Best practices to make your job more RESTful

by Bartek Pacia

In this article I would like to share 5 great techniques regarding REST API design. Before delving into them, I would implore you to refresh the fundamentals.
I'm assuming that you have some basic knowledge about REST APIs – i.e you know the 5 basic rules. If you aren't familiar with what REST is or don't feel comfortable when you hear this term, I would recommend reading a few articles[1] about it and then coming back to mine.

Table of Contents

A quick introduction to REST

If you code, then there's no chance you haven't downloaded data from some API (unless it was Scratch). You might even not know about it, but it was there, serving you obediently. If you are like me, at some point you noticed the recurring pattern in the URLs you were visiting and started to experiment with them. When reading a blog, you altered `posts/45` to `posts/42` and zonk! - another post got loaded.

APIs have become an inseparable part of everyday workflow of millions of software developers around the world. If something is used by so many people, a set of rules has to exist, so APIs built by those millions can communicate with each other. One of such rulesets is REST - an acronym for REpresentational State Transfer. It was proposed by Roy Fielding in his famous dissertation[2].

In the last decade REST, thanks to its simplicity and flexibility, has taken over the world by storm. Simplicity? Yes, REST consists of only 5 rules (in theory of 6, but the last one is rarely used), namely:

---

[1] Codecademy has an excellent article – https://www.codecademy.com/articles/what-is-rest
[2] Roy Fielding, Architectural Styles and the Design of Network-based Software Architectures https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

- Uniform interface
- Client–server
- Stateless
- Cacheable
- Layered system

The 5, rather broad, rules listed above a strong theoretical base that we can use to build our application using our technology of choice.
However, more often than not, APIs built by us are complex and these 5 rules just don't provide enough information on some important details.
You may ask – "Hey, REST has been around for almost 20 years, its specification *must* have been improved and adapted to needs of modern software development".
Well, it turns out that it's not the case – what Fielding described in his dissertation remains unchanged and is the only official resource about REST. What's more, the standard does not answer many of the problems we encounter when designing and implementing APIs. The best we can do in such case is to come up with our own guidelines that we will stick to while building our system. We can also follow some widely-used practices.
Let me digress a bit - this ambiguity and the lack of a single correct answer to many problems reminds me of the situation with HTTP response codes - except for a few basic cases where we return most common statuses (like 400..404 or 500...or 418), there's a never-ending debate over *which code should the program return in this specific case.*

---

5 Best REST Practices

1. Use plural nouns instead of verbs!

This simple rule refers to designing URLs. Now it's time to refer back to your theoretical knowledge about resources in REST. If you don't feel like googling, here's an synopsis - resources are data of any kind. Most often it's a JSON object, but it can be as well a photo, text document or some binary gibberish.
Consider the following HTTP request.
`GET my-api/users/getUser/2`

This GET request is accessing resource user of id 2 in the `usersCollection`.
It might seem that there is nothing wrong with this particular approach at the first glance of an unexperienced programmer. Now, recall what resources are: objects, files, *data*. In other words, resources are static entities, which cannot do anything on their own.

createUser doesn't sound like something static. It's a verb, it's an imperative sentence saying what to do. Now...wait, what? Data performing action? If I were you, I'd be scared of that data. Jokes aside, put short, this endpoint should be simply named GET my-api/users/2. Generally, it's considered a good practice to use plural forms of nouns.
One could also say that badly-written endpoint example has one more flaw: it is redundant. Why hard-code getUsers when we have HTTP methods? Well, this leads us to second good practice, which is…

2. Take advantage of HTTP methods

If resources are static and cannot perform any action, how can perform some modifying or destructive operation on them? Well-known quartet is the answer - GET, POST, PUT, DELETE. Instead of writing "bad" endpoints with verbs in their URL, we can simply include information about an action we want to perform in the request. This way, we avoid data duplication and we use HTTP methods for their intended purpose.
GET and DELETE are pretty self-explanatory, but I've seen some people (including me few years back) struggling to understand the difference between POST and PUT.

POST	PUT
Used to create new resource	Used to update existing *resource*

In my opinion, POST should be renamed to CREATE and PUT to UPDATE so that their nature would be more obvious. But considering it's been like that for almost 30 years, we'd better get used to the current nomenclature.

3. Plan ahead – version your API

No software is able to stand the test of time, and APIs are no different. That's why you should think ahead and give yourself a possibility to have multiple versions of the API working simultaneously. Think of a simple example: Let's suppose you're building a new social-like website with your team. Your API for performing operations on likes of a post looks like website.com/api/posts/<post_id>/likes. After a few months, upon users' suggestions, you decide you want "like object" to store more data – for example, you want to save the date when the like was created. You start coding and you realize you have a serious problem – if you deploy updated code, there's high chance that everybody who's still using old clients (for example, mobile apps) will encounter strange and unexpected behavior. Why is that?
In the original API version, like model consisted of only 2 properties: user_id and post_id. Then you added creation_date property to the model of like model and happily pushed new version to hosting service.

	original API	updated API
expected response body:	{ user_id, post_id }	{ user_id, post_id, creation_date }

But did you think about the users of your mobile app? Not all of them are going to update the client app immediately after you make a new version available on Google Play Store. Some may even use old version that were deprecated a long time ago. You can never be that no one someone is using them

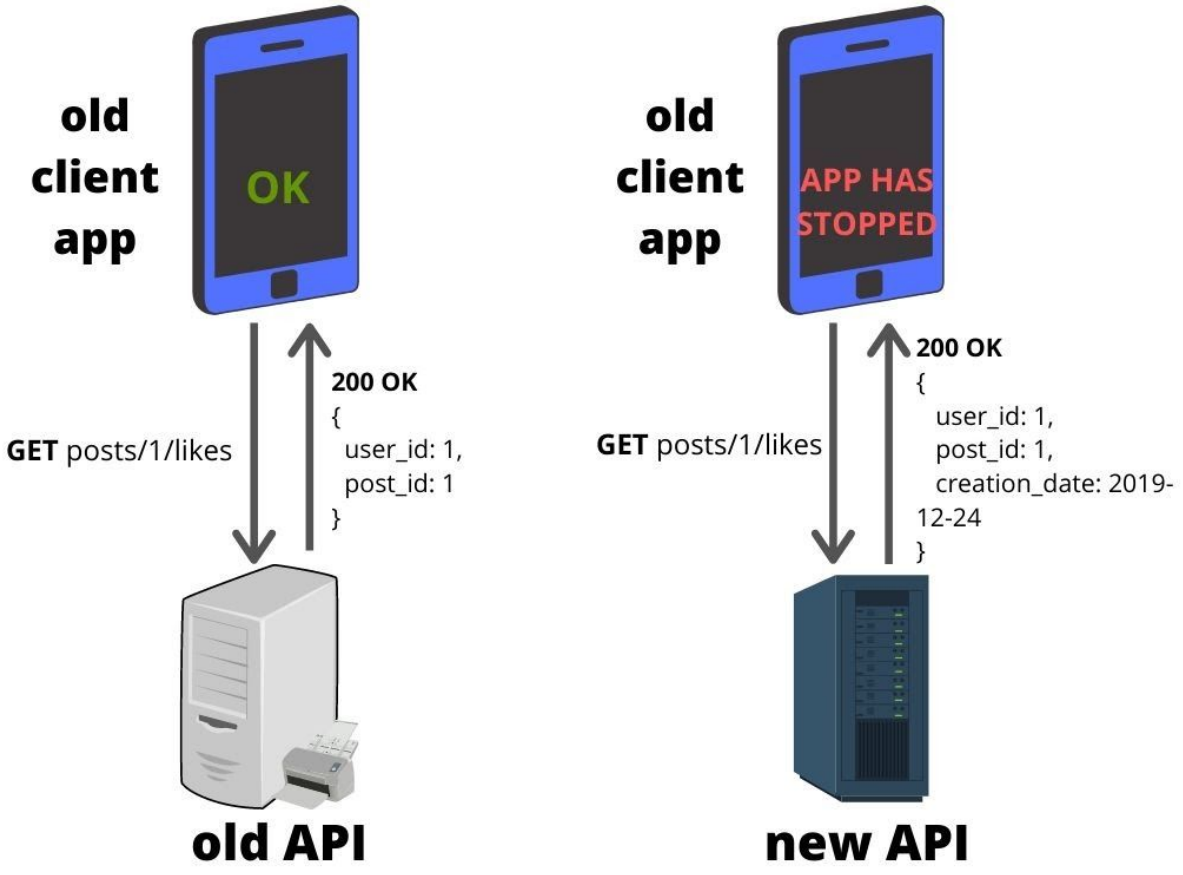

Figure 3.1 Unversioned API system

What happens when an old client app tries to GET likes for a particular post from the updated API? It expects to see two properties - `user_id` and `post_id`. But your API gives it three properties and it is highly possible that a crash will occur.

What is the solution to this problem? The answer is simpler than you think: include version info in the URL of your API. How would it look like? `website.com/api/v1/posts/<post_id>/likes` and `website.com/api/v2/posts/<post_id>/likes`

Old client app will continue to use API version it was designed for, and you will be able to develop and deploy bleeding-edge features without worrying about backwards compatibility.

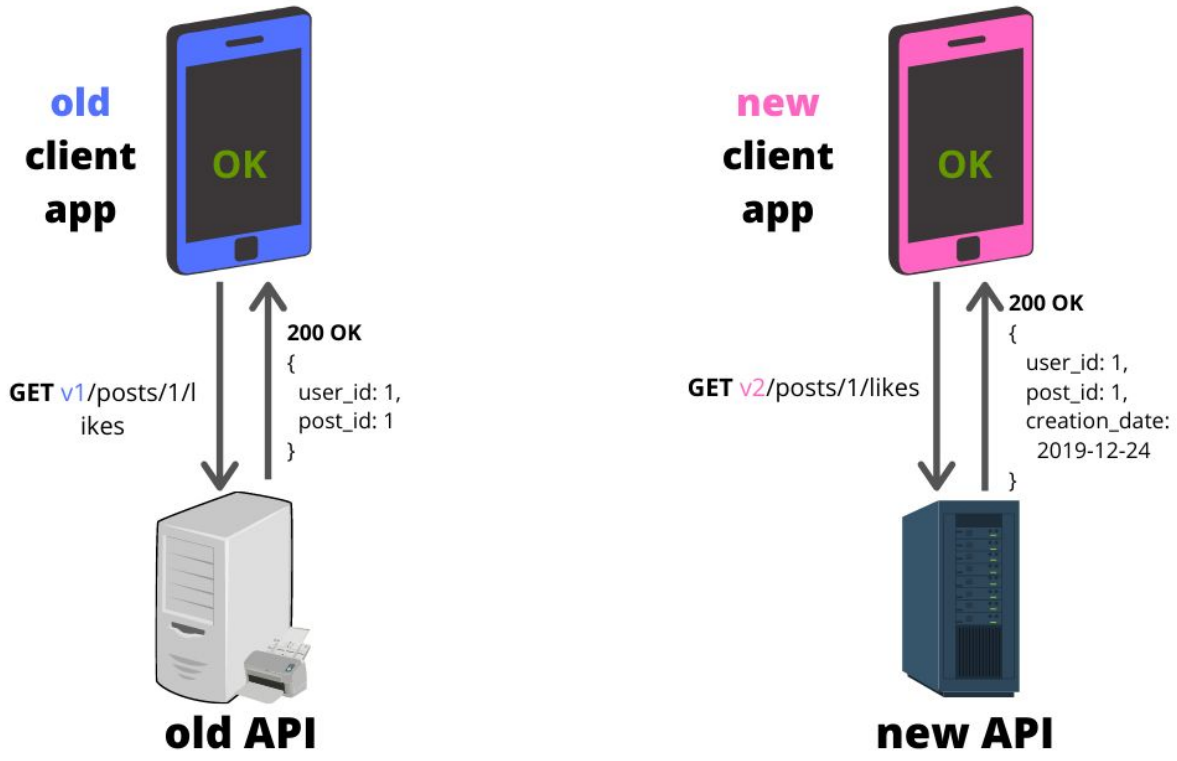

Figure 3.2 Versioned API system

4. Build for people, not for databases

When planning REST endpoints, I often feel tempted to design it the way so that it reflects the database structure. For us, devs who also built the database, creating endpoints that way may seem obvious. Consider the following database structure.

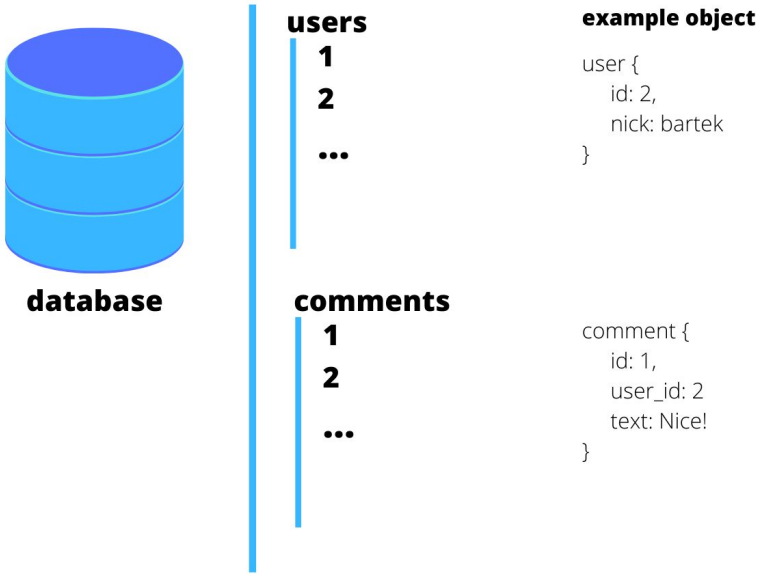

Figure 4.1 Example of database structure

Now think of a developer who wants to get nick of the user who posted comment with the id of 1. What HTTP request would he find the most intuitive? api/comments/1/user, of course. But it wouldn't work if we structured our API exactly the same way as we structured our database. That's because user model doesn't have 'user' property. That aforementioned, poor developer would have to make 2 requests: first to get a whole first comment resource, grab one single id property from there and make another request to users collection, passing id as the parameter. That's how it would look like.
GET api/comments/1 -> extract user_id value
GET api/users/2 -> extract nick value

Aside from this approach being sub-optimal (we're downloading a whole comment object in order to get id), it's also counter-intuitive.
What's more, REST specification says nothing about the storage system underpinning the API, so you shouldn't have any doubts whether to make your API feel fluent and easy to use.
Okay, but how could it be implemented?
When endpoint api/comments/1/user is hit, I'd also download whole user object from the database, parse it and return whole user object, but all this stuff would happen on the server. As a result we have nice API, and as a bonus we aren't contributing to our users' large bills for mobile data usage.

5. Use HTTP status code wisely

Alternative title: No, 500 isn't a cure for everything!
HTTP statuses, like HTTP methods, exist for a reason – for you to use them! Most webservices I've worked with return only few most common status code like 400 and 404.
I admit, choosing *the* appropriate response code can be quite challenging*, but it the long run it's going to save you a lot of headache while debugging.

*If you have ever had to make a choice between 401 and 403, you'll know what I mean.

Good practices that Systers mentorship-backend encorporates

Layered system

Mentorship-backend is a great example of well-designed, well-architected backend. The structure is nice and clean. When I first looked at the it, the amount of directories seemed unnecessary and files, but as I were coding more and more and getting to know it, I appreciated its logical division into layers.
I would really like my API, which I've created a few months back for my private project, to have such a clear structure - after the contest I will definitely refactor it.

Excellent documentation

If you're the one and only user of the API powering some side-project, then it's

---

What would I improve in Systers mentorship-backend

1. General pagination support

Pagination is one of the things that beginners rarely think of when creating backend systems and most of the time, they'll never experience issues caused by its lack. Ultimately, do you really think that your n-th instagram clone will ever generate enough traffic to bottleneck even the cheapest hosting solutions (like basic dynos on Heroku)?
This situation applies to mentorship-backend – it's currently in the development phase. None of the endpoints support pagination and as of now, it isn't really a problem, but imagine what would happen if we had 5000 users?
First actions aiming to change this has been already taken – namely issue #279[3] in mentorship-backend repository. However, it only asks for implementation of paging for `/users` endpoint, but it's a step in the right direction.

2. Search and filtering support

Search is such a common feature and useful that I was very surprised mentorship-backend hasn't got it. Do I have to say that having to download whole (non-paged!) collection and then do filtering on-device isn't the most optimal solution?

3. Proper API versioning

The importance of supporting multiple versions of the API simultaneously has been described earlier, so I'm not going to repeat myself. Again, while we're on development stage, it's not a problem, but we must think about the future if we take this project seriously.
I'd suggest leaving current API "as-is" to maintain backwards compatibility (so we won't "throw the baby out with the bathwater") and begin versioning with the `v1`.

---

[3] Issue # 279 https://github.com/systers/mentorship-backend/issues/279

Okay, so that'd be it! I hope you liked my 5 tips and actually learned something. Now open your IDE and go build delightful REST APIs which will bring joy to lives of developers using them!

Resources used:

https://code-maze.com/top-rest-api-best-practices/#urlformat [access 19.12.2019]
https://www.kennethlange.com/7-tips-for-designing-a-better-rest-api/ [access 18.12.2019]
https://www.codecademy.com/articles/what-is-rest [access 18.12.2019]
https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm [access 18.12.2019]
https://devszczepaniak.pl/wstep-do-rest-api/ [access 18.12.2019, Polish language]
https://www.devmobile.pl/rest-api-od-podstaw/ [access 18.12.2019, Polish language]