May 31, 2017 · 13 min read

RESTful API Design Tips from Experience

A working guide of API design tips and trend evaluations.

© Nathaniel Merz—Imaginary Peaks via. 500px

A Quick Word on Code Reuse

I am reminded of a lovely quote that I found through another Medium post by Ken Rogers, of which was originally written by Hemingway:

> *We are all apprentices in a craft where no one ever becomes a master.*

As I write this, I chuckle to myself in seeing a great parallel behind myself referencing Hemingway's quote from someone else; the sheer notion that I need not labour away at creating a different implementation of the passage with similar functionality for the result value (or in this case, meaning) is a literary testament to code reuse!

But I'm not here to write about the benefits of code packages, but more to mention some of the traits I've come to appreciate, and actively implement in present and future projects. And of these features and

implementation details, I grow my own package of API rules and primitives.

A Note Regarding Amendments

From publishing this article, many threads of discussion in channels such as Reddit have helped me adjust and tweak some of my explanations and stances on API design, particularly with ***enveloping*** *and* ***media types****. I would like to thank all who have contributed in the discussion, and I hope this helps build this article into a more valuable resource for others. Thanks!*

Use API Versioning

If you're going to develop an API for any client service, you're going to want to prepare yourself for eventual change. This is best realised by providing a "version-namespace" for your RESTful API.

We do this with simply adding the version as a prefix to all URLs.

```
GET www.myservice.com/api/v1/posts
```

However, through studying other API implementations, I've grown to like a shorter URL style offered by accessing the API as part of a subdomain, and then dropping the `/api` from the route; *shorter and more concise is better.*

```
GET api.myservice.com/v1/posts
```

Cross-Origin Resource Sharing (CORS)

It is important to consider that when placing your API into a *different subdomain* such as `api.myservice.com` it will require implementing CORS for your backend if you plan to host your frontend site at `www.myservice.com` and expect to use AJAX calls without throwing `No Access-Control-Allow-Origin header is present` errors.

Use Plurals

It makes semantic sense when you request many posts from `/posts` .

And for goodness sake don't consider `/post/all` with `/post/:id` .

```
// plurals are consistent and make sense
GET /v1/posts/:id/attachments/:id/comments
```

```
// don't consider ambiguity
// is it just one comment? is it a form? etc.
GET /v1/post/:id/attachment/:id/comment
```

> *"I like the idea of using plurals for resource names, but sometimes you get names that can't be pluralised." (source)*

In cases like these you should simply try to get as close to plural as you can!

More about mixing types in: "**Use a Root-Level 'Me' Endpoint"** below.

Avoid Unnecessary Query Strings

> *Query strings should be used for further filtering results beyond the initial grouping of a logical set offered by a relationship.*

Seek to design endpoint paths that avoid unnecessary query string parameters as they are generally harder to read and to work with, when compared to paths whose structure promotes an initial logical filtering and grouping of such items the deeper it goes.

> *This* `/projects/:id/collections` *is better than* `/collections?projectId=:id` . *And this* `/projects/:id/collections/:id/items` *is even better than* `/items?projectId=:id&collectionId=:id` .

More in: **"Avoid Operations on Nested Routes"** below.

Use HTTP Methods

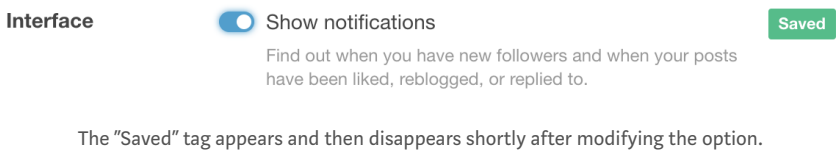
Use methods such as:

- `GET` for fetching data.
- `POST` for adding data.
- `PUT` for updating data (as a whole object).
- `PATCH` for updating data (with partial information for the object).

- `DELETE` for deleting data.

I would like to add that I think `PATCH` is great way to cut down the size of requests to change parts of bigger objects, but also that it fits well with commonly implemented auto-submit/auto-save fields.

A nice example of this is with Tumblr's "Dashboard Settings" screen, where non-critical options about the user experience of the service can be edited and saved, per item, without the need of a final form submission button. It is simply a much more organic way to interact with the user's preference data.



The "Saved" tag appears and then disappears shortly after modifying the option.

And it's also good practice to return the updated object in response to a successful `POST` , `PUT` , or `PATCH` request, instead of just `null` .

Read more in **"Media Types—JSON Responses and Requests"** below.

Use Envelopes

> *"I do not like enveloping data. It just introduces another key to navigate a potentially dense tree of data. Meta information should go in headers."*

> *"One argument for nesting data is to provide two distinct root keys to indicate the success of the response,* `data` *and* `error` *. However, I delegate this distinction to the HTTP status codes in cases of errors."*

Originally, I held the stance that enveloping data is not necessary, and that HTTP provided an adequate "envelope" in itself for delivering a response. However, after reading through responses on Reddit, various vulnerabilities and potential hacks can occur if you do not envelope JSON arrays.

I now recommend enveloping; you should envelope your data responses!

```
// enveloped, a top level object is secure and succinct
{
```

```
  data: [
    { ... },
    { ... },
    // ...
  ]
}
```

```
// non-enveloped, potential security risk
[
  { ... },
  { ... },
  // ...
]
```

> *"Additionally, if you like to use a tool like normalizr for parsing data from responses client-side, removing an envelope removes the need for constantly extracting the data from the response payload to pass it to be normalised."*

On the contrary, providing an extra key for accessing your `data` allows for reliably checking if anything was actually returned, and if not, may refer to a non-colliding `error` key separate from the body of a response.

It is also important to consider that unlike some, languages such as JavaScript will evaluate empty objects as `true` ! Hence it is important to not return an empty object for `error` as part of a response in the case of:

```
// enveloped, error extraction from payload
const { data, error } = payload
```

```
// processing errors if they exist
if (error) { throw ... }
```

```
// otherwise
const normalizedData = normalize(data, schema)
```

Media Types—JSON Responses and Requests

> *"Everything should be serialised into JSON. If you're expecting JSON from the server, be polite, and provide the server with JSON.* Consistency!"

Obviously “everything” is an overstatement as some comments point out, but was intended to refer to any simple, plain object that should be serialised for the process of consuming and/or returning from the API.

It is essential to “*define your media types”* through headers on both responses and requests for a RESTful API. When dealing with JSON ensure that you include a `Content-Type: application/json` header, and respectively for other response types, be it CSVs or binaries.

There have been cases where I’ve had nothing to return from the success of an action (i.e. `DELETE`), however I feel that returning an empty object can in some languages (such as Python) be evaluated as false and may not be as obvious to a human debugging their application.

Support the `204 – No Content` response status code in cases where the request was successful but has no content to return.

> *“Because of this, I like to return* `"OK"` *from such endpoints, which although is a string, is consistently wrapped into a simple response object.”*

Through much public consideration into handling “empty” responses, and ideas of returning *some* unique signal to represent success, *I have reevaluated the appropriateness of using an arbitrary* `"OK"` *string to indicate success*.

The envelope of the response, coupled with a `2XX` HTTP success code is enough to indicate a successful response without arbitrary “information”.

```
DELETE /v1/posts/:id
```

```
// response - 204
{
  "data": null
}
```

Use HTTP Status Codes and Error Responses

Because we are using HTTP methods, we should use HTTP status codes. Although a challenge here is to select a distinct slice of these

codes, and then depend on response data to detail any response errors. Keeping a small set of codes helps you consume and handle errors consistently.

I like to use:

for Data Errors

- `400` for when the requested information is incomplete or malformed.
- `422` for when the requested information is okay, but invalid.
- `404` for when everything is okay, but the resource doesn't exist.
- `409` for when a conflict of data exists, even with valid information.

for Auth Errors

- `401` for when an access token isn't provided, or is invalid.
- `403` for when an access token is valid, but requires more privileges.

for Standard Statuses

- `200` for when everything is okay.
- `204` for when everything is okay, but there's no content to return.
- `500` for when the server throws an error, completely unexpected.

Furthermore, returning responses after these errors is also very important. I want to consider not only the presentation of the status itself, but also a reason behind it.

In the case of trying to create a new account, imagine we provide an `email` and `password` . Of course we would like to have our client app prevent any requests with an invalid email, or password that is too short, but outsiders have as much access to the API as we do from our client app when it's live.

- If the `email` field is missing, return a `400` .
- If the `password` field is too short, return a `422` .
- If the `email` field isn't a valid email, return a `422` .

- If the `email` is already taken, return a `409` .

> *"It's much better to specify a more specific 4xx series code than just plain 400. I understand that you can put whatever you want in the response body to break down the error but codes are much easier to read at a glance." (source)*

Now from these cases, two errors returned `422s` regardless of their reasons being different. This is why we need an error code, and maybe even an error description. It's important to make a distinction between code and description as I intend to have `error` (code) as a machine consumable constant, and `description` as a human consumable string that may change.

In the case of per-field errors, the presence of the field as a key in the error is enough of a "code" to indicate that it is a target of a validation error.

Field Validation Errors

For returning those per field errors, it may be returned as:

```
POST /v1/register
```

```
// request
{
  "email": "end@@user.comx"
  "password": "abc"
}
```

```
// response - 422
{
  "error": {
    "status": 422,
    "error": "FIELDS_VALIDATION_ERROR",
    "description": "One or more fields raised validation
errors."
    "fields": {
      "email": "Invalid email address.",
      "password": "Password too short."
    }
  }
}
```

Operational Validation Errors

And for returning operational validation errors:

```
POST /v1/register
```

```
// request
{
  "email": "end@user.com",
  "password": "password"
}
```

```
// response - 409
{
  "error": {
    "status": 409,
    "error": "EMAIL_ALREADY_EXISTS",
    "description": "An account already exists with this
email."
  }
}
```

This way, your fetch logic watches out for non-200 errors, and can then straight-up check the `error` key from the response and then compare it to any further logic in the client app.

Bundling the `status` seems useful too if you don't want to introspect the response meta-data and conditionally add the key in if you ever need it.

The `description` acts as a fallback human-readable error message in the case an appropriate localisation string implementation cannot be used.

Avoid Password Composition Rules

After doing a lot of research into password rules, I've come to agree that password rules are bullshit and are part of NIST's "don'ts".

I've collated some points (from the above links) for password handling:

1. Only enforce a minimum *unicode* password length (min 8-10).
2. Check against common passwords ("password12345")
3. Check for basic entropy (don't allow "aaaaaaaaaaaa").
4. Don't use password composing rules (at least one "!@#$%&").
5. Don't use password hints (rhymes with "assword").
6. Don't use knowledge-based authentication ("security" questions).

7. Don't expire passwords without reason.
8. Don't use SMS for two-factor authentication.
9. Use a password salt of 32-bits or more.

In a way, all these "don'ts" should make password validation far easier!

Use Access and Refresh Tokens

Modern stateless, RESTful APIs implement authentication with tokens. This eliminates the need to handle sessions and cookies on a now stateless server, and makes it really easy to debug network requests by simply utilising the `Authorization` header (or even an `access_token` query param).

Access Tokens

The access token is used for authenticating all future API requests, has a short life span and can't be revoked*.

> *"[..] one downside to [JWTs] is that you can't revoke [them] before their expiry without making the authentication flow stateful in some way. If a user account is compromised, the attacker can't be forcefully logged out. You just have to wait for the token to expire." (source)*

I write about a potential solution to this problem in this post here. (link)

However, JWTs can be revoked if the API expects a different "secret" for signing, however this would invalidate **all currently issued tokens, but that shouldn't matter too much considering they are intended to be short-lived.*

Refresh Tokens

The refresh token is returned in the initial login response, but it is hashed and stored in the database with an expiry timestamp and relationship to the user. This acts as a long-life, password-like secret token, which can be used to request a new short-life JWT access tokens. Refresh tokens also extend their life every time they are used for renewal, meaning that a user doesn't need to log in again if they continually use the service.

Logging In

In my implementation, a normal login process would look like:

1. Receive email and password through a `/login` endpoint.
2. Check the email and password hash against the database.
3. Create a new refresh token and JWT access token.
4. Return both.

Renewing a Token

And a normal renew auth flow would look like:

1. JWT has expired when attempting to create request from client.
2. Submit refresh token to a `/renew` token endpoint.
3. Check refresh token exists by matching hash against database.
4. On success, create new JWT access token and extend expiry timeout.
5. Return access token only.

Validating a Token

Validating a JWT access token is a matter of checking the expiry date and signature hash. If either don't pass, it's an invalid token.

However if it does pass, the payload of the JWT contains a `uid` which is used in the APIs response context to derive a corresponding `user` object to check permissions/roles, and appropriately create/read/update/delete data.

Terminating a Session

Because the refresh token is stored in the database, it can be deleted to effectively "terminate a session". This provides an element of control for the user as they may wish to secure their account with control over their active refresh token "sessions", and potentially with how often they wish to reauthenticate (by adjusting the behaviour behind the expiry timestamp).

Example Code

> *"[..] can you add examples for the authentication?" (source)*

Coming up soon!

Keep JWTs Small

When serialising information into an JWT access token, keep it to the bare minimum, it's nicer when authentication tokens aren't super long, because they don't have to be. Serialise only the `uid` (id) of the user if you can, the rest can be delivered by a `GET /me` endpoint.

It's also worth noting that any sensitive information stored in a JWT payload is not inherently secure, because it is simply a base64 encoded string.

Use a Root-Level "Me" Endpoint

As a matter of preference `/profile` is cool too, but the general idea is the same; deliver the basic information about the user relevant to themselves.

I feel as though it may be nice to bundle information like personal `settings` in there too if that happens to be server-side data too.

However, I've seen implementations that feel rather messy in their own right, such as mixing a `/users/:id` endpoint that is supposed to accept integers and then allowing the string `me` to point to your own profile.

To me this needlessly challenges the size of your APIs "route-space" per se, where you should prefer to create less nested routes, and removing any ambiguity through mixing string and integer inputs.

Having `/me` accessible from the root makes accessing deeper levels of your personal information more consistent, in the way of `/settings` or even `/billing` information from `/me`, of which you wouldn't typically have access to through `users/:id/billing` for any other user.

```
// not preferred
GET /v1/users/me
```

```
// preferred, shorter, no int/str mixing
GET /v1/me
```

Avoid Operations on Nested Routes

In another project which was built afresh with a few of these design ideals, it fell into designing a URL system that became hard to work with.

```
// consistently long url
PATCH /v1/projects/:id/collections/:id/items/:id/attachments
```

Now this may seem practical if the intent was to `POST` a new attachment, however, if in developing the client-side application you wanted to do something as simple as “star” the attachment, you would need to rewrite supporting code for dispatching an action similar to this:

What I mean by supporting code would be this:

Wouldn’t it be nice if you could delegate the resolution of all meta data about the attachment (and even permissions of whether or not you can view/star an attachment) to the server, and just use a root-level endpoint?

Overall, I think there are advantages to either method of reference, so there isn't any reason why both can't be implemented, although I'd like to make a distinction in preferring a **longer path for creating/fetching** nested resources, and a **shorter path for updating/deleting** them.

The reason I maintain for **fetching** is that a longer path provides a natural filter to the queried resources, without the need to tag on query parameters for general results in that collection.

This `/projects/:id/collections` is better than `/collections?projectId=:id` . And this `/projects/:id/collections/:id/items` is even better than `/items?projectId=:id&collectionId=:id` .

Provide Pagination

Pagination is really important because you don't want a simple request to be incredibly expensive if there are thousands of rows of results. It seems obvious, but many neglect this functionality.

There are multiple ways to do this:

"From" Parameter

Arguably the easiest to implement, where the API accepts a `from` query string parameter and then returns a limited number of results from that offset (commonly `20` results).

Also best to provide a `limit` parameter which has a hard-maximum, such as the case of Twitter, with a maximum of `1000` and default limit of `200` .

Next Page Token

Google's Places API returns a `next_page_token` in its responses if there is more information available beyond the limited `20` results per page. It then accepts `pagetoken` as a parameter for a new request which

continues returning more results with a new `next_page_token` until it is exhausted.

Twitter does a similar thing instead using a param called `next_cursor` .

Implement a "Health-Check" Endpoint

Through developing with AWS, it been necessary to provide a way to output a simple response that can indicate that the API instance is alive and does not need to be restarted. It's also useful for easily checking what version of the API is on any machine at any time, without authentication.

```
GET /v1
```

```
// response - 200
{
  "status": "running",
  "version": "fdb1d5e"
}
```

I provide `status` and `version` (which refers to the git commit ref of the API at the time it was built). It's also worth mentioning that this value is not derived from an active `.git` repo being bundled with the APIs container for EC2. Instead, it is read (and stored in memory) on initialisation from a `version.txt` file (which is generated from the build process), and defaults to `__UNKNOWN__` in case of a read error, or the file does not exist.

. . .

That's all I can think of for now. Thanks for reading!

Please leave a comment below; let's have a conversation.