Contents

[REST API 1](#_Toc118391276)

[REST API BEST PRACTICES 1](#_Toc118391277)

[**Using HTTP VERB** 1](#_Toc118391278)

[**API END NAMING** 2](#_Toc118391279)

[**Using HTTP Response Codes** 3](#_Toc118391280)

[**Creating Fine-Grained Resources** 5](#_Toc118391281)

## REST API

Representational State Transfer is referred to as REST. RESTful refers to any API (Application Programming Interface) that follows to the REST design principle.

In plain English, a REST API is a way to communicate among two computers using the HTTP (Hypertext Transfer Protocol), just like clients and servers do.

## REST API BEST PRACTICES

API is a reflective interface that allows developers to interact with data. The developer's life can be made easier and more comfortable with the help of a beautifully structured and designed API. The critical point here, however, is perfectly designed REST APIs. If the REST API is not flawlessly designed, it can cause problems for developers rather than improving the user experience. As a result, it is critical to follow the commonly used API design conventions to provide the best solution to your clients or developers.

### **Using HTTP VERB**

API consumers are capable of sending GET, POST, PUT, and DELETE verbs, which greatly enhance the clarity of a given request.

The four main HTTP verbs are typically used as follows:

**GET**

Read a specific resource (by an identifier) or a collection of resources.

**PUT**

Update a specific resource (by an identifier) or a collection of resources. Can also be used to create a specific resource if the resource identifier is known before-hand.

**DELETE**

Remove/delete a specific resource by an identifier.

**POST**

Create a new resource. Also, a catch-all verb for operations that don't fit into the other categories.

### **API END NAMING**

* **No CRUD Operation in URIs:** Putting verbs in the URI is not a good idea. Restful URIs should not contain any CRUD (Create, Read, Update, and Delete) functionality. This is due to the fact that HTTP verbs should be sufficient to describe the action performed on the resource.

Example: /users/{id} instead of /getUser

* **For Hierarchy:** Forward slashes are commonly used to indicate the hierarchy of individual resources and collections. When there is no hierarchical relationship (as in lists), punctuation marks such as the semicolon or, more commonly, the comma should be used.

Example: /users/{id1},{id2}

Upper example is used for accessing multiple user resources.

* **Query Parameters:** A situation may arise in which a collection of resources must be sorted, filtered, or limited based on some specific resource attribute. Instead of creating new APIs for this requirement, allow sorting, filtering, and pagination capabilities in the resource collection API and pass the input parameters as query parameters.

Example: /users?location=BD

* **Lowercase Letters and Dashes and No Underscore:** Resource names should not contain any capital letters. Resource names should use exclusively lowercase letters. Similarly, dashes (-) are conventionally used in place of underscores (\_).

Example: /users/{id}/pending-orders instead of /users/{id}/Pending\_Orders

* **No File Extensions:** Extensions should never be included in URIs. It makes the URIs unattractive and lengthy. Apart from the above reason, if highlighting the media type of API using file extension is required, then relying on the media type is better, as communicated through the Content-Type header, to determine how to process the body’s content.

Example: /users/{id}/pending-orders instead of /users/{id}/pending-orders.xml

* **In URIs, No Trailing Forward Slash (/):** It’s best to avoid adding a forward slash (/) to the end of the URIs. It doesn’t add any semantic value and may confuse. It is preferable to remove it from the URI.

Example: /users/{id}/pending-orders instead of /users/{id}/pending-orders/

### **Using HTTP Response Codes**

Response status codes are part of the HTTP specification. There are quite a number of them to address the most common situations. In the spirit of having our RESTful services embrace the HTTP specification, our Web APIs should return relevant HTTP status codes.

* **200 OK:** General success status code. This is the most common code. Used to indicate success.
* **201 CREATED:** Successful creation occurred (via either POST or PUT). Set the Location header to contain a link to the newly-created resource (on POST). Response body content may or may not be present.
* **204 NO CONTENT:** Indicates success but nothing is in the response body, often used for DELETE and PUT operations.
* **400 BAD REQUESTS:** General error for when fulfilling the request would cause an invalid state. Domain validation errors, missing data, etc. are some examples.
* **401 UNAUTHORIZED:** Error code response for missing or invalid authentication token.
* **403 FORBIDDEN:** Error code for when the user is not authorized to perform the operation or the resource is unavailable for some reason (e.g., time constraints, etc.).
* **404 NOT FOUND:** Used when the requested resource is not found, whether it doesn't exist or if there was a 401 or 403 that, for security reasons, the service wants to mask.
* **405 METHOD NOT ALLOWED:** Used to indicate that the requested URL exists, but the requested HTTP method is not applicable. For example, POST */users/12345* where the API doesn't support creation of resources this way (with a provided ID). The Allow HTTP header must be set when returning a 405 to indicate the HTTP methods that are supported. In the previous case, the header would look like "Allow: GET, PUT, DELETE"
* **409 CONFLICTS:** Whenever a resource conflict would be caused by fulfilling the request. Duplicate entries, such as trying to create two customers with the same information, and deleting root objects when cascade-delete is not supported are a couple of examples.
* **500 INTERNAL SERVER ERROR:** Never return this intentionally. The general catch-all error when the server-side throws an exception. Use this only for errors that the consumer cannot address from their end.

### **Creating Fine-Grained Resources**

It's best to start out by developing APIs that closely resemble the system's underlying application domain or database architecture. In the end, you'll need aggregate services that draw on a variety of underlying sources to cut down on chatter. However, it is much simpler to create fine-grained or individual resources from larger aggregates than it is to create larger resources later from smaller resources. Start with simple, easily defined resources and add CRUD functionality to them to make life easier for yourself. Later, you can develop those use-case-focused, chit-busting resources.