REST API design best practices – Part 1

Description

Introduction

In this post we are going to see some of the basic REST API design best practices. Most of the developers underestimate the design aspect while working with REST APIs. There are a lot of ways in which we can design REST APIs, but we are not going to cover in a single post rather we are make a series of blog post just to keep the content short and concise.

Rule#1 – Stateless API

Statelessness means the servers will not maintain any state with the clients, therefore the client can interact with any server without being tied to a specific server. This is important because statelessness makes the APIs more scalable without relying on any state.

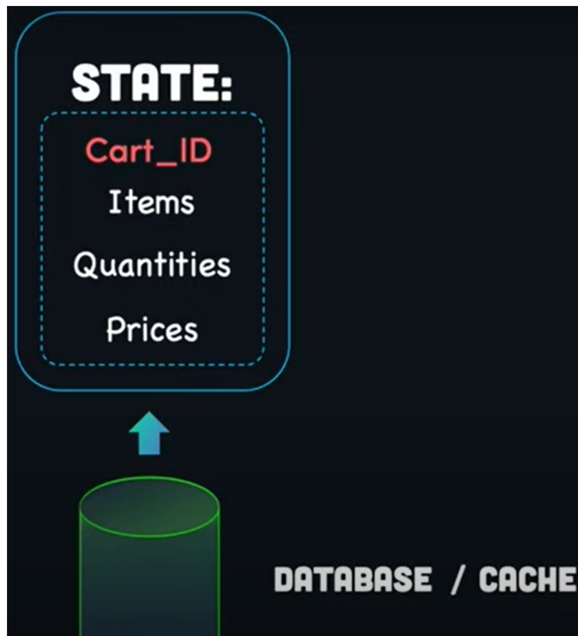
default watermark

So each client request to the API ***contains all the necessary information*** needed to process the request, and the app does not maintain any session state or context information between request.

Rule#2 – Making Stateful Apps Stateless

There are some scenarios where we need to maintain the state of the application E.g. maintaining the state of a Shopping cart in an ecommerce website. The actions can be add, update, retrieve or delete the items from the shopping cart.

In order to maintain the state, first we need to identify what all things the state should contain. In case of shopping cart we can keep data such as **Items**, **Quantities** & **Prices**.



Instead of storing the state within the application we need to store it externally, For example a database or a cache. By doing this, we ensure that the app does not rely on any internal state, and app can operate independently. We can make use of IDs such as **CART_ID** in this case to get the state of the user.

default watermark

Rule#3 – Use noun to represent resources

We should always use Noun to represent resources. Most of the inexperienced developers sometimes use verbs to represent resources, which should be avoided in order to adhere to REST API design best practices. Lets see with an example

https://e-commerce.com/v1/store/**items**/{id}

https://e-commerce.com/v1/store/**getItems**/{id}

In the second link, if you see, the URL contains ***getItems*** which is a verb. The reason this should be avoided is because the HTTP method i.e. GET, it is understood that the API is returning response and is self explanatory.

Rule#4 – Use Plural form to represent resources

Always use plural noun form to represent resources.

examplewebsite.com/v1/store/**items**/{id}

examplewebsite.com/v1/store/**item**/{id}

Rule#5 – Use Hyphens to improve readability

Hyphen separates words and we should use hyphens to improve the readability. Some may think why not use underscore! because underscore are considered as part of the word and hyphens as word separators. Hyphens enable search engines to index each individual word.

This rule comes from Mark Masse's "REST API Design Rulebook" from Oreilly.

If you see the link of this post, it has hyphens – https://springmicroservices.com/best-practices-for-rest-api-design

Hence hyphen qualifies as one of the important best practices for REST API design.

Rule#6 – Avoid deeper collection hierarchy

default watermark

Collection is a group of resources.

/orders This endpoint represents a collection of orders.
/orders/102 This endpoint represents information about a specific order.

Ideally we should not go deeper than **/collection/resource/collection**

For complex scenario such as **/customers/3/orders/101/products** which is three levels deep. This level of complexity is difficult to maintain and it will not be flexible in case the relationship between resources will change in the future. Instead we can create two different URIs for the same requirement.

/customers/3/orders
/orders/99/products

Rule#7 – Maintain API versioning

Imagine there are thousand of consumers consuming your API, and some of the consumers require a change in the payload. But if you change the response payload, chances are high that some of the other consumers may break or function improperly. Hence versioning the APIs is very important best practices for REST API design.

Below example shows two variations you can implement for versioning. Most of the developers use the first approach.

examplewebsite.com/**v1**/store
examplewebsite.com/store**?version=2**

Rule#8 – HATEOAS

HATEOAS is an acronym for “Hypermedia As The Engine Of Application State”. It means that hypertext should be used to find your way through the API. This allows us navigation to related resources without prior knowledge of the URI scheme. This is just a good thing to know that it exists.

A simple JSON presentation is traditionally rendered as:

```
{
    "accountnumber" : "123456",
    "currentbalance" : "2000",
}
```

A HATEOAS-based response would look like below. Apart from the fact that accountnumber has 2000 dollars (US) in our account, we can see option to deposit more money.

default watermark

```
{
    "accountnumber" : "123456",
    "currentbalance" : "2000",
    "links": [ {
        "rel": "self",
        "href": "http://localhost:8080/account/12345/deposit"
    } ]
}
```

The disadvantage of including links to the allowed operations and related resources, increases the size of the payload.

Rule#9 – Return correct HTTP code

There are so many HTTP status codes available. Returning the correct HTTP code is very important to avoid any misbehavior of the business scenario. Below are five type of responses and their HTTP status code range.

1. Informational responses (100 â€“ 199)
2. Successful responses (200 â€“ 299)
3. Redirection messages (300 â€“ 399)
4. Client error responses (400 â€“ 499)
5. Server error responses (500 â€“ 599)

One example can be different status code for the same operation.

- First delete request of a resource returns **204** (**no content** was returned) or 200 (**OK**)
- Second delete request of a resource returns 404 which mean the resource **not found** as it was already deleted in the first request.
- Empty body of the GET request should return 204 **(no content)** else response with payload should return 200 (OK)

Rule#10 – Idempotency

Idempotency means making multiple identical requests for the same resource should result in same state as if making a single request. The **PUT** and **DELETE** methods are defined to be idempotent.

But however **DELETE** method has some caveat. Note that while idempotent operations produce the same result on the server (no side effects), the response itself may not be the same (e.g. a resource's state may change between requests). We have seen this in **Rule#9**.

Conclusion

default watermark

I tried to keep the part one as simple as possible mentioning very basic REST design best practices. Some interviewers prefer asking this question to see how knowledgeable the candidate is. Please share this post as much as possible if you found it insightful.
Happy Learning ðŸ˜Š

Category

1. Design

Date Created
December 30, 2023
Author
kk-ravi144gmail-com