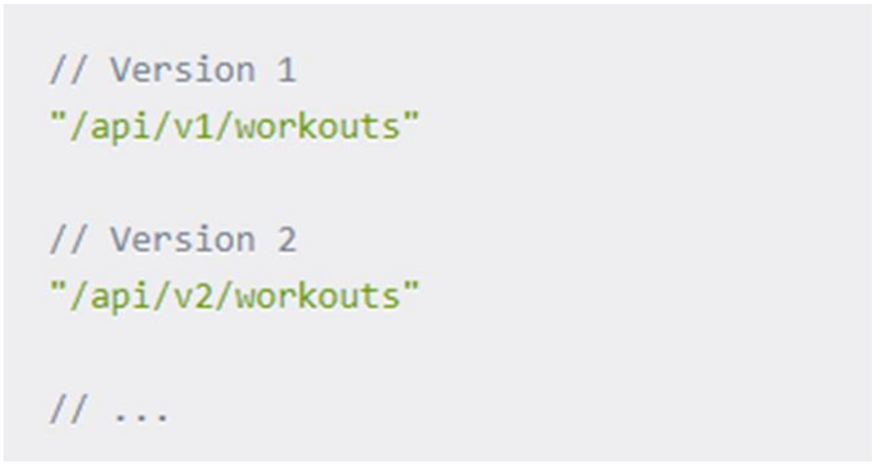
1. REST API BEST PRACTICES

- ❖ VERSIONING
  - ➢ Before we write any API-specific code we should be aware of versioning. Like in other applications there will be improvements, new features, and stuff like that. So it's important to version our API as well.



```
// Version 1
"/api/v1/workouts"

// Version 2
"/api/v2/workouts"

// ...
```

- ❖ API END NAMING BEST PRACTICE
  - ✓ NAME RESOURCES IN PLURAL
  - ✓ AVOID VERBS IN ENDPOINT NAMES
  - ✓ USE NOUNS TO NAME URIS
  - ✓ USE LOWERCASE LETTERS
- ❖ ACCEPT AND RESPOND WITH DATA IN JSON FORMAT
- ❖ RESPOND WITH STANDARD HTTP ERROR CODES
- ❖ REQUEST RESPONSE MODEL
- ❖ GOOD SECURITY PRACTICES

2. HTTP METHODS BEST PRACTICES

❖ **What is HTTP?**

The Hypertext Transfer Protocol (HTTP) is designed to enable communications between clients and servers. HTTP works as a request-response protocol between a client and server.

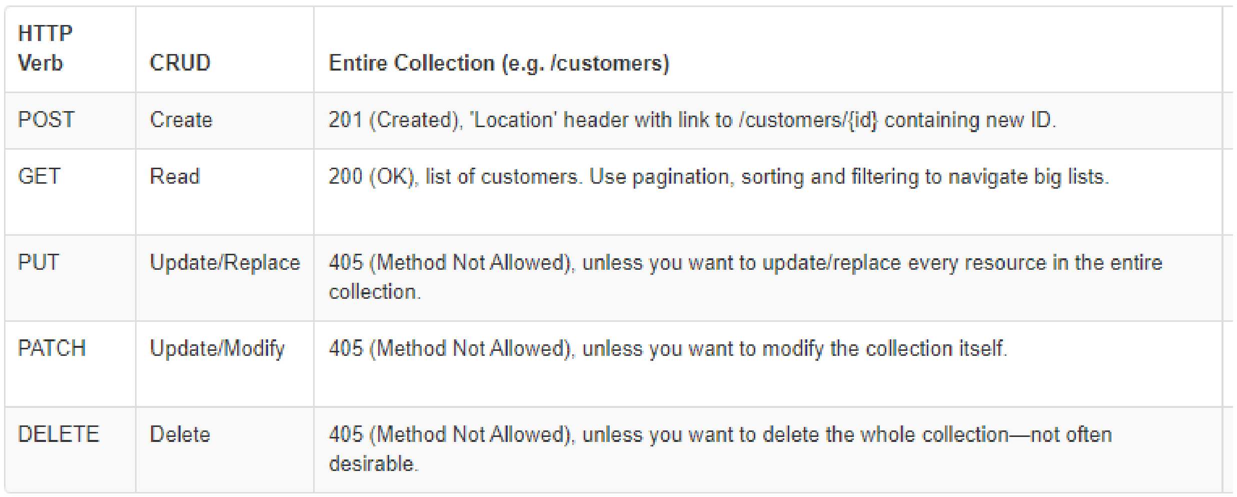
Example: A client (browser) sends an HTTP request to the server; then the server returns a response to the client. The response contains status information about the request and may also contain the requested content.

❖ **HTTP Methods**

- GET
- POST
- PUT
- HEAD
- DELETE
- PATCH
- OPTIONS
- CONNECT
- TRACE

✓ **The primary or most-commonly-used HTTP verbs (or methods, as they are properly called) are POST, GET, PUT, PATCH, and DELETE. These correspond to create, read, update, and delete (or CRUD) operations, respectively. There are a number of other verbs, too, but are utilized less frequently. Of those less-frequent methods, OPTIONS and HEAD are used more often than others.**

✓ **The two most common HTTP methods are: GET and POST.**



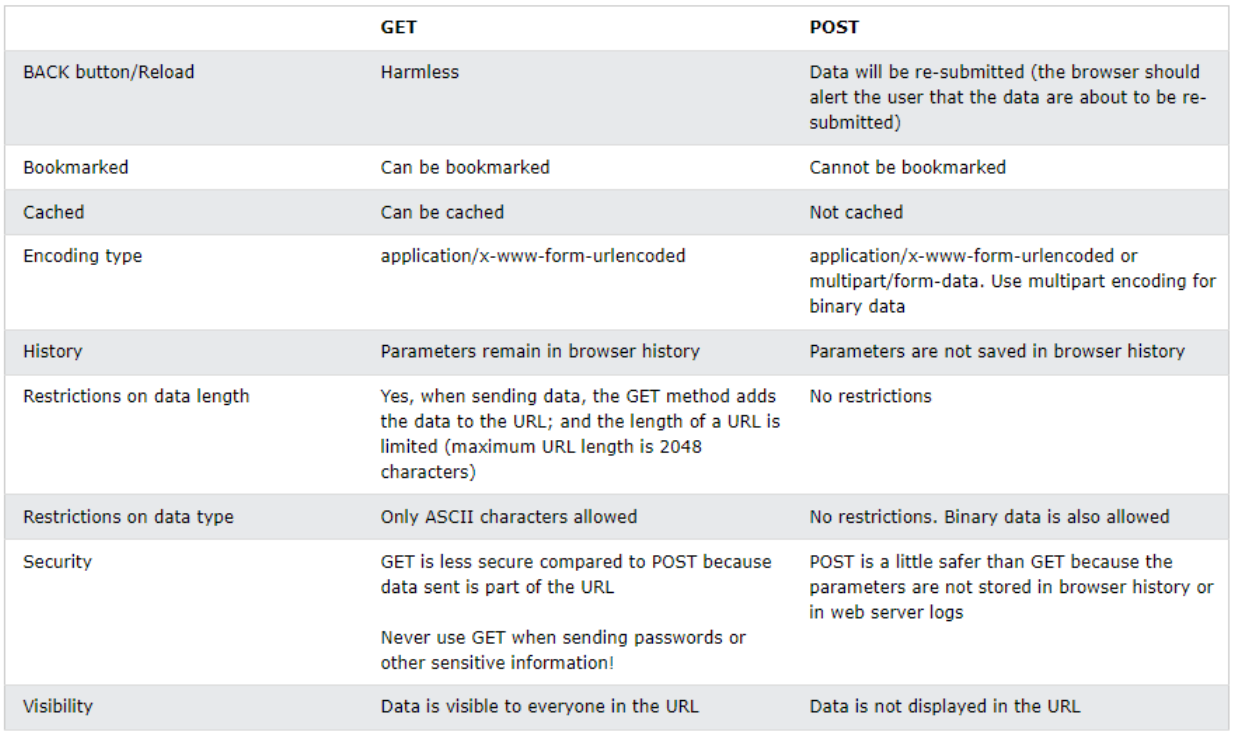
HTTP Verb	CRUD	Entire Collection (e.g. /customers)
POST	Create	201 (Created), 'Location' header with link to /customers/{id} containing new ID.
GET	Read	200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists.
PUT	Update/Replace	405 (Method Not Allowed), unless you want to update/replace every resource in the entire collection.
PATCH	Update/Modify	405 (Method Not Allowed), unless you want to modify the collection itself.
DELETE	Delete	405 (Method Not Allowed), unless you want to delete the whole collection—not often desirable.

- **THE GET METHOD:** GET is used to request data from a specified resource.

- **SOME NOTES ON GET REQUESTS:**
  - ✓ GET requests can be cached
  - ✓ GET requests remain in the browser history
  - ✓ GET requests can be bookmarked
  - ✓ GET requests should never be used when dealing with sensitive data
  - ✓ GET requests have length restrictions
  - ✓ GET requests are only used to request data (not modify)

- **THE POST METHOD:** POST is used to send data to a server to create/update a resource.

- **SOME NOTES ON POST REQUESTS:**
  - ✓ POST requests are never cached
  - ✓ POST requests do not remain in the browser history
  - ✓ POST requests cannot be bookmarked
  - ✓ POST requests have no restrictions on data length



	GET	**POST**
BACK button/Reload	Harmless	Data will be re-submitted (the browser should alert the user that the data are about to be re-submitted)
Bookmarked	Can be bookmarked	Cannot be bookmarked
Cached	Can be cached	Not cached
Encoding type	application/x-www-form-urlencoded	application/x-www-form-urlencoded or multipart/form-data. Use multipart encoding for binary data
History	Parameters remain in browser history	Parameters are not saved in browser history
Restrictions on data length	Yes, when sending data, the GET method adds the data to the URL; and the length of a URL is limited (maximum URL length is 2048 characters)	No restrictions
Restrictions on data type	Only ASCII characters allowed	No restrictions. Binary data is also allowed
Security	GET is less secure compared to POST because data sent is part of the URL Never use GET when sending passwords or other sensitive information!	POST is a little safer than GET because the parameters are not stored in browser history or in web server logs
Visibility	Data is visible to everyone in the URL	Data is not displayed in the URL

3. JSON Best PRACTICES

❖ WHAT IS JSON?

➢ JSON is a data interchange format that is easy to parse and generate. JSON is an extension of the syntax used to describe object data in JavaScript. Yet, it’s not restricted to use with JavaScript. It has a text format that uses object and array structures for the portable representation of data. All modern programming languages support these data structures, making JSON completely language independent.

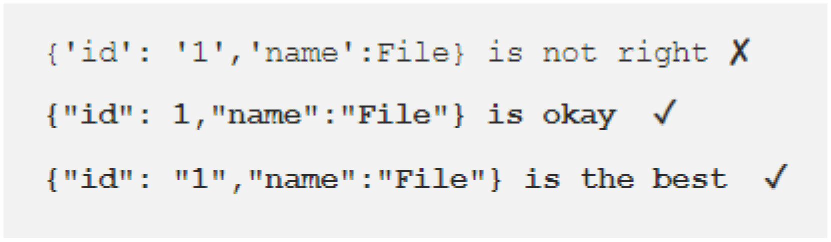
JSON is no doubt a flexible way to share data across systems. But that doesn’t mean JSON can be created any way.

Here are some best practices which will help consumers to use consume output.

- **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.

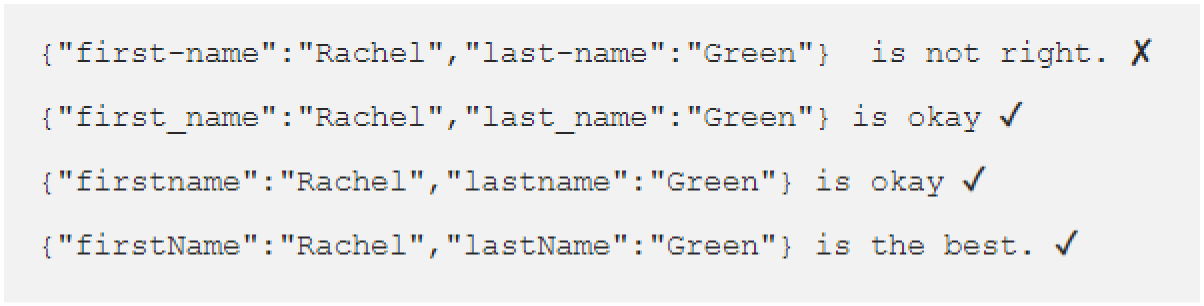


```
{'id': '1','name':File} is not right ✗
{"id": 1,"name":"File"} is okay ✓
{"id": "1","name":"File"} is the best ✓
```

- **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.

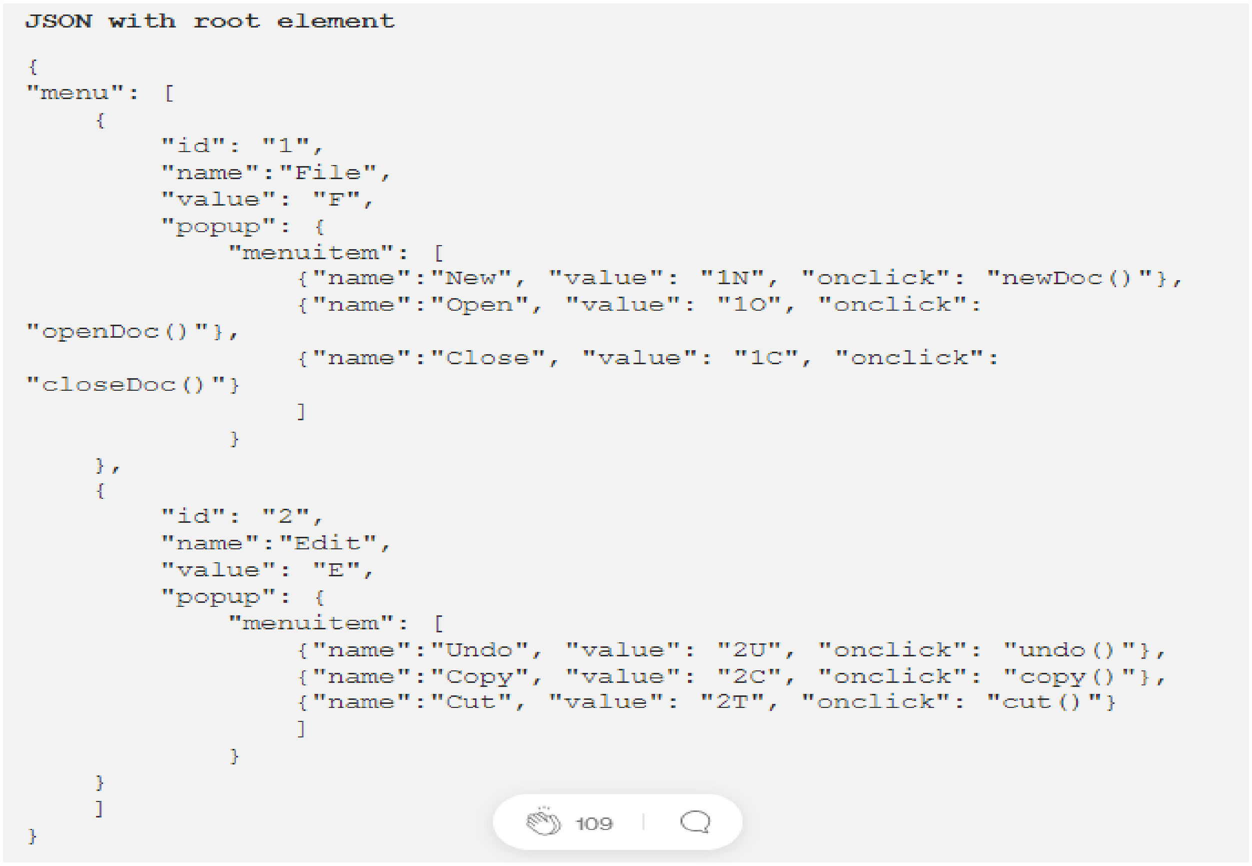
- ✓ Instead of Hyphens use underscores (_). But using all lower case or camel Case is the best.



```
{"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. ✓
```

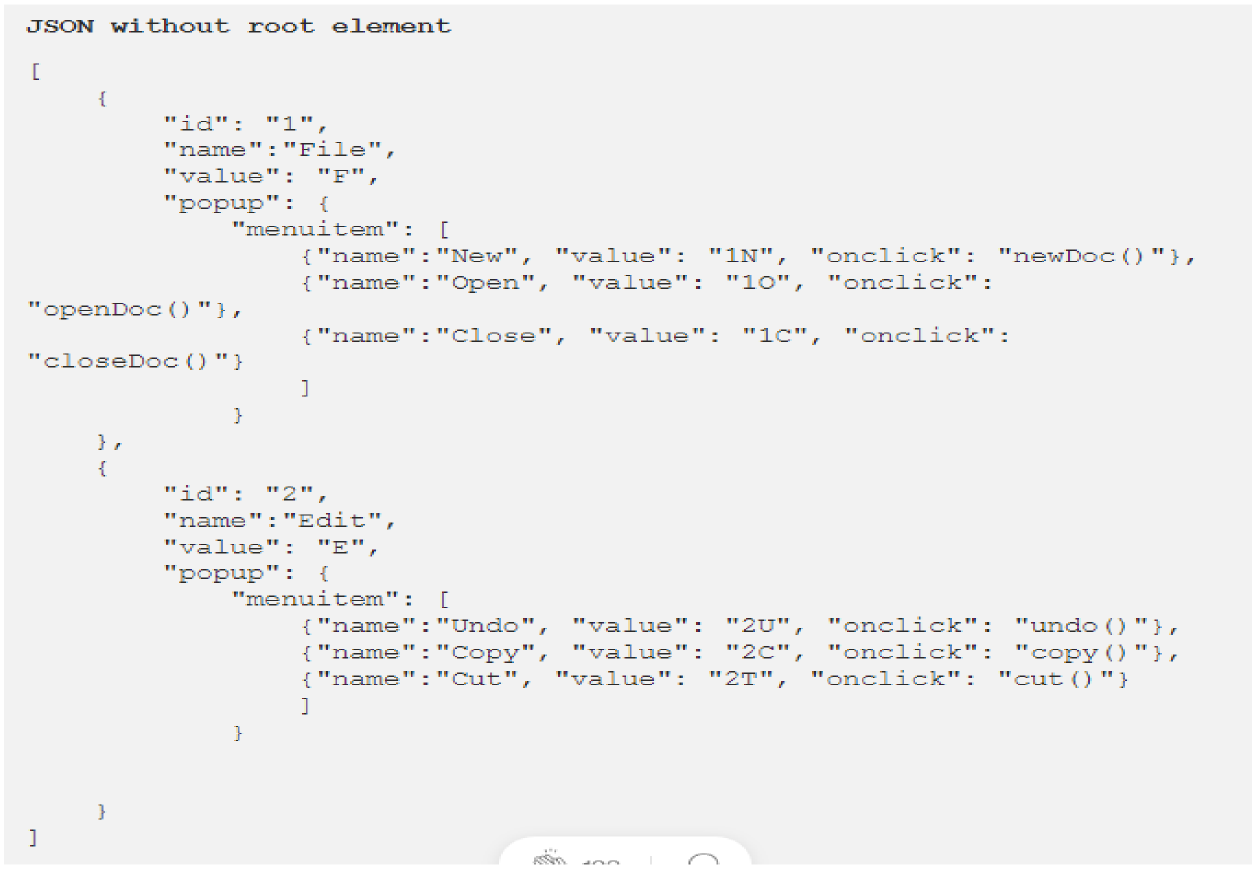
- **ALWAYS CREATE A ROOT ELEMENT.**

  - ✓ 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 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()"}
                ]
            }


    }
]
```

- **PROVIDE META SAMPLE**

    ✓ The idea of JSON is flexibility so you don't have to restrict your data feed within few columns.

    ✓ But at the same time, if you are providing large data set with nested levels, the consumer will go crazy. Provide them with the meta / sample, so it helps them to understand what data to look for and what to skip.

```
{
"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()"}
                ]
            }
    }
    ]
}
```

> *Meta sample can be*
>
> *menu.id : integer — unique identifier*
> *menu.name : string — name of the menu*
> *menu.value : string — internval id of the menu*
> *menu.popup.menuitem.name : string — name of the submenu*
> *menu.popup.menuitem.value : string — interlal id of the submenu*
> *menu.popup.menuitem.onclick : string — client event of the submenu*

4. REQUEST - RESPONSE BEST PRACTICES

❖ HTTP REQUEST:

- HTTP Request is the first step to initiate web request/response communication. Every request is a combination of request header, body and request URL.

REQUEST AREA	STANDARD DATA TYPE
Body	Simple String, JSON, Download, Redirect, XML
Header	Key Pair Value
URL Parameter	String

Think about an HTTP request as your browser connecting to the server and either asking for a specific resource or sending data to it. There are several types of HTTP request methods, which completely alter the type of response that you get from the server. The most common ones are:

GET.

This is the most frequently used HTTP request method by far. A GET request asks the server for a specific piece of information or resource. When you connect to a website, your browser usually sends several GET requests to receive the data that it needs for the page to load.

HEAD.

With a HEAD request, you only receive the header information of the page that you want to load. You can use this type of HTTP request to find out the size of a document before you download it using GET.

POST.

Your browser uses the POST HTTP request method when it needs to send data to the server. For example, if you fill out a contact form on a website and submit it, you’re using a POST request so the server receives that information.

PUT.

PUT requests are similar in functionality to the POST method. However, instead of submitting data, you use PUT requests to update information that already exists on the end server.

There are some other types of HTTP requests that you can use, including the DELETE, PATCH, and OPTIONS methods.

WHEN USE GET():

- ✓ GET is used to request something from server with less amount of data to pass.
- ✓ When nothing should change on the server because of your action.
- ✓ When request only retrieves data from a web server by specifying parameters
- ✓ Get method only carries request url & header not request body.

WHEN USE POST():

- ✓ POST should be used when the server state changes due to that action.
- ✓ When request needs its body, to pass large amount of data.
- ✓ When want to upload documents , images , video from client to server

Request Body:

- Request body should be structured in JSON Array/ Object pattern
- Request body hold multipart/ form-data like images, audio, video etc.
- Request body should not hold any auth related information.
- Request body should associated with specific request data model, setter getter can used for this

Request Header:

- Request header should carry all security related information, like token, auth etc.
- Only string **Key:Pair** value is allowed for header .
- Request header should provide user agent information of client application.
- If necessary CSRF/ XSRF should provide via header.
- Request header should associated with middleware controller, where necessary

HTTP RESPONSE:

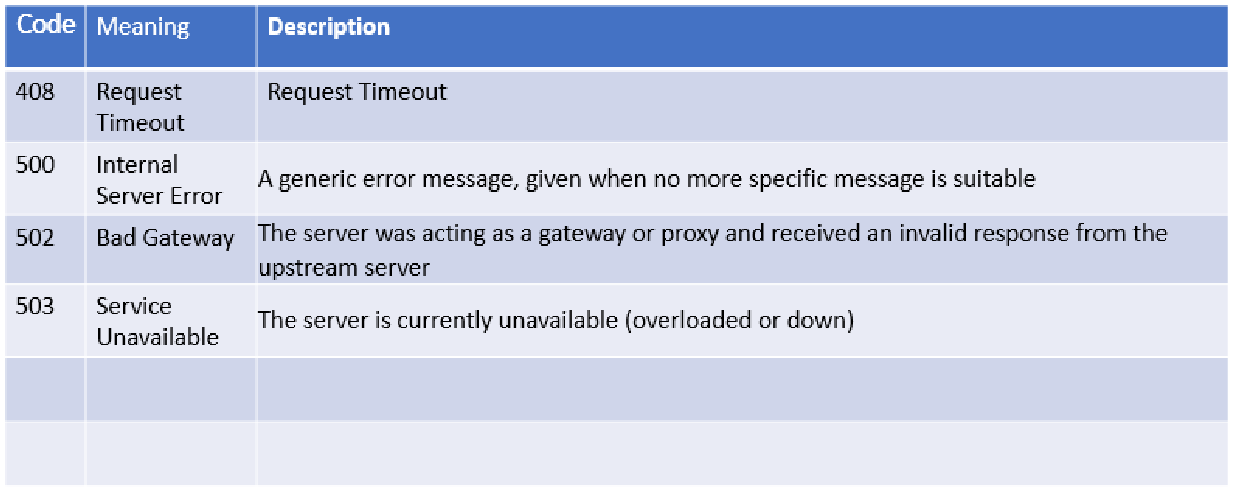
Http response is the final step of request-response communication. Every response is a combination of response header, body and cookies.

REQUEST AREA	STANDARD DATA TYPE
Body	Simple String, JSON, Download, Redirect, XML
Header	Key Pair Value
Cookies	Key Pair Value

HTTP RESPONSE STATUS MESSAGES:

Code	Meaning	**Description**
200	OK	The request is OK (this is the standard response for successful HTTP requests)
201	Created	The request has been fulfilled, and a new resource is created
202	Accepted	The request has been accepted for processing, but the processing has not been completed
203	Non-Authoritative Information	The request has been successfully processed, but is returning information that may be from another source
204	No Content	The request has been successfully processed, but is not returning any content
205	Reset Content	The request has been successfully processed, but is not returning any content, and requires that the requester reset the document view

Code	Meaning	**Description**
206	Partial Content	The server is delivering only part of the resource due to a range header sent by the client
400	Bad Request	The request cannot be fulfilled due to bad syntax
401	Unauthorized	The request was a legal request, but the server is refusing to respond to it.
403	Forbidden	The request was a legal request, but the server is refusing to respond to it
404	Not Found	The requested page could not be found but may be available again in the future
405	Method Not Allowed	A request was made of a page using a request method not supported by that page



Code	Meaning	Description
408	Request Timeout	Request Timeout
500	Internal Server Error	A generic error message, given when no more specific message is suitable
502	Bad Gateway	The server was acting as a gateway or proxy and received an invalid response from the upstream server
503	Service Unavailable	The server is currently unavailable (overloaded or down)

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 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 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

What is Web Application Security?

Web application security is a central component of any web-based business. The global nature of the Internet exposes web properties to attack from different locations and various levels of scale and complexity. Web application security deals specifically with the security surrounding websites, web applications and web services such as APIs.

What are common web app security vulnerabilities?

Attacks against web apps range from targeted database manipulation to large-scale network disruption. Let's explore some of the common methods of attack or "vectors" commonly exploited.

Cross site scripting (XSS) - XSS is a vulnerability that allows an attacker to inject client-side scripts into a webpage in order to access important information directly, impersonate the user, or trick the user into revealing important information.

SQL injection (SQi) - SQi is a method by which an attacker exploits vulnerabilities in the way a database executes search queries. Attackers use SQi to gain access to unauthorized information, modify or create new user permissions, or otherwise manipulate or destroy sensitive data.

Denial-of-service (DoS) and distributed denial-of-service (DDoS) attacks - Through a variety of vectors, attackers are able to overload a targeted server or its surrounding infrastructure with different types of attack traffic. When a server is no longer able to effectively process incoming requests, it begins to behave sluggishly and eventually deny service to incoming requests from legitimate users.

Memory corruption - Memory corruption occurs when a location in memory is unintentionally modified, resulting in the potential for unexpected behavior in the software. Bad actors will attempt to sniff out and exploit memory corruption through exploits such as code injections or buffer overflow attacks.

Buffer overflow - Buffer overflow is an anomaly that occurs when software writing data to a defined space in memory known as a buffer. Overflowing the buffer's capacity results in adjacent memory locations being overwritten with data. This behavior can be exploited to inject malicious code into memory, potentially creating a vulnerability in the targeted machine.

Cross-site request forgery (CSRF) - Cross site request forgery involves tricking a victim into making a request that utilizes their authentication or authorization. By leveraging the account privileges of a user, an attacker is able to send a request masquerading as the user. Once a user's account has been compromised, the attacker can exfiltrate, destroy or modify important information. Highly privileged accounts such as administrators or executives are commonly targeted.

Data breach - Different than specific attack vectors, a data breach is a general term referring to the release of sensitive or confidential information, and can occur through malicious actions or by mistake. The scope of what is considered a data breach is fairly wide, and may consist of a few highly valuable records all the way up to millions of exposed user accounts.

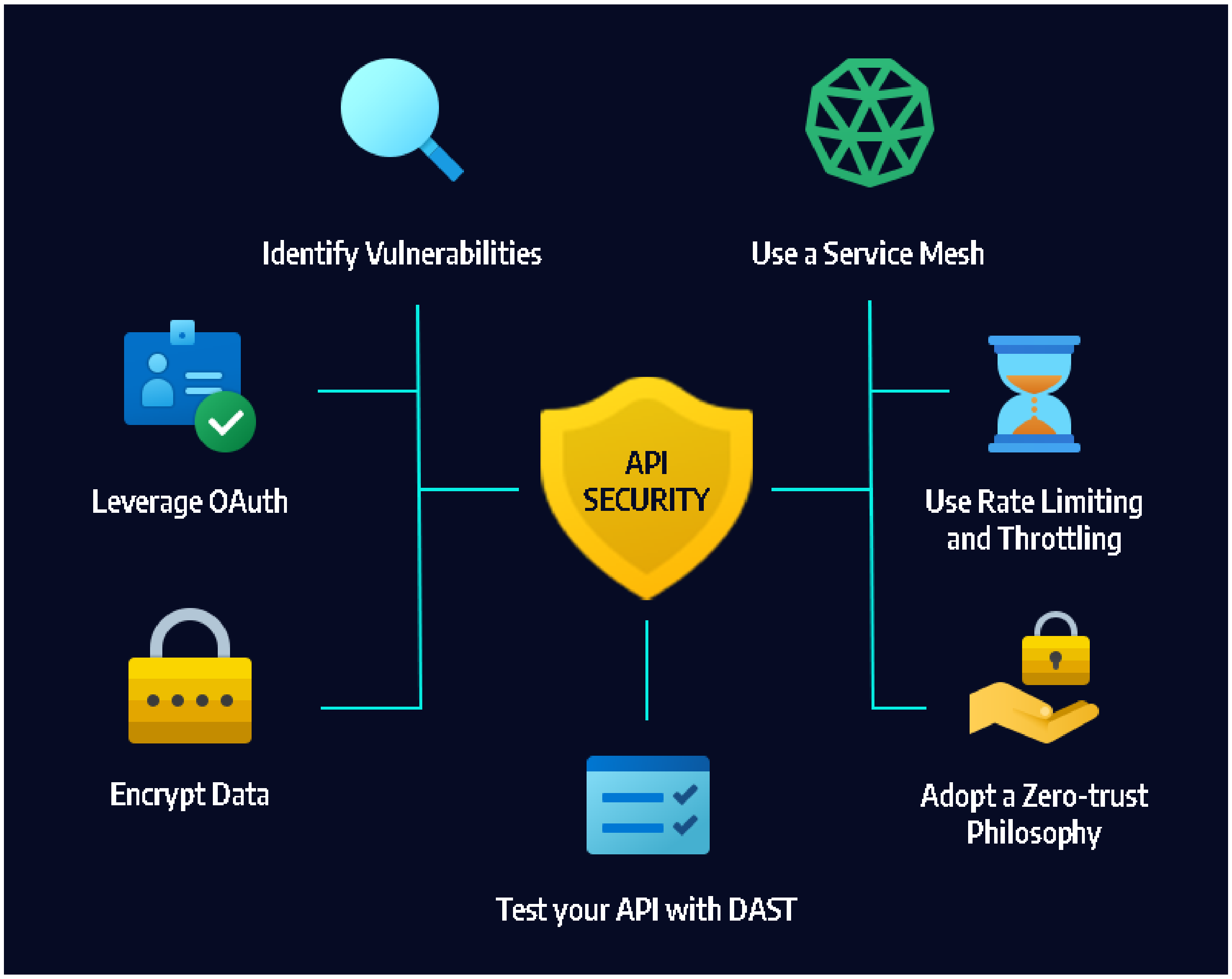
How to secure your Web?

Security Practices May Varies

- ✓ May varies from application to application
- ✓ May varies from developer to developer
- ✓ May varies from environment to environment
- ✓ May varies from use case to use case

But we have to know

- ✓ The best practices
- ✓ Know about the security layers
- ✓ Security placement



- **Require Injection & Input Validation**
  A good rule of thumb is to consider all input to be hostile until proven otherwise. Input validation is done so that only properly-formed data passes through the workflow in a web application. This prevents bad or possibly corrupted data from being processed and possibly triggering the malfunction of downstream components.

Some types of input validation are as follows:

- Data type validation (ensures that parameters are of the correct type: numeric, text, et cetera).
- Data format validation (ensures data meets the proper format guidelines for schemas such as JSON or XML).
- Data value validation (ensures parameters meet expectations for accepted value ranges or lengths).

There is a whole lot more to input validation and injection prevention, however, the basic thing to keep in mind is that you want to validate inputs with both a syntactical as well as a semantic approach. Syntactic validation should enforce correct syntax of information (SSN, birth date, currency or whole numbers) while semantic validation should enforce the correctness of their values within a very specific business context (end date is greater than the start date, low price is less than high price).

- **Encrypt your data**

Encryption is the basic process of encoding information to protect it from anyone who is not authorized to access it. Encryption itself does not prevent interference in transmit of the data but obfuscates the intelligible content to those who are not authorized to access it.

Not only is encryption the most common form of protecting sensitive information across transit, but it can also be used to secure data "at rest" such as information that is stored in databases or other storage devices.

When using Web Services and APIs you should not only implement an authentication plan for entities accessing them, but the data across those services should be encrypted in some fashion. An open, unsecured web service is a hacker's best friend (and they have shown increasingly smarter algorithms that can find these services rather painlessly).

- **Apply Authentication, Role Management & Access Control**

Implementing effective account management practices such as strong password enforcement, secure password recovery mechanisms and multi-factor authentication are some strong steps to take when building a web application. You can even force re-authentication for users when accessing more sensitive features.

When designing a web application, one very basic goal should be to give each and every user as little privileges as possible for them to get what they need from the system. Using this principle of minimal privilege, you will vastly reduce the chance of an intruder performing operations that could crash the application or even the entire platform in some cases (thus adversely affecting other applications running on that same platform or system).

Other considerations for authentication and access control include things such as password expiration, account lock-outs where applicable, and of course SSL to prevent passwords and other account-related information being sent in plain view.

- **Implement HTTPS**

Encryption at the service level is also extremely helpful (and sometimes necessary) preventative measure that can be taken to safeguard information. This is typically done by using HTTPS (SSL or Secure Sockets Layer).

SSL is a technology used to establish an encrypted link between a web server and a browser. This ensures that the information passed between the browser and the webserver remains private. SSL is used by millions of websites and is the industry standard for protecting online transactions.

In addition, blanket use of SSL is advised not only because it simply will then protect your entire website, but also because many issues can crop up with resources like stylesheets, JavaScript or other files if they aren’t referenced via HTTPS over an SSL.

- **Request Rate limit- Throttling**

We need to make sure our APIs are running as efficiently as possible. Otherwise, everyone using your database will suffer from slow performance. Performance isn’t the only reason to limit API requests, either. API limiting, which also known as rate is limiting, is an essential component of Internet security, as DoS attacks can tank a server with unlimited API requests.

Rate limiting also helps make your API scalable. If your API blows up in popularity, there can be unexpected spikes in traffic, causing severe lag time.

- **CSRF/XSRF Protection**

Cross-site request forgery attacks (CSRF or XSRF for short) are used to send malicious requests from an authenticated user to a web application.

- ✓ Use request-response header to pass CSRF token
- ✓ CSRF token should be unique for every session
- ✓ For self API CSRF token works well.

- **User Agent Protection**

User agent is a request header property, describe client identity like operating system, browser details, device details etc. Moreover every web crawler like Google crawler, Facebook crawler has specific user-agent name.

  - ✓ Using user agent we can prevent REST API from search engine indexing, social media sharing.
  - ✓ Can stop subspecies request from who is hiding his identity.
  - ✓ We can add user agent along with REST API usage history.
  - ✓ We can add device/OS usage restriction.

- **API Key**

    - ✓ This is the most straightforward method and the easiest way for auth.
    - ✓ With this method, the sender places a username:password/ ID / Keys into the request header.
    - ✓ The credentials are encoded and decode to ensure safe transmission.
    - ✓ This method does not require cookies, session IDs, login pages, and other such specialty solutions.

- **Output Validation**

  - **Output 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.

  - **Output 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.

- **CSRF/XSRF Protection**

  Cross-site request forgery attacks (CSRF or XSRF for short) are used to send malicious requests from an authenticated user to a web application.

    - ✓ Use request-response header to pass CSRF token
    - ✓ CSRF token should be unique for every session
    - ✓ For self API CSRF token works well.

- **User Agent Protection**

  User agent is a request header property, describe client identity like operating system, browser details, device details etc. Moreover every web crawler like Google crawler, Facebook crawler has specific user-agent name.

- ✓ Using user agent we can prevent REST API from search engine indexing, social media sharing.
- ✓ Can stop subspecies request from who is hiding his identity.
- ✓ We can add user agent along with REST API usage history.
- ✓ We can add device/OS usage restriction.