Assignment on :

Make A note on Web Back-End Development

Submitted by
MD Shirulla Khan.

1. Rest API Best Practices

Web API is a web development architecture that decouples the client GUI from the database and the server's logic. This architecture enables to serve of the same interface to multiple clients running on various platforms. Also, the same application can communicate with numerous interfaces.
REST stands for Representational state transfer and it's a stateless protocol over HTTP that provides interactions with the resources stored in the database which contains four basic CRUD actions - Create, Read, Update and Delete.

- Prioritize Nouns over Verbs
- Prefer using Plural naming conventions
- Utilize Resource Nesting Efficiently
- Systematic Documentation
- Data Filtering options
- Utilize SSL/TLS security layers
- Focus on Error Handling
- Always choose JSON

API End Naming Best Practice:

- Use Nouns to Name URIs.
- Use Clear, Unabridged Names That Are Intuitive.
- Use Forward Slashes to Denote URI Hierarchy.
- Separate Words with Hyphens.
- Use Lowercase Letters.
- Avoid Special Characters.
- Avoid File Extensions
- Be Consistent with Naming REST API Endpoints.

- **Use Nouns to Name URIs.**

All REST APIs have a URL at which they can be accessed, e.g. https://api.example.com. Subdirectories of this URL denote different API resources, which are accessed using a Uniform Resource Identifier (URI). This is an easy way for developers to ensure the name makes sense and can be understood by anyone. For example, the URI https://api.example.com/users will return a list containing the users of a particular service. The web APIs make it easier to understand which web services are in use.

In general, URIs should be named with nouns that specify the contents of the resource, rather than adding a verb for the function being performed. For example, you should use https://api.example.com/users instead of https://api.example.com/getUsers. This is because CRUD (create, read, update, delete) functionality should already be specified in the HTTP request (e.g. HTTP GET https://api.example.com/users). There is no need to repeat the information, which keeps the URI easy to read while showing that it is the Get Method needed. A content-type header can be a good way to name the URI. In rare cases, you can use HTTP verbs, but it's best to stick to nouns for the rest endpoint name.

Using nouns for naming URIs is a REST API naming best practice, but you may wonder if plural or singular nouns are best. When should you use singular or plural nouns? In general, you should name your URIs with plural nouns. The exception to this is when you have a concept that is obviously

singular, which rarely happens. (e.g.https://api.example.com/users/admin for the administrative user).

- **Use Clear, Unabridged Names That Are Intuitive.**

When naming REST API endpoints, you should use URI names that are intuitive and easy to understand. Consider the person checking the URI when they've never used your API previously. They should be able to easily guess what words are used and how they are structured. You should definitely avoid abbreviations and shorthand (e.g. https://api.example.com/users/123/fn instead of https://api.example.com/users/123/first-name). In some cases, the accepted or popular term for something is the abbreviation, which means you can use it. (e.g. https://api.example.com/users/ids instead of https://api.example.com/users/identification-numbers). Remember to keep it simple enough that anyone new to your API can simply guess the URI. If everyone uses the same methods to name their REST API endpoints, it makes it easier to work with them.

- **Use Forward Slashes to Denote URI Hierarchy**

REST APIs are typically structured in a hierarchy. For example, https://api.example.com/users/123/first-name will retrieve the user's first name with ID number 123. The forward-slash ("/") character should be used to navigate this hierarchy and to indicate where you are. It moves from general to specific when going from left to right in the URI.

While forward slashes are suitable for denoting the hierarchy of your API, they're not necessary at the very end of the URL. Adding this extraneous slash increases complexity without adding clarity. For example, you should use https://api.example.com/users instead of https://api.example.com/users/.

- **Separate Words with Hyphens.**

When a REST API endpoint contains multiple words (e.g. https://api.example.com/users/123/first-name), you should separate the words with hyphens. It's a good way to make the URI easier to read and is a universal method that everyone can understand.

It's generally accepted that a hyphen is clearer and more user-friendly than using underscores (e.g. first_name) or camel case (e.g. firstName), which is discouraged due to its use of capital letters (see below). The hyphen is easy to type, and with all developers on the same page, it can streamline the URIs to ensure everyone has access.

- **Use Lowercase Letters.**

Whenever possible, use lowercase letters in your API URLs. This is mainly because the RFC 3986 specification for URI standards denotes that URIs are case-sensitive (except for the scheme and host components of the URL). Lowercase letters for URIs are in widespread use, and also help avoid confusion about inconsistent capitalization. If you add capital letters, you should be aware that this will cause confusion and result in user error more often than not.

- **Avoid Special Characters.**

Special characters are not only unnecessary, they may confuse users who are familiar with API design and naming. They aren't available to everyone easily and are technically complex. Because URLs can only be sent and received using the ASCII character set, all of your API URLs should contain only ASCII characters.

In addition, try to avoid the use of “unsafe” ASCII characters, which are typically encoded in order to prevent confusion and security issues (e.g. “%20” for the space character). “Unsafe” ASCII characters for URLs include the space character (“ “), as well as brackets (“[]”), angle brackets (“<>”), braces (“{}”), and pipes (“|”). Keep your names as simple as possible and you shouldn’t have any problems. In most cases, these are the same as HTTP methods.

- **Avoid File Extensions**

While the result of an API call may be a particular filetype, file extensions such as .HTML are largely seen as unnecessary in URIs, as they add length and complexity. For example, you should use https://api.example.com/users instead of https://api.example.com/users.xml. In fact, using a file extension can create issues for end users if you change the filetype of the results later on. You don’t need to use node.js or similar, for example. It can be simplified.

File extensions in URIs will often create confusion and also make it harder to intuit the URI if it is unknown. File extensions do not need to be included and there are other ways to indicate the file type. These should not be included in the REST API design and names.

If you want to specify the file type of the results, you can use the Content-Type entity-header instead. This lets the user know which media type was used for the original resource. It is not always necessary, but if you wish to maintain records of the original file type, you can use this method.

- **Be Consistent with Naming REST API Endpoints**

Choose a system for naming your API endpoints and stick with it. You should document your methods so everyone working with you knows the naming protocols. When you are consistent in your names, this ensures a uniform system across the board. Everyone working with the APIs will find

it easy to use them. If they're unsure of a specific URI, they can assume what it will be, based on the naming protocols.

2. HTTP methods Best Practices:

REST API development is very popular today, fulfilling rapid growing of cloud services and apps. You know, one of REST architectural constraints is Uniform Interface - stating that developers should use common, well-known HTTP methods and status codes in their APIs, in a way that ensures conformity across the web.

So what is the best practice widely used by the industry? In this article, I'd like to share with you guys how to use the right HTTP methods and status codes in your REST APIs.

1. How to use the Right HTTP Methods for REST APIs

Basically, for CRUD operations (Create, Retrieve, Update and Delete) you can use the HTTP methods as follows:
- POST: create resource or search operation
- GET: read resource operation
- PUT: update resource operation
- DELETE: remove resource operation
- PATCH: partial update resource operation

Let's go into detail of each HTTP method.

API end point with POST method:

Used mainly for Create resource operations, or Read operations in some certain cases:

- Read resource if URL / query string exceeds maximum allowed characters. Maximum length of URL and query string is 2,048 characters.
- Read resource if query parameters contain sensitive information

Return status code:

- 201 Created for successful create operation
- 200 OKfor successful read operation if the response contains data
- 204 No Content for successful read operation if the response contains NO data

API end point with GET method:

Used primary for Read resource operations.
Return status code:

- 200 OK if the response contains data
- 204 No Content if the response contains no data

API end point with PUT method:
Used for Update resource operations.
Return status code:

- 200 OK if the response contains data
- 204 No Content if the response contains no data

API end point with DELETE method:
Used for Delete resource operations.
Return status code: 204 No Content for successful delete operation.

API end point with PATCH method:
Used for Partial update resource operations.
Return status code: 200 OKfor successful partial update operation.

2. How to use the Right Status Codes for REST APIs

Returning the right HTTP status codes in REST APIs is also important, to ensure uniform interface architectural constraint. Besides the status codes above, below is the guideline for common HTTP status codes:

- 200 OK: for successful requests
- 201 Created: for successful creation requests
- 202 Accepted: the server accepts the request, but the response cannot be sent immediately (e.g. in batch processing)
- 204 No Content: for successful operations that contain no data
- 304 Not Modified: used for caching, indicating the resource is not modified
- 400 Bad Request: for failed operation when input parameters are incorrect or missing, or the request itself is incomplete
- 401 Unauthorized: for failed operation due to unauthenticated requests
- 403 Forbidden: for failed operation when the client is not authorized to perform
- 404 Not Found: for failed operation when the resource doesn't exist
- 405 Method Not Allowed: for failed operation when the HTTP method is not allowed for the requested resource
- 406 Not Acceptable: for failed operation when the Accept header doesn't match. Also can be used to refuse request
- 409 Conflict: for failed operation when an attempt is made for a duplicate create operation
- 429 Too Many Requests: for failed operation when a user sends too many requests in a given amount of time (rate limiting)
- 500 Internal Server Error: for failed operation due to server error (generic)
- 502 Bad Gateway: for failed operation when the upstream server calls fail (e.g. call to a third-party service fails)
- 503 Service Unavailable: for a failed operation when something unexpected happened at the server (e.g. overload of service fails)

3. JSON Best Practices

JavaScript Object Notation (JSON)

- JSON is a lightweight data-interchange format that is completely language-independent.
- It was derived from JavaScript, but many modern programming languages include code to generate and parse JSON-format data
- The official Internet media type for JSON is application/json.
- It was designed for human-readable data interchange.
- The filename extension is .json.

Uses of JSON

- It is used while writing JavaScript based applications that includes browser extensions and websites.

- JSON format is used for serializing and transmitting structured data over network connection.

- It is primarily used to transmit data between a server and web applications.

- Web services and APIs use JSON format to provide public data.

Characteristics of JSON

- JSON is easy to read and write.
- It is a lightweight text-based interchange format.
- JSON is language-independent.

Enclose within DOUBLE Quotes

Always enclose the Key : Value pair within double quotes. It may be convenient (not sure how) to generate with Single quotes, but JSON parser don't like to parse JSON objects with single quotes.

For numerical Values, quotes are optional but is a good practice to enclose them in double quote.

```
{'id': '1','name':File} is not right ✗
{"id": 1,"name":"File"} is okay  ✓
{"id": "1","name":"File"} is the best  ✓
```

No Hyphens please

Never Never Never use Hyphens in your Key fields. It breaks python, scala parser and developers have to escape it to use those fields.

Instead of Hyphens use underscores (_). But using alllower case or camel Case is the best. See samples below.

```
{"first-name":"Rachel","last-name":"Green"}  is not right. ✗
{"first_name":"Rachel","last_name":"Green"} is okay ✓
{"firstname":"Rachel","lastname":"Green"} is okay ✓
{"firstName":"Rachel","lastName":"Green"} is the best. ✓
```

Always create a Root element.

Creation of Root element is optional, but it helps when you are generating complicated JSON.

```
JSON with root element

{
"menu": [
    {
        "id": "1",
        "name":"File",
        "value": "F",
        "popup": {
            "menuitem": [
                {"name":"New", "value": "1N", "onclick": "newDoc()"},
                {"name":"Open", "value": "1O", "onclick":
"openDoc()"},
                {"name":"Close", "value": "1C", "onclick":
"closeDoc()"}
                ]
            }
    },
    {
        "id": "2",
        "name":"Edit",
        "value": "E",
        "popup": {
            "menuitem": [
                {"name":"Undo", "value": "2U", "onclick": "undo()"},
                {"name":"Copy", "value": "2C", "onclick": "copy()"},
                {"name":"Cut", "value": "2T", "onclick": "cut()"}
                ]
            }
    }
    ]
```

JSON without a root element

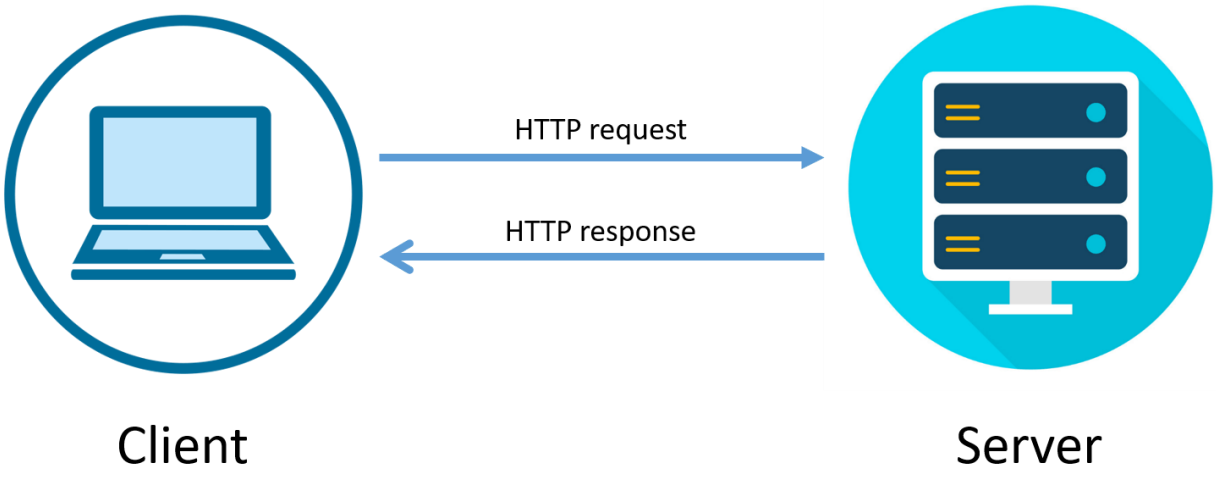
```
[
    {
        "id": "1",
        "name":"File",
        "value": "F",
        "popup": {
            "menuitem": [
                {"name":"New", "value": "1N", "onclick": "newDoc()"},
                {"name":"Open", "value": "1O", "onclick":
"openDoc()"},
                {"name":"Close", "value": "1C", "onclick":
"closeDoc()"}
                ]
            }
    },
    {
        "id": "2",
        "name":"Edit",
        "value": "E",
        "popup": {
            "menuitem": [
                {"name":"Undo", "value": "2U", "onclick": "undo()"},
                {"name":"Copy", "value": "2C", "onclick": "copy()"},
                {"name":"Cut", "value": "2T", "onclick": "cut()"}
                ]
            }


    }
]
```

4. Request - Response Best Practices

HTTP/HTTPS Request Response Communication

- In request/response communication mode.
- One software module sends a request to a second software module and waits for a response.
- The First software module performs the role of the client.
- The second, the role of the server,
- This is called client/server interaction.



HTTP request methods

HTTP defines a set of request methods to indicate the desired action to be performed for a given resource. Although they can also be nouns, these request methods are sometimes referred to as HTTP verbs. Each of them implements a different semantic, but some common features are shared by a group of them: e.g. a request method can be safe, idempotent, or cacheable.

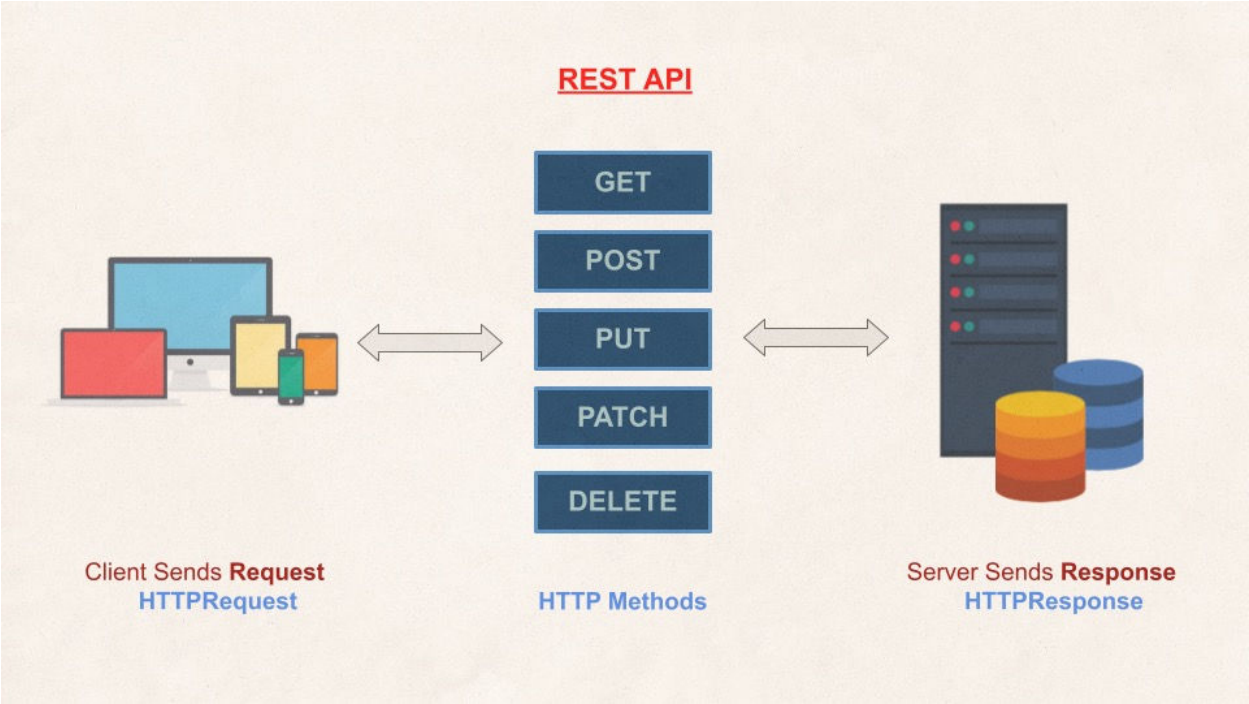
GET
The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.

HEAD
The HEAD method asks for a response identical to a GET request, but without the response body.

POST
The POST method submits an entity to the specified resource, often causing a change in state or side effects on the server.

PUT
The PUT method replaces all current representations of the target resource with the request payload.

DELETE
The DELETE method deletes the specified resource.

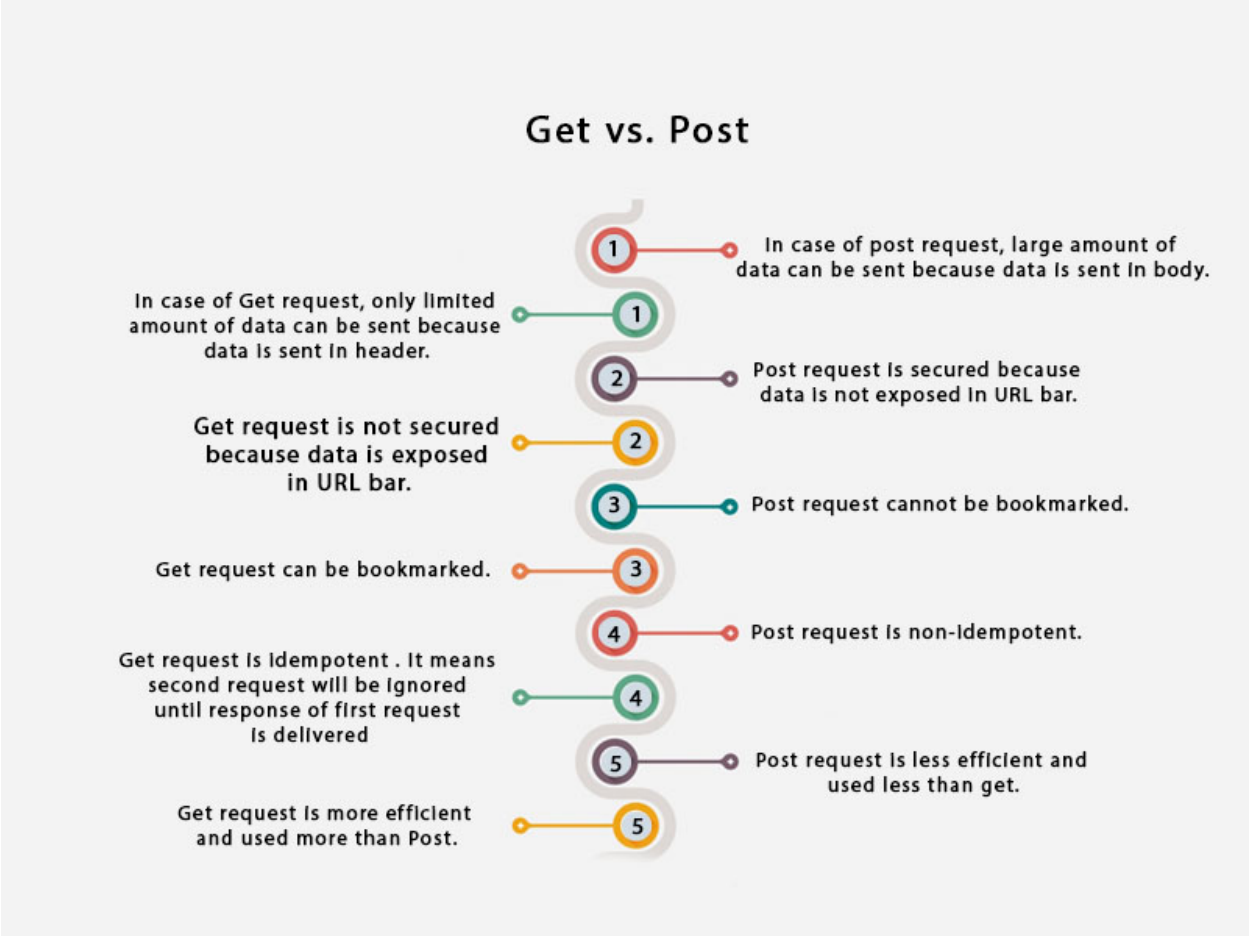


HTTP Request Throttling:

Throttle Request refers to a process in which a user is allowed to hit the application maximum time in per second or per minute. Throttling is also known as request rate limiting.

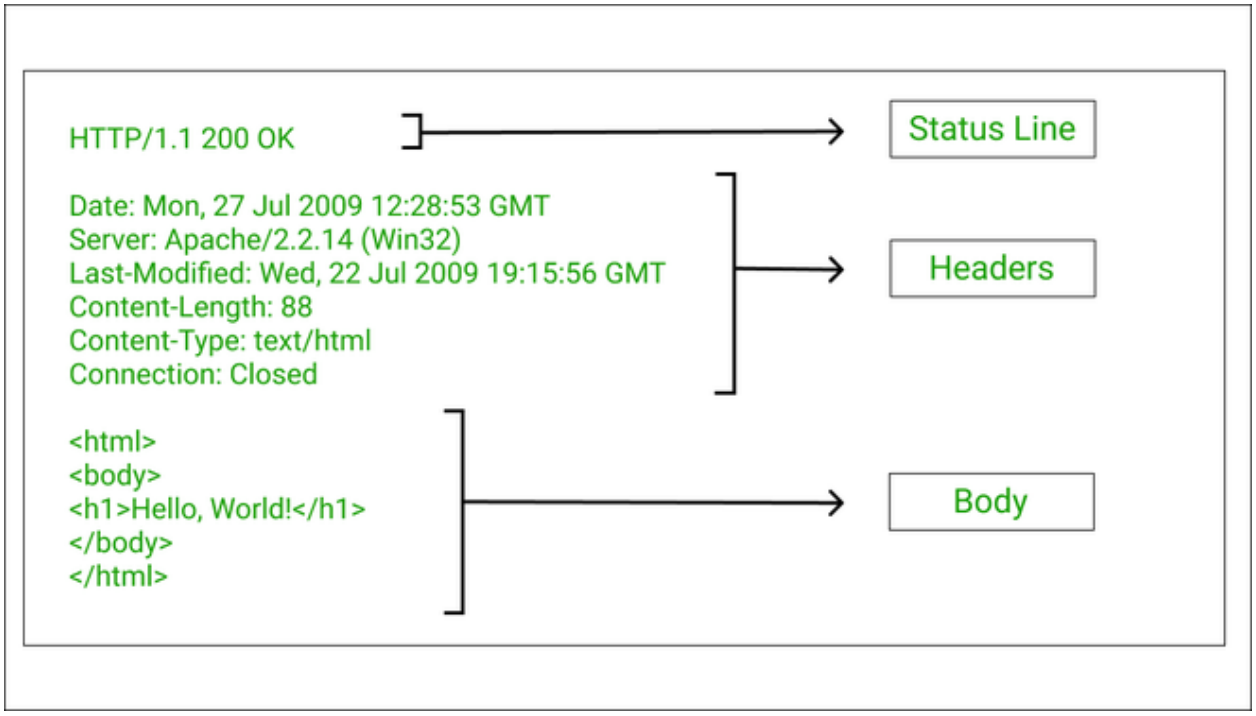
- Essential component of Internet security, as DoS attacks can tank a server with unlimited requests.
- Rate limiting also helps make your API scalable by avoid unexpected spikes in traffic, causing severe lag time.

Request Compare GET vs. POST:



HTTP Response:

An HTTP response is made by a server to a client. The aim of the response is to provide the client with the resource it requested, or inform the client that the action it requested has been carried out; or else to inform the client that an error occurred in processing its request.



An HTTP response is made by a server to a client. The aim of the response is to provide the client with the resource it requested, or inform the client that the action it requested has been carried out; or else to inform the client that an error occurred in processing its request.

An HTTP response contains:

1. A status line.
2. A series of HTTP headers, or header fields.
3. A message body, which is usually needed.

As in a request message, each HTTP header is followed by a carriage return line feed (CRLF). After the last of the HTTP headers, an additional CRLF is used (to give an empty line), and then the message body begins.

The status line is the first line in the response message. It consists of three items:

1. The HTTP version number, showing the HTTP specification to which the server has tried to make the message comply.
2. A status code, which is a three-digit number indicating the result of the request.
3. A reason phrase, also known as status text, which is human-readable text that summarizes the meaning of the status code.

An example of a response line is:
HTTP/1.1 200 OK

In this example:

- the HTTP version is HTTP/1.1
- the status code is 200
- the reason phrase is OK

HTTP headers

The HTTP headers for a server's response contain information that a client can use to find out more about the response, and about the server that sent it. This information can assist the client with displaying the response to a user, with storing (or caching) the response for future use, and with making further requests to the server now or in the future. For example, the following series of headers tell the client when the response was sent, that it was sent by CICS®, and that it is a JPEG image:

Date: Thu, 09 Dec 2004 12:07:48 GMT
Server: IBM_CICS_Transaction_Server/3.1.0(zOS)
Content-type: image/jpg

In the case of an unsuccessful request, headers can be used to tell the client what it must do to complete its request successfully.
An empty line (that is, a CRLF alone) is placed in the response message after the series of HTTP headers, to divide the headers from the message body.

Message body

The message body of a response may be referred to for convenience as a response body.

Message bodies are used for most responses. The exceptions are where a server is responding to a client request that used the HEAD method (which asks for the headers but not the body of the response), and where a server is using certain status codes.

For a response to a successful request, the message body contains either the resource requested by the client, or some information about the status of the action requested by the client. For a response to an unsuccessful request, the message body might provide further information about the reasons for the error, or about some action the client needs to take to complete the request successfully.

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 date 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 String, Boolean directly.
- Provide proper JSON encode-decode before writing JSON Body.
- Follow discussion on JSON described before.
-

Response Cookies:

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

5. Web Security Practices

REST API Security isn't an afterthought. It has to be an integral part of any development project and also for REST APIs.

There are multiple ways to secure a RESTful API e.g. basic auth, OAuth, etc. but one thing is sure that RESTful APIs should be stateless – so request authentication/authorization should not depend on sessions.

Instead, each API request should come with some sort of authentication credentials that must be validated on the server for every request.

Keep it Simple

Secure an API/System – just how secure it needs to be. Every time you make the solution more complex "unnecessarily," you are also likely to leave a hole.

Always Use HTTPS

By always using SSL, the authentication credentials can be simplified to a randomly generated access token. The token is delivered in the username field of HTTP Basic Auth. It's relatively simple to use, and you get a lot of security features for free.

If you use HTTP 2, to improve performance – you can even send multiple requests over a single connection, that way you avoid the complete TCP and SSL handshake overhead on later requests.

Use Password Hash

Passwords must always be hashed to protect the system (or minimize the damage) even if it is compromised in some hacking attempts. There are many such hashing algorithms that can prove really effective for password security e.g. PBKDF2, bcrypt, and scrypt algorithms.

Never expose information on URLs

Usernames, passwords, session tokens, and API keys should not appear in the URL, as this can be captured in web server logs, which makes them easily exploitable.

https://api.domain.com/user-management/users/{id}/someAction?apiKey=abcd123456789 //Very BAD !!

The above URL exposes the API key. So, never use this form of security.

Consider OAuth

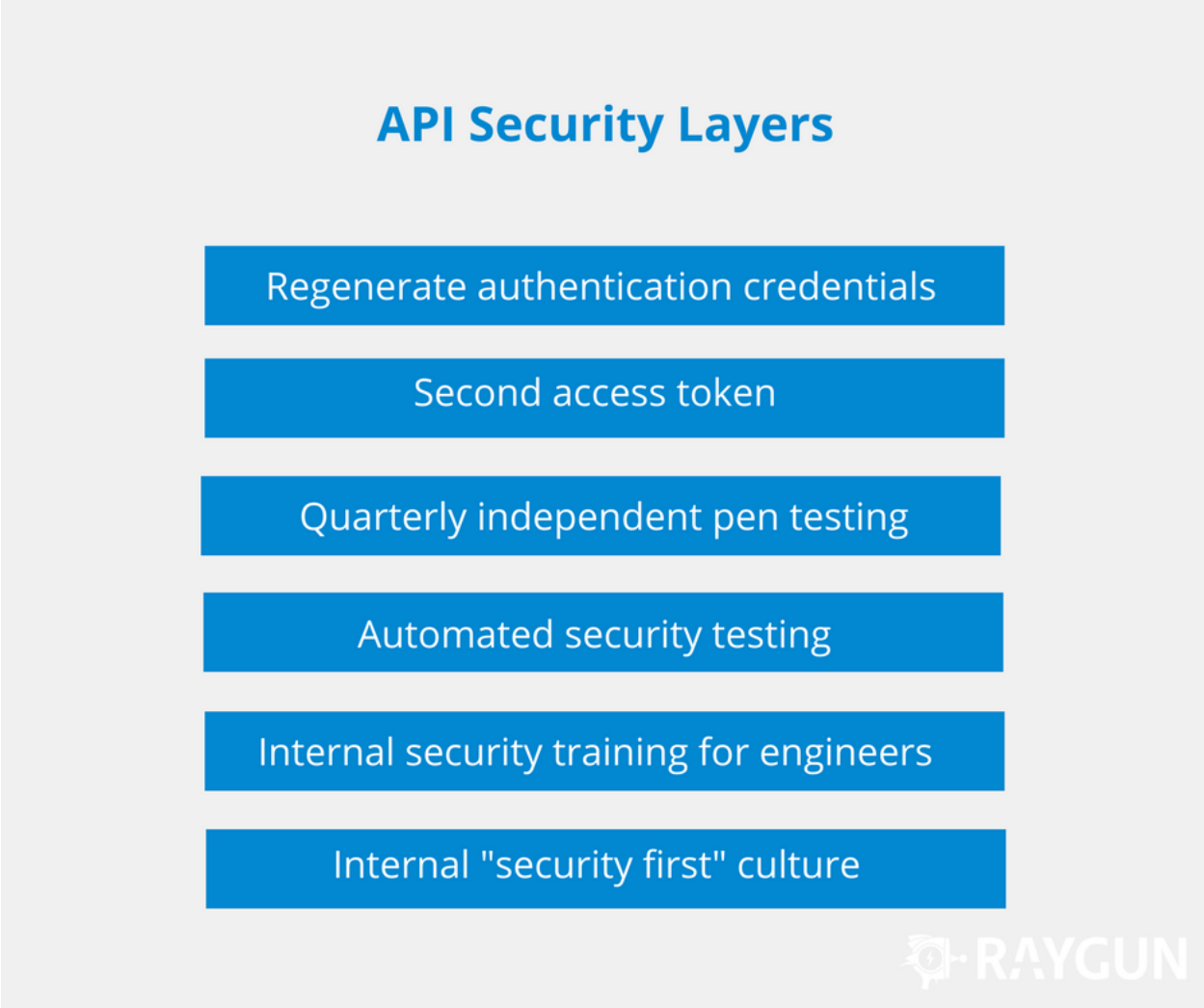
Though basic auth is good enough for most of the APIs and if implemented correctly, it's secure as well – yet you may want to consider OAuth as well.
The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its behalf.

Consider Adding Timestamp in Request

Along with other request parameters, you may add a request timestamp as an HTTP custom header in API requests.
The server will compare the current timestamp to the request timestamp and only accepts the request if it is after a reasonable timeframe (30 seconds, perhaps).
This will prevent very basic replay attacks from people who are trying to brute force your system without changing this timestamp.

Input Parameter Validation

Validate request parameters on the very first step, before it reaches application logic. Put strong validation checks and reject the request immediately if validation fails.
In API response, send relevant error messages and examples of correct input format to improve user experience.



JSON Web Token

A JSON web token(JWT) is JSON Object which is used to securely transfer information over the web(between two parties). It can be used for an authentication system and can also be used for information exchange. The token is mainly composed of header, payload, signature. These three parts are separated by dots(.).

Is JWT Secure?

A JSON web signature (JWS) does not inherently provide security. A service's security or vulnerability emerges from the implementation. The most common vulnerabilities created by a JWT/JWS implementation are easily avoided but not always obvious. Some of the more common vulnerabilities arise from implementations allowing the JWT header values to drive the validation of the JWT

Best Practices When Implementing JWTs

Don't Allow the JWT Header to Drive the Authentication of the Token

Avoid Weak Token Secrets

Prevent Token Sidejacking

Use Secure Token Storage

Set Reasonable Expiration Times

Limit the Scope of Authorization

Require Claims That Limit the Scope of Access a Token Provides

Do Not Store Privileged Data in the JWT

6. Collect 50 Interview Question on Web Back-End Development

- **Explain the purpose of the backend?**
- **What is a typical workflow for implementing a new feature on the backend?**
- **Explain the essence of DRY and DIE principles?**
- **What is a web server?**
- **What is the difference between a GET and a POST request?**
- **When should you use asynchronous programming?**
- **What is the difference between promises and callbacks?**
- **What is a closure?**
- **What is a software development kit (SDK)?**
- **What are high-order functions? Why are they useful?**
- **What is a microservice?**
- **How would you design an API?**
- **How do you handle errors when making API calls?**

- **What are pure components in MERN Stack?**
-
- **What are the key features of Node.js?**
-
- **How are child threads handled in Node.js?**
-
- **What is the purpose of ExpressJS?**
-
- **What is the purpose of MongoDB?**

-
- **What is the function of Node.js?**
-
- **Name the IDEs that are commonly used for Node.JS development?**
- **Define DATA modeling?**
-
- **What is REPL In Node.Js?**
-
- **Define Scope in JavaScript.**
-
- **Define Containerization.**
-
- **Define a Test Pyramid. How can you actualize a Test Pyramid when discussing HTTP APIs?**
-
- **What purpose do Indexes serve in MongoDB?**
-
- **What is meant by “Callback” in Node.js?**
-
- **Define Cross-site Scripting (XSS).**

- **What Is CAP Theorem?**

- **What REST stands for?**

- **What are NoSQL databases? What are the different types of NoSQL databases?**
- **What do you understand by NoSQL databases? Explain.**

- **What is SQL injection?**

- **What is meant by *Continuous Integration***

- **How to mitigate the SQL Injection risks?**

- **Name some performance testing steps**

- **Name the difference between *Acceptance Test* and *Functional Test***

- **What are the advantages of Web Services?**
- **What is meant by replication in MongoDB?**

- **What is Mongoose?**

- **What do you know about Asynchronous API?**
-
- **What is CallBack Hell?**
-

- **How does node prevent blocking code?**
-
- **What is Key?**
-
- **How can we make node modules available externally?**
-
- **List the abbreviation of MERN**
-
- **What are the features of NodeJS?**
-
- **Explain occasion circle in NodeJS**
-
- **Which are the 2 arguments that async.queue accepts as input in Node.js?**
-
- **Explain stub in Node.js**
-
- **Explain concept of a thread pool**
- **What do you know about smart components and dumb components?**
- **What do you know about Dependency Injection?**
- **What are some common security risks when building a web application?**
- **How would you implement authentication and authorization on a new project?**
- **What is the difference between a cookie and a session?**
- **What is a database?**