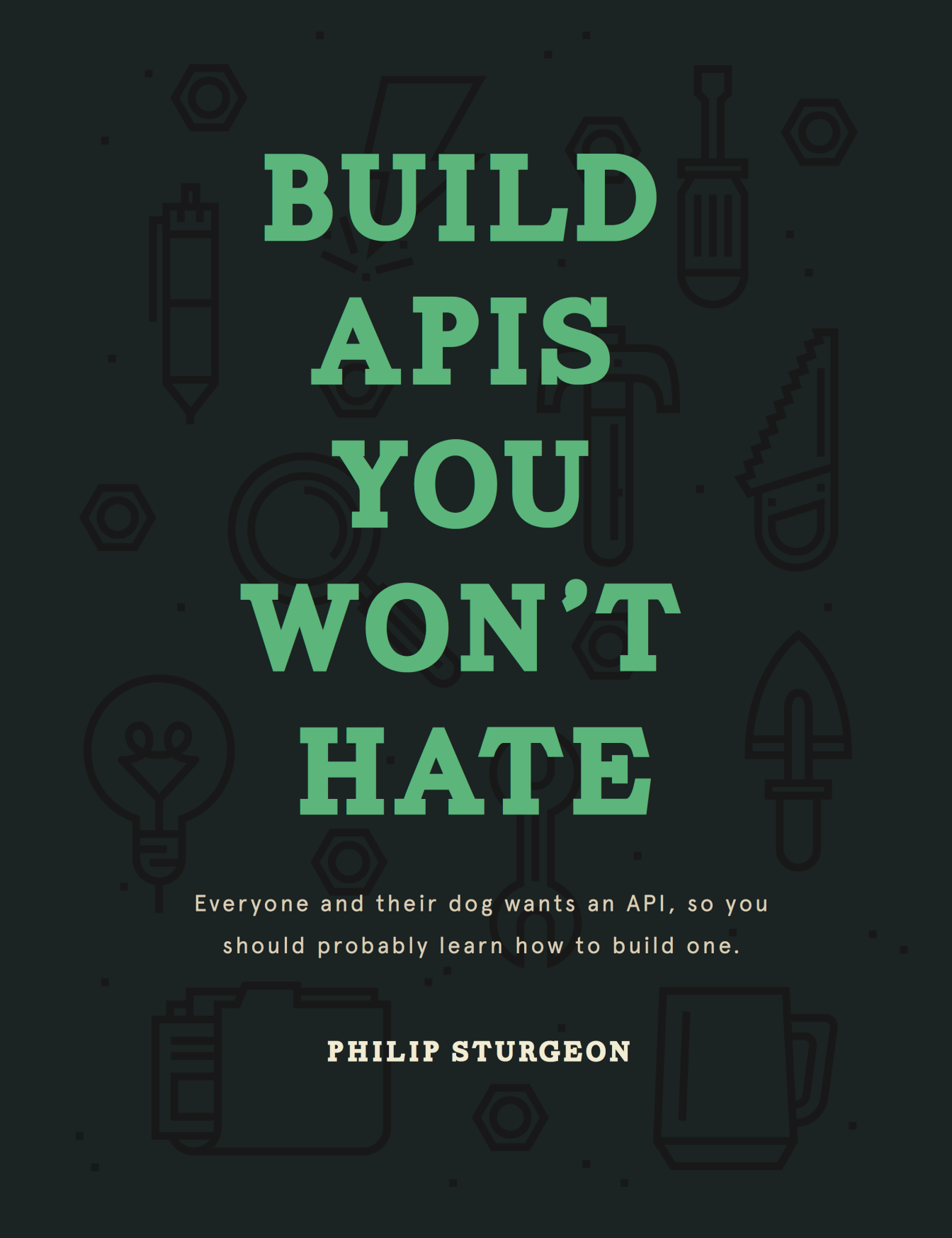
BUILD
APIS
YOU
WON'T
HATE
Everyone and their dog wants an API, so you should probably learn how to build one.
PHILIP STURGEON

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 2016-03-30

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.

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://apisyouwonthate.com

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

A huge thank you to all the developers and other folks who built the technologies this book talks about.

I would also like to thank everyone who bought an early copy of this book on LeanPub. 2014 was a really messed up year for me, and those book sales kept me going, and kept me motivated to finish the book on time.

Without you, I would be much further away from getting my boat.

Contents

1. HATEOAS

1.1 Introduction

HATEOAS is a tricky subject to explain, but it is actually rather simple. It stands for Hypermedia as the Engine of Application State, and is pronounced as either *hat-ee-os*, *hate O-A-S* or *hate-ee-ohs*; the latter of which sounds a little like a cereal for API developers.

However you want to try and say it, it basically means two things for your API:

1. Content negotiation
2. Hypermedia controls

In my experience, content negotiation is one of the first things many API developers implement. When building my CodeIgniter Rest-Server extension, it was the first feature I added, because hey, it is fun! Changing the `Accept` header and seeing the `Content-Type` header in the response switch from JSON to XML or CSV is great, and also super easy to do.

1.2 Content Negotiation

Some self-proclaimed RESTful APIs (Twitter, you are to blame for this) handle content negotiation with file extensions. Their URLs often look like:

- `/statuses/show.json?id=210462857140252672`
- `/statuses/show.xml?id=210462857140252672`

This is a bit of a misuse of the concept of a resource and forces users to know not only that the endpoint show exists, but that they must pick a content type extension and that the `id` parameter must be used.

A good API would simply have `/statuses/210462857140252672`. This has the dual benefit of letting the API respond with a default content type, or respecting the `Accept` header and either outputting the request content type or spitting out a `415` status code if the API does not support it. The second benefit is that the consumer does not need to know about `?id=`.

URIs are not supposed to be a bunch of folders and file names and an API is not a list of JSON files or XML files. They are a list of resources that can be represented in different formats depending on the `Accept` header, **and nothing else**.

A simple example of content negotiation requesting JSON

```
GET /places HTTP/1.1
Host: localhost:8000
Accept: application/json
```

A response would then contain JSON if the API supports JSON as an output format.

A shortened example of the HTTP response with JSON data

```
HTTP/1.1 200 OK
Host: localhost:8000
Connection: close

{
    "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"
        },
        ...
```

```
    ]
}
```

Most popular APIs will support JSON by default, or maybe *only* JSON as our sample app has done so far. This is not realistic, but has been done throughout the book so far, mainly for the sake of simplicity.

XML is still a tricky one to do as you need to require view files, and that is out of scope of this chapter.

YAML, however, is rather easy to achieve, so we can see how content negotiation works with a little change to our app.

Check `~/apisyouwonthate/chapter12/` for the updated sample app.

The main change other than including the Symfony YAML component[1] was to simply update the `respondWithArray()` method to check the `Accept` header and react accordingly.

Updated respondWithArray() method with accept header detection

```
protected function respondWithArray(array $array, array $headers = [])
{
    // You will probably want to do something intelligent with charset if provided.
    // This chapter just ignores everything and takes the main MIME type value

    $mimeParts = (array) explode(';', Input::server('HTTP_ACCEPT'));
    $mimeType = strtolower($mimeParts[0]);

    switch ($mimeType) {
        case 'application/json':
            $contentType = 'application/json';
            $content = json_encode($array);
            break;

        case 'application/x-yaml':
            $contentType = 'application/x-yaml';
            $dumper = new YamlDumper();
            $content = $dumper->dump($array, 2);
            break;

```

[1] http://symfony.com/doc/current/components/yaml/introduction.html

```
        default:
            $contentType = 'application/json';
            $content = json_encode([
                'error' => [
                    'code' => static::CODE_INVALID_MIME_TYPE,
                    'http_code' => 406,
                    'message' => sprintf('Content of type %s is not supported.', $mim\
eType),
                ]
            ]);
    }

    $response = Response::make($content, $this->statusCode, $headers);
    $response->header('Content-Type', $contentType);

    return $response;
}
```

Very basic, but now if we try a different MIME type we can expect a different result:

An HTTP request specifying the preferred response MIME type

```
GET /places HTTP/1.1
Host: localhost:8000
Accept: application/x-yaml
```

The response will be in YAML.

A shortened example of the HTTP response with YAML data

```
HTTP/1.1 200 OK
Host: localhost:8000
Connection: close

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 }
    ...
```

Making these requests programmatically is simple.

Using PHP and the Guzzle package to request a different response type

```
use GuzzleHttp\Client;

$client = new Client(['base_url' => 'http://localhost:8000']);

$response = $client->get('/places', [
        'headers' => ['Accept' => 'application/x-yaml']
]);

$response->getBody(); // YAML, ready to be parsed
```

This is not the end of the conversation for content negotiation as there is more to talk about with vendor-based MIME types for resources, which can also be versioned. To keep this chapter on point, that discussion will happen in Chapter 13: API Versioning.

1.3 Hypermedia Controls

The second part of HATEOAS, however, is drastically underused, and is the last step in making your API technically a RESTful API.

Batman provides a standard response to often futile bucket remark "But it's not RESTful if you..." Credit to Troy Hunt (@troyhunt)

While you often hear complaints like "but that is not RESTful!" from people about silly things, this is one instance where they are completely right. Roy Fielding says that without hypermedia controls an API is not RESTful[2], writing back in 2008. People have been ignoring that ever since, and the last estimate was that 74% of APIs claiming to be "RESTful" do not actually use hypermedia.

[2] http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

RESTful Nirvana

There is something floating around the REST/Hypermedia community called the Richardson Maturity Model[3], written about here by Martin Fowler[4] but originally invented by Leonard Richardson[5]. It covers what he considers to be 'the four levels of REST':

1. **"The Swamp of POX."** You're using HTTP to make RPC calls. HTTP is only really used as a tunnel.
2. **Resources.** Rather than making every call to a service endpoint, you have multiple endpoints that are used to represent resources, and you're talking to them. This is the very beginnings of supporting REST.
3. **HTTP Verbs.** This is the level that something like Rails gives you out of the box: You interact with these Resources using HTTP verbs, rather than always using POST.
4. **Hypermedia Controls.** HATEOAS. You're 100% REST compliant.
   – **Source:** Steve Klabnik, "Haters gonna HATEOAS"[6]

Some dispute this model because, as Roy says, unless you have hypermedia then it is not REST. The model is good as long as you understand that steps 1, 2 and 3 are still "not REST" and step 4 is "REST".

So, what are hypermedia controls? They are just links to other content, relationships, and further actions. These allow a consumer to browse around the API, discovering actions as it goes.

Basically, your data needs to have "hyperlinks", which you have probably been using in your HTML output for years. I said early on in the book that REST is just using the same conventions as the actual Internet, instead of inventing new ones, so it makes sense that linking to other resources should be the same in an API as it is in a web page.

The general underlying theme of hypermedia is that an API should be able to make perfect sense to an API client application and the human looking

[3] http://martinfowler.com/articles/richardsonMaturityModel.html
[4] http://martinfowler.com/
[5] http://www.crummy.com/
[6] http://timelessrepo.com/haters-gonna-hateoas

at the responses, entirely without having to hunt through documentation to work out what is going on.

Small HATEOAS concepts have been sneakily sprinkled throughout this book, from suggesting error codes be combined with human readable error messages and documentation links, to helping the client application avoid maths when interacting with pagination. The underlying theme is always to make controls such as next, previous (or any other sort of related interaction) clearly obvious to either a human or a computer.

Understanding Hypermedia Controls

This is the easiest part of building a RESTful API, so I am going to try really hard not to leave this section at "just add links mate" (my normal advice for anyone asking about HATEOAS).

Our usual data is output in such a way that only represents one or more resources. By itself, this one piece of data is an island, completely cut off from the rest of the API. The only way to continue interacting with the API is for the developer to read the documentation and understand what data can be related, and to discover where that data might live. This is far from ideal.

To tie one `place` to the related resources, subresources or collections is easy.

```
{
    "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",
        "links": [
                {
```

```
16                          "rel": "self",
17                          "uri": "/places/2"
18                  },
19                  {
20                          "rel": "place.checkins",
21                          "uri": "/places/2/checkins"
22                  },
23                  {
24                          "rel": "place.image",
25                          "uri": "/places/2/image"
26                  }
27          ]
28      }
29  }
```

Here are three simple entries, with the first linking to itself. They all contain a `uri` (Universal Resource Indicator) and a `rel` (Relationship).

URI vs. URL

The acronym "URI" is often used to refer to only content after the protocol, hostname and port (meaning URI is the path, extension and query string), whilst "URL" is used to describe the full address. While this is not strictly true, it is perpetuated by many software projects such as CodeIgniter. Wikipedia[7] and the W3[8] say a bunch of conflicting things, but I feel like a URI is easily described as being simply any sort of identifier for the location of a resource on the Internet.

A URI can be partial, or absolute. URL is considered by some to be a completely non-existent term, but this book uses URL to describe an absolute URI, which is what you see in the address bar. Rightly or wrongly. Got it?

Some people scoff at the `self` relationship suggesting that it is pointless. While you certainly know what URL you just called, that URL is not always going to match up with the `self` URI. For example, if you just created a `place` resource, you will have called `POST /places`, and that is not what you would want to call again to get updated information on the same

[7] http://wikipedia.org/wiki/Uniform_Resource_Identifier
[8] http://www.w3.org/TR/uri-clarification/

resource. Regardless of the context, outputting a `place` always needs to have a `self` relationship, and that `self` should not just output whatever is in the address bar. Basically put, the `self` relationship points to where the resource lives, not the current address.

As for the other `rel` items, they are links to subresources that contain related information. The content of the tags can be anything you like, just keep it consistent throughout. The convention used in this example is to namespace relationships so that they are unique. Two different types of resources could have a `checkins` relationship (eg: `users` and `places`), so keeping them unique could be of benefit for the sake of documentation at least. Maybe you would prefer to remove the namespace, but that is up to you.

Those custom relationships have fairly unique names, but for more generic relationships you can consider using the Registry of Link Relations[9] defined by the IANA, which is used by Atom (RFC 4287[10]) and plenty of other things.

Creating Hypermedia Controls

This is literally a case of shoving some links into your data output. However you chose to do that, it can be part of your "transformation" or "presentation" layer.

If you are using the PHP component Fractal - which has been used as an example throughout the book - then you can simply do the following:

PlaceTransformer with links included in the response data.

```
1  public function transform(Place $place)
2  {
3      return [
4          'id'           => (int) $place->id,
5          'name'         => $place->name,
6          'lat'          => (float) $place->lat,
7          'lon'          => (float) $place->lon,
8          'address1'     => $place->address1,
9          'address2'     => $place->address2,
```

[9] http://www.iana.org/assignments/link-relations/link-relations.xhtml
[10] http://atompub.org/rfc4287.html

```
        'city'          => $place->city,
        'state'         => $place->state,
        'zip'           => $place->zip,
        'website'       => $place->website,
        'phone'         => $place->phone,

        'links'         => [
            [
                'rel' => 'self',
                'uri' => '/places/'.$place->id,
            ],
            [
                'rel' => 'place.checkins',
                'uri' => '/places/'.$place->id.'/checkins',
            ],
            [
                'rel' => 'place.image',
                'uri' => '/places/'.$place->id.'/image',
            ]
        ],
    ];
}
```

People try to get smarter and have various relationships based on their `$_SERVER` settings or based on their ORM relationships, but all of that is just going to cause you problems. If you have these transformers then you only need to write this lot out once. This then avoids exposing any database logic and keeps your code readable and understandable.

Once you have input these links, other people need to know how to interact with them. You might think, "surely I should put `GET` or `PUT` in there so people know what to do". Wrong. They are links to resources, not actions. An image exists for a place, and we can either blindly assume we can make certain actions on it, or we can ask our API what actions are available and cache the result.

Discovering Resources Programmatically

Taking a shortened example from earlier on in this chapter, we can expect to see output like this:

```
{
    "data": {
        ...
        "links": [
            {
                "rel": "self",
                "uri": "/places/2"
            },
            {
                "rel": "place.checkins",
                "uri": "/places/2/checkins"
            },
            {
                "rel": "place.image",
                "uri": "/places/2/image"
            }
        }
    }
}
```

We can assume that a `GET` will work on both the `self` and the `place.checkins` endpoints, but what else can we do with them? Beyond that, what on Earth do we do with the `place.image` endpoint?

HTTP has us covered here with a simple and effective verb that has so far not been discussed: `OPTIONS`.

An HTTP request using the OPTIONS verb

```
OPTIONS /places/2/checkins HTTP/1.1
Host: localhost:8000
```

The response to the previous HTTP request

```
HTTP/1.1 200 OK
Host: localhost:8000
Connection: close
Allow: GET,HEAD,POST
```

By inspecting the `Allow` header, we as humans (or programmatically as an API client application), can work out what options are available to us on

any given endpoint. This is what JavaScript is often doing in your browser for AJAX requests and you might not even know it.

Doing this programmatically is pretty easy too, and most HTTP clients in any given language will let you make an `OPTIONS` call just as easily as making a `GET` or `POST` call. If your HTTP client does not let you do this, then change your HTTP client.

Making an OPTIONS HTTP request using PHP and the Guzzle package

```
use GuzzleHttp\Client;

$client = new Client(['base_url' => 'http://localhost:8000']);
$response = $client->options('/places/2/checkins');
$methods = array_walk('trim', explode(',', $response->getHeader('Accept'));
var_dump($methods); // Outputs: ['GET', 'HEAD', 'POST']
```

So in this instance, we know that we can get a list of check-ins for a place using `GET` and we can add to them by making a `POST` HTTP request to that URL. We can also do a `HEAD` check, which is the same as a `GET` but skips the HTTP body. You will probably need to handle this differently in your application, but this is handy for checking if a resource or collection exists without having to download the entire body content (i.e: just look for a `200` or a `404`).

It might seem a little nuts to take this extra step to interact with an API, but really it should be considered much easier than hunting for documentation. Think about it: trying to find that little "Developers" link on the website, then navigating to the documentation for the correct API (because they are so cool they have about three), then wondering if you have the right version... not fun. Compare that to a programmatically self-documenting API, which can grow, change and expand over time, rename URLs and... well that is a real win. Trust me.

If you know that an API follows RESTful principles then you *should* be confident that it follows HATEOAS because advertising it as RESTful without following HATEOAS is a big stinking lie. Sadly, most of the popular APIs out there are big stinking liars.

> GitHub responds with a 500, Reddit with 501 Not Implemented, Google maps with 405 Method Not Allowed. You get the idea.

> I've tried many others, and the results are usually similar. Sometimes it yields something identical to a GET response. None of these are right.
> – **Source:** Zac Stewart, "The HTTP OPTIONS method and potential for self-describing RESTful APIs"[11]

If you are building your own API, then you can easily do this yourself and your clients know that you know how to build a decent API.

And that, is about all there is for HATEOAS. You should now know enough to go out and build up an API that in theory you won't hate. Sadly, you will probably need to build a new version within a few months regardless, so for that we will now take a look at API versioning.

[11] http://zacstewart.com/2012/04/14/http-options-method.html