Rest API Practices

Ali-Akbor

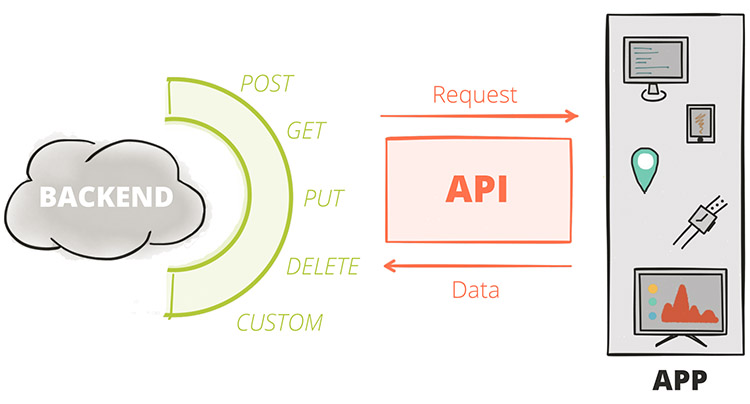
## What is REST API?

## 

REST API is an API that follows a set of rules for an application and services to communicate with each other. As it is constrained to REST architecture, REST API is referred to as RESTful API. REST APIs provide a way of accessing web services in a flexible way without massive processing capabilities.

## Characteristics of a well-designed API

* **Flexible**: REST API is flexible with multiple types of calls like returning different data formats and changing structurally with the correct implementation of hypermedia. It allows users to communicate back and forth with clients and servers, even if they are hosted on different servers.
* **Adaptable**: REST API is adaptable to any modification done in data that resides in the database, even when hosted on the different back- and front-end servers. Since it depends to a certain extent on codes, it helps synchronize data within websites without any issue.
* **Ease of Understanding**: As REST uses HTTP verbs (GET, POST, PUT or DELETE) methods for communication, these methods are self-explanatory. In addition, REST architecture helps increase developers’ productivity, allowing them to display the information on the client-side and store or manipulate the data on the server-side.

****

## REST API Best Practices

### **Always choose JSON.**

### **Focus on Error Handling.**

### **Prefer using Plural naming conventions.**

### **Utilize Resource Nesting Efficiently.**

### **Provide Accurate API Documentation.**

### **Utilize SSL/TLS security layers.**

### **Prioritize Nouns over Verbs in URI.**

### **Filtering options.**

1. **API End Naming Best Practice**

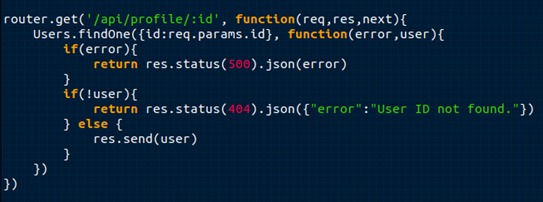
### **Always choose JSON**

JavaScript Object Notation is one of the easiest languages and an easy-to-use format. One of the important best practices to follow is to always choose JSON. The key feature of JSON is that it is very easy to parse and supports most frameworks. JSON can be used by any programming language.

## 

**Focus on Error Handling**

You should always use regular HTTP status codes in responses to requests made to your API. This will help your users to know what is going on – whether the request is successful, or if it fails, or something else.

****

Idel error handling code consists of 3 parts:

* **Error –** a unique identifier of the error
* **Message –** a comprehensive, readable message
* **Detail –** lengthier explanation of the message

### **Prefer using Plural naming conventions**

Generally, it is the best practice to use plural nouns for collections. This plural naming convention becomes a global code. This also helps normal people to understand that these groups of APIs form a collection.

The following table helps you in understanding the right and wrong usage of plural names in REST API :

| Do’s | Dont’s |
| --- | --- |
| GET/bikes/123 | GET/bike/123 |
| POST/bikes | POST/bike |
| GET/bikes | GET/bike |

### **Utilize Resource Nesting Efficiently**

Resource nesting is a practice of clubbing two functions that have some hierarchy or are linked to each other. Nesting to one level is one of the best practices to group resources that are logically coherent. For example, ‘order’ and ‘users’ are two resources of the same category in an online shop. The ‘user’ makes the ‘order’ and the ‘order’ belongs to the ‘user’. The following code explains the scenario discussed above.

/users // list all users

/users/123 // specific user

/users/123/orders //list of orders that belong to a specific user

/users/123/orders/0001 // specific orders of a specific users order

list

Overusing Nesting is not good in any case. When overused, Nesting loses its appeal and creates unwanted dependency issues. So the REST API best practice that can be followed is limiting the use of nesting to one level

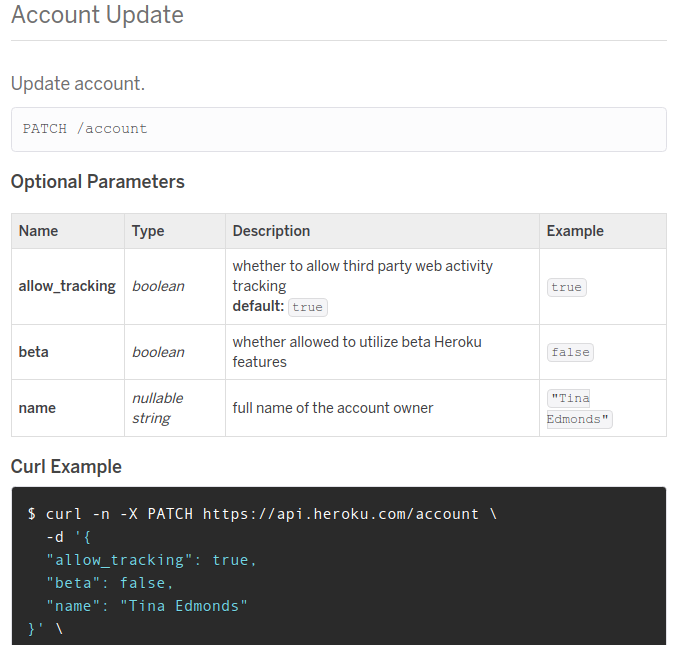
**Provide Accurate API Documentation**

When you make a REST API, you need to help clients (consumers) learn and figure out how to use it correctly. The best way to do this is by providing good documentation for the API.

The documentation should contain:

* relevant endpoints of the API
* example requests of the endpoints
* implementation in several programming languages
* messages listed for different errors with their status codes

One of the most common tools you can use for API documentation is Swagger. And you can also use Postman, one of the most common API testing tools in software development, to document your APIs.

****

**Utilize SSL/TLS security layers**

One of the REST API Best practices is to encrypt the communication using SSL/TLS. It is very essential to ensure database security for any API developer. The earned trust of the customers to keep their sensitive details private is a must. To avoid security breaches, you need to use SSL (Secure Socket Layer) and TLS (Transport Layer Security). SSL/TSL provides a public and private key to give a secured connection. TSL is an advanced version of SSL and hence provides better protection and security.

The clear difference between the URL of a REST API that runs over SSL and the one which does not is the “s” in HTTP:

https://mysite.com/posts runs on SSL.  
http://mysite.com/posts  does not run on SSL.

**Prioritize Nouns over Verbs in URI**

Since REST API is mostly developed for resources like services, it is essential to use Nouns and not verbs. So it is better to use only Nouns to represent an entity in REST endpoint paths. This is because the HTTP request method already consists of verbs. So having verb in REST API endpoints will not pull any new information. You must use tags to change the resource’s state.

The following table helps you in understanding the REST API Verbs:

| **REST Verb** | **Action** |
| --- | --- |
| GET | Fetches a record or set of resources from the server |
| OPTIONS | Fetches all available REST operations |
| POST | Creates a new set of resources or a resource |
| PUT | Updates or replaces the given record |
| PATCH | Modifies the given record |
| DELETE | Deletes the given resource |

Here are a few examples to show how the endpoints should look like,

* GET/books/123
* DELETE/ books/123
* POST/books
* PUT/books/123
* PATCH/book/123

**Filtering options**

* [Filtering](https://hevodata.com/learn/rest-api-best-practices/#s1)
* [Sorting](https://hevodata.com/learn/rest-api-best-practices/#s2)
* [Paging](https://hevodata.com/learn/rest-api-best-practices/#s3)
* [Field Selection](https://hevodata.com/learn/rest-api-best-practices/#s4)

#### **Filtering:**

Using this you can filter results that satisfy your required conditions. You can use search parameters like country, creation, date and etc for this.

ET /users?country=UK

GET /users?creation\_date=2021-10-11

GET /users?creation\_date=2021-10-11

#### **Sorting:**

You can sort your results in ascending and descending order using this option.

GET /users?sort=birthdate\_date:asc

GET /users?sort=birthdate\_date:desc

#### **Paging:**

Using the ‘limit’ option, you can narrow down the results to the required number. You can also use ‘offset’ to show the part of the overall results displayed.

GET /users?limit=120

GET /users?offset=3

#### **Field Selection:**

Using the field selection function, you can request to display a specific part of the data available for that object. While you query an object with many fields, you can specify the fields in your response. An object will have ‘Name’, ‘Surname’, ‘Birthdate’, ‘Email’, ‘Phone’ as its fields.

For example, when you want to retrieve the birthdate and email to automate birthday wishes. You can use a query like this:

For a specific user:

GET/ users/123?fields=name,birthdate,email

For a full list of users:

GET/ users?fields=name,birthdate,email

**API End Naming Best Practice**

**URIs as resources as nouns:**

One of the most recognizable characteristics of REST is the predominant use of nouns in URIs. Restful URIs should

not indicate any kind of CRUD (Create, Read, Update, and Delete) functionality. Instead, REST APIs should allow you to

manipulate a resource.

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

**Forward slashes for hierarchy:**

As shown in the examples above, forward slashes are conventionally used to show the hierarchy between individual resources and collections

|  |
| --- |
| Example: /users/{id}/address clearly falls under the /users/{id} resource which falls under the /users collection |

**Punctuation for lists:**

When there is no hierarchical relationship (such as in lists), punctuation marks such as the semicolon, or, more frequently, the comma should be used.

|  |
| --- |
| Example: /users/{id1},{id2} to access multiple user resources |

**Query parameters where necessary:**

In order to sort or filter a collection, a REST API should **allow query parameters** to be passed in the URI.

|  |
| --- |
| Example: /users?location=USA to find all users living in the unaited States |

**Lowercase letters and dashes:**

By convention, 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:**

**Leave file extensions (such as .xml) out of your URIs.** We’re sorry to say it, but they’re ugly and add length to URIs. If you need to specify the format of the body, instead use the Content-Type header

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

**No trailing forward slash:**

Similarly, in the interests of **keeping URIs clean, do not add a trailing forward slash** to the end of URIs.

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

## Key requirements for the API

* It should use web standards where they make sense
* It should be friendly to the developer and be explorable via a browser address bar
* It should be simple, intuitive and consistent to make adoption not only easy but pleasant
* It should provide enough flexibility to power majority of the [Enchant](http://www.enchant.com/) UI
* It should be efficient, while maintaining balance with the other requirements