**REST API development best practices**

In web-application or web-service development, REST APIs are most evident part of it. REST APIs are HTTP / HTTPS based protocol interfaces which helps client and backend service to communicate with each other.

Client can be front-end applications running on any browser or any web-service which need to interact with other web-services. Backend Service can be any web-service which can accept Http request from client to process it or send back response. Therefore, web-service need to expose REST based Application Programming Interface (API) to establish communication between either party.

All programmer who are working on web-applications development must have work on requirements where they need to create new REST APIs for their web-services. So, it is important to know some of the best design practices to develop REST APIs. Just keeping these guidelines in your mind while developing APIs will provide your web-services a robust and comprehensive design.

Let’s go through some of the best REST APIs development practices:

1. **Use nouns instead of verbs in endpoints**

Endpoint paths should always noun. Avoid using verb in path. Name of endpoint should refer to the entity, not the action be performed. e.g.

<https://example.com/service-name/createArticle>

<https://example.com/service-name/article>

The action should be indicated by the HTTP request method type. Most likely HTTP request method types used for CURD operation are:

* **GET** - Retrieves data from server.
* **POST** - Submits new data to the server.
* **PUT** - Updates existing data to the server.
* **DELETE** - Removes data from server.

For examples:

**GET** [/service-name/article](https://example.com/service-name/article)/{articleId}

**POST** [/service-name/article](https://example.com/service-name/article)

**PUT** [/service-name/article](https://example.com/service-name/article)

**DELETE** [/service-name/article](https://example.com/service-name/article)/{articleId}

1. **Use plural nouns for collections endpoints**

Name of the endpoint should clearly indicate that whether it is using single resource or collection of resources. Therefore, if endpoint is acting on multiple entities, then plural nouns should be used to indicate clear intentions. For example:

//Get all articles by author

**GET** [/service-name/article](https://example.com/service-name/article)s/{authorId}

//Create multiple articles in one go.

**POST** [/service-name/article](https://example.com/service-name/article)s/

// Get all articles.

**GET** [/service-name/article](https://example.com/service-name/article)s/

// Delete all article written by author

**DELETE** [/service-name/article](https://example.com/service-name/article)s/{authorId}

1. **Use endpoint nesting**

Applying logical nesting naming on endpoint is important to construct a context of endpoint. This nested naming should communicate clear objective of that endpoint. Also avoid too many levels of nesting because it starts losing it significance and objective. Take best of your judgement for nesting level. Ideally more than 3 levels should be avoided.

//Get all comments for an article.

GET [/service-name/article](https://example.com/service-name/article)s/{articleId}/comments

// Specific article for a specific author.

GET [/service-name/](https://example.com/service-name/article)authors/{authorId}/articles/{articleId}

1. **Use JSON for request and response**

JSON is an open, standardised, and most suitable format for data transfer. Endpoint should accept JSON payload and return JSON response. JavaScript has built-in support to parse JSON format quickly. All programming languages support JSON format and can be easily interpreted by human.

1. **Use HTTP Status codes for error handling**

Error response is always a confusion, whenever any error occurs it’s hard to understand reason of that error. Therefore, to avoid confusion, error should be handled gracefully. To provide more clarity to error response, always return HTTP response codes.

Common HTTP error status codes are:

* **400 Bad Request**: It means that client-side input is not correct and fails validation.
* **401 Unauthorized:** It means the request isn’t authorized to access an endpoint. It usually happens when the user isn’t authenticated.
* **403 Forbidden:** It means the user is authenticated, but it isn’t allowed to access an endpoint. It happens when the user isn’t authorized to access endpoint due to its role.
* **404 Not Found:** Its indicates that endpoint is not found, which user is trying to access.
* **500 Internal server error**: It is a generic server error. It probably shouldn’t be thrown explicitly.
* **502**[**Bad Gateway**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/502)**:** It indicates that the server, while acting as a gateway or proxy, received an invalid response from the upstream server.
* **503 Service Unavailable:** It indicates that the server is not ready to handle the request.

1. **Return error details in the response body**

Whenever internal server error happens, endpoint throws error response with HTTP status code **500 Internal Server Error**. But this is very generic error and does not provide any information about actual error. Therefore, it’s always a good practice to return error details in the JSON or response body to help a user with debugging.

{

"error": "Database server is not available.",

"detail": {

“error-message“: “Database server is down, webserver not able to establish connection with database. Please try after some time.”

}

}

1. **Allow filtering, sorting, and pagination**

Sometime, databases behind endpoint can be incredibly large. Therefore, such large data can’t be returned in one go because it’s way too slow or will bring down the systems. Therefore, we need ways to filter items. So filtering, pagination, and sorting are some techniques to handler large data response.

Endpoint needs a way to paginate data so that endpoint can only return a certain number of results at a time. Such endpoint should be designed in a way that it can support pagination using parameters. The endpoint can be hit again with next parameters to fetch next set of results.

Filtering and pagination both increase performance by reducing the usage of server resources. Sorting endpoint response can make data representation more logical, searching on result data become easier on sorted data.

1. **Implement timeout**

Timeouts cause a request to fail after a specified amount of time. This is useful when there is a network issue, and the request cannot be completed, or a user sends too much data and takes longer to handler request. When timeout occurs, connection is closed instead of remaining open. So, implementing timeout on endpoint is a good practice to avoid resource abuse.

1. **Use caching**

Using **Cache-Control** headers will allow users to make effective use of cached data. Caching allows users to access data faster because it is stored locally, meaning another request to the server retrieve data from cache instead for database. The good thing about caching is that users can get data faster. However, the data that users get may be outdated. So, its depends on developers wisdom that which endpoint is good candidate for enable caching.

1. **Maintain good security practices**

Most of time client and service exchange very confidential data among them, so it become very important to secure data from any type of attach. Therefore, using SSL/TLS security protocol is a must.

SSL certificates are not hard to load to a server and are available for free mostly during the first year. They are not expensive to buy in cases where they are not available for free.

Using SSL certificate data got encrypt while transition. It enables HTTPS protocol to provide one extra layer of security. So, enabling SSL/TLS certification for your web application is always a best practice.

1. **Implement Versioning**

While web application development there are incremental changes in existing endpoints. But we don’t want to modify existing endpoint because it could break exiting client. Therefore, endpoints versioning can be a way to keep different versions of endpoint to support different types of implementations.

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

<https://example.com/service-name/v1/article>  
<https://example.com/service-name/v2/article>

1. **API documentation**

Documentation is vital to communicate to other developers how to use your APIs and all its features. API docs should contain info for your API’s endpoints, code snippet examples in various programming languages, tutorials, debugging info, and more.