REST API

ARIHARASUDHAN

REST

REST is an Architectural Style or a design pattern for building API's. Before getting into a deeper overview, we must have a proper understanding of the keyterms such as CLIENT and RESOURCE. A Client can be a person or a software who uses the API. We can consider a browser that access a Weather API as an analogy. A Resource is provided by the API. If you want to access a data about a person in Instagram, you might use Instagram API to obtain those data. The Data you obtain is the Resource. REST stands for Representational State Transfer.

It means, when a REST API is called, it will return the client a representation of the state of the requested resource. For an instance, when Instagram API is called, the API will return the state of that user, including their name, the number of posts that user posted on Instagram so far, how many followers they have, and more.

That representation of the state can be in JSON , XML or HTML format. What the server does when you, the client, call one of its APIs depends on two main things that you need to provide to the server.

★ An identifier for the resource you are interested in. This is the URL for the resource, also known as the **endpoint**. In fact, URL stands for Uniform Resource Locator.

★ The operation you want the server to perform on that resource, in the form of an **HTTP method**. The common HTTP methods are GET, POST, PUT, and DELETE.

For example, fetching a specific Instagram user, using Instagram's RESTful API, will require a URL that identify that user and the HTTP method GET.

www.instagram.com/aravind_ariharasudhan_ has the unique identifier for the instagram page of Ariharasudhan. Instagram uses the username as the identifier, and indeed Instagram usernames are unique. The HTTP method **GET** indicates that we want to **get** the state of that user.

The Growth Of The Web

Tim Berner's Lee developed the World Wide Web which began to grow, at times exponentially. Within five years, the number of web users skyrocketed to 40 million. At one point, the number was doubling every two months. In fact, the Web was growing too large, too fast, and it was heading toward collapse. The Web's traffic was outgrowing the capacity of the Internet infrastructure. With such rapid expansion, it was unclear if the Web would scale to meet the increasing demand.

Web Architecture

The constraints, which Roy Fielding grouped into six categories and collectively referred to as the Web's architectural style, are

- ★ Client-server
- ★ Layered system
- ★ Stateless
- ★ Uniform interface
- ★ Cache
- ★ Code-on-demand

★ Client-Server

The Web is a client-server based system, in which clients and servers have distinct parts to play. They may be implemented and deployed independently, so long as they conform to the Web's uniform interface.

★ Uniform Interface

The interactions between the Web's components—meaning its clients, servers, and network-based intermediaries—depend on the uniformity of their interfaces. The Uniform Interface's four constraints are,

1. Identification of Resources : Each Web-content can be addressed by a unique identifier, such as a URL. For an example, a particular web page URI, like www.google.com/images, uniquely identifies the respective page content.

2.Manipulation of Resources through Representations Clients manipulate representations of resources. A document might be represented as HTML to a web browser, and as JSON to an automated program. The key idea here is that the representation is a way to interact with the resource but it is not the resource itself.

3. Self-Descriptive Messages : A resource's desired state can be represented within a client's request message. A resource's current state may be represented within the response message that comes back from a server. The self-descriptive messages may include metadata to convey additional details regarding the resource state, the representation format and size, and the message itself. An HTTP message provides headers to organize the various types of metadata into uniform fields.

4. Hypermedia as the Engine of Application State (HATEOAS): A resource's state representation includes links to related resources. The presence, or absence, of a link on a page is an important part of the resource's current state.

★ Layered System

The layered system constraints enable network-based intermediaries such as proxies and gateways to be transparently deployed between a client and server using the Web's uniform interface.

★ Caching

A cache may exist anywhere along the network path between the client and server. They can be in an organization's web server network, within specialized content delivery networks (CDNs), or inside a client itself.

★ Stateless

The stateless constraint dictates that a web server is not required to memorize the state of its client applications. As a result, each client must include all of the contextual information that it considers relevant in each interaction with the web server.

★ Code On Demand

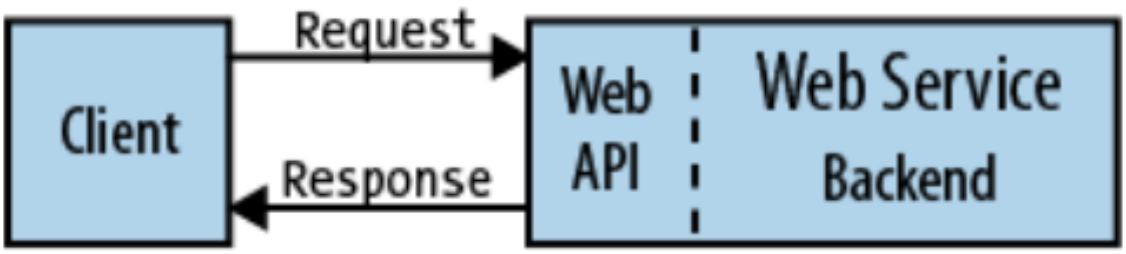
The Web makes heavy use of code-on-demand, a constraint which enables web servers to temporarily transfer executable programs, such as scripts or plug-ins, to clients.

HBD REST

In the year 2000, after the Web's scalability issue was overcome, Fielding named and described the Web's architectural style in his Ph.D. dissertation. "Representational State Transfer" (REST) is the name that Fielding gave to his description of the Web's architectural style, which is composed of the constraints outlined above.

REST API's

A web service is a resource that is available over the internet. It's valuable because it provides functionality other applications can use, such as payment processing, logins, and database storage. In other hand, An Application Programming Interface (API) is a way for two or more computer programs to communicate with each other. It is a type of software interface. Web API is the face of a web service, directly listening and responding to client requests.



URI's

The REST API uses URI (Uniform Resource Identifier) to address resources. URI designs range from Crystal Clear Design which is so clear:

http://api.example.restapi.org/india/tamilnadu/tirunelveli/ariharasudhan

to those that are much harder for people to understand, such as:

http://api.example.restapi.org/68dd0-a9d3-11e0-9f1c-0802320ac9b66

URI Format

A proper URI format must follow the syntax,

URI = scheme "://" authority "/" path ["?" query] ["#" fragment]

★ Forward slash separator (/) must be used to indicate a hierarchical relationship

★ A trailing forward slash (/) should not be included in URIs. Many web components and frameworks will treat the following two URIs equally :

http://api.example.restapi.org/india/

http://api.example.restapi.org/india/

Different URIs map to different resources. If the URIs differ, then so do the resources, and vice versa.

★ Hyphens (-) should be used to improve the readability of URIs

http://arihara-sudhan.github.io

★ Underscores (_) should not be used in URIs

Text viewer applications (browsers, editors, etc.) often underline URIs to provide a visual cue that they are clickable. Depending on the application's font, the underscore (_) character can either get partially obscured or completely hidden by this underlining.

★ Lowercase letters should be preferred in URI

When convenient, lowercase letters are preferred in URI paths since capital letters can sometimes cause problems. URIs are described as case-sensitive except for the scheme and host components.

http://arihara-sudhan.github.io/resume.html 1

HTTP://ARIHARA-SUDHAN.GITHUB.IO/resume.html 2

http://arihara-sudhan.github.io/Resume.html 3

URI 1 is fine. The URI format specification considers URI 2 to be identical to URI 1. But, URI 3 is not the same as URIs 1 and 2, which may cause unnecessary confusion.

★ File extensions should not be included in URIs

A REST API should not include artificial file extensions in URIs to indicate the format of a message's entity body. Instead, they should rely on the media type, as communicated through the Content-Type header, to determine how to process the body's content.

URI Authority Design

This section covers the naming conventions that should be used for the authority portion of a REST API.

★ Consistent subdomain names should be used. The full domain name of an API should add a subdomain named api. For example :

http://api.soccer.restapi.org

★ Consistent subdomain names should be used for your client developer portal

If an API provides a developer portal, by convention it should have a subdomain labeled developer. For example:

http://developer.soccer.restapi.org

URI - Resource Modeling

The URI path conveys a REST API's resource model, with each forward slash separated path segment corresponding to a unique resource within the model's hierarchy. For example, this URI design http://football.org/leagues/november/league1 indicates that each of these URIs should also identify an addressable resource:

http://football.org/leagues/november/league1

http://football.org/leagues/november

http://football.org/leagues

URI - Resource Archetypes

When modeling an API's resources, we can start with the some basic resource archetypes. The resource archetypes help us consistently communicate the structures and behaviors that are commonly found in REST API designs. A REST API is composed of four distinct resource archetypes : document, collection, store, and controller.

★ DOCUMENT SINGULAR

A document's state representation typically includes both fields with values and links to other related resources. Each URI below identifies a document resource:

http://mypage.org/pages/firstpage/childpages/firstchildpage

http://mypage.org/pages/firstpage

With its ability to bring many different resource types together under a single parent, a document is a logical candidate for a REST API's root resource, which is also known as the docroot. The example URI below identifies the docroot :

http://mypage.org

★ COLLECTION PLURAL

A collection resource is a server-managed directory of resources. Clients may propose new resources to be added to a collection. Following define such collection

http://mypage.org/pages/firstpage/childpages

http://mypage.org/pages

★ STORE PLURAL

A store is a client-managed resource repository. A store resource lets an API client put resources in, get them back out, and decide when to delete them. Let's store elephant in our favourite store.

PUT /users/1234/favorites/elephant

★ CONTROLLER VERB

It defines the procedural concept. Controller names typically appear as the last segment in a URI path, with no child resources to follow them in the hierarchy. The example below shows a controller resource that allows a client to resend an alert

POST /alerts/245743/resend

Besides, Variable path segments may be substituted with identity-based values. http://api.example.restapi.org/india can be changed as http://api.example.restapi.org/68dd0-a9d3 and CRUD operations shouldn't be mentioned in the URL like DELETE /deleteUser/1234

URI - Query Design

As a component of a URI, the query contributes to the unique identification of a resource. Consider the following example:

```
http://mycollege.org/students                                    1
http://mycollege.org/students/students?name=ari                  2
```

Here, URI 1 returns the students list. Meanwhile,URI 2 returns the student named ‘ari’. The query component can provide clients with additional interaction capabilities such as searching and filtering. Besides these operations, it can also be used to perform pagination of collection or for storing the results.

A REST API client can use the query component to paginate collection and store results with the pageSize and pageStartIndex parameters. The pageSize parameter specifies the maximum number of contained elements to return in the response. The pageStartIndex parameter specifies the zero-based index of the first element to return in the response. For example:

```
GET /users?pageSize=25&pageStartIndex=50
```

Request Methods

Clients specify the desired interaction method in the Request-Line part of a request message. The Request-Line syntax is,

Request-Line = Method SP Request-URI SP HTTP-Version CRLF

Each method has specific, well-defined semantics within the context of a REST API's resource model.

GET : To retrieve a representation of a resource's state

```
$ curl -v http://api.example.restapi.org/greeting ❶

> GET /greeting HTTP/1.1 ❷
> User-Agent: curl/7.20.1 ❸
> Host: api.example.restapi.org
> Accept: */*

< HTTP/1.1 200 OK ❹
< Date: Sat, 20 Aug 2011 16:02:40 GMT ❺
< Server: Apache
< Expires: Sat, 20 Aug 2011 16:03:40 GMT
< Cache-Control: max-age=60, must-revalidate
< ETag: text/html:hello world
< Content-Length: 130
< Last-Modified: Sat, 20 Aug 2011 16:02:17 GMT
< Vary: Accept-Encoding
< Content-Type: text/html

<!doctype html><head><meta charset="utf-8"><title>Greeting</title></head> ❻
<body><div id="greeting">Hello World!</div></body></html>
```

HEAD : To retrieve the metadata associated with the resource's state

```
$ curl --head http://api.example.restapi.org/greeting

HTTP/1.1 200 OK ❶
Date: Sat, 20 Aug 2011 16:02:40 GMT ❷
Server: Apache
Expires: Sat, 20 Aug 2011 16:03:40 GMT
Cache-Control: max-age=60, must-revalidate
ETag: text/html:hello world
Content-Length: 130
Last-Modified: Sat, 20 Aug 2011 16:02:17 GMT
Vary: Accept-Encoding
Content-Type: text/html
```

PUT : To insert or update a stored resource

```
PUT /accounts/4ef2d5d0-cb7e-11e0-9572-0800200c9a66/buckets/objects/4321
```

DELETE : To remove a resource from its parent

```
DELETE /accounts/4ef2d5d0-cb7e-11e0-9572-0800200c9a66/buckets/objects/4321
```

POST : To create a new resource within a collection

```
POST /leagues/seattle/teams/trebuchet/players
```

POST : To execute a controller

```
POST /alerts/245743/resend
```

OPTIONS : To retrieve resource metadata that includes an Allow header value

```
Allow: GET, PUT, DELETE
```

Response Status Codes

REST APIs use the Status-Line part of an HTTP response message to inform clients of their request's overarching result. The Syntax of such Response Status will be,

Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase

Some categories are,

1xx - Informational

2xx - Success

3xx - Redirection

4xx - Client Error

5xx - Server Error

Some Important Status Codes

200 (“OK”) should be used to indicate nonspecific success

201 (“Created”) must be used to indicate successful resource creation

202 (“Accepted”) is for successful start of an asynchronous action

204 (“No Content”) is for when the response body is intentionally empty

301 (“Moved Permanently”) should be used to relocate resources

302 (“Found”) should not be used

303 (“See Other”) should be used to refer the client to a different URI

304 (“Not Modified”) should be used to preserve bandwidth

307 (“Temporary Redirect”) is for tell to resubmit request to another URI

400 (“Bad Request”) may be used to indicate nonspecific failure

401 (“Unauthorized”) is when there is a problem with the credentials

403 (“Forbidden”) is to forbid access

404 (“Not Found”) is when a URI cannot be mapped to a resource

405 (“Method Not Allowed”) is when the HTTP method is not supported

406 (“Not Acceptable”) is when the a mediatype can’t be served

409 (“Conflict”) should be used to indicate a violation of resource state

412 (“Precondition Failed”) is to support conditional operations

415 (“Unsupported Media Type”) : The media type can’t be processed

500 (“Internal Server Error”) should be used to indicate API malfunction

Headers

Various forms of metadata may be conveyed through the entity headers contained within HTTP's request and response messages. HTTP defines a set of standard headers, some of which provide information about a requested resource. Other headers indicate something about the representation carried by the message. Finally, a few headers serve as directives to control intermediary caches.

Request URI: http://www.example.com

```
HTTP/1.1 200 OK
Content-Encoding: gzip
Age: 521648
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Date: Fri, 06 Mar 2020 17:36:11 GMT
Etag: "3147526947+gzip"
Expires: Fri, 13 Mar 2020 17:36:11 GMT
Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT
Server: ECS (dcb/7EC9)
Vary: Accept-Encoding
X-Cache: HIT
Content-Length: 648
```

★ Content-Type must be specified

★ Content-Length must be specified

★ Last Modified must be specified

★ ETag should be used in response

The value of ETag is an opaque string that identifies a specific “version” of the representational state contained in the response’s entity. Clients may choose to save an ETag header’s value for use in future GET requests, as the value of the conditional If-None-Match request header. If the REST API concludes that the entity tag hasn’t changed, then it can save time and bandwidth by not sending the representation again.

★ Location must be used to specify the URI of a newly created resource

★ Cache-Control, Expires, and Date response headers should be used

CONTENT-TYPE

To identify the form of the data contained within a request or response message body, the Content-Type header’s value references a media type. Media types have the following syntax:

```
type "/" subtype *( ";" parameter )
```

The two examples below demonstrate a Content-Type header value that references a media type with a single charset parameter :

```
Content-type: text/html; charset=ISO-8859-4
```

```
Content-type: text/plain; charset="us-ascii
```

Some commonly used registered types are,

- text/plain
- text/html
- image/jpeg
- application/xml
- application/javascript
- application/json

★ Application-specific media types should be used

```
Accept:  application/json
```

★ Type selection using a query parameter may be supported

```
GET /bookmarks/mikemassedotcom?accept=application/xml
```

Message Body

A REST API commonly uses a response message's entity body to help convey the state of a request message's identified resource. REST APIs often employ a text-based format to represent a resource state as a set of meaningful fields. Today, the most commonly used text formats are XML and JSON.

★ JSON should be supported for resource representation

★ JSON should be well-formed

The following example shows well-formed JSON with all names enclosed in double quotes.

```
{
    "firstName" : "Osvaldo",
    "lastName" : "Alonso",
    "firstNamePronunciation" : "ahs-VAHL-doe",
    "number" : 6,  ❶
    "birthDate" : "1985-11-11"  ❷
}
```

❶ JSON supports number values directly, so they do not need to be treated as strings.

❷ JSON does not support date-time values, so they are typically formatted as strings.

★ XML and other formats can be optionally used

A REST API must leverage the message “envelope” provided by HTTP. In other words, the body should contain a representation of the resource state, without any additional, transport-oriented wrappers.

A HyperMedia Representation

★ A General Response

```
{} mine.json > ...
1   {
2     "_links": {
3       "self": {
4         "href": "http://example.com/api/book/hal-cookbook"
5       }
6     },
7     "id": "hal-cookbook",
8     "name": "HAL Cookbook"
9   }
```

★ An Embedded Response

```
{} mine.json > ...
1   {
2       "_links": {
3         "self": {
4           "href": "http://example.com/api/book/hal-cookbook"
5         }
6       },
7       "_embedded": {
8         "author": {
9           "_links": {
10            "self": {
11              "href": "http://example.com/api/author/shahadat"
12            }
13          },
14          "id": "shahadat",
15          "name": "Shahadat Hossain Khan",
16          "homepage": "http://author-example.com"
17        }
18      },
19      "id": "hal-cookbook",
20      "name": "HAL Cookbook"
21    }
```

★ Collection

mine.json > ...

```
{
    "_links": {
      "self": {
        "href": "http://example.com/api/book/hal-cookbook"
      },
      "next": {
        "href": "http://example.com/api/book/hal-case-study"
      },
      "prev": {
        "href": "http://example.com/api/book/json-and-beyond"
      },
      "first": {
        "href": "http://example.com/api/book/catalog"
      },
      "last": {
        "href": "http://example.com/api/book/upcoming-books"
      }
    },
    "_embedded": {
      "author": {
        "_links": {
          "self": {
            "href": "http://example.com/api/author/shahadat"
          }
        },
        "id": "shahadat",
        "name": "Shahadat Hossain Khan",
        "homepage": "http://author-example.com"
      }
    },
    "id": "hal-cookbook",
    "name": "HAL Cookbook"
}

```

Error Representation

When formatted with JSON, an `Error` has the following consistent form:

```
{
    "id" : Text,  ❶
    "description" : Text  ❷
}
```

Error-Representing-Schema must be consistent.

mine.json > ...

```
1   {
2   "elements" : [{
3   "id" : "Update Failed",
4   "description" : "Failed to update /users/1234"
5   }]
6   }
```

REST API Design Practices

★ Use JSON as the Format for Sending and Receiving Data

```
JS some.js > ...
1   const express = require('express');
2   const bodyParser = require('body-parser');
3
4   const app = express();
5
6   app.use(bodyParser.json());
7
8   app.post('/', (req, res) => {
9     res.json(req.body);
10  });
11  app.listen(3000, () => console.log('server started'));
```

With XML for example, it's often a bit of a hassle to decode and encode data. JavaScript, for example, has an inbuilt method to parse JSON data through the fetch API because JSON was primarily made for it. But if you are using any other programming language such as Python or PHP, they now all have methods to parse and manipulate JSON data as well.

★ Use Nouns Instead of Verbs in Endpoints

When you're designing a REST API, you should not use verbs in the endpoint paths. The endpoints should use nouns, signifying what each of them does. This is because HTTP methods such as GET, POST, PUT, PATCH, and DELETE are already in verb form for performing basic CRUD (Create, Read, Update, Delete) operations. GET, POST, PUT, PATCH, and DELETE are the commonest HTTP verbs. There are also others such as COPY, PURGE, LINK, UNLINK, and so on.

So, for example, an endpoint should not look like this: https://mysite.com/getPosts or https://mysite.com/createPost Instead, it should be something like this: https://mysite.com/posts

JS some.js > ...

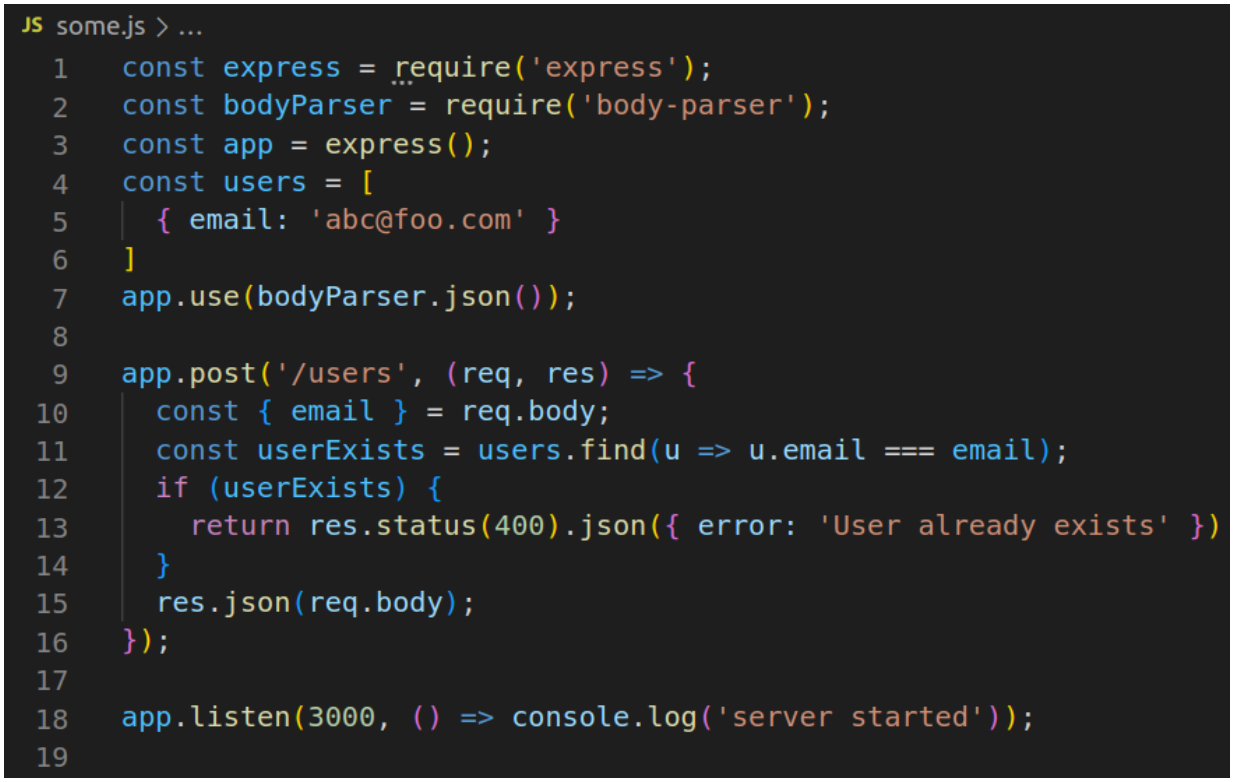
```
1   const express = require('express');
2   const bodyParser = require('body-parser');
3   const app = express();
4   app.use(bodyParser.json());
5
6   app.get('/articles', (req, res) => {
7     const articles = [];
8     res.json(articles);
9   });
10
11  app.post('/articles', (req, res) => {
12    res.json(req.body);
13  });
14
15  app.put('/articles/:id', (req, res) => {
16    const { id } = req.params;
17    res.json(req.body);
18  });
19
20  app.delete('/articles/:id', (req, res) => {
21    const { id } = req.params;
22    res.json({ deleted: id });
23  });
24
25  app.listen(3000, () => console.log('server started'));
```

★ Name Collections with Plural Nouns

You can think of the data of your API as a collection of different resources from your consumers. If you have an endpoint like https://mysite.com/post/123, it might be okay for deleting a post with a DELETE request or updating a post with PUT or PATCH request, but it doesn't tell the user that there could be some other posts in the collection. This is why your collections should use plural nouns. So, instead of https://mysite.com/post/123, it should be https://mysite.com/posts/123

★ Use Status Codes in 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.



★ Use Nesting on Endpoints to Show Relationships

```
JS some.js > ...
1   const express = require('express');
2   const bodyParser = require('body-parser');
3   const app = express();
4   app.use(bodyParser.json());
5   app.get('/articles/:articleId/comments', (req, res) => {
6     const { articleId } = req.params;
7     const comments = [];
8     res.json(comments);
9   });
10
11  app.listen(3000, () => console.log('server started'));
```

Oftentimes, different endpoints can be interlinked, so you should nest them so it's easier to understand them. For example, in the case of a multi-user blogging platform, different posts could be written by different authors, so an endpoint such as https://mysite.com/posts/author would make a valid nesting in this case. In the same vein, the posts might have their individual comments, so to retrieve the comments, an endpoint like https://mysite.com/posts/postId/comments would make sense. You should avoid nesting that is more than 3 levels deep as this can make the API less elegant and readable.

★ Use Filtering, Sorting, and Pagination to Retrieve the Data Requested

Sometimes, an API's database can get incredibly large. If this happens, retrieving data from such a database could be very slow. Filtering, sorting, and pagination are all actions that can be performed on the collection of a REST API. This lets it only retrieve, sort, and arrange the necessary data into pages so the server doesn't get too occupied with requests. Eg : https://mysite.com/posts?tags=javascript This endpoint will fetch any post that has a tag of JavaScript.

```
JS some.js > ...
1   const express = require('express');
2   const bodyParser = require('body-parser');
3   const app = express();
4
5   const employees = [
6     { firstName: 'Jane', lastName: 'Smith', age: 20 },
7     //...
8     { firstName: 'John', lastName: 'Smith', age: 30 },
9     { firstName: 'Mary', lastName: 'Green', age: 50 },
10  ]
11  app.use(bodyParser.json());
12  app.get('/employees', (req, res) => {
13    const { firstName, lastName, age } = req.query;
14    let results = [...employees];
15    if (firstName) {
16      results = results.filter(r => r.firstName === firstName);
17    }
18
19    if (lastName) {
20      results = results.filter(r => r.lastName === lastName);
21    }
22
23    if (age) {
24      results = results.filter(r => +r.age === +age);
25    }
26    res.json(results);
27  });
28
29  app.listen(3000, () => console.log('server started'));
```

★ Use SSL for Security

SSL stands for secure socket layer. It is crucial for security in REST API design. This will secure your API and make it less vulnerable to malicious attacks. Other security measures you should take into consideration include: making the communication between server and client private and ensuring that anyone consuming the API doesn't get more than what they request. 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. 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.

★ Be Clear with Versioning

REST APIs should have different versions, so you don't force clients (users) to migrate to new versions. This might even break the application if you're not careful. One of the commonest versioning systems in web development is semantic versioning. An example of semantic versioning is 1.0.0, 2.1.2, and 3.3.4. The first number represents the major version, the second number represents the minor version, and the third represents the patch version. Many RESTful APIs from tech giants and individuals usually comes like this: https://mysite.com/v1 for version 1 https://mysite.com/v2 for version 2.

```
JS some.js > ...
1   const express = require('express');
2   const bodyParser = require('body-parser');
3   const app = express();
4   app.use(bodyParser.json());
5
6   app.get('/v1/employees', (req, res) => {
7     const employees = [];
8     res.json(employees);
9   });
10
11  app.get('/v2/employees', (req, res) => {
12    const employees = [];
13    res.json(employees);
14  });
15
16  app.listen(3000, () => console.log('server started'));
```

★ 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.

FINE