**Request-Response Best Practices**

**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.

**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.

**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](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete) 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.

**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.

**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.

**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.

**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 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](https://redis.io/), in-memory caching, and more. We can change the way data is cached as our needs change.

**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.