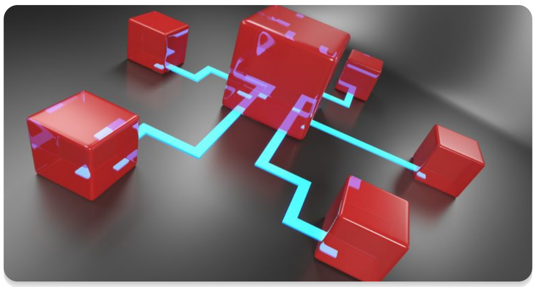
code-for-a-living MARCH 2, 2020

Best practices for REST API design



In this article, we'll look at how to design REST APIs to be easy to understand for anyone consuming them, future-proof, and secure and fast since they serve data to clients that may be confidential.

John Au-Yeung and **Ryan Donovan**

REST APIs are one of the most common kinds of web services available today. They allow various clients including browser apps to communicate with a server via the REST API. Therefore, it's very important to design REST APIs properly so that we won't run into problems down the road. We have to take into account security, performance, and ease of use for API consumers.

Otherwise, we create problems for clients that use our APIs, which isn't pleasant and detracts people from using our API. If we don't follow commonly accepted conventions, then we confuse the maintainers of the API and the clients that use them since it's different from what everyone expects.

In this article, we'll look at how to design REST APIs to be easy to understand for anyone consuming them, future-proof, and secure and fast since they serve data to clients that may be confidential.

What is a REST API?

A REST API is an application programming interface that conforms to specific architectural constraints, like stateless communication and cacheable data. It is not a protocol or standard. While REST APIs can be accessed through a number of communication protocols, most commonly, they are called over HTTPS, so the guidelines below apply to REST API endpoints that will be called over the internet.

Note: For REST APIs called over the internet, you'll like want to follow the best practices for REST API authentication.

Accept and respond with JSON

REST APIs should accept JSON for request payload and also send responses to JSON. JSON is the standard for transferring data. Almost every networked technology can use it: JavaScript has built-in methods to encode and decode JSON either through the Fetch API or another HTTP client. Server-side technologies have libraries that can decode JSON without doing much work.

There are other ways to transfer data. XML isn't widely supported by frameworks without transforming the data ourselves to something that can be used, and that's usually JSON. We can't manipulate this data as easily on the client-side, especially in browsers. It ends up being a lot of extra work just to do normal data transfer.

Form data is good for sending data, especially if we want to send files. But for text and numbers, we don't need form data to transfer those since—with most frameworks—we can transfer JSON by just getting the data from it directly on the client side. It's by far the most straightforward to do so.

To make sure that when our REST API app responds with JSON that clients interpret it as such, we should set `Content-Type` in the response header to `application/json` after the request is made. Many server-side app frameworks set the response header automatically. Some HTTP clients look at the `Content-Type` response header and parse the data according to that format.

The only exception is if we're trying to send and receive files between client and server. Then we need to handle file responses and send form data from client to server. But that is a topic for another time.

We should also make sure that our endpoints return JSON as a response. Many server-side frameworks have this as a built-in feature.

Let's take a look at an example API that accepts JSON payloads. This example will use the Express back end framework for Node.js. We can use the `body-parser` middleware to parse the JSON request body, and then we can call the `res.json` method with the object that we want to return as the JSON response as follows:

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();

app.use(bodyParser.json());

app.post('/', (req, res) => {
  res.json(req.body);
});

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

`bodyParser.json()` parses the JSON request body string into a JavaScript object and then assigns it to the `req.body` object.

Set the `Content-Type` header in the response to `application/json; charset=utf-8` without any changes. The method above applies to most other back end frameworks.

stack overflow
FOR TEAMS
Collaboration without distraction
Enterprise-level digital collaboration made easy
Learn how →

Use nouns instead of verbs in endpoint paths

We shouldn't use verbs in our endpoint paths. Instead, we should use the nouns which represent the entity that the endpoint that we're retrieving or manipulating as the pathname.

This is because our HTTP request method already has the verb. Having verbs in our API endpoint paths isn't useful and it makes it unnecessarily long since it doesn't convey any new information. The chosen verbs could vary by the developer's whim. For instance, some like 'get' and some like 'retrieve', so it's just better to let the HTTP GET verb tell us what and endpoint does.

The action should be indicated by the HTTP request method that we're making. The most common methods include GET, POST, PUT, and DELETE.

- GET retrieves resources.
- POST submits new data to the server.
- PUT updates existing data.
- DELETE removes data.

The verbs map to CRUD operations.

With the two principles we discussed above in mind, we should create routes like GET `/articles/` for getting news articles. Likewise, POST `/articles/` is for adding a new article , PUT `/articles/:id` is for updating the article with the given `id`. DELETE `/articles/:id` is for deleting an existing article with the given ID.

`/articles` represents a REST API resource. For instance, we can use Express to add the following endpoints for manipulate articles as follows:

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();

app.use(bodyParser.json());

app.get('/articles', (req, res) => {
  const articles = [];
  // code to retrieve an article...
  res.json(articles);
});

app.post('/articles', (req, res) => {
  // code to add a new article...
  res.json(req.body);
});

app.put('/articles/:id', (req, res) => {
  const { id } = req.params;
  // code to update an article...
  res.json(req.body);
});

app.delete('/articles/:id', (req, res) => {
  const { id } = req.params;
  // code to delete an article...
  res.json({ deleted: id });
});

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

In the code above, we defined the endpoints to manipulate articles. As we can see, the path names do not have any verbs in them. All we have are nouns. The verbs are in the HTTP verbs.

The POST, PUT, and DELETE endpoints all take JSON as the request body, and they all return JSON as the response, including the GET endpoint.

Use logical nesting on endpoints

When designing endpoints, it makes sense to group those that contain associated information. That is, if one object can contain another object, you should design the endpoint to reflect that. This is good practice regardless of whether your data is structured like this in your database. In fact, it may be advisable to avoid mirroring your database structure in your endpoints to avoid giving attackers unnecessary information.

For example, if we want an endpoint to get the comments for a news article, we should append the `/comments` path to the end of the

`/articles` path. We can do that with the following code in Express:

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();

app.use(bodyParser.json());

app.get('/articles/:articleId/comments', (req, res) => {
  const { articleId } = req.params;
  const comments = [];
  // code to get comments by articleId
  res.json(comments);
});


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

In the code above, we can use the GET method on the path `'/articles/:articleId/comments'`. We get `comments` on the article identified by `articleId` and then return it in the response. We add `'comments'` after the `'/articles/:articleId'` path segment to indicate that it's a child resource of `/articles`.

This makes sense since `comments` are the children objects of the `articles`, assuming each article has its own comments. Otherwise, it's confusing to the user since this structure is generally accepted to be for accessing child objects. The same principle also applies to the POST, PUT, and DELETE endpoints. They can all use the same kind of nesting structure for the path names.

However, nesting can go too far. After about the second or third level, nested endpoints can get unwieldy. Consider, instead, returning the URL to those resources instead, especially if that data is not necessarily contained within the top level object.

For example, suppose you wanted to return the author of particular comments. You could use

`/articles/:articleId/comments/:commentId/author`. But that's getting out of hand. Instead, return the URI for that particular user within the JSON response instead:

`"author": "/users/:userId"`

Handle errors gracefully and return standard error codes

To eliminate confusion for API users when an error occurs, we should handle errors gracefully and return HTTP response codes that indicate what kind of error occurred. This gives maintainers of the API enough information to understand the problem that's occurred. We don't want errors to bring down our system, so we can leave them unhandled, which means that the API consumer has to handle them.

Common error HTTP status codes include:

- 400 Bad Request – This means that client-side input fails validation.
- 401 Unauthorized – This means the user isn't not authorized to access a resource. It usually returns when the user isn't authenticated.
- 403 Forbidden – This means the user is authenticated, but it's not allowed to access a resource.
- 404 Not Found – This indicates that a resource is not found.
- 500 Internal server error – This is a generic server error. It probably shouldn't be thrown explicitly.
- 502 Bad Gateway – This indicates an invalid response from an upstream server.
- 503 Service Unavailable – This indicates that something unexpected happened on server side (It can be anything like server overload, some parts of the system failed, etc.).

We should be throwing errors that correspond to the problem that our app has encountered. For example, if we want to reject the data from the request payload, then we should return a 400 response as follows in an Express API:

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();

// existing users
const users = [
  { email: 'abc@foo.com' }
]

app.use(bodyParser.json());

app.post('/users', (req, res) => {
  const { email } = req.body;
  const userExists = users.find(u => u.email === email);
  if (userExists) {
    return res.status(400).json({ error: 'User already exists'
})
  }
  res.json(req.body);
});


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

In the code above, we have a list of existing users in the `users` array with the given email.

Then if we try to submit the payload with the `email` value that already exists in `users`, we'll get a 400 response status code with a `'User already exists'` message to let users know that the user already exists. With that information, the user can correct the action by changing the email to something that doesn't exist.

Error codes need to have messages accompanied with them so that the maintainers have enough information to troubleshoot the issue, but attackers can't use the error content to carry our attacks like stealing information or bringing down the system.

Whenever our API does not successfully complete, we should fail gracefully by sending an error with information to help users make corrective action.

Allow filtering, sorting, and pagination

The databases behind a REST API can get very large. Sometimes, there's so much data that it shouldn't be returned all at once because it's way too slow or will bring down our systems. Therefore, we need ways to filter items.

We also need ways to paginate data so that we only return a few results at a time. We don't want to tie up resources for too long by trying to get all the requested data at once.

Filtering and pagination both increase performance by reducing the usage of server resources. As more data accumulates in the database, the more important these features become.

Here's a small example where an API can accept a query string with various query parameters to let us filter out items by their fields:

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();

// employees data in a database
const employees = [
  { firstName: 'Jane', lastName: 'Smith', age: 20 },
  //...
  { firstName: 'John', lastName: 'Smith', age: 30 },
  { firstName: 'Mary', lastName: 'Green', age: 50 },
]
```

```
app.use(bodyParser.json());

app.get('/employees', (req, res) => {
  const { firstName, lastName, age } = req.query;
  let results = [...employees];
  if (firstName) {
    results = results.filter(r => r.firstName === firstName);
  }

  if (lastName) {
    results = results.filter(r => r.lastName === lastName);
  }

  if (age) {
    results = results.filter(r => +r.age === +age);
  }
  res.json(results);
});

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

In the code above, we have the `req.query` variable to get the query parameters. We then extract the property values by destructuring the individual query parameters into variables using the JavaScript destructuring syntax. Finally, we run `filter` on with each query parameter value to locate the items that we want to return.

Once we have done that, we return the `results` as the response. Therefore, when we make a GET request to the following path with the query string:

`/employees?lastName=Smith&age=30`

We get:

```
[
    {
        "firstName": "John",
        "lastName": "Smith",
        "age": 30
    }
]
```

as the returned response since we filtered by `lastName` and `age`.

Likewise, we can accept the `page` query parameter and return a group of entries in the position from `(page - 1) * 20` to `page * 20`.

We can also specify the fields to sort by in the query string. For instance, we can get the parameter from a query string with the fields we want to sort the data for. Then we can sort them by those individual fields.

For instance, we may want to extract the query string from a URL like:

`http://example.com/articles?sort=+author,-datepublished`

Where `+` means ascending and `-` means descending. So we sort by author's name in alphabetical order and `datepublished` from most recent to least recent.

Maintain good security practices

Most communication between client and server should be private since we often send and receive private information. Therefore, using SSL/TLS for security is a must.

A SSL certificate isn't too difficult to load onto a server and the cost is free or very low. There's no reason not to make our REST APIs communicate over secure channels instead of in the open.

People shouldn't be able to access more information that they requested. For example, a normal user shouldn't be able to access information of another user. They also shouldn't be able to access data of admins.

To enforce the principle of least privilege, we need to add role checks either for a single role, or have more granular roles for each user.

If we choose to group users into a few roles, then the roles should have the permissions that cover all they need and no more. If we have more granular permissions for each feature that users have access to, then we have to make sure that admins can add and remove those features from each user accordingly. Also, we need to add some preset roles that can be applied to a group users so that we don't have to do that for every user manually.

Cache data to improve performance

We can add caching to return data from the local memory cache instead of querying the database to get the data every time we want to retrieve some data that users request. The good thing about caching is that users can get data faster. However, the data that users get may be outdated. This may also lead to issues when debugging in production environments when something goes wrong as we keep seeing old data.

There are many kinds of caching solutions like Redis, in-memory caching, and more. We can change the way data is cached as our needs change.

For instance, Express has the `apicache` middleware to add caching to our app without much configuration. We can add a simple in-memory cache into our server like so:

```
const express = require('express');
const bodyParser = require('body-parser');
const apicache = require('apicache');
const app = express();
let cache = apicache.middleware;
app.use(cache('5 minutes'));

// employees data in a database
const employees = [
  { firstName: 'Jane', lastName: 'Smith', age: 20 },
  //...
  { firstName: 'John', lastName: 'Smith', age: 30 },
  { firstName: 'Mary', lastName: 'Green', age: 50 },
]

app.use(bodyParser.json());

app.get('/employees', (req, res) => {
  res.json(employees);
});

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

The code above just references the `apicache` middleware with `apicache.middleware` and then we have:

`app.use(cache('5 minutes'))`

to apply the caching to the whole app. We cache the results for five minutes, for example. We can adjust this for our needs.

If you are using caching, you should also include `Cache-Control` information in your headers. This will help users effectively use your caching system.

Versioning our APIs

We should have different versions of API if we’re making any changes to them that may break clients. The versioning can be done according to semantic version (for example, 2.0.6 to indicate major version 2 and the sixth patch) like most apps do nowadays.

This way, we can gradually phase out old endpoints instead of forcing everyone to move to the new API at the same time. The v1 endpoint can stay active for people who don’t want to change, while the v2, with its shiny new features, can serve those who are ready to upgrade. This is especially important if our API is public. We should version them so that we won’t break third party apps that use our APIs.

Versioning is usually done with `/v1/`, `/v2/`, etc. added at the start of the API path.

For example, we can do that with Express as follows:

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();
app.use(bodyParser.json());

app.get('/v1/employees', (req, res) => {
  const employees = [];
  // code to get employees
  res.json(employees);
});

app.get('/v2/employees', (req, res) => {
  const employees = [];
  // different code to get employees
  res.json(employees);
});

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

We just add the version number to the start of the endpoint URL path to version them.

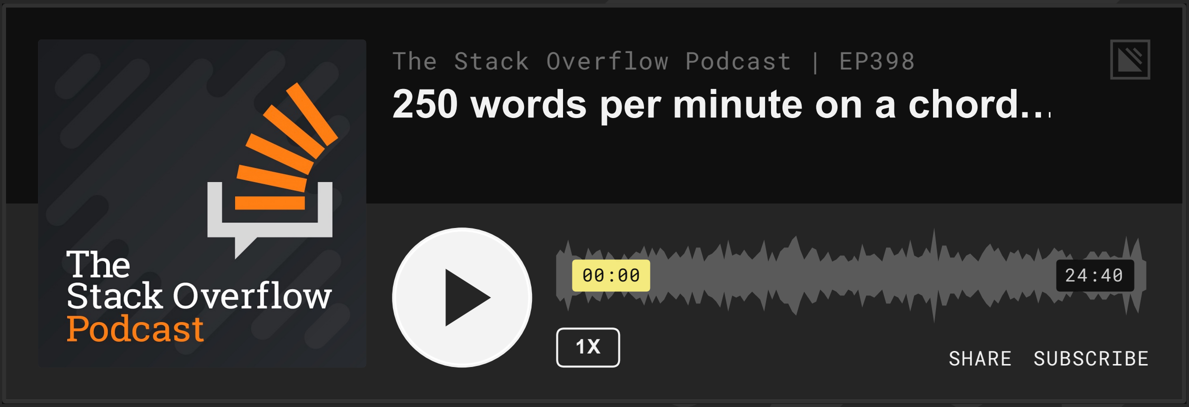
Conclusion

The most important takeaways for designing high-quality REST APIs is to have consistency by following web standards and conventions. JSON, SSL/TLS, and HTTP status codes are all standard building blocks of the modern web.

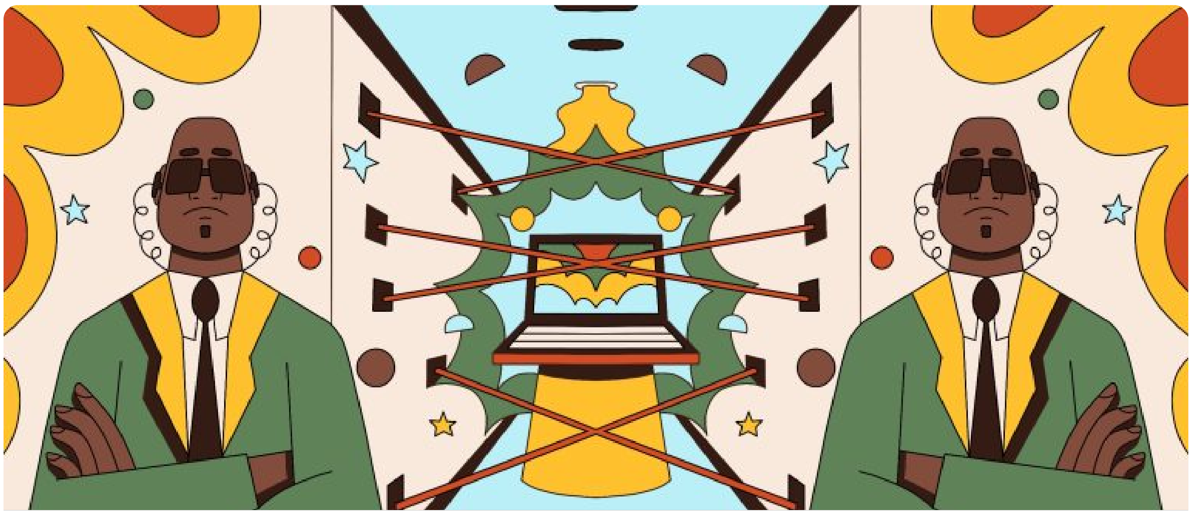
Performance is also an important consideration. We can increase it by not returning too much data at once. Also, we can use caching so that we don’t have to query for data all the time.

Paths of endpoints should be consistent, we use nouns only since the HTTP methods indicate the action we want to take. Paths of nested resources should come after the path of the parent resource. They should tell us what we're getting or manipulating without the need to read extra documentation to understand what it's doing.

Tags: express, javascript, rest api, stackoverflow


The Stack Overflow Podcast | EP398
250 words per minute on a chord...
The Stack Overflow Podcast
00:00
24:40
1X
SHARE
SUBSCRIBE

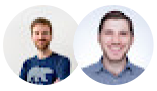
Related



code-for-a-living OCTOBER 6, 2021

Best practices for REST API security: Authentication and authorization

If you have a REST API accessible on the internet, you're going to need to secure it. Here's the best practices on how to do that.



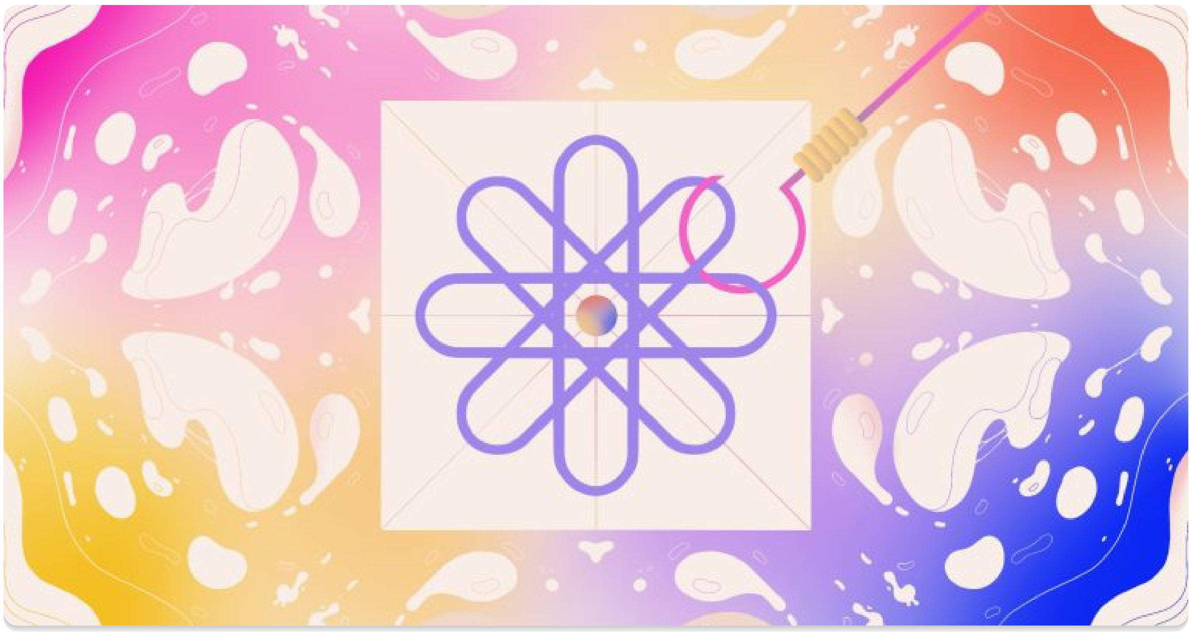
Sam Scott and **Graham Neray**

newsletter OCTOBER 15, 2021

The Overflow #95: Image search, but for any object IRL

Welcome to ISSUE #95 of The Overflow! This newsletter is by developers, for developers, written and curated by the Stack Overflow team and Cassidy Williams at Netlify. This week: the best practices for REST API auth, a physics lesson with air conditioners, and the wonders of the CSS Paint API. From the blog Extracting text from any file…

Ryan Donovan and Cassidy Williams



code-for-a-living OCTOBER 20, 2021

Why hooks are the best thing to happen to React

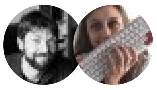
Originally, React mainly used class components, which can be strenuous at times as you always had to switch between classes, higher-order components, and render props. With React hooks, you can now do all these without switching, using functional components.

Doro Onome

newsletter AUGUST 20, 2021

The Overflow #87: Advance your developer career by writing

Welcome to ISSUE #87 of the Overflow! This newsletter is by developers, for developers, written and curated by the Stack Overflow team and Cassidy Williams at Netlify. This week: we bought some weird domain names, found creative ways to kill flies, and designed a swanky initial drop caps using CSS. From the blog How writing can advance your…



Ryan Donovan and Cassidy Williams

79 Comments

TJ 2 Mar 20 at 3:50

How on earth can you write an article on the REST best practices without mentioning HATEOAS (one of the most ignored yet fundamental, and required REST principles)?

Reply

Bernardo 2 Mar 20 at 4:24

I believe he tried to be succint here, as I've posted we could be a lot more pedantic and pragmatic, but since this is not a RFC just a blog post to guide general best practices. HATEOAS is great, but too heavy perhaps for a blog post, another example could be that he didn't even mention CORS.

Reply

Dave 3 Mar 20 at 7:04

This is a guide to general best practices of creating URLs. HATEOAS ensures you never have to build a URL yourself.

Reply

Luís 27 Apr 21 at 3:47

Can you elaborate on why?

I understand that if I want to explore some API or build an interface to consume REST APIs and allow users to freely navigate (something like postman or a crawler), this is very useful, because my application doesn't need to know how to access a "post comment" to allow the user to do so.

But I don't see this kind of application (or I don't perceive). Most use cases that I have is one application talking with another about specifc things that I must implement knowing the URL. Otherwise I could navigate using URLs provided by the resources themselves, but probably would cost more (network, time, cpu etc).

Reply

Roger Araque 7 Oct 20 at 2:31

Excellent article very good for advanced users and novices

Reply

Mallick 5 Sep 21 at 11:34

Indeed it is.

Reply

David Hillier 24 Feb 21 at 6:08

What does CORS have to do with REST?

Reply

BAHATI Robben 28 Apr 21 at 4:14

It helps restrict the clients origin that can access your server. you can open it to all clients, but you can restrict it to some clients

Reply

Leonardo Forero 3 Mar 20 at 6:37

Yes, he could point to http://stateless.co/hal_specification.html or to https://jsonapi.org/ for hateoas introduction

Reply

Saf 3 Mar 20 at 8:14

REST api, not necessarily RESTFull…

Reply

Raz Luvaton 2 Jun 20 at 12:11

In pagination you should avoid using page and use cursor instead. [Explanation article](https://engineering.mixmax.com/blog/api-paging-built-the-right-way/)

Reply

Decoder143 15 Jun 20 at 2:58

It's a pretty cool article!.
I am using the same from beginning.

Client sends the ID of the last item it received.Server finds that item in the table (sorted by “timestamp of item creation”) and sends back the “Next N” items.

Reply

Matt Trask 1 Jun 20 at 9:28

Ignoring HATEOAS is on point for most REST articles.

Reply

David Hillier 24 Feb 21 at 6:07

Exactly, I was disappointed to see this was about HTTP APIs and had nothing to do with REST. It’s like writing an article titled about sports cars and then just talking about cars in general.

Reply

Mahek Dougherty 12 Mar 21 at 9:53

99.99999% of the time you can replace the term REST with HTTP in tech articles. The richardson maturity model is also bad because there are no “levels” of REST. HATEOAS is part of REST fullstop.

Reply

wysohn 20 Aug 21 at 1:32

Good thing you have mentioned it here, now all I need to do is searching for what HATEOAS is

Reply

Bernardo 2 Mar 20 at 4:10

I agree with most things written here, except mostly with the part regarding status codes. I think that it is missing the part about success status codes, as there are multiple ones that have different meanings, as well with the error status codes.
About success status codes here are a few examples: 200 – as the default success status code; 201 – for resource creation success; 202 – for asynchronous requests (this one is a little bit more complex so it is ok to be out ;D); 204 – for resource deletion or update, as there is no reason to return a deleted resource or the updated state of the resource (this last one is debatable); 206 – for partial content as when returning from a large collection of resources, perhaps pagination, you can assert that there are resources unreceived.
Now looking at the error status codes here are a few things that I diverge, for example the 400 status code for "User already exists", I believe it is wrong as the request body is in a correct format, the only thing wrong is the information conveyed by it. For this problem of conflict there is a much better response, the 409 – Conflict, as it simply states there is a conflict between the data provided by the client and the current state of the server. 400 could be the default error code, for that I agree when an implementation does not desire to use 409. But I believe that 400 should be used only by default behaviour and when the format of the data is wrong, everything else should lead to 422 (Unprocessable Entity) as it states that the syntax is correct but it is semantically incorrect by some business rule.
Quite pedantic here, but I like to use HTTP status codes to help to the triage of responses.

Reply

John Au-Yeung 4 Mar 20 at 9:35

I think 400 is for a general invalid input response. It shouldn't be the response code if there's more specific info. If there're more specific issues that we know about from the request payload, then we can use the other ones.

Few exceptions include 403 vs 404 for attempting to accessing off-limits resources.

Reply

Maetko 4 Jun 20 at 10:27

Fully agree with your whole comment, 409 is a way to go here, just came here to comment the same.

Reply

gggeek 24 Feb 21 at 10:16

In my own experience, conflating HTTP status codes with business-logic status codes can lead to hard-to-troubleshoot problems, and should really ve avoided, not recommended.

The classical case is when I replace the API server at the url you are using with a plain dumb webserver serving html pages.

The app sending REST requests will then start receiving back 404, 401 responses, and, unless carefully coded (but who does check for the response's content-type header anyway? lolz), and assume that those simply mean 'data not found' and go on processing, instead of halting all operations.

Reply

Ehab Qadah 2 Mar 20 at 4:43

are there better/recommend options for versioning the Rest APIs apart from the mentioned mechanism /v1/ … ?

Reply

steg 2 Mar 20 at 8:06

There's many opinions out there. This is a good place to start: https://www.troyhunt.com/your-api-versioning-is-wrong-which-is/

Like many (but not all) people I prefer URL versioning because it's the easiest to use.

Reply

cstreet 2 Mar 20 at 8:45

> HTML status codes are all standard building blocks of the modern web.

I love when my HTML has status codes. 🙂

Reply

Abel Callejo 2 Mar 20 at 9:26

Finally! StackOverflow have just spoken about this important topic.

Other important factors to discuss are:

1. Number of descendant object nodes in a JSON response: Keep it to a minimum ~ ex: user.address.country.city.street.name=”acme”
2. Token regeneration and expiration

Reply

Chris McBrien 9 Mar 20 at 11:09

I agree I would also like to see a follow up with discussion of authentication/authorization. Tokens are an important aspect. It would be helpful to also touch on services like OAUTH.

Reply

Abel Callejo 2 Mar 20 at 9:29

Also I would like to add “camelCase vs snake_case” in JSON

Reply

Olaoluwa 3 Mar 20 at 10:50

Nice article for an introduction!

Reply

Ananth 3 Mar 20 at 12:37

I would like to add idempotency for APIs to serve consistent data

Reply

Lalit Sharma 3 Mar 20 at 1:36

Nice article as a starting point for REST API

Reply

API 3 Mar 20 at 2:24

A reference to the OWASP API Security Top 10 is probably in order:
https://github.com/OWASP/API-Security

Reply

cassiomolin 3 Mar 20 at 3:59

HTML status codes?!

Reply

Dragica 3 Mar 20 at 5:21

“HTML status code” should be “HTTP status code”

Reply

Sahi4j 6 Mar 20 at 2:32

XML isn’t widely supported ??
Where did you get that one? I thought you were serious guys.

Reply

Hans 3 Mar 20 at 5:38

> using standard HTML codes

HTML codes? what’s that? do you mean HTTP status codes? (eg 404 Not Found, 500

Internal Server Error, and so on?) –
https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

Reply

Marco Thomazini

3 Mar 20 at 8:33

One thing that bothers me on using standard HTTP status codes is the possible ambiguities that may arise. Let me explain. Assume a server application returns 404 to indicate a resource doesn't exist. A client application which receives this 404, could in response call the corresponding POST endpoint to create such resource. That would be OK, until we added a piece of infrastructure, such as nginx or HAProxy, between client and server.
This infrastructure could response 404, for misconfiguration or during a maintenance, and induce the client applications to error.
How scenarios like that should be handled?

Reply

Rasmus Schuktz

3 Mar 20 at 4:45

Yep, this was mainly the part of the article that I deeply disagree with.

HTTP status codes were designed to provide the status of the HTTP request, which is why you can't always find a suitable status code – these codes are intended to communicate information about the transport of information, which is why they're standardized and generalize to all kinds of web-based resources.

In my opinion, attempting to overload HTTP status codes with domain-specific meaning is a mistake that leads to countless confusing situations like the one you point out. These meanings are are often contrived and not helpful, they just add complexity to client code and response handking for no practical reason behind the misguided satisfaction of being on so HTTP compliant.

Yes, use a 404 status code for paths that don't resolve to a resource – of course use 500 for unhandled exceptions and so on, but do this at the framework level, so a client can always trust that HTTP status codes convey general information about the status of the request itself, so that it always has the same meaning to to an HTTP client.

200 means your request was handled and the response contains what you asked for. Leave it at that, and encode any domain-specific information in the body of the response itself – there's absolutely no practical, compelling reason to encode domain details as cryptic HTTP status codes, requiring you to read a manual and

handle all sorts of ambiguous status codes with a new meaning for every type of resource.

This nonsense caught on because it looks cool and feels good, not because it has any practical merit or value.

90% of the time, just encode your information as JSON. It's much easier to understand and document and consume, and your users will have much simpler client code with fewer errors and smoother adoption.

As far as overloading the meaning of HTTP verbs, I'm leaning towards "no" here as well. I've seen more than a few cases where we "ran out" of verbs and had to add another resource – CRUD are not the only 4 possible operations, unless your app is essentially a key/value database. Most domains are much richer and more nuanced than that.

Verbs in the path itself can often better communicate meaning in domain-terminology than you can by overloading 4 generic verbs with all sorts of contrived and misleading meaning. Just use GET for read-only requests and POST for requests that make changes – users can understand this right away for your entire API without further explanation for individual resources.

Don't overthink it. Just let HTTP be HTTP, and use JSON for anything that's resource/application-specific.

Reply

Thomas Kjørnes 4 Mar 20 at 10:46

Excellent points! I agree 100%

Reply

Carl Clarke 6 Mar 20 at 1:09

I disagree, In the suggested scenario where a proxy is responding due to misconfiguration/maintenance/etc. then there will be no API specific JSON just HTTP response codes so the client is now going to have to handle 'with and without JSON' because of infrastructure that it should not need to know about. I think it is a case of either using HTTP/RESTFul and fitting your solutions to its strengths and weaknesses or maybe do something completely different (Graph API, gRPC). On my travels I see many people thinking and doing RPC but using HTTP/WebAPI frameworks – they had a name for that, it was called SOAP.

I do agree that POST is the ‘catch-all’ verb for anything that doesn’t fit with the other verbs of which there are 5 most used and 9 in total.

Actually I think the author has produced a reasonable high-level article. The only advice that I would add is that as he has mentioned versioning APIs he might also want to mention versioning the requests and responses. For example there should be a \v1\orders path and a v1 JSON order object probably with a V1 part of the namespace in the supporting code, a lot of businesses only have a current/latest representation of entities and then really struggle to maintain API versions even though they have put a ‘v1\’ in the path.

Reply

Chris Newey 17 Jun 20 at 1:13

Very much disagree with this. RPC style APIs can be (and often are) implemented over http without resorting to bloated technologies like SOAP. My advice would be to re-read Rasmus Schuktz’ answer. I think he nails it.

Reply

Eli Kay 3 Mar 20 at 11:42

I agree. I never understood why the development community hijacked this code or others for that matter that had long been established as messages created the server. But the practice seems to be written in stone so I guess that’s what makes it “best”

Reply

Larry Lustig 3 Mar 20 at 11:53

This is a comprehensive post which the author obviously put a great deal of thought and effort into. Unfortunately, I disagree with almost everything in it.

Having verbs in our API endpoint paths isn’t useful and it makes it unnecessarily long since *it doesn't convey any new information.*

While it's true that RESTful API endpoints should contain only nouns, it's not because using verbs is redundant. It's because using verbs is antithetical to the concept behind REST, which is that you're transferring *state* and not processing instructions. This is the most important (and, to many people, the hardest) concept about REST. You need to "think" in terms of transferring object state and not in terms of an expected action on the part of the recipient of the message. If you're thinking in terms of verbs in your API endpoints then simply removing the verbs won't fix the design — you need to change your thinking. If RESTful thinking doesn't apply to the system you're building, you should abandon REST and use a more RPC-type approach, not try to make your approach *look* RESTful by blindly following some rules about naming.

We should name collections with plural nouns.

I find the issue of naming tables in databases (it has to be plural! it has to be singular!) is pretty meaningless. If you can't figure out that the table Customer or Customers contains multiple customer records then you have bigger problems than the letter "s"!

When it comes to naming API endpoints, it's pretty clear to me that singular/plural should be determined by whether the endpoint returns a single object or a collection (in JSON, probably a list) of objects. Therefore /Customers?state=NJ should be plural, and should always return a list while /Customer/:custId should be singular if it returns a single customer object (or 404) but plural (/Customers/:custId) if it returns a max-length-of-one list of customers. Now, you're actually describing to the consumer the nature of the message they will receive back from your service.

We use plurals to be consistent with what's in our databases.

While there exist simple cases in which our RESTful API is intended only to expose an underlying database it's much more common that our API is intended to expose a business model (a domain of information) that does not reflect the structure of the database used to persist the objects — if there even is a database involved! The RESTful server exists to expose our information in the most useful way to our clients and not to our back-end systems.

We have to make sure that it makes sure what we considered a nested resources matches what we have in our database tables.

Definitely not! A common situation would be one in which a RESTful server constructs business objects by consulting several *different* back-end servers or databases. In the example used, the articles and the comments could be stored completely separately (articles in a JSON database, or even in the file system and comments in a SQL table somewhere) or in different microservices and this implementation might change over time. The domain model we present to our consumers should absolutely not be based on something as trivial and changeable as our storage mechanism.

Sometimes, there's so much data that it shouldn't be returned all at once because it's way *too slow or will bring down our systems.*

Yes, this is *sometimes* true. But not always. Blindly adding filtering and pagination strikes me as premature optimization. You should not add these features if the expected maximum size of the resource collection is limited (like the number of departments in an organization, for instance) or if the "normal" use case is for the consumer to want the entire list. Conversely, if the expected collection size is large, or the normal use case is to want only a small subset of records, or design requirements state that you never want the consumer to be able to know the entire collection, then you should add these features.

A better "best practice" here would be to say "Consider payload size / network congestion, and what features will make your service most useful to the client."

REST APIs should accept JSON for request payload and also send responses to JSON.

There is absolutely no relationship between following RESTful principles and formatting the messages as JSON. REST was originally designed for media file transfers and one of its guiding principles is that messages should be 100% self-describing. In that sense, no service that returns structured data where the consumer needs to have a specific understanding of the structure is actually RESTful. But that ship has long since sailed. Still, there's absolutely no reason to tie the RESTful principles to JSON.

JSON is a concise, fairly readable, widely used format for data persistence and transfer. Also, it's in fashion. So it's a good "go to" format for data transfer and persistence. But there's nothing more or less RESTful about sending JSON vs XML vs Excel vs some format of your own devising (if it's more useful in the context of your application). The only rule is that you must state what the content is in the content-type header field.

We should be throwing errors that correspond to the problem that our app has encountered.
. . .
Whenever our API does not successfully complete, we should fail gracefully by sending an error with information to help users make corrective action.

Firstly, I think this confuses the issue of "throwing" exceptions and returning status codes that indicate an error in the API request.

When a server formats and returns a non-2xx status code to the consumer it's *not* the same thing as throwing an exception. The API call did, in fact, succeed (our back-end code ran, computed a result, and returned it to the caller). True, the result isn't the most common case in which the call did exactly what the consumer was expecting.

Secondly, it's difficult to have a "best practice" in the area of returning non-success HTTP codes since the specification around this is not great and usage is not consistent. Here are some of the problems in trying to specify best practices in this area:

* Should the reason phrase of the HTTP header to return specific information beyond the textual description of the status code (that is, should the reason phrase say "Not Found"

or “No employees matched your search of name=Smith.”?

* Some people return message *bodies* (to supply additional information) in cases in which the HTML spec states no body should be returned.

* Some status codes are extremely ambiguous. For instance, in the endpoint /departments/:deptID/employees?name=Smith, it’s obvious that if there are no Smiths in the specified department, 404 should be returned. But what if there is no department named deptID? Is that 400 Bad Request (eg, Hey, we don’t have a Legal department so we can’t even begin to look for employees that match your request) or 404 Not Found (eg, Well, we checked the list of departments but didn’t find Legal in it). And what if there’s no /departments endpoint at all? Bad Request or Not Found?

* The amount of information you should be returning (and even whether you should return anything at all) depends on the nature of your API — in particular whether it’s purely internal (when the only people seeing response are your own programmers) or external (in which case you might not want the wider world to see any information about what went wrong).

We can add caching to return data from the local memory cache instead of querying the database to get the data every time we want to retrieve some data that users request.

Caching is a valuable and interesting topic. But it’s largely orthogonal to RESTful APIs. One advantage of the RESTful approach is that it’s well suited to caching (and caching in a much broader sense than discussed in this article, since it makes it relatively easy to insert proxy caches into the workflow without modifying the RESTful server in any way).

But is using a cache a *REST* “best practice”? Absolutely not. Some problem domains are not amenable to caching. Still others will not benefit sufficiently from the extra complexity of caching to make the effort worthwhile.

Reply

Ben Tilford 3 Mar 20 at 5:14

Only thing I’d add to this would be related to

“Form data is good for sending data, especially if we want to send files. But for text and numbers, we don’t need form data to transfer those since—with most frameworks—we can transfer JSON by just getting the data from it directly on the client side. It’s by far the most straightforward to do so.”

JSON is not native to html but forms are. You also don’t have to write anything to pull the data out of the html in order to send it to the server. Unless you’re dealing with complex nested structures form data is much easier to work with and most modern servers can parse the body of either then route it in a way you don’t need to concern yourself with the actual content type of the request.

Reply

John Au-Yeung 4 Mar 20 at 9:40

I think JSON is more common now, but either way, it's good.

Reply

John Doe 9 Apr 20 at 4:47

I disagree with your explanation of collections being plural or singular. The convention is usually {plural collection name}/{id of a single entity within the collection}. You look at it like the collection is a directory, and you are retrieving a single entity within that collection when you add /{id} to it. I'm not saying your way is wrong, but it's definitely not the usual convention to have both "/Customers" and "/Customer". You usually have on path for drilling down further into a resource. It starts with "/customers" to get the collection of customers, and you append additional path arguments to the end to get a subset of the collection, not have two distinct paths "/customers" and "/customer".

Either way, the URI should be opaque to the consumer because they should be receiving the URI dynamically via HATEOS, not manually creating it anyway, so there really isn't a requirement for the URI to be constructed in any certain way.

Reply

ingo 9 Aug 20 at 11:41

/customer/ vs. /customer
The slash has a meaning.

Reply

Filip 3 Jun 20 at 11:46

This guy should have written the article.

Reply

Jonny Olliff-Lee 5 Dec 20 at 6:26

Thank you for writing this response, I hope people who read this article get to read this comment as well!

Reply

Tony 3 Mar 20 at 1:11

I don't get why REST API resources must follow database structure. To me how data is represented and how it's stored are different things. Especially if the data model is complicated, I don't see why that complexity should be brought to API level also.

Reply

John Au-Yeung 4 Mar 20 at 9:37

It's just being consistent with the database operations and the verbs. It reduces the cognitive load for users of the API.

Reply

Jonny Olliff-Lee 5 Dec 20 at 6:28

You're right Tony. They maybe the same, but it shouldn't be a constraint for your API design.

Reply

Momin Riyadh 4 Mar 20 at 6:24

Subject oriented article! Looking for a more details article on Rest API, and GraphQL.

Reply

JL 4 Mar 20 at 9:42

I would not call these “Best Practice”, only “most-common practice”. As such, an API designed this way will suffer from the most common pitfalls of “REST”: over/under-fetching and excess chattiness.

A REST API should not be designed around exposing the domain/data model as CRUD-over-http, but around actual use cases and process flow. If you can’t be bothered to figure out what those are, and what your API consumers actually need, you are abdicating design responsibility to your consumers, and virtually guaranteeing that they will need to make multiple API calls just to get the data they need for any given view. This only adds fuel to the “REST is dead. Long live GraphQL” dumpster fire.

An API should be designed with the same care and attention as a UI.

Reply

John Au-Yeung 4 Mar 20 at 9:36

That’s true. However, I think some commonly accepted conventions are needed to make the API easy to understand. We also don’t have to remember as many things if we follow common conventions.

Reply

The God Brother 27 Jan 21 at 6:11

Can you please give a site or a learning resource where this concept is explained… please.

Reply

Danish Kamal 6 Mar 20 at 6:05

Are there any server-side frameworks/libraries that allow accepting sort=+firstName like query parameters?

Reply

Prateek 8 Mar 20 at 1:43

I’m actually developing a REST API service and randomly saw this article. Really helpful, one aspect I think that is missing is good practice to structure your JSON data while accepting and responding. 🙂

Reply

John Doe 9 Apr 20 at 4:52

This article barely scratches the surface of REST, and there are some things mentioned that are debatable. I believe the one thing that is not debatable and is definitely just plain wrong, and very bad practice, is associating any of your design of your REST API with your underlying database. Your method of exposing your data to the outside (via a REST API) needs to be completely independent of your database design. You want to be able to change your database design without having to redesign your API. So not only is it wrong that you said you should model your API based on your database tables, you should go one step further and make sure you design your API completely independently of any database design.

Reply

zayra 15 Jul 20 at 9:53

how can you design and code your rest api completely independently of your database tables? Could you give some example or at least point to a site where this concept is explained? I appreciate your help in this matter

Reply

Jonny Olliff-Lee 5 Dec 20 at 6:46

Have a look at Ports and Adapters / Clean Architecture, and probably CQRS.

Separation of the code that does the querying from the code that handles API responses is how you achieve this.

Reply

Macaroni King 5 May 20 at 12:48

Hi!

I would change this part

```
...
const userExists = users.find(u => u.email === email);
if (userExists) {
return res.status(400).json({ error: 'User already exists' })
}
...
});
```

as

```
const userExists = users.some(u => u.email === email);
```

Otherwise IMO this is really nice article for new comers to REST API.

I would have avoided shooting my own leg a couple of times if I would have known the things listed in this one.

Reply

Agbeze Obinna Ephraim 31 Jul 20 at 10:19

hey Macaroni King! The problem with 'find' method is that, it stops searching when it finds the first match. so always go with filter as it filters out all match.

Reply

Abhijit Jadhav 27 May 20 at 2:58

Hey John Au-Yeung

can you please add uploading image API also,and how to get links in json response with different relations

Reply

Daniel 1 Jun 20 at 4:40

Sad to see the article mention caching but not cache headers[1], error codes but not problem details[2], and nested resources but not hypermedia[3]. The advice that IS here seems solid though, and it would make the life of an API consumer much easier if it were consistently followed!

[1]: https://web.dev/http-cache/
[2]: https://tools.ietf.org/html/rfc7807
[3]: https://apisyouwonthate.com/blog/rest-and-hypermedia-in-2019/

Reply

Michał Szumnarski 5 Jun 20 at 3:05

If you are dealing with really large dataset, you should not filter results on the server but you should form appropriate query to the DATABASE, and the database should handle you filtered results that you can serve via your API.

Reply

Simone Simone 9 Jun 20 at 1:58

Strongly disagree about using a time based cache

I also disagree with most of them, I mean, they are good but not “best”

Someone with a few experience could start adopt them as “ultimate guide” because stackoverflow said so, then get in some trouble on project growing

For example:

409 on a formerly correct request that server cannot accomplish, for example DELETE /item/123 if element 123 can’t be deleted.

The article also lacks of foundamental part, request data validation and error response

Reply

Shawn 22 Jun 20 at 3:24

Here is a video I made that shows you how to do DAST of web services specifically using AppScan Enterprise: https://www.youtube.com/watch?v=8IUg_Nz-TsQ

Reply

Tony 6 Aug 20 at 3:23

Why on earth do you return req.body as response payload for put and post api ? I know, you want to read what you send, but this is not even logically correct. You could log sseparately for the body you sent but not putting it into response. This is nonsense.

Reply

Leonardo 13 Aug 20 at 10:13

I have a question, why do you use ‘body-parser’?

I ask because of this:

https://stackoverflow.com/questions/47232187/express-json-vs-bodyparser-json/47232318

“Earlier versions of Express used to have a lot of middleware bundled with it. bodyParser was one of the middlewares that came it. When Express 4.0 was released they decided to remove the bundled middleware from Express and make them separate packages instead. The syntax then changed from app.use(express.json()) to app.use(bodyParser.json()) after installing the bodyParser module.

bodyParser was added back to Express in release 4.16.0, because people wanted it bundled with Express like before. That means you don’t have to use bodyParser.json() anymore if you are on the latest release. You can use express.json() instead.”

Reply

Vijay Singh 1 Sep 20 at 8:49

Some basic things are missing in this article which is essential now days e.x cache headers, security headers, error codes like 429, and some other best practices

[1]: https://www.loginradius.com/engineering/blog/best-practice-guide-for-rest-api-security/
[2]: https://www.loginradius.com/engineering/blog/http-security-headers/

Reply

Code Topology 5 Sep 20 at 9:46

is this a good idea of adding cryptic endpoints for the sake of security? so even if it disclosed, no one should understand it for which purpose this API is made?

Reply

Charlyarg 29 Oct 20 at 10:14

Great article an even better comments. I'm streamlining my API (it's internal usage for now so it becomes a bit easier) and I use stuff like get_articles so I'll try to introduce VERBS. Sensible HTTP codes as a result is something where I've been coming back and forward; either return a 4xx code or a 200 code when for example the id does not exist.. In general I'd prefer a 200 code with a success indicator of false, and a message that can be handled in the client. For authorization errors, though, now i'm ok with sending 4xx. JSON with the proper header for response is also something that improved things. I had to do some tweaking as I'm testing using either Jquery .ajax/Fetch/axios in the client and .ajax obviously send the data parameters in a different format. All in all quite interesting. Now I'd need an article on more advanced authentication.

Reply

Kyle 24 Nov 20 at 12:50

Why does this article not mention anything about Swagger / OpenAPI? Having defined contracts answers many ambiguities as well as allows for clients to be autogenerated, making much of the conventions still desirable, but not as necessary especially for edge cases.

Reply

omid 26 Nov 20 at 6:09

if we have some posts and users can like or dislike them.

PUT /posts/:postId/like
PUT /posts/:postId/dislike

(I checked Instagram they have something like that.)

Is it better way for these cases which HTTP verbs is not enough to clear operation?

Reply

Alex99 3 Jan 21 at 2:01

Well, these express, .net thing are not the language of web. These are weird substnaces. The only Language of web is PHP

Reply

Jeremy Carter 22 Feb 21 at 4:39

Why does the sort query string value contain two values (delimited by comma). You can have multiple query string values eg. ?sort=-name&sort=+timestamp.

Reply

Harry McIntyre 24 Feb 21 at 5:32

This article needs a find-and-replace for “REST” with “HTTP RPC”

Reply

Adam 2 Mar 21 at 4:29

Great article!

Do you have any thoughts on when data should be “processed”. For example, if my database model has a “gender” field with values like “m”, “f”, etc. should I send this response and then process it on the client side? I suppose there are some reasons to process on the server side, for example turning a date of birth into an age so nobody sees it. Any thoughts?

Reply

Todd McAllister 28 Apr 21 at 7:38

Did anyone else find it bizarre read a blog about “best practice” when they change the color of their background using javascript instead of CSS?

Reply

Syed

14 Sep 21 at 1:16

Is it ok for Api A to talk to Api B and vice versa? Is this circular dependency bad?

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

☐ Save my name, email, and website in this browser for the next time I comment.

Post Comment