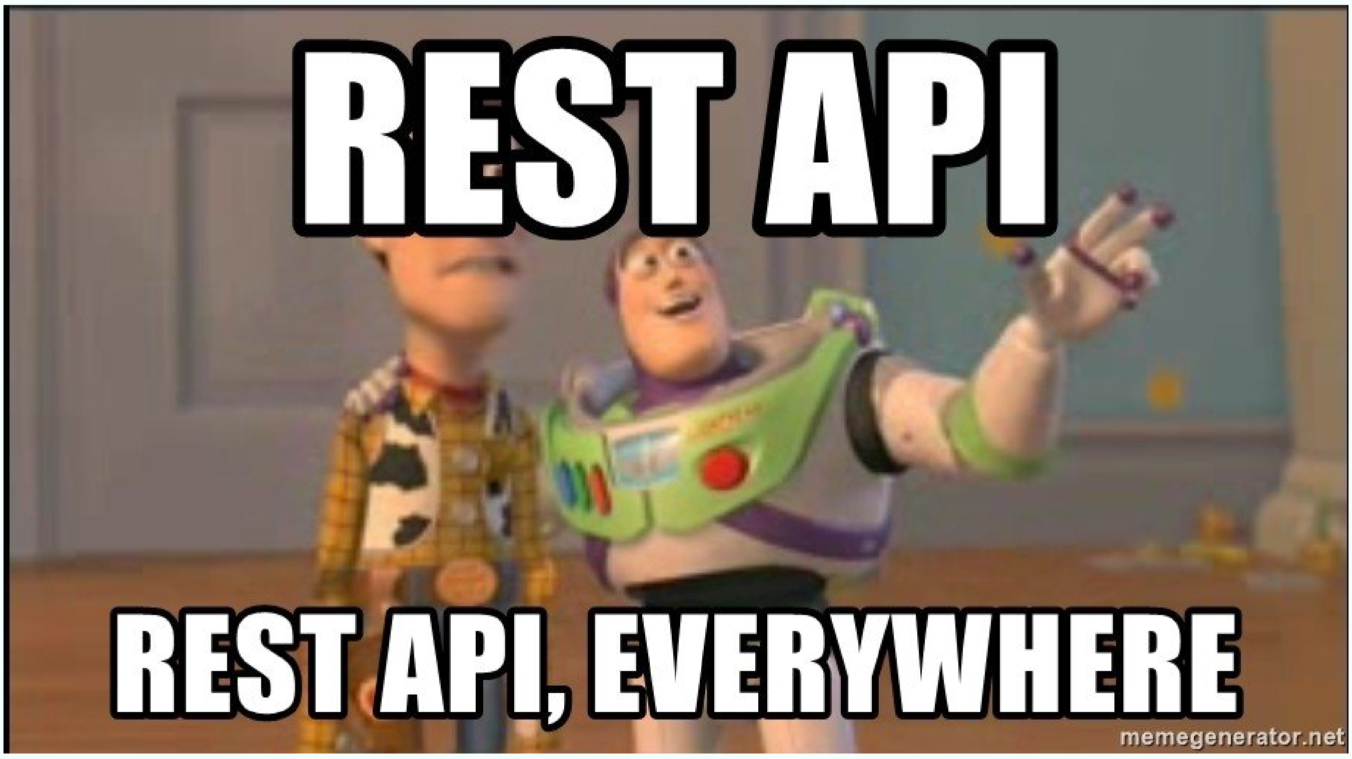
API Architecture — Design Best Practices for REST APIs

Abdul Wahab Oct 31 · 12 min read

REST APIs everywhere!



Source: memegenerator.net

In general, web services have been in existence for just as long as the HTTP protocol has existed. But, since the beginning of cloud computing, they have become *the* ubiquitous method of enabling client interaction with services and data.

As a developer, I have been lucky enough to work with some SOAP services that are still around @ work. But, I've largely played with **REST**, which is a resource-based architectural style for developing APIs and web services.

For a great chunk of my career, **I have been involved in projects either building, designing, and using APIs.**

Most of the APIs I have seen "claimed" to be "**RESTful**"—*meaning compliant with the principles and constraints of REST architecture*.

Yet, there are a few handful I have worked with that **give REST a very, very bad rep**.

Inaccurate usage of HTTP status codes, plain text responses, inconsistent schemas, verbs in endpoints… **I feel like I've seen it all** (or at least, a good chunk).

So, I decided to write up a piece describing what *I personally think* are some **best practices when it comes to designing REST APIs**.

Just so we're clear…

Source: giphy

I do not claim to be the authority, or mean to infer that the following practices are 100% in sync with any "holy REST principles" (*if there even is such a thing in existence*). I have pieced these thoughts from my own experiences building, and working with different APIs throughout my career.

Also, I do not pretend to have mastered REST API design, either! I believe it is an **art/sport** — the more you practice, the better you get.

I will list out some code snippets as “examples of bad design”. If they look like something you would write, that’s fine! 🙂 The only thing that matters is that we learn together.

Here are some tips, advice, and guidance to designing great REST APIs that will make your consumers (*and developers*) happy.

1. Learn the basics of HTTP

Source: imgflip.com

If you aspire to build a well-designed **REST API**, you must know the basics of the **HTTP protocol**. I firmly believe **this will help you make good design choices**.

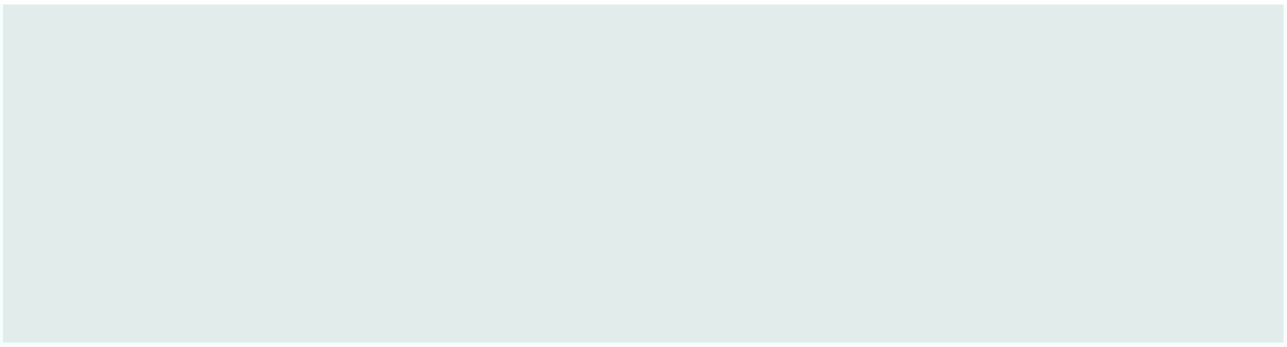
I find the Overview of HTTP on the Mozilla Developer Network docs to be a pretty comprehensive reference for this topic.

Although, as far as **REST API design** is concerned, here is a TLDR of **HTTP applied to RESTful Design**:

- HTTP has **verbs** (actions or methods): GET, POST, PUT, PATCH and DELETE are most common.
- REST is **resource-oriented** and a resource is represented by an **URI**: `/library/`
- An **endpoint** is the combination of a verb and an URI, example: `GET: /books/`
- An endpoint can be interpreted as an *action performed on a resource*. Example: `POST: /books/` may mean "Create a new book".
- At a high-level, **verbs map to CRUD operations**: `GET` means `Read`, `POST` means `Create`, `PUT` and `PATCH` mean `Update`, and `DELETE` means `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 other things the HTTP protocol offers for REST API design, but these are the basic things I believe you must keep in mind.

2. Do not return plain text



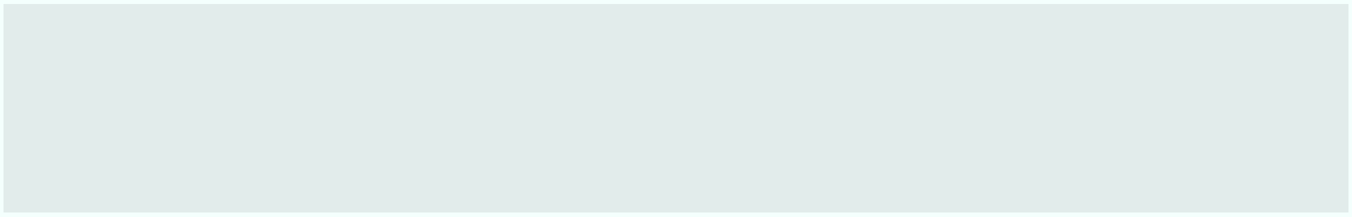
Source: meme-arsenal

Although this is not imposed or mandated by any REST architectural style, most REST APIs by convention use JSON as the data format.

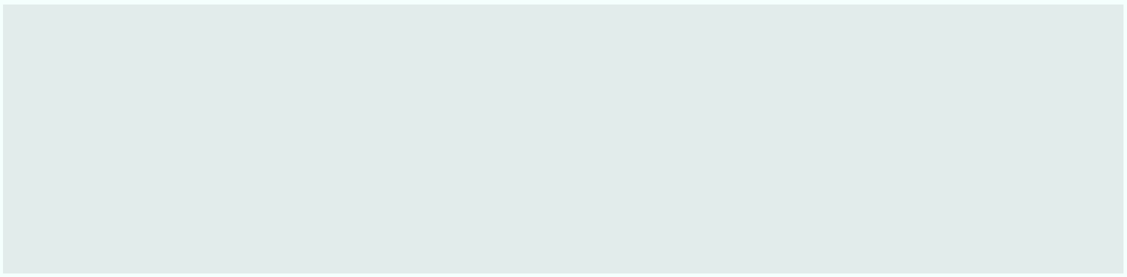
However, it is not good enough to just return a response body containing a JSON-formatted String. You should still **specify the `Content-Type` header.** It must be set to the value `application/json`.

This is especially important when dealing with **application/programmatic clients** (example, another service/API interacting with your API via the `requests` library in Python)—some of them rely on this header to accurately decode the response.

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

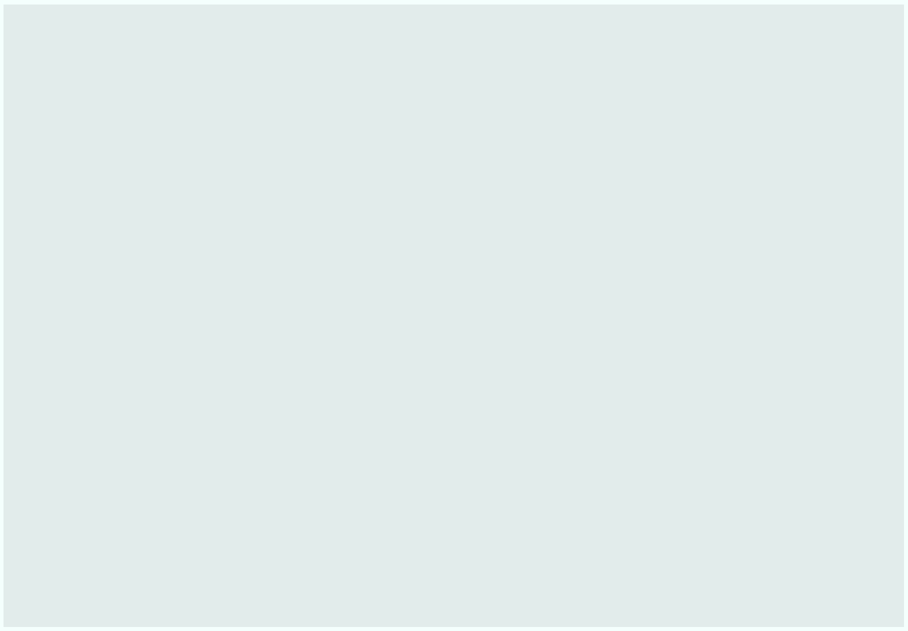


In Firefox, "Content-Type: text/plain" looks… plain.



"Content-Type: application/json" Nice, how pretty and functional this is. 🕺

3. Do not use verbs in URIs



Source: quickmeme

By now if you’ve understood the basics, you’ll start to realize that it is **not RESTful** to put verbs in the URI.

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

Example: Let’s say that you are providing an endpoint to generate and retrieve a book cover for a book. I will note `:param` a placeholder for an URI parameter (like an ID or a slug). Your first idea might be to create a similar endpoint to this one:

```
GET: /books/:slug/generateBookCover/
```

But, the `GET` method is syntactically sufficient here to say that we are retrieving ("GETting") a book’s cover. So, let's just use:

```
GET: /books/:slug/bookCover/
```

Likewise, for an endpoint that creates a new book:

```
# Don’t do this
POST: /books/createNewBook/

# Do this
POST: /books/
```

HTTP verbs (*verbalizses*) all the things!

4. Use plural nouns for resources

Source: gifer.com

This may be hard to determine, whether or not you should use plural or singular form for resource nouns.

Should we use `/book/:id/` (singular) or `/books/:id/` (plural)?

My personal advice is to *use the plural form*.

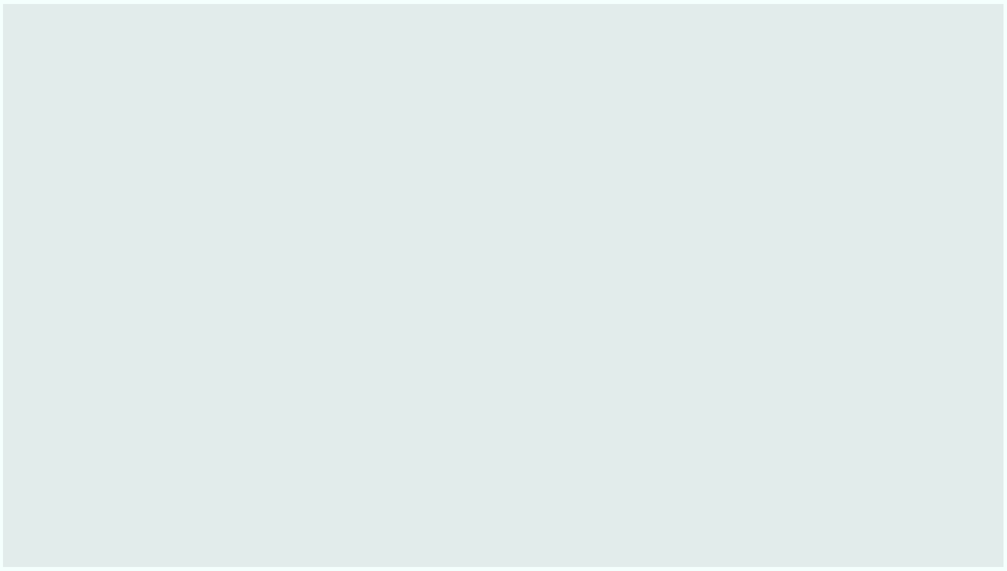
Why? Because it fits *all types* of endpoints very well.

I can see that `GET /book/2/` is fine. But what about `GET /book/`? Are we GETting the one and only book in the library, couple of them, or *all of them*?

To prevent this kind of ambiguity, **let’s be consistent** (💡Software career advice!) and use plural everywhere:

```
GET: /books/2/
POST: /books/
...
```

5. Return the error details in the response body

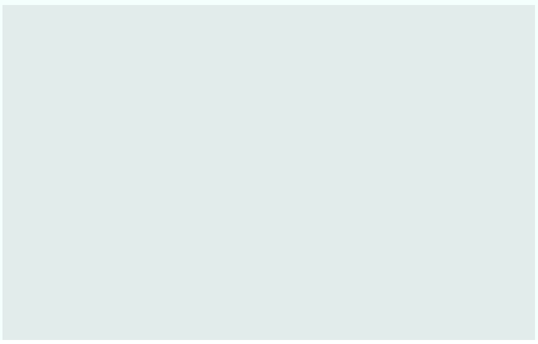


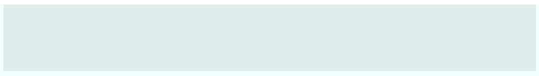
Source: Gif Abyss

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

```
{
    "error": "Invalid payload.",
    "detail": {
        "name": "This field is required."
    }
}
```

6. Pay special attention to HTTP status codes




Source: Giphy

I feel this one is *pretty important*. If there is 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 HTTP status code** that *accurately* describes the type of error.

Still, you're probably wondering, *"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 integrate an API that returned `200 OK` for every response and indicated whether the request had succeeded via a `status` field:

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

Despite the HTTP status code was returning `200 OK`, I could not be *absolutely* sure it that it didn't fail to process my request.

In fact, the API could return responses like so:

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

{
    "status": "failure",
    "data": {
        "error": "Expected at least three items in the 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* that everything was fine before I would read the `data`.

SO ANNOYING! □♂

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

Source: tenor

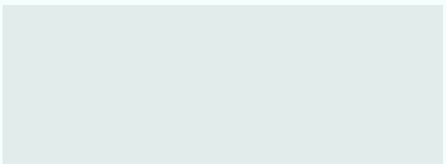
All of this is *tremendously* un-RESTful. What should you do instead?

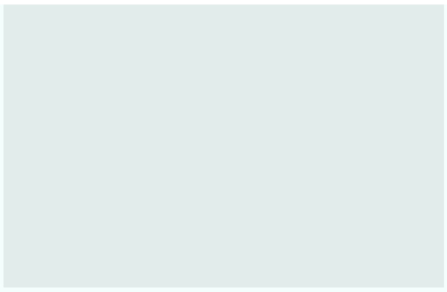
Make use of the HTTP status code, and only use the response body to provide error details.

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

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

7. You should use HTTP status codes consistently




Source: Tenor

Once you’ve mastered HTTP status codes, you should aim to use them **consistently**.

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

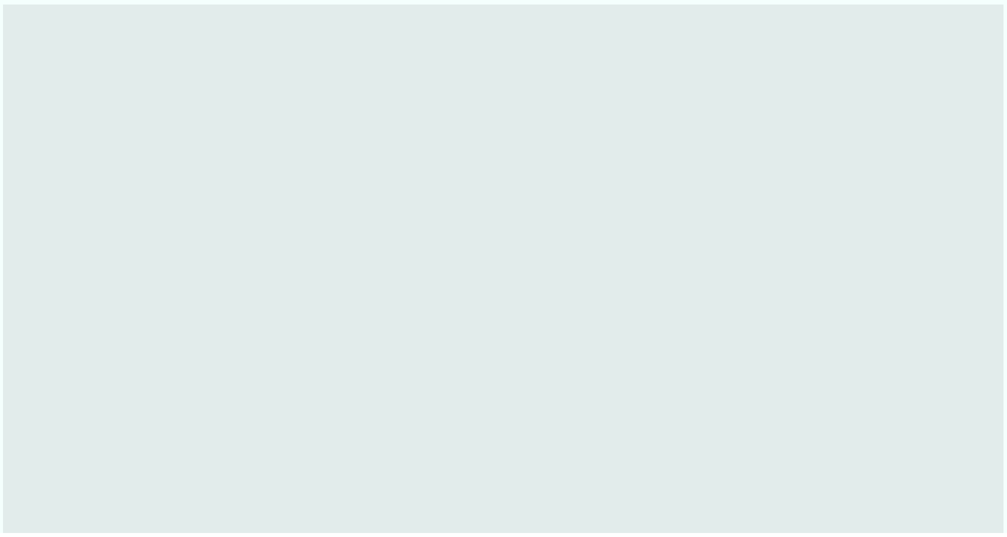
Why? Because consumers should not have to worry about *which method on which endpoint will return which status code in which circumstances*.

So, be predictable (**consistent).** If you have to stray away from conventions, **document it** somewhere with big signs.

Typically, I stick to the following pattern:

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

8. Do not nest resources



Source: memes.com

You are probably noticing by now that REST APIs deal with resources. Retrieving a list, or a single instance of a resource is straightforward. But, what happens when you deal with **related resources**?

For example, let's say we want to retrieve the list of books for a particular author — the one with `name=Cagan`. There are basically two options.

The first option would be to **nest** the `books` resource under the `authors` resource, example:

```
GET: /authors/Cagan/books/
```

Some architects recommend this convention because it does indeed represent the **one-to-many relationship** between an author and their books.

But, **it is not clear** anymore what type of resource you are requesting. Is it authors? Is it books? …

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

My personal recommendation is to **use query string parameters** to filter the `books` resource directly:

```
GET: /books?authorName=Cagan
```

And this clearly means: "Get all books for author name Cagan", right? 🙂

9. Handle trailing slashes gracefully

Source: tenor

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

(I will admit, I have been guilty of this one myself more than once. 🙈)

Story time! 📓 One day, as I was integrating a REST API into one of my projects, and I kept receiving `HTTP 500 Internal Error` on *every single call*. The endpoint I was using looked something like this:

```
POST: /buckets
```

I was fuming and for the life of me couldn't figure out what the hell I was doing wrong. □

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

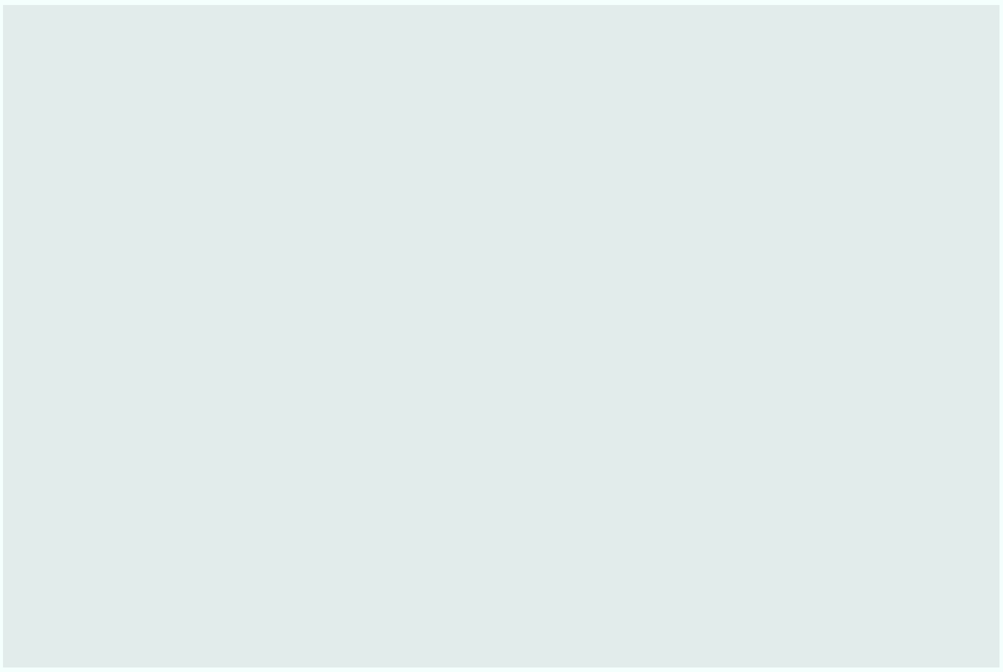
```
POST: /buckets/
```

Aaaand everything went fine afterwards. □♂

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

💡**Pro-Tip:** Most web-based frameworks (Angular, React, etc.) have an option to gracefully redirect to the trailed or untrailed version of the URL. Find that option and activate it as early as possible.

10. Make use of the querystring for filtering and pagination



Source: tenor

Majority of the times, a simple endpoint is not enough to satisfy various complex business cases.

Your consumers 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**, consumers can specify parameters (or properties) that the returned items should have.

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

Now, the question is: H**ow do you incorporate such features in a REST API?**

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

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

```
GET: /books?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 *only published* books:

```
GET: /books/published/
```

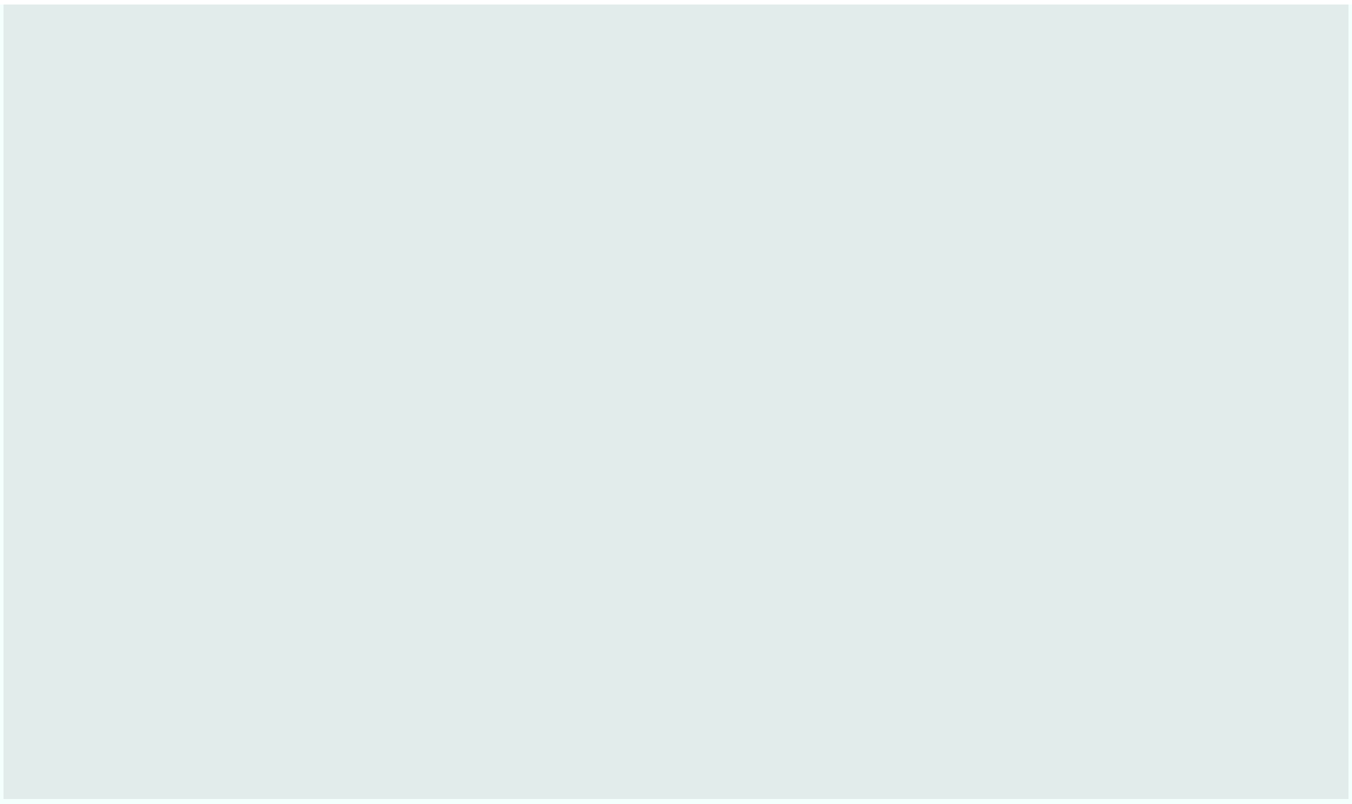
Design issue: **`published` is not** *a resource!* Instead, it is a *trait* of the data you are retrieving. That kind of thing should go in the **querystring**.

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

```
GET: /books?published=true&page=2&page_size=10
```

Beautifully explicit, isn't it?

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



Source: http.cat

If I had a quarter for every single time I have seen developers and even some experienced architects mess this up…

When handling security errors in a REST API, it is extremely easy to get confused about whether the error relates to **Authentication** or **Authorization** (a.k.a. *permissions*)— used to happen to me all of the time.

This is my cheat sheet for knowing what I am dealing with, depending on the situation:

- Has the consumer not provided authentication credentials? Was their SSO Token invalid/timed out? ☞ `401 Unauthorized`.
- Was the consumer *correctly authenticated*, but they *don't have the required permissions/proper clearance* to access the resource? ☞ `403 Forbidden`.

12. Make good use of HTTP `202 Accepted`

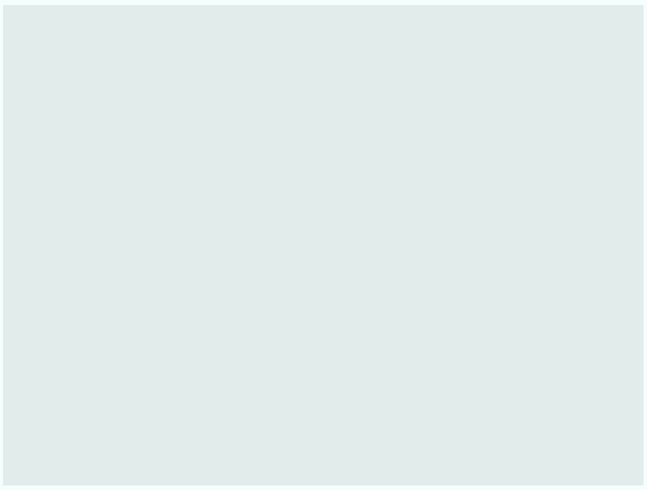
I find `202 Accepted` to be a very handy alternative to `201 Created`. It basically means:

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

There are two main scenarios which I find `202 Accepted` to be especially suitable:

- If the resource will be created as a result of future processing — example: After a job/process has finished.
- If the resource already existed in some way, but this should not be interpreted as an error.

13. Use a web framework specialized in REST APIs


Source: memegenerator.net

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

Most of the time, you want to create a quick API so that a few services can interact with one another.

Python developers would grab Flask, JavaScript developers would grab Node (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**.

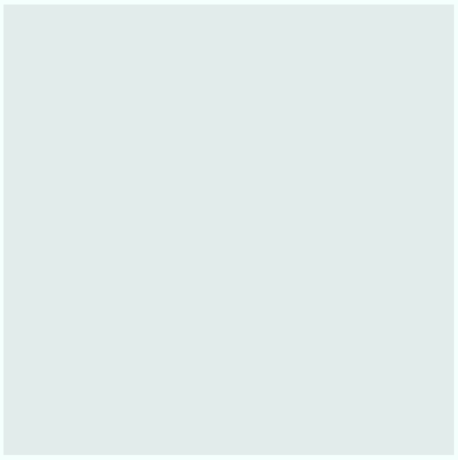
For example, both Flask and Express are two very versatile frameworks, but they were not *specifically* made to help you build REST APIs.

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 will not make the effort** — and leave your consumers with a quirky API.

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

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, nicely fast and perfect for building REST APIs within minutes.



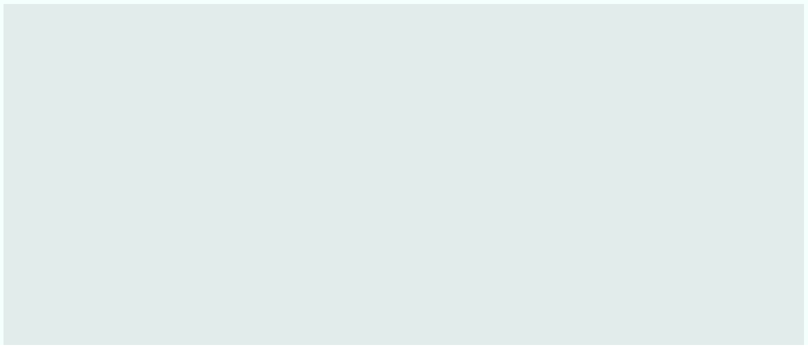
Falcon: Unburdening APIs for over 0.0564 centuries.

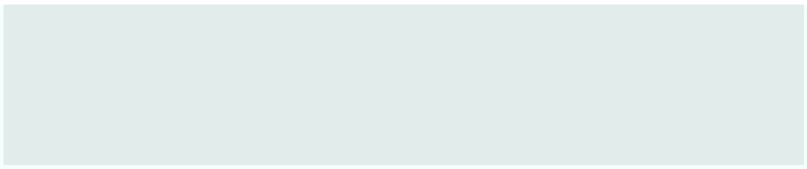
If you're more of a Django type of person, the go-to is the Django REST Framework. It's not as intuitive, but incredibly powerful.

In Node, Restify seems to be a good candidate as well, although I haven't gotten around to trying it yet.

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

Closing thoughts





Source: memegenrator.net

We should all strive to make APIs a pleasure to use. ***Both, for consumers and our own fellow developers***.

I hope this article helped you learn some tips, and inspired techniques to build **better REST APIs**. To me, it just boils down to **good semantics**, **simplicity,** and **common sense**.

REST API design is an **art,** more than anything else.

If you have a different approach to any of the tips I have shared above, please share. I would love to hear about it.

In the meantime, keep ’em APIs coming! 💻

1.8K   25

Rest Api   Technical Architecture   Technical Design   Software Development   Best Practices

More from Abdul Wahab

Follow

Software Engineer skilled in crafting experiences customers 💙 Today I manage 🔐 data access in AWS ☁️ I'm a holistic problem solver. All views are 💯 my own.

More From Medium

Maven on Java 8 and 9

Nicolai Parlog in nipafx news

How To Share 3D Models With IPFS

Justin Hunter in Pinata

Continuous Integration — what, why, who?

Louise Norris

Facelock + AWS +Email + Whatsapp

Pratikkorgaonkar

Deploy Angular and Express app in Docker Containers

kartik agarwal

Install Raspbian Buster Lite Headless (setup Wi-Fi and activate ssh without access to the command…

Dani Dudas

Image API to Impressionism

trent schultz

10 Hot Things a Php Expert Can Do to Make Your Website Sizzle

Punch