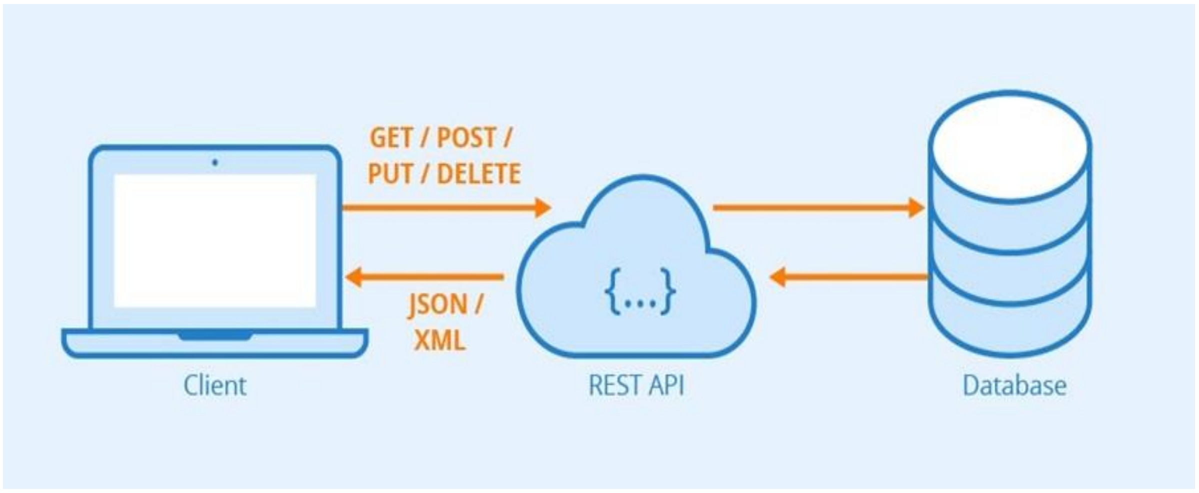
Rest API Best Practices

REST stands for Representational State Transfer. It is a software architectural style created by Roy Fielding in 2000 to guide the design of architecture for the web.

API is a reflective interface through which developers communicate with the data. With the support of a beautifully structured and designed API, ease and comfort can be introduced into the developer's life. However, the critical point here is – perfectly designed REST APIs. If the REST API isn't designed flawlessly, it can create problems for developers instead of easing the user experience. Thus, it is highly critical to use the commonly followed conventions of API design to serve the best solution to your clients or developers.

BEST PRACTICES TO DESIGN REST APIS:

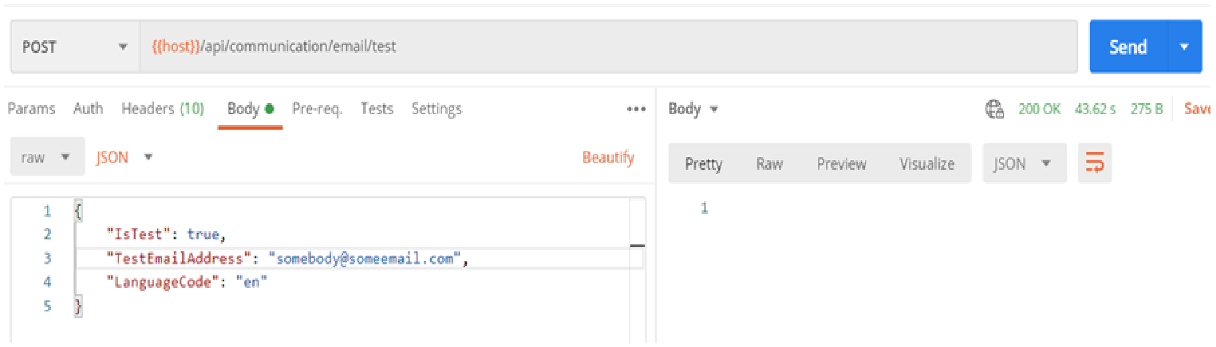


When we are aiming to bring ease and smoothness in your API user's life, then we have to follow the path of the best REST API design practices to avoid tripping over our API's syntax mess. The tried and tested conventions to follow while designing REST APIs are:

1. REST API MUST ACCEPT AND RESPOND WITH JSON

It is a common practice that APIs should accept JSON requests as the payload and also send responses back. JSON is a open and standardized format for data transfer. It is derived from JavaScript in a way to encode and decode JSON via the Fetch API or another HTTP client. Moreover, server-side technologies have libraries that can decode JSON without any hassle.

Let's have a look at an example of API where JSON accepts payloads. A screenshot from Postman to send JSON to our API.



{{host}}/api/communication/email/test

```
{
    "IsTest": true,
    "TestEmailAddress": "somebody@someemail.com",
    "LanguageCode": "en"
}
```

Let's take another example API that accepts JSON payloads. This example will use the Express back end framework for Node.js. We can use the body-parser middleware to parse the JSON request body, and then we can call the res.json method with the object that we want to return as the JSON response as follows:

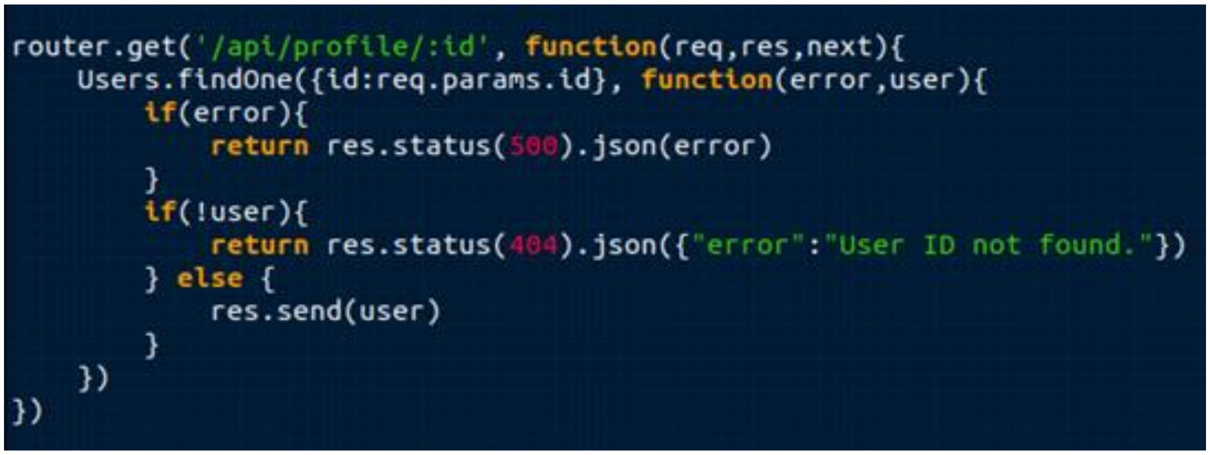
```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();

app.use(bodyParser.json());

app.post('/', (req, res) => {
  res.json(req.body);
});

app.listen(3000, () => console.log('server started'));
```

bodyParser.json() parses the JSON request body string into a JavaScript object and then assigns it to the req.body object.

Set the Content-Type header in the response to application/json; charset=utf-8 without any changes. The method above applies to most other back end frameworks.

2. GO WITH ERROR STATUS CODES

Over 100 status codes have already been built by HTTP. It is a boon for developers to use status codes in their REST API design. With the status codes, developers can instantly identify the issue, which reduces the time of writing parsers to address all the different types of errors. There’s a status code for everything – from finding out the cause of a denied session to locating the missing resource. Developers can quickly implement routines for managing numerous errors based on status codes.



```
router.get('/api/profile/:id', function(req,res,next){
    Users.findOne({id:req.params.id}, function(error,user){
        if(error){
            return res.status(500).json(error)
        }
        if(!user){
            return res.status(404).json({"error":"User ID not found."})
        } else {
            res.send(user)
        }
    })
})
```

3. DON’T USE VERBS IN URLS

If you understood the APIs' basics, you would know that inserting verbs in the URL isn’t a good idea. The reason behind this is that HTTP has to be self-sufficient to describe the purpose of the action. Let’s take an example when you want endpoint to produce a banner image for a post; you have to note the: param is a placeholder for a URI parameter. Your first extinct might be to create this endpoint:

GET: /articles/:slug/generateBanner/

The GET method is only able to say here that you just want to retrieve a banner. So, using this syntax might be beneficial:

GET: /articles/:slug/banner/

Similarly, for the endpoint, it might generate the new article, as shown in this example.

Don't use

POST: /articles/createNewArticle/

Do use

POST: /articles/

4. USE PLURAL NOUNS TO NAME A COLLECTION
When you have to develop the collection in REST API, just go with plural nouns. It makes it easier for humans to understand the meaning of collection without actually opening it. Let’s go through this example:

GET /cars/123

POST /cars

GET /cars

It is clear from the example that ‘car’ is referred to as number 123 from the entire list of "cars". The usage of a plural noun is merely indicating that this is a collection of different cars. Now, look at one another example:

GET /car/123

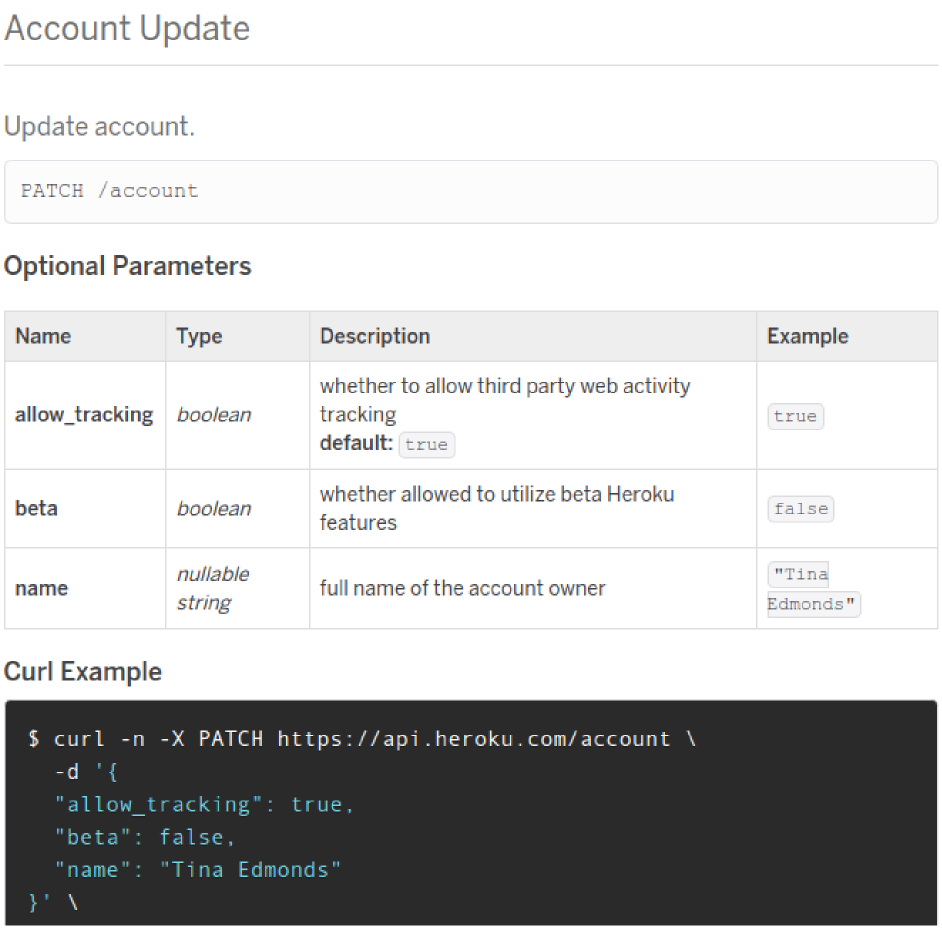
POST /car

GET /car

This example doesn’t clearly show whether there is more than one car in the system or not. For a human reader, it might be challenging to understand, as well.

5. WELL COMPILED DOCUMENTATION
Documentation is one of the important but highly ignored aspects of a REST API structure. The documentation is the first point in the hands of customers to understand the product and critical deciding factor whether to use it or not. One good documentation is neatly presented in a proper flow to make an API development process quicker.

It is a simple principle – the faster developers understand your API, the faster they start using it. Your API documentation must be compiled with precision. It must include all the relevant information such as the endpoint and compatible methods, different parameter options, numerous types of data, and so on. The documentation should be so robust that it can easily walk a new user through your API design.



Account Update

Update account.

```
PATCH /account
```

Optional Parameters

Name	Type	Description	Example
allow_tracking	*boolean*	whether to allow third party web activity tracking **default:** `true`	`true`
beta	*boolean*	whether allowed to utilize beta Heroku features	`false`
name	*nullable string*	full name of the account owner	`"Tina Edmonds"`

Curl Example

```
$ curl -n -X PATCH https://api.heroku.com/account \
  -d '{
  "allow_tracking": true,
  "beta": false,
  "name": "Tina Edmonds"
}' \
```

6. RETURN ERROR DETAILS IN THE RESPONSE BODY

It is convenient for the API endpoint to return error details in the JSON or response body to help a user with debugging. If you can explicitly include the affected field in error, this will be special kudos to you.

```
{
   "error": "Invalid payoad.",
   "detail": {
      "surname": "This field is required."
   }
}
```

7. USE RESOURCE NESTING

Resource objectives always contain some sort of functional hierarchy or are interlinked to one another. However, it is still ideal to limit the nesting to one level in the REST API. Too many nested levels can lose their elegant appeal. If you take a case of the online store into consideration, we can see “users” and “orders” are part of stores. Orders belong to some user; therefore the endpoint structure looks like:

/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 order of a specific users order list

8. USE SSL/TLS
When you have to encrypt the communication with your API, *always* use SSL/TLS. Use this feature without asking any questions.

9. SECURE YOUR API
It is a favorite pastime for hackers to use automated scripts to attack your API server. Thus, your API needs to follow proactive security measures to run operations while safeguarding your sensitive data smoothly. Foremost, your API must have an HTTP Strict Transport Security (HSTS) policy. Up next, you should secure your network from middle man attacks, protocol downgrade attacks, session hijacking, etc., Just use all the relevant security standards to the security of your API.

Perfectly designed REST API stays on the positive side of technical constraints along with taking user experience-based solutions. API is a part of the business strategy; it is a marketing tool for the organization; thus, it is essential to execute APIs in the right manner. That's because unstructured API is a liability rather than an asset.

API End Naming Best Practices:

- **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 example 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 URL.

Example: */users?location=USA* to find all users living in the United 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 interest 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/*

API Response Best Practices:

- **Response Header:**
  - ✓ Provide proper http response status code.
  - ✓ Provide proper content type, file type if any.
  - ✓ Provide cache status if any.

- ✓ Authentication token should provide via response header.
- ✓ Only string data is allowed for response header.
- ✓ Provide content length if any.
- ✓ Provide response data and time.
- ✓ Follow request-response model described before.

❖ **Response Body:**

- ✓ Avoid providing response status, code, message via response body.
- ✓ Use JSON best practices for JSON response body.
- ✓ For single result, can use Sting, Boolean directly.
- ✓ Provide proper JSON encode-decode before writing JSON body.

❖ **Response Cookies:**

- ✓ A Restful API may send cookies just like a regular Web Application that serves HTML.
- ✓ Avoid using response cookies as it is violated stateless principle.
- ✓ If required use cookie encryption, decryption and other policies.

API Request Handling Best Practices:

❖ **When use GET():**

- ✓ GET is used to request something from server with less amount of data to pass.
- ✓ When nothing should change on the server because of your action.
- ✓ When request only retrieves data from a web server by specifying parameters.
- ✓ Get method only carries request url & header not request body.

❖ **When use POST():**

- ✓ POST should be used when the server state changes due to that action.
- ✓ When request needs its body, to pass large amount of data.

- ✓ When want to upload documents, images, video from client to server.

- ❖ **Request Body:**
  - ✓ Request body should be structured in JSON Array/Object pattern.
  - ✓ Request body hold multipart/from-data like images, audio, video etc.
  - ✓ Request body should not hold any auth related information.
  - ✓ ‘Request body should associate with specific request data model, setter getter can used for this.

- ❖ **Request Header:**
  - ✓ Request header should carry all security related information like token, auth etc.
  - ✓ Only string key-pair value is allowed for header.
  - ✓ Request header should provide user agent information of client application.
  - ✓ If necessary CSRF/XSRF should provide via header.
  - ✓ Request header should associate with middleware controller, where necessary.

- ❖ **API Controller Best Practices:**
  - ✓ ·The controllers should always be as clean as possible. We shouldn’t place any business logic inside it.
  - ✓ ·Controllers should be responsible for accepting http request
  - ✓ ·Consider API versioning
  - ✓ ·Use async/await if at all possible.
  - ✓ ·Follow solid principles to manage controller classes.
  - ✓ ·Mention which method is responsible for GET() and which for POST().
  - ✓ ·Controller should be only responsible for calling model, return response, redirect to action etc.

- ❖ **API Middleware Controller Best Practices:** Middleware is a special type of controller executed after request but before in response. It is a type of filtering mechanism to ensure API securities and more. Middleware acts as a bridge between a request and a response.

  Middleware Uses:

- ✓ Use to implement API key, user-agent restriction, CSRF, XSRF security, token-based API authentication.
- ✓ ·Use to implement API request rate limit.
- ✓ ·Logging of incoming HTTP requests.
- ✓ ·Redirecting the users based on requests.
- ✓ ·Middleware can inspect a request and decorate it, or reject it, based on what it finds.
- ✓ ·Middleware is most often considered separate from your application logic.
- ✓ ·Middleware gives you enough freedom to create your own security mechanism.

Conclusion:
In this article, you learned about the several best practices to bear in mind when you're building REST APIs.

It is important to put these best practices and conventions into practice so you can build highly functional applications that work well, are secure, and ultimately make the lives of your API consumers easier.

Thank you for reading. Now, go make some APIs with these best practices.