**RESTful API Designing guidelines — The best practices**

**Best way to design API**:

There is a long debate going on the internet, about the best ways to design the APIs, and is one of the most nuanced. There are no official guidelines defined for the same.

The API is an interface, through which many developers interact with the data. A good designed API is always very easy to use and makes the developer’s life very smooth. API is the GUI for developers, if it is confusing or not verbose, then the developer will start finding the alternatives or stop using it. Developers’ experience is the most important metric to measure the quality of the APIs.

The API is like an artist performing on stage, and its users are the audience



**The following points are most important to design REST API**:

1. **Terminologies**
2. **API endpoint**
3. **HTTP methods (verbs)**
4. **HTTP response status codes**
5. **Field name casing convention**
6. **Searching, sorting, filtering and pagination**
7. **Versioning**

**1) Terminologies**:

The following are the most important terms related to REST APIs

**Resource**:

Resources is an object or representation of something, which has some associated data with it and there can be set of methods to operate on it. E.g. Animals, schools and employees are resources and delete, add, update are the operations to be performed on these resources.

**Collections**:

Collections are set of resources, e.g. Companies is the collection of Company resource.

**URL**:

URL (Uniform Resource Locator) is a path through which a resource can be located and some actions can be performed on it.

**2) API endpoint**:

Let’s write few APIs for **Companies** which has some **Employees**, to understand more.

**/getAllEmployees** is an API which will respond with the list of employees. Few more APIs around a **Company** will look like as follows:

* /addNewEmployee
* /updateEmployee
* /deleteEmployee
* /deleteAllEmployees
* /promoteEmployee
* /promoteAllEmployees

And there will be tons of other API endpoints like these for different operations. All of those will contain many redundant actions. Hence, all these API endpoints would be burdensome to maintain, when API count increases.

**What is wrong**:

The URL should only contain resources(nouns) not actions or verbs. The API path **/addNewEmployee** contains the action addNewalong with the resource name Employee.

**Then what is the correct way:**

/companies endpoint is a good example, which contains no action. But the question is how do we tell the server about the actions to be performed on companies resource viz. whether to add, delete or update?

This is where the HTTP methods (GET, POST, DELETE, PUT), also called as verbs, play the role.

The resource should always be **plural** in the API endpoint and if we want to access one instance of the resource, we can always pass the id in the URL.

* Method GET path /companies should get the list of all companies
* Method GET path /companies/34 should get the detail of company 34
* Method DELETE path /companies/34 should delete company 34

In few other use cases, if we have resources under a resource, e.g. Employees of a Company, then few of the sample API endpoints would be:

* GET /companies/3/employees should get the list of all employees from company 3
* GET /companies/3/employees/45 should get the details of employee 45, which belongs to company 3
* DELETE /companies/3/employees/45 should delete employee 45, which belongs to company 3
* POST /companies should create a new company and return the details of the new company created

Isn’t the APIs being now more precise and consistent?

**Conclusion**:

The paths should contain the plural form of resources and the HTTP method should define the kind of action to be performed on the resource.

**3) HTTP methods (verbs)**:

HTTP has defined few sets of methods which indicates the type of action to be performed on the

resources.

The URL is a sentence, where resources are nouns and HTTP methods are verbs.

**The important HTTP methods are as follows**:

1. **GET** method requests data from the resource and should not produce any side effect.

E.g. **/companies/3/employees** returns list of all employees from company 3.

1. **POST** method requests the server to create a resource in the database, mostly when a web form is submitted.

E.g. **/companies/3/employees** creates a new Employee of company 3. **POST** is non-idempotent which means multiple requests will have different effects.

1. **PUT** method requests the server to update resource or create the resource, if it doesn’t exist.

E.g. **/companies/3/employees/john** will request the server to update, or create if doesn’t exist, the john resource in employee's collection under company 3.

**PUT** is idempotent which means multiple requests will have the same effects.

1. **DELETE** method requests that the resources, or its instance, should be removed from the database.

E.g. **/companies/3/employees/john/** will request the server to delete john resource from the employee's collection under the company 3.

There are few other [methods](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods) which is not so much important.

**4) HTTP response status codes**:

When the client raises a request to the server through an API, the client should know the feedback, whether it failed, passed or the request was wrong. HTTP status codes are bunch of standardized codes which has various explanations in various scenarios. The server should always return the right status code.

The following are the important categorization of HTTP codes

**2xx (Success category)**:

These status codes represent that the requested action was received and successfully processed by the server.

**200 Ok**:

The standard HTTP response representing success for **GET**, **PUT** or **POST**.

**201 Created**:

This status code should be returned whenever the new instance is created. E.g. on creating a new instance, using **POST** method, should always return 201 status code.

**204 No Content**:

This status code represents the request is successfully processed, but has not returned any content.

DELETE can be a good example of this.

The API DELETE /companies/43/employees/2 will delete the employee 2 and in return we do not need any data in the response body of the API, as we explicitly asked the system to delete. If there is any error, like if employee 2 does not exist in the database, then the response code would be not be of 2xx Success Category but around 4xx Client Error category.

**3xx (Redirection Category)**:

**304 Not Modified**:

Indicates that the client has the response already in its cache. And hence there is no need to transfer the same data again.

**4xx (Client Error Category)**:

These status codes represent that the client has raised a faulty request.

**400 Bad Request**:

Indicates that the request by the client was not processed, as the server could not understand what the client is asking for.

**401 Unauthorized**:

Indicates that the client is not allowed to access resources, and should re-request with the required credentials.

**403 Forbidden**:

Indicates that the request is valid and the client is authenticated, but the client is not allowed access the page or resource for any reason.

E.g. sometimes the authorized client is not allowed to access the directory on the server.

**404 Not Found**:

Indicates that the requested resource is not available now.

**410 Gone**:

Indicates that the requested resource is no longer available which has been intentionally moved.

**5xx (Server Error Category)**:

**500 Internal Server Error**:

Indicates that the request is valid, but the server is totally confused and the server is asked to serve some unexpected condition.

**503 Service Unavailable**:

Indicates that the server is down or unavailable to receive and process the request. Mostly if the server is undergoing maintenance.

**5) Field name casing convention**:

We can follow any casing convention, but make sure it is consistent across the application. If the request body or response type is JSON then please follow camelCase to maintain the consistency.

6) **Searching, sorting, filtering and pagination:**

All of these actions are simply the query on one dataset. There will be no new set of APIs to handle these actions. We need to append the query params with the **GET** method API.

Let’s understand with few examples how to implement these actions.

**Sorting**:

In case, the client wants to get the sorted list of companies, the **GET** **/companies** endpoint should accept multiple sort params in the query.

E.g. GET **/companies?sort=rank\_asc** would sort the companies by its rank in ascending order.

**Filtering**:

For filtering the dataset, we can pass various options through query params.

E.g. GET **/companies?category=banking&location=india** would filter the companies list data with the company category of Banking and where the location is India.

**Searching**:

When searching the company name in companies list the API endpoint should be GET **/companies?search=Digital Mckinsey**

**Pagination**:

When the dataset is too large, we divide the data set into smaller chunks, which helps in improving the performance and is easier to handle the response.

E.g. **GET /companies?page=23** means get the list of companies on 23rd page.

If adding many query params in **GET** methods makes the URI too long, the server may respond with 414 URI Too long HTTP status, in those cases params can also be passed in the request body of the **POST** method.

**7) Versioning**:

When your APIs are being consumed by the world, upgrading the APIs with some breaking change would also lead to breaking the existing products or services using your APIs.

**http://api.yourservice.com/v1/companies/34/employees** is a good example, which has the version number of the API in the path. If there is any major breaking update, we can name the new set of APIs as v2 or v1.x.x

Link: <https://hackernoon.com/restful-api-designing-guidelines-the-best-practices-60e1d954e7c9>