REST API BEST PRACTICES

Representational State Transfer is referred to as REST. This indicates that the server sends back the resource's current state in a uniform format in response to a client request for a resource made using a REST API.

Benefits:

- Scalable
- Lightweight
- Independent

Best Practices:

- **REST API MUST ACCEPT AND RESPOND WITH JSON**

In order to convey data, JSON is an open and standardized format. Its ability to encode and decode JSON using the Fetch API or another HTTP client is a JavaScript-derived feature. Additionally, server-side technologies include libraries that can easily decode JSON.

- **USE ERROR STATUS CODE CODES**

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)
        }
    })
})
```

- **AVOID VERB USE IN URLS**

If we know 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

Use `POST: /articles/` Instead of `POST: /articles/createNewArticle/`

- **USE PLURAL NOUNS TO NAME A COLLECTION**

`GET /schools/50`

`POST /bikes`

- **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.

Example:

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

Copy

- **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.

Example:

`/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.

- **SECURE YOUR API**
  Automated script attacks on theAPI server are a favorite sport of hackers. As a result, in order to maintain seamless operations while securing sensitive data, the API must adhere to proactive security procedures. The HTTP Strict Transport Security (HSTS) policy is the first requirement for the API. Next, we must protect the network from attacks such as session hijacking, protocol downgrade assaults, and middleman attacks. Use all applicable security requirements to ensure that your API is secure.
  Perfectly built REST APIs take solutions based on the user experience while being on the good side of technical limitations. API implementation is crucial because it is a component of the business plan and a marketing tool for the company.
- **VERSIONING**

Versioning REST API is a good approach to take right from the start. Versioning is usually done with `/v1/`, `/v2/`, etc. added at the start of the API path.

- **Provide Documentation**

HTTP BEST PRACTICES

- GET, POST, PUT, PATCH, and DELETE are the commonest HTTP verbs. There are also others such as COPY, PURGE, LINK, UNLINK, and etc.

Use https://mysite.com/posts instead of https://mysite.com/getPosts or https://mysite.com/createPost

- common collections with plural nouns

instead of **https://mysite.com/post/123**, **use** **https://mysite.com/posts/123**.

- Use Nesting on Endpoints to Show Relationships

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.

- Use Filtering, Sorting, and Pagination to Retrieve the Data Requested
  Filtered endpoint:

https://mysite.com/posts?tags=javascript

- Provide accurate API documentation

JSON BEST PRACTICES

• The names of array fields Must be plural (e.g. "orders": [])

• Member names SHOULD contain the letters "a-z" (U+0061 to U+007A) at the beginning and end.

• The member's name MUST be in camel case (i.e., wordWordWord)

• Characters "a-z" (U+0061 to U+007A) should begin and conclude member names.

• The only ASCII alphanumeric letters allowed in member names are "a-z", "A-Z," and "0-9."

Boolean fields Must Not have null values, and empty arrays and objects May Not have null values (use [] or instead). Fields with null values SHOULD also be removed.

• The names of array fields Must be plural (e.g. "orders": [])

- identify the key-value pair within double quotes at all times.

```
{ "name": "Katherine Johnson" }
```

- **Avoid using hyphens in your key fields.** Use underscores (_), all lower case, or camel case.

```
{ "first_name":"Katherine", "last_name":"Johnson" } No comments in JSON objects.
```

- Choose meaningful property names.
- Array types should have plural property names. All other property names should be singular
- Avoid naming conflicts by choosing a new property name or versioning the API.

REQUEST RESPONSE BEST PRACTICE:

Response header

- Provide an appropriate HTTP response status code.
- Give the actual content type and, if applicable, file type.
- Describe any cache status.

- The response header should include the authentication token.
- Response headers can only contain string data.
- Describe any content length.
- Give a date and time for your response.
- adhere to the request-response mechanism previously outlined.

Response body

- Avoid sending response code, message, or status via the response body.
- For the JSON response body, useJSON best practices.
- String and Boolean can be used straight for a single result.
- Before writing the JSON body, provide correct JSON encode-decode.
- Continually discuss JSON as previously described.

Response cookies

- Avoid utilizing response cookies since they violate the stateless concept and can be sent by a Restful API just like a typical Web application serving HTML.
- Encryption, decryption, and other policies should be used when necessary for cookies.

Request body

- The request body ought to be organized using the JSON Array/Object pattern.
- Hold multipart/form data in the request body, such as photos, audio, and video, etc.
- There shouldn't be any authentication-related information in the request body.
- Setter getters can be used to connect the request body to a specified request data model.

Request Header

- All security-related information, such as token, auth, etc., should be carried in the request header.
- For header, only strings with Key:Pair values are permitted.
- Information on the client application's user agent should be included in the request header.
- CSRF/XSRF should be sent through header if necessary.
- When necessary, request headers should be linked to middleware controllers.

Middleware

Middleware is a software that works as a bridge between request and response to ensure API security.

Uses:

- Use to build token-based API authentication, user-agent restriction, CSRF, and XSRF security.
- Utilize to establish an API request rate cap.
- Incoming HTTP request logging.
- redirecting users in response to their inquiries.
- A request can be examined by middleware, and depending on its findings, either be decorated or rejected.
- The majority of the time, middleware is regarded as distinct from your application logic.
- You have enough freedom with middleware to design your own security system.

WEB SECURITY BEST PRACTICES

- **Carry Out A Full-Scale Security Audit**

It's vital to carry out a full-scale security audit of your web application and all its elements, including Webserver, Application server, Database server, Web application code

- **Ensure if Data Is Encrypted**

We need to make sure that the data is encrypted when it is stored and when it's moving between computers.

Whenever a user submits data via their browser, like filling out a form or logging in, it gets encrypted and then submitted over an encrypted connection before reaching the server.

- Implement Real-Time Security Monitoring
- Follow Proper **Logging** Practices

Log files are helpful to make changes to your application or its code. But following proper logging practices will ensure that we're not storing sensitive data in log files directly accessible by anyone with physical access to your computer.

- Continuously Check For Common Web Application Vulnerabilities
- Implement Security Hardening Measures
- Carry Out Regular Vulnerability Scans and Updates

50 INTERVIEW QUESTIONS BASED ON WEB DEVELOPMENT

1.How did you first get involved in computer science?

2.What has been your role in development projects in the past?

3.Can you identify limitations within the development languages you prefer?

4.Where do you see yourself professionally in five years?

5.Which technical skills do backend developers need to have?

6.Which soft skills do backend developers need to be successful?

7.Name the main backend development responsibilities you had in your previous role.

8.Which backend developer skills do you lack? How are you trying to improve?

9.Which is your favorite programming language?

10.Which programming language is your least favorite?

11.How are web services beneficial?

12.Which method do you use to remain up-to-date with the latest trends in backend development?

13.Describe your greatest coding strength.

14.Explain how your coding career began.

15.How do you receive and make use of negative feedback as a backend developer?

16.How do you share negative feedback with your co-workers?

17.Explain the difference between software design and architecture.

18.How do you handle errors when making API calls?

19 What is a typical workflow for implementing a new feature on the backend?

20 Explain the essence of DRY and DIE principles?

21. What is a web server?

22.What is the difference between a GET and a POST request?

23.What is an example of when you would use caching?

24. How would you select a cache strategy (e.g., LRU, FIFO)?

25. What are some common issues with ORMs?

26. When should you use asynchronous programming?

27.What is the difference between promises and callbacks?

28. What is a closure?

29. What is the difference between a Class and an Interface in Java?

30. What is continuous integration?

31. What is a software development kit (SDK)?

33. What are high-order functions? Why are they useful?

34. What is a microservice?

35. How would you design an API?

36. What is the difference between a RESTful and a SOAP API?

37. How do you handle errors when making API calls?

38. What is a database?

39. How would you handle optimizing an existing database?

40. What is the difference between a relational and a non-relational database?

41. How would you query data from a MongoDB database?

42. What are some benefits of using a NoSQL database?

43. How would you normalize data in a relational database?

Scalability Questions

44. How would you design a software system for scalability?

45. What are some common scalability issues? How can they be addressed?

46. Scale-out vs. scale-up: how are they different? When to apply one, when the other?

47. What are some common security risks when building a web application?

48. How would you implement authentication and authorization on a new project?

49. What is the difference between a cookie and a session?

50. What are some performance testing steps?