Best Practices for Designing a Pragmatic RESTful API

vinaysahni.com/best-practices-for-a-pragmatic-restful-api

Your data model has started to stabilize and you're in a position to create a public API for your web app. You realize it's hard to make significant changes to your API once it's released and want to get as much right as possible up front. Now, the internet has no shortage on opinions on API design. But, since there's no one widely adopted standard that works in all cases, you're left with a bunch of choices: What formats should you accept? How should you authenticate? Should your API be versioned?

Key requirements for the API

Many of the API design opinions found on the web are academic discussions revolving around subjective interpretations of fuzzy standards as opposed to what makes sense in the real world. My goal with this post is to describe best practices for a pragmatic API designed for today's web applications. I make no attempt to satisfy a standard if it doesn't feel right. To help guide the decision making process, I've written down some requirements that the API must strive for:

- It should use web standards where they *make sense*
- It should be friendly to the developer and be explorable via a browser address bar
- It should be simple, intuitive and consistent to make adoption not only easy but pleasant
- It should provide enough flexibility to power majority of the Enchant UI
- It should be efficient, while maintaining balance with the other requirements

An API is a developer's UI - just like any UI, it's important to ensure the user's experience is thought out carefully!

Use RESTful URLs and actions

If there's one thing that has gained wide adoption, it's RESTful principles. These were first introduced by Roy Fielding in Chapter 5 of his dissertation on network based software architectures.

The key principles of REST involve separating your API into logical resources. These resources are manipulated using HTTP requests where the method (GET, POST, PUT, PATCH, DELETE) has specific meaning.

But what can I make a resource? Well, these should be nouns (not verbs!) that make sense from the perspective of the API consumer. Although your internal models may map neatly to resources, it isn't necessarily a one-to-one mapping. The key here is to not leak irrelevant implementation details out to your API! Some of Enchant's nouns would be *ticket*, *user* and *group*.

Once you have your resources defined, you need to identify what actions apply to them and how those would map to your API. RESTful principles provide strategies to handle CRUD actions using HTTP methods mapped as follows:

- GET /tickets - Retrieves a list of tickets
- GET /tickets/12 - Retrieves a specific ticket
- POST /tickets - Creates a new ticket
- PUT /tickets/12 - Updates ticket #12
- PATCH /tickets/12 - Partially updates ticket #12
- DELETE /tickets/12 - Deletes ticket #12

The great thing about REST is that you're leveraging existing HTTP methods to implement significant functionality on just a single /tickets endpoint. There are no method naming conventions to follow and the URL structure is clean & clear. *REST FTW!*

Should the endpoint name be singular or plural? The keep-it-simple rule applies here. Although your inner-grammatician will tell you it's wrong to describe a single instance of a resource using a plural, the pragmatic answer is to keep the URL format consistent and always use a plural. Not having to deal with odd pluralization (person/people, goose/geese) makes the life of the API consumer better and is easier for the API provider to implement (as most modern frameworks will natively handle /tickets and /tickets/12 under a common controller).

But how do you deal with relations? If a relation can only exist within another resource, RESTful principles provide useful guidance. Let's look at this with an example. A ticket in Enchant consists of a number of messages. These messages can be logically mapped to the /tickets endpoint as follows:

- GET /tickets/12/messages - Retrieves list of messages for ticket #12
- GET /tickets/12/messages/5 - Retrieves message #5 for ticket #12
- POST /tickets/12/messages - Creates a new message in ticket #12
- PUT /tickets/12/messages/5 - Updates message #5 for ticket #12
- PATCH /tickets/12/messages/5 - Partially updates message #5 for ticket #12
- DELETE /tickets/12/messages/5 - Deletes message #5 for ticket #12

Alternatively, if a relation can exist independently of the resource, it makes sense to just include an identifier for it within the output representation of the resource. The API consumer would then have to hit the relation's endpoint. However, if the relation is commonly requested alongside the resource, the API could offer functionality to automatically embed the relation's representation and avoid the second hit to the API.

What about actions that don't fit into the world of CRUD operations?

This is where things can get fuzzy. There are a number of approaches:

1. Restructure the action to appear like a field of a resource. This works if the action doesn't take parameters. For example an *activate* action could be mapped to a boolean activated field and updated via a PATCH to the resource.
2. Treat it like a sub-resource with RESTful principles. For example, GitHub's API lets you star a gist with PUT /gists/:id/star and unstar with DELETE /gists/:id/star.
3. Sometimes you really have no way to map the action to a sensible RESTful structure. For example, a multi-resource search doesn't really make sense to be applied to a specific resource's endpoint. In this case, /search would make the most sense even though it isn't a resource. This is OK - just do what's right from the perspective of the API consumer and make sure it's documented clearly to avoid confusion.

SSL everywhere - all the time

Always use SSL. No exceptions. Today, your web APIs can get accessed from anywhere there is internet (like libraries, coffee shops, airports among others). Not all of these are secure. Many don't encrypt communications at all, allowing for easy eavesdropping or impersonation if authentication credentials are hijacked.

Another advantage of always using SSL is that guaranteed encrypted communications simplifies authentication efforts - you can get away with simple access tokens instead of having to sign each API request.

One thing to watch out for is non-SSL access to API URLs. Do**not** redirect these to their SSL counterparts. Throw a hard error instead! The last thing you want is for poorly configured clients to send requests to an unencrypted endpoint, just to be silently redirected to the actual encrypted endpoint.

Documentation

An API is only as good as its documentation. The docs should be easy to find and publically accessible. Most developers will check out the docs before attempting any integration effort. When the docs are hidden inside a PDF file or require signing in, they're not only difficult to find but also not easy to search.

The docs should show examples of complete request/response cycles. Preferably, the requests should be pastable examples - either links that can be pasted into a browser or curl examples that can be pasted into a terminal. GitHub and Stripe do a great job with this.

Once you release a public API, you've committed to not breaking things without notice. The documentation must include any deprecation schedules and details surrounding externally visible API updates. Updates should be delivered via a blog (i.e. a changelog) or a mailing list (preferably both!).

Versioning

Always version your API. Versioning helps you iterate faster and prevents invalid requests from hitting updated endpoints. It also helps smooth over any major API version transitions as you can continue to offer old API versions for a period of time.

There are mixed opinions around whether an API version should be included in the URL or in a header. Academically speaking, it should probably be in a header. However, the version needs to be in the URL to ensure browser explorability of the resources across versions (remember the API requirements specified at the top of this post?).

I'm a big fan of the approach that Stripe has taken to API versioning - the URL has a major version number (v1), but the API has date based sub-versions which can be chosen using a custom HTTP request header. In this case, the major version provides structural stability of the API as a whole while the sub-versions accounts for smaller changes (field deprecations, endpoint changes, etc).

An API is never going to be completely stable. Change is inevitable. What's important is how that change is managed. Well documented and announced multi-month deprecation schedules can be an acceptable practice for many APIs. It comes down to what is reasonable given the industry and possible consumers of the API.

Result filtering, sorting & searching

It's best to keep the base resource URLs as lean as possible. Complex result filters, sorting requirements and advanced searching (when restricted to a single type of resource) can all be easily implemented as query parameters on top of the base URL. Let's look at these in more detail:

Filtering: Use a unique query parameter for each field that implements filtering. For example, when requesting a list of tickets from the /tickets endpoint, you may want to limit these to only those in the open state. This could be accomplished with a request like GET /tickets?state=open. Here, state is a query parameter that implements a filter.

Sorting: Similar to filtering, a generic parameter sort can be used to describe sorting rules. Accommodate complex sorting requirements by letting the sort parameter take in list of comma separated fields, each with a possible unary negative to imply descending sort order. Let's look at some examples:

- GET /tickets?sort=-priority - Retrieves a list of tickets in descending order of priority
- GET /tickets?sort=-priority,created_at - Retrieves a list of tickets in descending order of priority. Within a specific priority, older tickets are ordered first

Searching: Sometimes basic filters aren't enough and you need the power of full text search. Perhaps you're already using ElasticSearch or another Lucene based search technology. When full text search is used as a mechanism of retrieving resource instances for a specific type of resource, it can be exposed on the API as a query parameter on the resource's endpoint. Let's say q. Search queries should be passed straight to the search engine and API output should be in the same format as a normal list result.

Combining these together, we can build queries like:

- GET /tickets?sort=-updated_at - Retrieve recently updated tickets
- GET /tickets?state=closed&sort=-updated_at - Retrieve recently closed tickets
- GET /tickets?q=return&state=open&sort=-priority,created_at - Retrieve the highest priority open tickets mentioning the word 'return'

Aliases for common queries

To make the API experience more pleasant for the average consumer, consider packaging up sets of conditions into easily accessible RESTful paths. For example, the recently closed tickets query above could be packaged up as GET /tickets/recently_closed

Limiting which fields are returned by the API

The API consumer doesn't always need the full representation of a resource. The ability select and chose returned fields goes a long way in letting the API consumer minimize network traffic and speed up their own usage of the API.

Use a fields query parameter that takes a comma separated list of fields to include. For example, the following request would retrieve just enough information to display a sorted listing of open tickets:

GET /tickets?fields=id,subject,customer_name,updated_at&state=open&sort=-updated_at

Updates & creation should return a resource representation

A PUT, POST or PATCH call may make modifications to fields of the underlying resource that weren't part of the provided parameters (for example: created_at or updated_at timestamps). To prevent an API consumer from having to hit the API again for an updated representation, have the API return the updated (or created) representation as part of the response.

In case of a POST that resulted in a creation, use a HTTP 201 status code and include a Location header that points to the URL of the new resource.

Should you HATEOAS?

There are a lot of mixed opinions as to whether the API consumer should create links or whether links should be provided to the API. RESTful design principles specify HATEOAS which roughly states that interaction with an endpoint should be defined within metadata that comes with the output representation and not based on out-of-band information.

Although the web generally works on HATEOAS type principles (where we go to a website's front page and follow links based on what we see on the page), I don't think we're ready for HATEOAS on APIs just yet. When browsing a website, decisions on what links will be clicked are made at run time. However, with an API, decisions as to what requests will be sent are made when the API integration code is written, not at run time. Could the decisions be deferred to run time? Sure, however, there isn't much to gain going down that route as code would still not be able to handle significant API changes without breaking. That said, I think HATEOAS is promising but not ready for prime time just yet. Some more effort has to be put in to define standards and tooling around these principles for its potential to be fully realized.

For now, it's best to assume the user has access to the documentation & include resource identifiers in the output representation which the API consumer will use when crafting links. There are a couple of advantages of sticking to identifiers - data flowing over the network is minimized and the data stored by API consumers is also minimized (as they are storing small identifiers as opposed to URLs that contain identifiers).

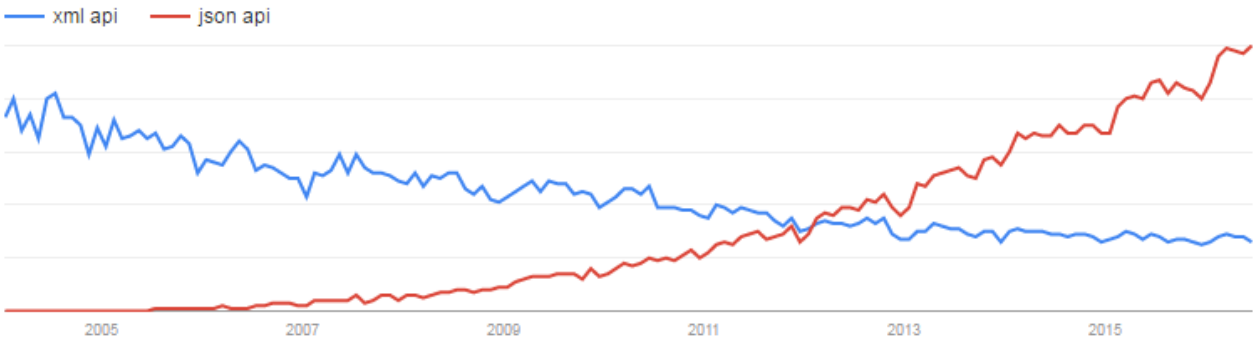
Also, given this post advocates version numbers in the URL, it makes more sense in the long term for the API consumer to store resource identifiers as opposed to URLs. After all, the identifier is stable across versions but the URL representing it is not!

JSON only responses

It's time to leave XML behind in APIs. It's verbose, it's hard to parse, it's hard to read, its data model isn't compatible with how most programming languages model data and its extendibility advantages are irrelevant when your output representation's primary needs are serialization from an internal representation.

I'm not going to put much effort into explaining the above breathful as it looks like others (YouTube, & Box) have already started the XML exodus.

I'll just leave you the following Google Trends chart (XML API vs JSON API) as food for thought:



However, if your customer base consists of a large number of enterprise customers, you may find yourself having to support XML anyway. If you must do this, you'll find yourself with a new question:

Should the media type change based on Accept headers or based on the URL? To ensure browser explorability, it should be in the URL. The most sensible option here would be to append a .json or .xml extension to the endpoint URL.

snake_case vs camelCase for field names

If you're using JSON (*JavaScript* Object Notation) as your primary representation format, the "right" thing to do is to follow JavaScript naming conventions - and that means camelCase for field names! If you then go the route of building client libraries in various languages, it's best to use idiomatic naming conventions in them - camelCase for C# & Java, snake_case for python & ruby.

Food for thought: I've always felt that snake_case is easier to read than JavaScript's convention of camelCase. I just didn't have any evidence to back up my gut feelings, until now. Based on an eye tracking study on camelCase and snake_case (PDF) from 2010, **snake_case is 20% easier to read than camelCase**! That impact on readability would affect API explorability and examples in documentation.

Many popular JSON APIs use snake_case. I suspect this is due to serialization libraries following naming conventions of the underlying language they are using. Perhaps we need to have JSON serialization libraries handle naming convention transformations.

Pretty print by default & ensure gzip is supported

An API that provides white-space compressed output isn't very fun to look at from a browser. Although some sort of query parameter (like ?pretty=true) could be provided to enable pretty printing, an API that pretty prints by default is much more approachable. The cost of the extra data transfer is negligible, especially when you compare to the cost of not implementing gzip.

Consider some use cases: What if an API consumer is debugging and has their code print out data it received from the API - It will be readable by default. Or if the consumer grabbed the URL their code was generating and hit it directly from the browser - it will be readable by default. These are small things. Small things that make an API pleasant to use!

But what about all the extra data transfer?

Let's look at this with a real world example. I've pulled some data from GitHub's API, which uses pretty print by default. I'll also be doing some gzip comparisons:

```
$ curl https://api.github.com/users/veesahni > with-whitespace.txt
$ ruby -r json -e 'puts JSON JSON.parse(STDIN.read)' < with-whitespace.txt > without-whitespace.txt
$ gzip -c with-whitespace.txt > with-whitespace.txt.gz
$ gzip -c without-whitespace.txt > without-whitespace.txt.gz
```

The output files have the following sizes:

- without-whitespace.txt - 1252 bytes
- with-whitespace.txt - 1369 bytes
- without-whitespace.txt.gz - 496 bytes
- with-whitespace.txt.gz - 509 bytes

In this example, the whitespace increased the output size by 8.5% when gzip is not in play and 2.6% when gzip is in play. On the other hand, the act of **gzipping in itself provided over 60% in bandwidth savings**. Since the cost of pretty printing is relatively small, it's best to pretty print by default and ensure gzip compression is supported!

To further hammer in this point, Twitter found that there was an 80% savings (in some cases) when enabling gzip compression on their Streaming API. Stack Exchange went as far as to never return a response that's not compressed!

Don't use an envelope by default, but make it possible when needed

Many APIs wrap their responses in envelopes like this:

```
{
 "data" : {
  "id" : 123,
  "name" : "John"
 }
}
```

There are a couple of justifications for doing this - it makes it easy to include additional metadata or pagination information, some REST clients don't allow easy access to HTTP headers & JSONP requests have no access to HTTP headers. However, with standards that are being rapidly adopted like CORS and the Link header from RFC 5988, enveloping is starting to become unnecessary.

We can future proof the API by staying envelope free by default and enveloping only in exceptional cases.

How should an envelope be used in the exceptional cases?

There are 2 situations where an envelope is really needed - if the API needs to support cross domain requests over JSONP or if the client is incapable of working with HTTP headers.

JSONP requests come with an additional query parameter (usually named callback or jsonp) representing the name of the callback function. If this parameter is present, the API should switch to a full envelope mode where it always responds with a 200 HTTP status code and passes the real status code in the JSON payload. Any additional HTTP headers that would have been passed alongside the response should be mapped to JSON fields, like so:

```
callback_function({
  status_code: 200,
  next_page: "https://..",
  response: {
    ... actual JSON response body ...
  }
})
```

Similarly, to support limited HTTP clients, allow for a special query parameter? envelope=true that would trigger full enveloping (without the JSONP callback function).

JSON encoded POST, PUT & PATCH bodies

If you're following the approach in this post, then you've embraced JSON for all API output. Let's consider JSON for API input.

Many APIs use URL encoding in their API request bodies. URL encoding is exactly what it sounds like - request bodies where key value pairs are encoded using the same conventions as one would use to encode data in URL query parameters. This is simple, widely supported and gets the job done.

However, URL encoding has a few issues that make it problematic. It has no concept of data types. This forces the API to parse integers and booleans out of strings. Furthermore, it has no real concept of hierarchical structure. Although there are some conventions that can build some structure out of key value pairs (like appending [ ] to a key to represent an array), this is no comparison to the native hierarchical structure of JSON.

If the API is simple, URL encoding may suffice. However, complex APIs should stick to JSON for their API input. Either way, pick one and be consistent throughout the API.

An API that accepts JSON encoded POST, PUT & PATCH requests should also require the Content-Type header be set to application/json or throw a 415 Unsupported Media Type HTTP status code.

Envelope loving APIs typically include pagination data in the envelope itself. And I don't blame them - until recently, there weren't many better options. The right way to include pagination details today is using the Link header introduced by RFC 5988.

An API that uses the Link header can return a set of ready-made links so the API consumer doesn't have to construct links themselves. This is especially important when pagination is cursor based. Here is an example of a Link header used properly, grabbed from GitHub's documentation:

```
Link: <https://api.github.com/user/repos?page=3&per_page=100>; rel="next",
<https://api.github.com/user/repos?page=50&per_page=100>; rel="last"
```

But this isn't a complete solution as many APIs do like to return the additional pagination information, like a count of the total number of available results. An API that requires sending a count can use a custom HTTP header like X-Total-Count.

Auto loading related resource representations

There are many cases where an API consumer needs to load data related to (or referenced) from the resource being requested. Rather than requiring the consumer to hit the API repeatedly for this information, there would be a significant efficiency gain from allowing related data to be returned and loaded alongside the original resource on demand.

However, as this does go against some RESTful principles, we can minimize our deviation by only doing so based on an embed (or expand) query parameter.

In this case, embed would be a comma separated list of fields to be embedded. Dot-notation could be used to refer to sub-fields. For example:

GET /tickets/12?embed=customer.name,assigned_user

This would return a ticket with additional details embedded, like:

```
{
 "id" : 12,
 "subject" : "I have a question!",
 "summary" : "Hi, ....",
 "customer" : {
  "name" : "Bob"
 },
 assigned_user: {
  "id" : 42,
  "name" : "Jim",
 }
}
```

Of course, ability to implement something like this really depends on internal complexity. This kind of embedding can easily result in an N+1 select issue.

Overriding the HTTP method

Some HTTP clients can only work with simple GET and POST requests. To increase accessibility to these limited clients, the API needs a way to override the HTTP method. Although there aren't any hard standards here, the popular convention is to accept a request header X-HTTP-Method-Override with a string value containing one of PUT, PATCH or DELETE.

Note that the override header should **only** be accepted on POST requests. GET requests should never change data on the server!

Rate limiting

To prevent abuse, it is standard practice to add some sort of rate limiting to an API. RFC 6585 introduced a HTTP status code 429 Too Many Requests to accommodate this.

However, it can be very useful to notify the consumer of their limits before they actually hit it. This is an area that currently lacks standards but has a number of popular conventions using HTTP response headers.

At a minimum, include the following headers (using Twitter's naming conventions as headers typically don't have mid-word capitalization):

- X-Rate-Limit-Limit - The number of allowed requests in the current period
- X-Rate-Limit-Remaining - The number of remaining requests in the current period
- X-Rate-Limit-Reset - The number of seconds left in the current period

Why is number of seconds left being used instead of a time stamp for X-Rate-Limit-Reset?

A timestamp contains all sorts of useful but unnecessary information like the date and possibly the time-zone. An API consumer really just wants to know when they can send the request again & the number of seconds answers this question with minimal additional processing on their end. It also avoids issues related to clock skew.

Some APIs use a UNIX timestamp (seconds since epoch) for X-Rate-Limit-Reset. Don't do this!

Why is it bad practice to use a UNIX timestamp for X-Rate-Limit-Reset?

The HTTP spec already specifies using RFC 1123 date formats (currently being used in Date, If-Modified-Since & Last-Modified HTTP headers). If we were to specify a new HTTP header that takes a timestamp of some sort, it should follow RFC 1123 conventions instead of using UNIX timestamps.

Authentication

A RESTful API should be stateless. This means that request authentication should not depend on cookies or sessions. Instead, each request should come with some sort authentication credentials.

By always using SSL, the authentication credentials can be simplified to a randomly generated access token that is delivered in the user name field of HTTP Basic Auth. The great thing about this is that it's completely browser explorable - the browser will just popup a prompt asking for credentials if it receives a 401 Unauthorized status code from the server.

However, this token-over-basic-auth method of authentication is only acceptable in cases where it's practical to have the user copy a token from an administration interface to the API consumer environment. In cases where this isn't possible, OAuth 2 should be used to provide secure token transfer to a third party. OAuth 2 uses Bearer tokens & also depends on SSL for its underlying transport encryption.

An API that needs to support JSONP will need a third method of authentication, as JSONP requests cannot send HTTP Basic Auth credentials or Bearer tokens. In this case, a special query parameter access_token can be used. Note: there is an inherent security issue in using a query parameter for the token as most web servers store query parameters in server logs.

For what it's worth, all three methods above are just ways to transport the token

across the API boundary. The actual underlying token itself could be identical.

Caching

HTTP provides a built-in caching framework! All you have to do is include some additional outbound response headers and do a little validation when you receive some inbound request headers.

There are 2 approaches: ETag and Last-Modified

ETag: When generating a response, include a HTTP header ETag containing a hash or checksum of the representation. This value should change whenever the output representation changes. Now, if an inbound HTTP requests contains a If-None-Match header with a matching ETag value, the API should return a 304 Not Modified status code instead of the output representation of the resource.

Last-Modified: This basically works like to ETag, except that it uses timestamps. The response header Last-Modified contains a timestamp in RFC 1123 format which is validated against If-Modified-Since. Note that the HTTP spec has had 3 different acceptable date formats and the server should be prepared to accept any one of them.

Errors

Just like an HTML error page shows a useful error message to a visitor, an API should provide a useful error message in a known consumable format. The representation of an error should be no different than the representation of any resource, just with its own set of fields.

The API should always return sensible HTTP status codes. API errors typically break down into 2 types: 400 series status codes for client issues & 500 series status codes for server issues. At a minimum, the API should standardize that all 400 series errors come with consumable JSON error representation. If possible (i.e. if load balancers & reverse proxies can create custom error bodies), this should extend to 500 series status codes.

A JSON error body should provide a few things for the developer - a useful error message, a unique error code (that can be looked up for more details in the docs) and possibly a detailed description. JSON output representation for something like this would look like:

```
{
 "code" : 1234,
 "message" : "Something bad happened :(",
 "description" : "More details about the error here"
}
```

Validation errors for PUT, PATCH and POST requests will need a field breakdown. This is best modeled by using a fixed top-level error code for validation failures and providing the detailed errors in an additional errors field, like so:

```
{
 "code" : 1024,
 "message" : "Validation Failed",
 "errors" : [
   {
    "code" : 5432,
    "field" : "first_name",
    "message" : "First name cannot have fancy characters"
   },
   {
     "code" : 5622,
     "field" : "password",
     "message" : "Password cannot be blank"
   }
 ]
}
```

HTTP status codes

HTTP defines a bunch of meaningful status codes that can be returned from your API. These can be leveraged to help the API consumers route their responses accordingly. I've curated a short list of the ones that you definitely should be using:

- 200 OK - Response to a successful GET, PUT, PATCH or DELETE. Can also be used for a POST that doesn't result in a creation.
- 201 Created - Response to a POST that results in a creation. Should be combined with a Location header pointing to the location of the new resource
- 204 No Content - Response to a successful request that won't be returning a body (like a DELETE request)
- 304 Not Modified - Used when HTTP caching headers are in play
- 400 Bad Request - The request is malformed, such as if the body does not parse
- 401 Unauthorized - When no or invalid authentication details are provided. Also useful to trigger an auth popup if the API is used from a browser
- 403 Forbidden - When authentication succeeded but authenticated user

doesn't have access to the resource
- 404 Not Found - When a non-existent resource is requested
- 405 Method Not Allowed - When an HTTP method is being requested that isn't allowed for the authenticated user
- 410 Gone - Indicates that the resource at this end point is no longer available. Useful as a blanket response for old API versions
- 415 Unsupported Media Type - If incorrect content type was provided as part of the request
- 422 Unprocessable Entity - Used for validation errors
- 429 Too Many Requests - When a request is rejected due to rate limiting

In Summary

An API is a user interface for developers. Put the effort in to ensure it's not just functional but pleasant to use.

Developers from Spotify, IBM and eBay already follow this blog.
You should too.

x
Developers from Spotify, IBM and eBay already follow this blog.
You should too.