SIGN IN

Blog / Programming / REST API Design Best Practices

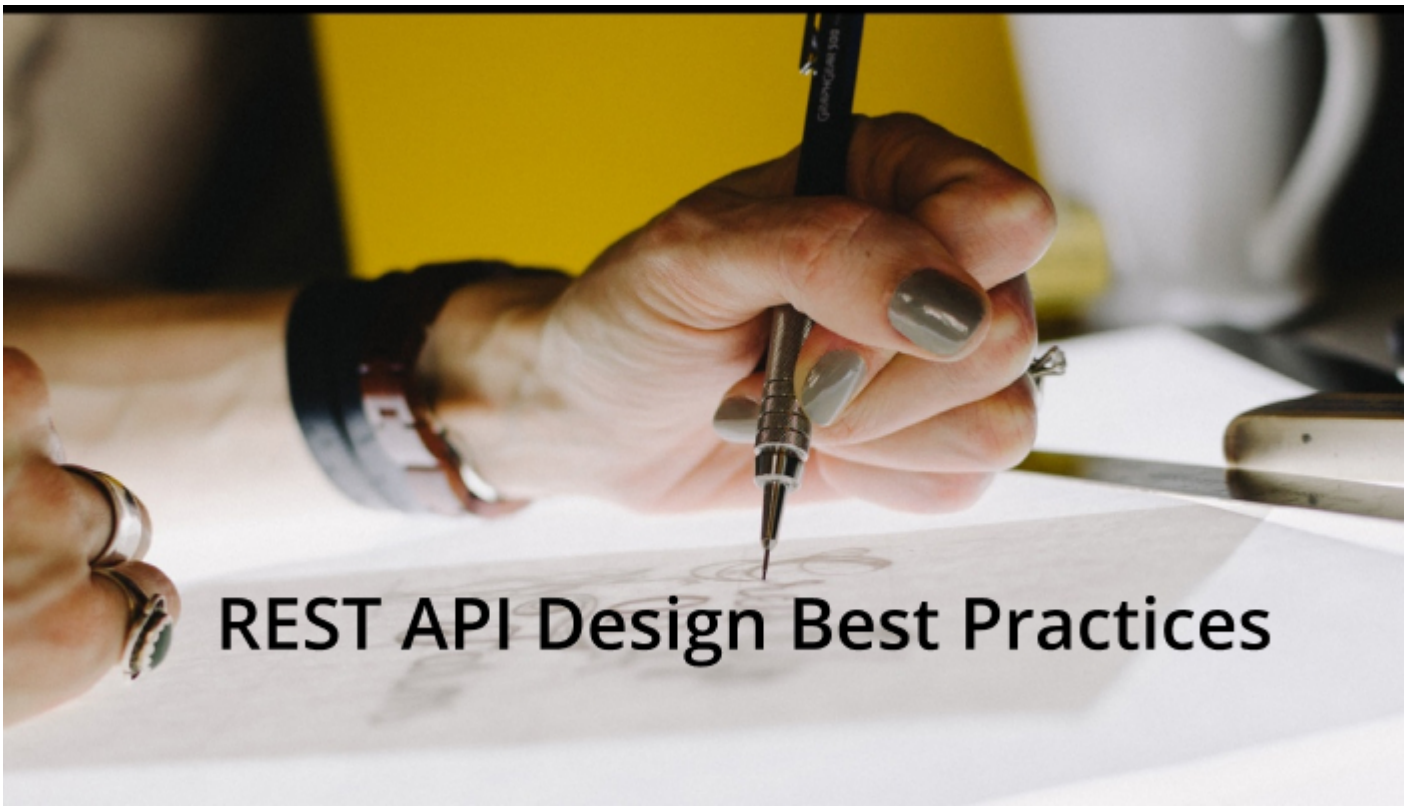
REST API Design Best Practices

3 1

Category: Programming

By John Au-Yeung Follow

14,346 February 28, 2020



REST API Design Best Practices

REST APIs are one of the most popular types of **APIs**. They're common and supported and easy to create. Like with anything else, we should think hard when we design them. In this article, we'll look at **how to design REST APIs,** which follow conventions and are easy to understand for everyone who's maintaining and using them.

Use JSON

We should use JSON for requests and responses. JSON can be easily parsed on the client and server-side since almost all libraries and frameworks that detail with send and receiving requests automatically parse JSON. Other formats aren't well supported and they require extra work for other people to parse them into usable data. `application/json` is a generic MIME type which makes it practical to use.

Use Nouns instead of Verbs

REST API should use HTTP verbs to represent their actions. Therefore, we don't need to use nouns in our URL to let developers know that we're doing some action. We should use GET requests to return data with our API. POST requests take data and create a new entity with our API. For example, we can use it to save a new entry to a database. PUT or PATCH requests take in data and then update an existing entity with the submitted data. DELETE requests are used for deleting items from a data source like a database.

Therefore, the correct way to design our routes would be:

- GET /posts/123
- DELETE /posts/123
- POST /posts
- PUT /posts/123
- PATCH /posts/123

The wrong way to design our routes would be:

- GET /addPost/123 (GET should be only used to read data and never to change its state in any way)
- GET /DeletePosts/123
- POST /DeleteAllPosts

- POST /posts/123/delete

As we can see, the incorrect way is much more confusing than the correct way. It's just much harder to understand and there's no consistency. We're using POST requests for multiple purposes, which isn't desirable in any scenario.

Name the Collections Using Plural Nouns

As we can see from the correct example above, we named the routes with plurals. This lets everyone know that we're manipulating a collection with our routes. Rarely do we only have one entry in our database in our collection, and it's probably a coincidence. Therefore, to represent what we actually have in our data source, we should name our API routes with plurals.

Use Resource Nesting to Show Relations or Hierarchy

If we have child entities that we want to return with parent entities, we can extend our route with extra entities by putting them after the end of the parent route.

For example, if we have comments with our posts, we can write:

- /posts/1/comments

to get all the comments associated with the post with ID 1.

If we want to get a specific comment from the comments list, we can write:

- /posts/1/comments/1

Then we can get a comment with ID 1 from the post with ID 1 if comment with ID 1 is actually a child of the post with ID 1. Since deep nesting is taxing on people's brains, it's a good idea to limit the nesting to one level in the REST API.

Alternatively, we can use query strings like:

- /comments?post=1

to get the comments for the post with ID 1.

Error Handling

We should return error codes that indicate the type of error that's encountered. The common ones include:

- 400 for bad request;
- 401 for unauthorized;
- 403 for forbidden;
- 404 for not found;
- 405 for HTTP method not accepted;
- 406 for the type of content that isn't accepted according to the accept header sent in the request;
- 409 for conflict between an existing entity and the request submitted.

400 series errors are errors that are caused by the client-side when making a request. Common server-side errors, which are 500 series errors, include 500 for interval server error, 501 for not implemented, 502 for a bad gateway, 503 for service unavailable, and 504 for gateway timeout. In addition to the HTTP response code, we should respond with some human-readable message so that the error can be corrected.

Filtering, Sorting, Pagination, and Field Selection

We should let the user filter information based on the query they submitted. This means that we don't have to return data that users don't want and therefore the responses are smaller. Sorting lets users get data in a particular order. This is useful for users and

developers to find data quickly. Pagination is used to limit the number of items shown at one time. Users and developers can go to different page numbers to get more data.

Example of filtering include:

- GET /posts?user=123

Sorting examples include:

- GET /posts?sort=published_date=desc

Pagination:

- GET /posts?page=1&limit=100

We get the posts by page number and limit them to a certain quantity.

All together, we can write:

- GET /posts?user=123&sort=published_date=desc&page=1&limit=100

to get the posts that are published by user 123, sorted by published date descending, page 1 and 100 entries per page. It's also useful to accept a field selection query string so that we don't have to return all the fields.

For example, we can write:

- GET /posts?fields=title,body

We can accept something like what we have above to only return the `title` and `body` fields of the posts.

Versioning

We can version APIs by having different routes for different versions.

For example, we can have routes that look like:

- GET /api.example.com/v1/posts

so that users can use the version of 1 of our API.

API Documentation

The API documentation is important so that everyone knows how to submit request header and body to the API and what responses they get back.

The documentation should have a list of routes with all those details and it's even better to have an API sandbox so that people can try out our APIs.

Using SSL/TLS

In a production environment, we should use SSL/TLS to send and receive data securely between the client and the server.

Simple Example

We can use Express to make a simple example of an API that follows the design principles we mentioned above. The following code is a simple example that follows the principles:

```
1  const express = require('express');
2  const bodyParser = require('body-parser');
3
4  const app = express();
5
6  app.use(bodyParser.json());
7  app.use(bodyParser.urlencoded({ extended: true }));
8
9  app.get('/v1/posts', (req, res) => {
```

```
  //... get data
  res.json({ message: 'post' });
});

app.get('/v1/posts/:id', (req, res) => {
  //... get data
  res.json({ message: 'post' });
});

app.post('/v1/posts', (req, res) => {
   //... saves data
  res.json({ message: 'post added' });
});

app.put('/v1/posts/:id', (req, res) => {
  //... updates data
  res.json({ message: 'post updated' });
});

app.delete('/v1/posts/:id', (req, res) => {
  //... deletes data
  res.json({ message: 'post deleted' });
});

app.get('/v1/posts/:id/comments', (req, res) => {
  //... get data
  res.json({ message: 'comments' });
});

app.listen(3000, () => console.log('server started'));
```

The API example is written with Express and JavaScript. In the code above, we have the routes that have the version number, the names in the route URLs are all plural and has no verbs. The request types also represent what they're doing. Also, when we have child entities in our routes, we put them after the parent entity. We also use the `json-parser` Express middleware to accept JSON. We return our response in JSON.

Conclusion

To make our REST APIs easy to read and consistent, we have to follow some conventions that are generally accepted. We don't want verbs in our name. Instead, we want to use HTTP verbs to represent them. Child entities come after a parent in route paths. Version numbers are needed if there're multiple versions of the API. Error codes should represent what's actually wrong with our system. We should also return meaningful messages to users so they can take corrective action. JSON should be used when accepting requests and returning responses.

About the author

John Au-Yeung

Registered 06-01-2020 | Last seen 1 year ago

0 Comments | 4 Publications

15

Share Share Share

Vacancies

Python Back-end web developer (remote) Soshace (Python, Django)

Python / Django Developer PULSE (Python, Django, MySQL)

Front-End React developer (remote) Soshace (React, JS, HTML, CSS)

Full-stack Angular and Node.js developer ... Soshace (JS, Node.js, Angular, HTML, CSS)

Stay Informed

It's important to keep up
with industry - subscribe!

Your Name

email@example.com

I agree with Privacy Policy

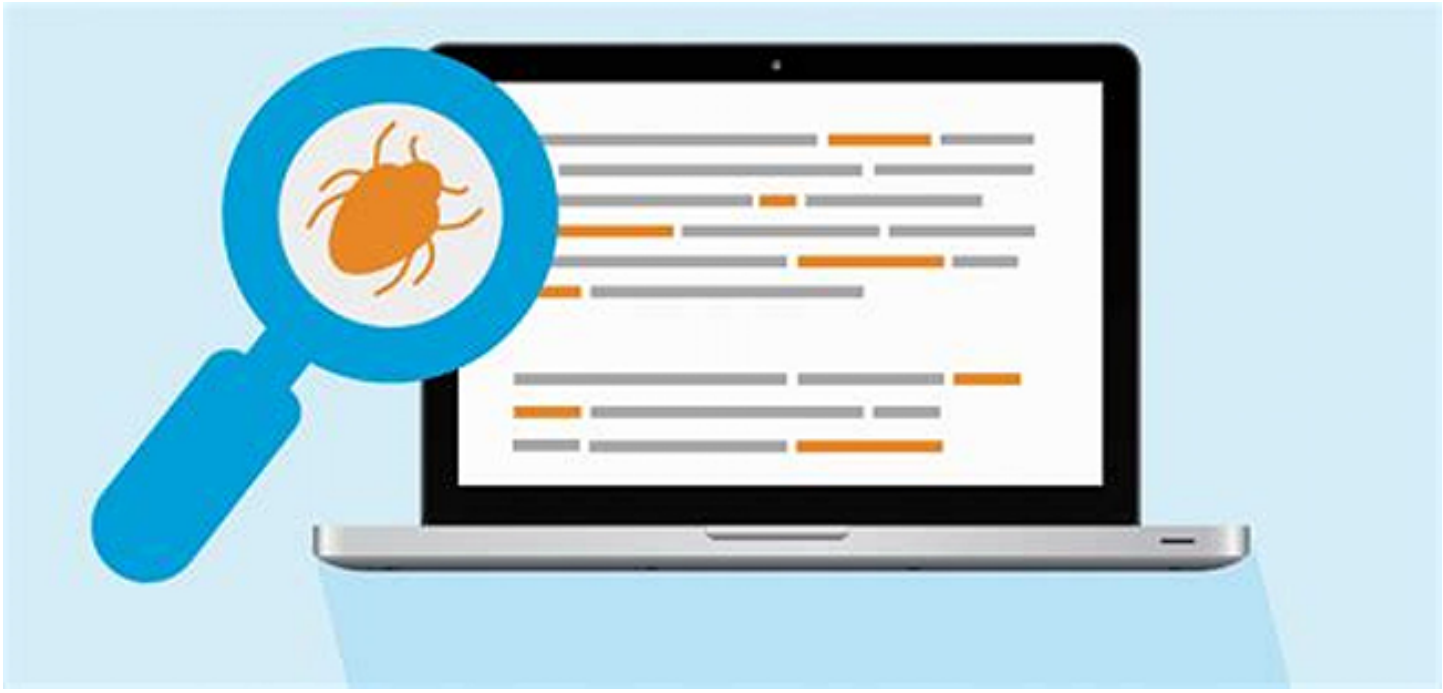
SUBSCRIBE!

Related articles

Building Rest API With Django Using Django Rest Framework and Django Rest Auth

Building Rest API seems to be complex when trying to achieve this on your own with Django, thanks to the Django Rest framework project which has ...

Django Programming Python



How To Use Prospector For Python Static Code Analysis

Prospector is a tool that analyzes Python code, outputs information about the errors, potential problems, convention violation, and complexity of the ...

Programming

MORE RELATED ARTICLES

No comments yet

Type your comment...

SEND

Categories

Programming (172) JavaScript (129) Tips (76)

React (73) Beginners (62) Node.js (47)

Project Managment (43) Interview (42) Human Resources (38)

Remote Job (33) Python (27) POS Tutorial (18)

React Lessons (17) Node.js Lessons (16) Events (16)

Trends (16) Job (15) Freelance (14)

React Native (12) Java (12) React Native Lessons (11)

Startups (11) Entrepreneurship (10) Regulations (10)

Node.js (8) CSS (7) Soshace (6)

Vue (6) Git (6) Comics (6)

Podcasts (5) Fortune 500 (5) Angular (5)

Blogs (4) PHP (3) Java Spring (3)

GraphQL (3)

Flutter (3)

Flask (3)

Django (2)

SEO (2)

Next.js (1)

ASP.NET (1)

Laravel (1)

GraphQl (1)

WordPress (1)

CONTACTS

197183, Russia, Saint-Petersburg,
Polevaya Sabirovskaya street, 54A, office 335

sales@soshace.com
hr@soshace.com
support@soshace.com

Write For Us

COMPANY

Reviews
Technologies
Blog
Contact Us
About Us
FAQ For Talents
FAQ For Customers
Privacy Policy

SERVICES

For Clients
For Developers
All Developers
Jobs

IN-DEMAND

React.Js Developer
Angular Developer
Node.Js Developer
Python Developer

Find us on: