Building a Decent API

PHP developers are increasingly moving over to API development, as are a lot of server-side developers. It's a trend that's been happening for the last few years and it's getting to the point where everyone and their dog are putting articles showing off how to build "awesome" API's. Unfortunately most of these are either woefully inadequate or are promoting bad practices.

I'm not going to link to any bad examples because that's just rude, but here are some golden rules that I stick to when building out API's.

Never Expose DB Results Directly

1. If you rename a field, then your users are fucked. Convert with a hardcoded array structure.
2. Most DB drivers [for PHP] will show integers as numeric strings and `false` as `"0"`, so you want to typecast as much of your output array as possible.
3. Unless you're using an ORM with "hidden" functionality, people will see passwords, salts and all sorts of fancy codes. If you add one and forget to put it in your `$hidden` array then OOPS! Manually declare your output, do NOT just return Users::all();

Use the URI sparingly, and correctly

1. Use the query string for paired params instead of `/users/id/5/active/true`. *Your API does not need to be SEO optimised.*
2. `?format=xml` is stupid, use an `Accept: application/xml` header. I added this to the CodeIgniter Rest-Server once for lazy people, and now people think it's a thing. It's not.
3. Make all of your resources plural. `/user/X` might make sense initially, but when you get a word like "opportunity" it gets weird fast gets. `/opportunities` and `/opportunity/X` is a pain.

Resources are EVERYTHING

1. You're always either asking for one resource, or multiple. If it's one, just return that data as an array. If you've asked for multiple, then return them in a plural parent element (such as "users").
2. Two resources in different locations should look identical (your iPhone dev will love you for this). This could be `/me/friends` and `/users/5/friends`, or embedded data.
3. If I want multiple resources in one call, then give them to me. `/users/X,Y,Z` in a `"users"` array works nicely.
4. If you ask for multiple and some results exist, shove that in a "users" array.
5. If you ask for multiple but find none, then a 404 makes sense.

JSON, XML or shut up

1. Don't spend forever trying to make your system output everything under the sun. Sure you can output lolcat, but you don't need to.
2. Tell your users to send you JSON or XML in the body. Messing around with `application/x-www-form-urlencoded` and its flat key/value formatting just to get at data with `$_POST['foo']` is

silly, especially when any decent framework (like Laravel 4) will allow `Input::json('foo')` anyway.

3. No payload parameters. I've seen APIs accept `application/x-www-form-urlencoded` with a `json={}` parameter. If you think that is a good idea it's time you re-train as a yoga teacher or something, the stress is effecting your judgement.

Authentication

1. OAuth 2 is the shit. A few people wrote ranty arguments about how it's insecure, because they weren't using SSL. Or because X company implemented it badly. Don't implement it badly.
2. Unless your API is firewalled off from the outside internets use SSL.
3. Make sure your OAuth 2 implementation is spec compliant, or you're going to have a bad time. This one is.

Caching

1. Your API needs a shorter memory than my favorite fruit is watermelon. No state, no sessions, no IP recognition. Don't guess, let them tell you who they are with their access token.
2. Caching should only happen on popular endpoints where there is no user context, and you know it's not going to change in the timeframe.
3. The `Cache-Control` header let's people know if they can (or should) cache stuff. If other devs ignore those headers then it's their problem.

Background all the things

1. "When the user sends us an image, resize it, upload it to S3, send an email then show a confirmation". Nope. That should be at least one background job, preferably 3, with an IMMEDIATE response. Ditch off to one "image_processing" job to get things going.
2. Create an "email" job, a "sms" job and a "APN" job for example, so you can generically send out all sorts of contact stuff from your API, your other jobs, etc without infecting your API with all that stuff. I can switch from Twilio to… well I don't know anyone better, but I could do it easy as hell by updating the SMS job.
3. Put your workers on a different server. Your API server is busy enough handling HTTP requests in responses, don't make it think about anything else.

Pagination

1. Do this. So many people just dump back an "get all this data" response and forget that N gets pretty big over time.
2. Add a `"paging"` element, which has a `"next"` or `"previous"` URL if there are more on either side.
3. Don't make the client do math.
4. OUTPUT TOTALS. I'm looking at you Facebook. Why do I have to poll "next" 20 times to manually count($response['data']) to see how many friends a specific user has when you obviously know the answer already? GAH!

Response Codes

1. Give me an actual error message, not just a code. "Oh yay, a E40255 just happened" –Nobody.
2. Use 410 instead of 404 if it's been deleted, or blocked, or rejected. "This content has been removed." is more useful than "Um, nope."

Documentation

1. If you have well written documentation, you get to say RTFM a lot.
2. Versioned API's are easiest to keep up to date, because they don't change (unlike the Facebook API, which might as well be nightly builds from develop).
3. Use a tool. Swagger-PHP + Swagger-UI does the trick.

Testing

1. Write tests (Behat is great at this), and get Jenkins to automate them.
2. If tests aren't automated, they might as well not exist.
3. Unit-test your components and models and whatnot.
4. No need to mock the DB, make a "testing" server and beat it hard, just mock FB/Twitter/"Foo-External API" calls.
5. TEST FOR ERRORS, not just success. Try and trigger every guard statement.
6. If tests aren't automated, they might as well not exist. *Nope, not a copy/paste error.*

Version your API like an adult

1. Sure throwing a `v1/` subfolder into your `app/controllers` or whatever might seem like a real clever way of doing things, but how are you gonna merge v1.0 tweaks with v1.1 and v2.0?
2. If you're going to use v1 then make sure its a different codebase, don't make one app do all versions ever because IT WONT WORK AND STOP TRYING WHY DO YOU KEEP TRYING STOP IT!
3. Nginx comes with a handy Map module, map custom Accept headers to a variable.
4. Those headers can look like this: `application/vnd.com.example-v1.0+json`. Gross? Whatever.
5. Use this variable to map to a different directory in your virtual host config, and these each have their own Git branch. Something like: "set $api_path api-v$api_version;" followed by a "root /var/www/$api_path/public;"
6. Merge changes upwards to as many codebases that share a common history, don't try and copy and paste changes like a dummy.
7. All rules in this section mean 1.0 could be PHP, 2.0 could be Node (you hipster you) and 3.0 could be Scala (I don't even…) and only your minor versions need to worry about merging changes upwards.

This message has been brought to you with the help of Dos Equis, and a little Scotch.

Build your API's better, because now you have no excuse.

Previous: Geeks Giving for Aids

Written by

Phil Sturgeon

Next: Beware the Route to Evil

Phil has contributed to CodeIgniter, FuelPHP, Laravel and handfuls of other projects, to try and make the PHP community a better place.

Published 12 Jul 2013

Comments for this thread are now closed. ×

20 Comments Phil Sturgeon's Blog 1 Login

Recommend Share Sort by Best

Rufus • 2 years ago
The advice on long-running jobs sounds sensible, but how do you deal with reporting progress and/or status to the user? Does the client poll for the status of the background job (and if so, isn't this similar to having sessions)?
Thanks for the article!
1 | • Share ›

Sprain • 2 years ago
@Adrian
You manage this by not actually deleting an intem but by setting a flag or timestamp. Of course you'll have to consider this by reading items as you probably will want to skip the ones flagged as deleted.
| • Share ›

Ran • 2 years ago
@Adrian Instead of actually deleting entries from the db, you can add a column "deleted" and set it to true on delete.
| • Share ›

Glen Scott • 2 years ago
Great write up, Phil.

I have a slight issue with this, though:

"Tell your users to send you JSON or XML in the body. Messing around with application/x-www-form-urlencoded and its flat key/value formatting just to get at data with $_POST['foo'] is silly, especially when any decent framework (like Laravel 4) will allow Input::json('foo') anyway."

I still prefer to build apps from the ground up -- I guess you would call it progressive enhancement -- so that forms generally work with no JS at all. For example, a user registration form would POST to a server side script that would redirect on success.

What would you suggest are the best practices for this kind of approach when you also want a JS client to be able to utilise the same functionality?
| • Share ›

Jason Judge • 2 years ago
There are arguments for and against the delivering of total counts when paging. Google only provides approximations. You can understand why, but it still feels wrong to me.

Error messages and response codes are a pain with many APIs. You need something to give to the end user, need something to log for diagnosis and need something to take an appropriate action on. Usually you need all three, and the requirements of each are very appropriate action on. Usually you need all three, and the requirements of each are very different.

An error code won't get munged in different languages or when the wording is changed to make more sense. An end user message should tell him, "the username is too long, try a shorter one". You may want to log, "field foobar does not match expression /[a-z]+/ (value: 'Dude...')".

Also the ability to log multiple errors can be very useful. I'm having fun with a UK CC payment gateway at the moment, because it will only report one validation error at a time, and does not point the error message at any particular field for helping the end user.

-- Jason

PS Hopefully see you at PHPNE13?

• Share ›

Adrian • 2 years ago

Forgive an inexperienced newb, but how do you track things in the db that were deleted vs never there?

• Share ›

Zack Kitzmiller • 2 years ago

@Adrian, don't actually every delete things. A deleted_at timestamp field with a default of NULL is incredibly handy for things like that.

• Share ›

Aken Roberts • 2 years ago

@Adrian: Soft delete. Have a flag column for if an entry should be marked as "deleted" or not.

• Share ›

Ryan • 2 years ago

First off, great article on API building!

Daksh, Modular codebases is an architectural pattern and a general good practice. i.e. Your controllers for your marketing site shouldn't be mucking about in the same folder structure and namespace as your business logic (when you scale you want to be able to tell a monkey, grab that folder and put it in the same framework and the marketing site is live now on a separate install).

APIs are information gateways for things things outside of your server to get data (usually securely through authorization). Common use cases are ajax calls for Javascripts (as simple as client-side validation all the way to full featured single page apps), Desktop client applications, or even mobile apps.

Of course it depends on your exact usecase, but in my experience, APIs for web apps or mobile apps are more often than not better off developed in a framework rather than a CMS. You want things as slim and quick as possible and while CMSs have their uses, you may be ripping out more than you are using to create an API. However, for something like a news app with an accompanying traditional served web experience (or in sharing info with an existing CMS), working with API tools within that CMS may work.

Share ›

Jono · 2 years ago

When getting multiple results, include two root elements: meta contains total count and pagination data, and records contains the items. You kinda mentioned this, but somewhat contradictory. Also, how do you handle errors on post and put?

Share ›

Sergey · 2 years ago

Great article, Phil.
May I suggest using tags (git) to make proper versioning, that way you can always get back to whatever you had knowing a simple tag name such as v0.3.0. Tags support annotations. Making a tag doesn't require making a new branch and doesn't create any branch pollution in your repository but as you are always working on one repo, you may use features and/or bugfixes from more recent versions by simply applying a single commit or whatever, aka backporting stuff. Acts more as milestones but that is what actually it is.

Another thing that came to my mind is that resources like /users/X,Y,Z may in some cases contain too much data, so I usually tend to send an array to something like a /users/multiple, it also helps processing in a more traditional manner.

Share ›

Daksh Mehta · 2 years ago

Hey phill,

Great article. However, can you explain me little when API development is good v.s. standard module development, specially with PyroCMS.

I think I will be needing to use API development soon with web + mobile app with PyroCMS

Thanks,
Daksh

Share ›

Grails · 2 years ago

My former boss designed our grids not to use pagination. What they do is keep the connection and cursor at the backend, such that when you need more records, no new query is executed. Instead, it just fetches the additional rows from the cursor.

Share ›

Ivo · 2 years ago

And look at jsonapi.org to standardize your responses. If Apis follow common patterns, they are easier to work with.

Share ›

Christian · 2 years ago

Expanding on pagination, you may want to discuss HATEOAS and its actual application; a link to roy fielding's rest principles also doesnt hurt. As for backgrounding shit, SRAW (synchronous reads asynchronous writes) is a good rule to follow.

(awesome video but on the long side)
https://blog.apigee.com/detail...

(summary of fielding's principles - ie, not the heavy phd dissertation)
https://www.servage.net/blog/2...

Share ›

Muhammad Faisal · 2 years ago

Great article Phil,

I was quite a help. I am creating an API these days and I used OAuth2 for authentication, but some anti OAuth2 articles created confusion in my mind about OAuth2. After reading this article I am feeling confident again :). Thanks Phil for clearing doubts about OAuth2.

Share ›

Phil Sturgeon · 2 years ago

Sergey: Absolutely, git-flow is just implied in any article I write at this point.

Glen Scott: Make a function that checks the HTTP Content-Type, and json decodes php://input, and you've got pretty much the same functionality as Input::json(). It's not rocket science, and you know that because I wrote the feature in Laravel 4.

Jason Judge: If a proprietary code is in the response then thats wonderful, as long as its not the only response and it well documented, and it's not in a 200. HTTP rules should give hints, or make it pretty much obvious, then adding proprietary codes and textual error messages is just a handy error. Those outputs are always going to be language specific if you're using language files and are respecting the Accept-Language header.

Share ›

Alex Bilbie · 2 years ago

@adrian never delete them, for every entity you have in the database have a "deleted" bool. If it isn't there then show 404, otherwise show 410 if deleted

Share ›

Steve · 2 years ago

This post makes me happy... especially the bit about running tasks in the background AND on a separate box. This should be required reading for anyone building an API.

Share ›

Matason · 2 years ago

Hey Phil, thanks for posting this, very helpful sound advice! I wondered if you'd mind sharing about the techniques you use for the backgrounding you mentioned sometime? I'm interesting in that and how you might handle long running processes.

Thanks again!

• Share ›