August 27, 2018 · #webdev #restapi #devtips

RESTful API Design: 13 Best Practices to Make Your Users Happy

First step to the RESTful way: make sure errors don't come back as 200 OK.

Architect at work. Daniel McCullough, unsplash.com

Web services have been around for as long as the HTTP protocol has existed. But especially since the advent of cloud computing, they have become a very common way of letting clients interact with our data.

I haven't been lucky enough to be a developer when SOAP APIs were still around. So I've mostly known **REST**, a resource-based architectural style for building APIs.

For a year or two already, **I have been working on projects involving either**

Yet, **some of them have been giving REST a very, very bad name.**

Bad usage of status codes, plain text responses, inconsistent schemas… **I've seen it all** (or at least, a good bunch). So I've decided to write up what I think are some **good practices when designing REST APIs**.

Disclaimer

I have no authority to say that the following practices comply 100% with the holy REST principles (if there is such a thing!). I gathered those from experience building and working with various APIs.

Likewise, I do not pretend to have mastered REST API design! It is a **craft** — the more you practice, the better you get.

I will expose some code snippets as "examples of bad design". If they look like something you'd write, it's okay! The only thing that matters is that we learn together. I hope this modest listicle will help us do just that.

Onwards: **here are tips, advice and recommendations to design REST APIs that make your users happy.**

1. Learn the basics of HTTP applied to REST

If you're to build a well-designed **REST API**, you'd better know the basics of the **HTTP protocol**. I truly believe **this will help you make better design decisions**.

I find the Overview of HTTP on the MDN web docs to be a very good read for this. However, as far as **REST API design** is concerned, here's a TL;DR of **HTTP applied to REST**:

- HTTP has **verbs** (or methods): GET, POST, PUT, PATCH and DELETE are the most common.

- REST is **resource-oriented** and a resource is represented by an **URI**: `/newspapers/`.
- An **endpoint** is the combination of a verb and an URI, e.g. `GET: /articles/`.
- An endpoint can be interpreted as an **action on a resource**. For example, `POST: /articles/` may mean "Create a new article".
- At a high-level, **verbs map to CRUD operations**: `GET` means `Read`, `POST` means `Create`, `PUT` and `PATCH` mean `Update`, and `DELETE` means... well, `Delete`.
- A response's status is specified by its **status code**: `1xx` for information, `2xx` for success, `3xx` for redirection, `4xx` for client errors and `5xx` for server errors.

Of course you can use anything the HTTP protocol offers for REST API design, but these are basic things I believe you need to keep in mind.

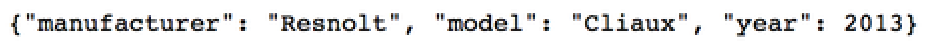
2. Don't return plain text

Although it is not imposed by the REST architectural style, most REST APIs use JSON as a data format.

However, it is not enough to return a body containing a JSON-formatted string. You need to **specify the `Content-Type` header too**! It must be set to the value **`application/json`**.

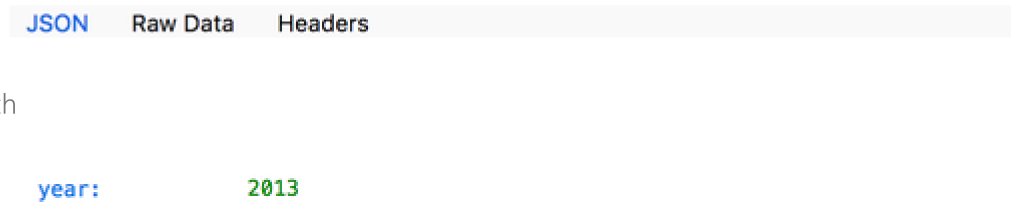
This is especially important for **programmatic clients** (for example, someone or a service interacting with your API via the `requests` library in Python) — some of them rely on this header to correctly decode the response.

Pro tip: you can verify a reponse's `Content-Type` very easily with Firefox. It has built-in pretty-display for responses with `Content-Type: application/json`. 🔥



```
{"manufacturer": "Resnolt", "model": "Cliaux", "year": 2013}
```

In Firefox, "Content-Type: text/plain" looks... plain.



```
year:            2013
```

"Content-Type: application/json" detected! Look how pretty and functional this is.

3. Avoid using verbs in URIs

If you've understood the basics, you'll now know it is **not RESTful** to put verbs in the URI.

This is because **HTTP verbs should be sufficient to describe the action being performed on the resource**.

Let's say you want to provide an endpoint to generate and retrieve a banner image for an article. I will note `:param` a placeholder for an URI parameter (like an ID or a slug). You might be tempted to create this endpoint:

```
GET: /articles/:slug/generateBanner/
```

But the `GET` method is semantically sufficient here to say that we're retrieving ("GETting") a banner. So, let's just use:

```
GET: /articles/:slug/banner/
```

Similarly, for an endpoint that creates a new article:

```
# Don't
POST: /articles/createNewArticle/
# Do
POST: /articles/
```

HTTP verbs all the things!

4. Use plural resource nouns

It may be hard to decide whether or not you should use plural or singular form for resource nouns.

Should you use `/article/:id/` (singular) or `/articles/:id/` (plural)?

I recommend using the plural form.

Why? Because it fits all types of endpoints very well.

I'd agree that `GET /article/2/` is fine, but what about `GET /article/`? Are we getting the one and only article in the system, or all of them?

To prevent this kind of ambiguity, **let's be consistent** (life advice!) and use plural everywhere:

```
GET: /articles/2/
POST: /articles/
...
```

5. Return error details in the response body

When an API server handles an error, it is convenient (and recommended!) to return **error details** in the JSON body to **help users with debugging**. Special kudos if you include which fields were affected by the error!

```
{
    "error": "Invalid payoad.",
    "detail": {
        "surname": "This field is required."
    }
}
```

6. Pay attention to status codes

This one is *super important*. If there's one thing you need to remember from this article, this is probably it:

> The worst thing your API could do is return an error response with a `200 OK` status code.

It's simply bad semantics. Instead, **return a meaningful status code** that correctly describes the type of error.

Still, you might wonder, "But I'm sending error details in the response body as you recommended, so what's wrong with that?"

Let me tell you a story.

I once had to use an API that returned `200 OK` for every response and indicated whether the request had succeeded via a `status` field:

```
{
    "status": "success",
    "data": {}
}
```

So even though the status was `200 OK`, I could not be absolutely sure it didn't fail to process my request.

In fact, the API could return responses such as:

```
HTTP/1.1 200 OK
Content-Type: text/html

{
    "status": "failure",
    "data": {
        "error": "Expected at least two items in list."
    }
}
```

(Yes — it also returned HTML content, because why not?)

As a result, I had to check the status code AND the ad-hoc `status` field to make absolutely sure everything was fine before I would read the `data`.

This kind of design is a real no-no because **it breaks the trust between the API and their users**. You come to fear that the API could be lying to you.

All of this is *extremely* un-RESTful. What should you do instead?

Make use of the status code and only use the response body to provide error details.

```
HTTP/1.1 400 Bad Request
Content-Type: application/json

{
    "error": "Expected at least two items in list."
}
```

7. Use status codes consistently

Once you've mastered status codes, you should strive to use them **consistenly**.

For example, if you choose that a `POST` endpoint returns a `201 Created` somewhere, use the same status code for every `POST` endpoint.

Why? Because users shouldn't have to worry about *which method on which endpoint will return which status code in which circumstances*.

So be **consistent**, and if you stray away from conventions, **document it** somewhere with big signs.

Generally, I stick to the following:

```
GET: 200 OK
POST: 201 Created
PUT: 200 OK
PATCH: 200 OK
DELETE: 204 No Content
```

8. Don't nest resources

REST APIs deal with resources, and retrieving a list or a single instance of a resource is straigforward. But what happens when you deal with **related resources**?

Let's say we want to retrieve the list of articles for a particular author — the one with `id=12`. There are basically two options.

The first option would be to **nest** the `articles` resource under the `authors` resource, e.g.:

```
GET: /authors/12/articles/
```

Some recommend it because it does indeed represent the one-to-many relationship between an author and their articles.

However, **it is not clear** anymore what kind of resource you're requesting. Is it authors? Is it articles?

Also flat is better than nested, so there must be a better way… And there is!

My recommendation is to **use the querystring** to filter the `articles` resource directly:

```
GET: /articles/?author_id=12
```

This clearly means: "get all articles for author #12", right? 👍

9. Handle trailing slashes gracefully

Whether or not URIs should have a trailing `/` is not really a debate. Simply choose one way or the other (i.e., with or without a trailing slash), stick to it and **gracefully redirect clients if they use the wrong convention**.

(I must say I have been guilty of this one myself more than once. 🙈)

Story time! One day, as I was integrating a REST API into a project, I kept receiving `500 Internal Error` on every single call. The endpoint I was using looked something like this:

```
POST: /entities
```

I was mad and couldn't figure out what I was doing wrong.

In the end, it turned out **the server was failing because I was missing a trailing slash!** So I started using

```
POST: /entities/
```

and everything went fine afterwards. 🤦‍♂

The API wasn't fixed, but hopefully *you* can prevent this type of issue for your users.

Pro tip: most web frameworks have an option to gracefully redirect to the trailed or untrailed version of the URL. Find that option and activate it.

10. Make use of the querystring for filtering and pagination

A lot of times, a simple endpoint is not enough to satisfy complex business cases.

Your users may want to retrieve items that fulfill a specific condition, or retrieve them in small amounts at a time to improve performance.

This is exactly what **filtering** and **pagination** are made for.

With **filtering**, users can specify properties that the returned items should have.

Pagination allows users to retrieve **fractions of a data set**. The simplest kind of pagination is **page number pagination**, which is determined by a `page` and a `page_size`.

Now the question is: **how do you incorporate such features in a RESTful API?**

My answer is: **use the querystring**.

I'd say it's pretty obvious why you should use the querystring for pagination. It would look like this:

```
GET: /articles/?page=1&page_size=10
```

But it may be less obvious for filtering. At first, you might think of doing something like this to retrieve a list of published articles only:

```
GET: /articles/published/
```

Design issue: `published` **is not a resource!** Instead, it is a trait of the data you're retrieving. That kind of thing should go in the **querystring**.

So in the end, a user could retrieve "the second page of published articles containing 20 items" like so:

```
GET: /articles/?published=true&page=2&page_size=20
```

Beautifully explicit, isn't it?

11. Learn the difference between `401 Unauthorized` and `403 Forbidden`

When handling security errors in an API, it's very easy to be confused about whether the error relates to **authentication** or **authorization** (a.k.a. permissions) — happens to me all the time.

Here's my cheat-sheet for knowing what I'm dealing with depending on the situation:

- Has the user not provided authentication credentials? Were they invalid? 👉 `401 Unauthorized`.
- Was the user correctly authenticated, but they don,t have the required permissions to access the resource? 👉 `403 Forbidden`.

12. Make good use of `202 Accepted`

I find `202 Accepted` to be a very handy alternative to `201 Created`. It basically means:

> I, the server, have understood your request. I have not created the resource (yet), but that is fine.

There are two cases which I find `202 Accepted` to be especially suitable for:

- If the resource will be created as a result of future processing — e.g. after a job has finished.
- If the resource already existed in some way, but this should not be interpreted as an error.

13. Use a web framework specialised in REST APIs

As a last best practice, let's discuss this question: **how do you actually implement best practices in your API?**

Oftentimes, you want to create a quick API so that a few services can interact with one another.

Python developers would grab Flask, JS developers would grab Express, and they would implement a few simple routes to handle HTTP requests.

The issue with this approach is that **generally, the framework is not targeted at building REST API servers**.

For example, both Flask and Express are two very versatile frameworks, but they were not *specifically* made to help you build REST APIs.

As a result, you have to take **extra steps** to implement best practices in your API. And most of the times, **laziness or a lack of time mean you won't make the effort** — and leave your users with a quirky API.

The solution is simple: **use the right tool for the job**.

HOW MANY TIMES DO I HAVE TO TELL YA?
USE THE RIGHT TOOL FOR THE RIGHT JOB!
memegenerator.net

Holy sayings from good lad Scotty.

New frameworks have emerged in various languages that are specifically made to build REST APIs. **They help you follow best practices hassle-free without sacrificing productivity.**

In Python, one of the best API framework I,ve found is Falcon. It's just as simple to use as Flask, incredibly fast and perfect for building REST APIs in minutes.

Falcon: Unburdening APIs for over 0.0564 centuries.

If you're more of a Django type of person, the go-to is the Django REST Framework. It's not as intuitive, but incredibly powerful.

In NodeJS, Restify seems to be a good candidate as well, although I haven't tried it yet.

I strongly recommend you give these frameworks a shot! They will help you build beautiful, elegant and well-designed REST APIs.

Let's give REST a great name!

We should all strive to make APIs a pleasure to use. Hopefully, this article helped you learn some tips and techniques to build **better REST APIs**. To me, it boils down to **good semantics**, **simplicity** and **common sense**.

REST API design is a craft more than anything else. If you have a different approach to some of the points above, I'd love to hear about it.

In the meantime, keep 'em great APIs coming! 💻

Stay in touch!