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.

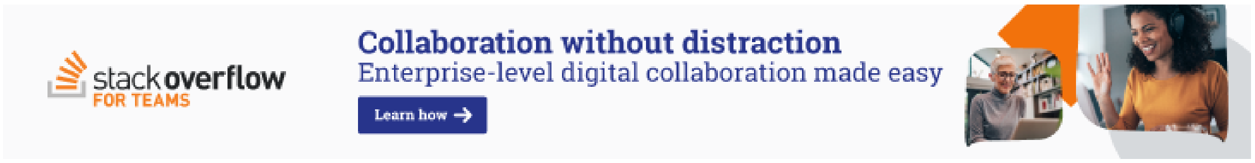
We should also make sure that our endpoints return JSON as a response. Many server-side frameworks have this as a built-in feature.

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.