**Why Is API Design So Important?**

People ask this question quite a lot, and to answer this:

REST APIs are the face of any service, and therefore they should:

1. Be easy to understand so that integration is straightforward

2. Be well documented, so that semantic behaviors are understood (not just syntactic)

3. Follow accepted standards such as HTTP.

**1.Keep it simple**

We need to make sure that the base URL of the API is simple. For example, if we want to design APIs for products, it should be designed like

/products  
/products/12345

**2.Use nouns and NOT the verbs**

A lot of developers make this mistake. They generally forget that we have HTTP methods with us to describe the APIs better and end up using verbs in the API URLs. For example, if we want get all products should be

/products

and NOT as shown below

/getAllProducts

**3.Plurals or Singulars**

Generally, we prefer to use plurals but there is no hard rule that one can’t use singular for resource name. The ideology behind using plurals is:

We are operating on one resource from collection of resources so to depict collection we use plural

For example, in the case of…

GET /users/123

the client is asking to retrieve a resource from users collection with id 123. While creating a resource we want to add one resource to current collection of resources, so the API looks like the below…

POST /users

**4.** **Use of right HTTP methods**

RESTful APIs have various methods to indicate the type of operation we are going to perform with this API —

GET — To get a resource or collection of resources.

POST — To create a resource or collection of resources.

PUT/PATCH — To update the existing resource or collection of resources.

DELETE — To delete the existing resource or the collection of resources.

We need to make sure we use the right HTTP method for given operation.

**5. Don’t Misuse Safe Methods (Idempotency)**

Safe methods are HTTP methods which return the same resource representation irrespective of how many times are called by client. GET, HEAD methods are defined as safe, meaning they are only intended for retrieving data and should not change a state of a resource on a server. Don’t use GET to delete content, for example…

GET /users/123/delete

It’s not like this can’t be implemented, but HTTP specification is violated in this case.

Use HTTP methods according to the action which needs to be performed.

**6. Use parameters**

Don’t create different URIs for fetching resources with filtering, searching, or sorting parameters. Try to keep the URI simple, and add query parameters to depict parameters or criteria to fetch a resource (single type of resource)

6.1.Filtering:

Use query parameters defined in URL for filtering a resource from server. For example, if we would like to fetch all published posts by user you can design an API such as:

GET /users/123/posts?state=published

In the example above, state is the filter parameter

6.2.Searching:

To get the results with powerful search queries instead of basic filters, one could use multiple parameters in a URI to request to fetch a resource from server.

GET /users/123/posts?state=published&ta=scala

The above query searches for posts which are published with the Scala tag. It’s very common today for Solr to be used as search tool as it provides advanced capabilities to search for a document and you can design your API such as:

GET /users/123/posts?q=sometext&fq=state:published,ta:scala

This will search posts for free text “sometext”(q) and filter results on fq state as published and having tag Scala.

6.3.Sorting:

ASC and DESC sorting parameters can be passed in URL such as:

GET /users/123/posts?sort=-updated\_at

Returns posts sorted with descending order of update date time.

**7. Use proper HTTP codes**

We have plenty of HTTP codes. Most of us only end up using two — 200 and 500! This is certainly not a good practice. Following are some commonly used HTTP codes.

200 OK — This is most commonly used HTTP code to show that the operation performed is successful.

201 CREATED — This can be used when you use POST method to create a new resource.

202 ACCEPTED — This can be used to acknowledge the request sent to the server.

400 BAD REQUEST — This can be used when client side input validation fails.

401 UNAUTHORIZED / 403 FORBIDDEN— This can be used if the user or the system is not authorised to perform certain operation.

404 NOT FOUND— This can be used if you are looking for certain resource and it is not available in the system.

500 INTERNAL SERVER ERROR — This should never be thrown explicitly but might occur if the system fails.

502 BAD GATEWAY — This can be used if server received an invalid response from the upstream server.

**8. HATEOAS**

Hypermedia As Transfer Engine Of Application State is a constraint of the REST application architecture that distinguishes it from other network application architectures.

It provides ease of navigation through a resource and its available actions. This way a client doesn’t need to know how to interact with an application for different actions, as all the metadata will be embedded in responses from the server.

To understand it better let’s look at the below response of retrieve user with id 123 from the server:

{

“name”: “John Doe”,

“links”: [

{

“rel”: “self”,

“href”: “http://localhost:8080/users/123"

},

{

“rel”: “posts”,

“href”: “http://localhost:8080/users/123/posts"

},

{

“rel”: “address”,

“href”: “http://localhost:8080/users/123/address"

}

]

}

Sometimes it’s easier to skip the links format, and specify links as fields of a resource as below:

{

“name”: “John Doe”,

“self”: “http://localhost:8080/users/123",

“posts”: “http://localhost:8080/users/123",

“address”: “http://localhost:8080/users/123/address"

}

**9. Security**

API security is the important aspects, having a vulnerability in the system opens a way for an attacker to perform malicious activity. Before deployment of restful API. Developer has to identify the vulnerabilities & fix the potential security bugs ASAP otherwise, it threatens the company’s database.

->Use SSL to secure all your API’s.

->Use Industry-standard for authentication and authorization like JWT, Oauth2. ->Authenticate the API, before responding to the request.

->Don’t store sensitive data in the JWT payload, as it is easy to decode.

->Use encryption on all sensitive data, do not store the raw password in databases, always encrypt the password before storing it.

Rate Limiting to protect against DDoS attacks/brute-force attacks.

->Return 429 “Too Many Requests” – used to notify that too many requests came quickly from the same origin.

Revoke the client credential or blacklist if the client violates the usage of API.

-> Validate all the inputs before responding to a request. Allowing this invalid data into our application could cause unpredictable results.

-> Don’t pass sensitive information in URLs like password, JWT token, API keys as this information stored in browser & server logs.

Example – https://example.com/login/username=abcd123456789&password=123321

Above URL expose password. So, never use this form of security.

**10. Swagger for Documentation**

Swagger is a widely-used tool to document REST APIs that provides a way to explore the use of a specific API, therefore allowing developers to understand the underlying semantic behavior. It’s a declarative way of adding documentation using annotations which further generates a JSON describing APIs and their usage.

**11. Use Proper Error Messages**

It is always a good practice to keep set of error messages application sends and respond that with proper id. For example, if you use Facebook graph APIs, in case of errors, it returns message like this —

{

"error": {

"message": "(#803) Some of the aliases you requested do not exist: products",

"type": "OAuthException",

"code": 803,

"fbtrace\_id": "FOXX2AhLh80"

}

}

**12.Versioning**

Versioning of APIs is very important. Many different companies use versions in different ways, some use versions as dates while some use versions as query parameters. I generally like to keep it prefixed to the resource. For instance —

/v1/products

/v2/products

I would also like to avoid using /v1.2/products as it implies the API would be frequently changing. Also dots (.) might not be easily visible in the URLs. So keep it simple.

It is always good practice to keep backward compatibility so that if you change the API version, consumers get enough time to move to the next version.

**Characteristics of Good Restful API’s**

* API should do one thing and do it well. And keep it simple
* Avoid long parameter lists.
* Use pagination & support sorting by date, numbers of records per page.
* Proper versioning of the restful API.
* Readable and intuitive: The interface should exactly do what its name and protocol suggest.
* Stateless: No method depends on the result of another one.
* Error handling should be done with HTTP status codes.
* Proper parameter names, naming conventions, lowercase letters preferred in URI paths. do not abbreviate.
* Highly available & secure.
* Version the API, Use pagination, Sorting
* API’s should be stateless.
* Good documentation: Developer love good documentation.
* Hard to exploit. All validation and edge cases should have to cover with proper HTTPS code.
* KISS [keep it simple silly].