# (http://) HTTP – Hyper Text Transfer Protocol

======================================================

* Responsible for communication between web servers and clients
* HTTP is connectionless, after making the request, the client disconnects from the server, when the response is ready the server re-establish the connection again and deliver the response.
* HTTP can deliver any sort of data, as long as the two computers are able to read it.
* HTTP Requests/Responses (With Start line, Headers & Body)
* HTTP is Stateless. Every request is completely independent. It doesn’t remember about previous transactions.
* Programming, Local Storage, Cookies, Sessions are used to create enhanced user experiences.

# HTTP Methods

==================

* Request is made to the server using one of the methods.
* The method is a command that tells the server what to do

1. GET 🡺 Retrieves data from the server.

Can be loading standard html page, CSS, images, json data, xml data

1. POST 🡺 Submit data to the server.

Adding some resource to server, as submitting a form data

1. PUT 🡺 Update data already on the server.

To Update resources already on the server.

1. DELETE 🡺 Deletes data from the server.

To delete resource available on the server.

1. Other method1
2. Other method 2

# HTTP Messages

===================

**Request Message:**

|  |  |
| --- | --- |
| **Start Line** | **Method** **URI**(path/to/file.ext)  **HTTP Version**  GET/products/myproducts.html HTTP/1.0 |
| **Headers** | **Host** : www.mywebsite.com  **Accept** **-language**: en-us  **Accept**: MIME type  **Note : (MIME type is filetype/ext** - image/gif, text/html,  application/xml, application/json**)** |
| **Body** | Nothing to send to server if method is GET. IF method is POST, PUT or Delete then send the required information in the body. |

**Response Message:**

|  |  |
| --- | --- |
| **Start Line** | **Http/version Status Code**  HTTP/1.0 200:OK |
| **Headers** | **Host :** www.mywebsite.com  **Accept-language :** en-us  **Accept:** MIME Type (e.g. text/html)  **Server :** |
| **Body** | **File Requested**  products/myproduct.html |

Typing URL <https://anamikanny28.blogspot.com/> in the address bar will fetch following header data:

**General Header Data:**

1. **Request URL --** https://anamikanny28.blogspot.com/
2. **Request Method --** GET
3. **Status Code --** 200
4. **Remote Address --** [2404:6800:4009:810::2001]:443
5. **Referrer Policy --** no-referrer-when-downgrade

**Request Header Data:**

1. **Authority –** anamikanny28.blogspot.com
2. **Method –** GET
3. **Path –** /
4. **Scheme –** https
5. **Accept –** text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,\*/\*;q=0.8,application/signed-exchange;v=b3;q=0.9
6. **Accept-Encoding –** gzip, deflate, br
7. **Accept-Language –** en-US,en;q=0.9,hi;q=0.8,en-IN;q=0.7,en-GB;q=0.6
8. **Cache-Control –** max-age=0
9. **If-Modified-Since –** Mon, 23 Dec 2019 12:21:19 GMT
10. **If-None-Match –** W/"648dd58046f400a8c9dec4c7389fa715eea4de7020d0bcaa4ba67a0c1a42e884"
11. **Sec-Fetch-Mode –** navigate
12. **Sec-Fetch-Site –** none
13. **Sec-Fetch-User –** ?1
14. **Upgrade-Insecure-Requests –** 1
15. **User-Agent –** Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.79 Safari/537.36

**Response Header Data:**

1. **Alt-SVC –**

quic=":443"; ma=2592000; v="46,43",h3-Q050=":443"; ma=2592000,h3-Q049=":443"; ma=2592000,h3-Q048=":443"; ma=2592000,h3-Q046=":443"; ma=2592000,h3-Q043=":443"; ma=2592000

1. **Cache-Control –** private, max-age=0
2. **Content-Encoding –** gzip
3. **Content-Length –** 20163
4. **Content-Type –** text/html; charset=UTF-8
5. **Date –** Wed, 25 Dec 2019 15:14:31 GMT
6. **eTag –** W/"648dd58046f400a8c9dec4c7389fa715eea4de7020d0bcaa4ba67a0c1a42e884"
7. **Expires –** Wed, 25 Dec 2019 15:14:31 GMT
8. **Last-Modified –** Mon, 23 Dec 2019 12:21:19 GMT
9. **Server –** GSE
10. **Status –** 304
11. **X-Content-Type-Options –** nosniff
12. **X-Xss-Protection –** 1; mode=block

**HTTP STATUS CODES:**

1. **1xx: Informational**

Request received / processing

1. **2xxx: Success**

Successfully Received, Understood and accepted.

**200 – OK**

**201 – OK Created**

1. **3xx: Redirect**

Further action must be taken / redirect.

**301 – Moved to new URL**

**304 – Not modified (Cached Version)**

1. **4xx: Client Error**

Request does not have what it needs

**400 – Bad Request**

**401 – Unauthorized**

**404 – Not Found**

1. **5xx: Server Error**

Server failed to fulfil an apparent valid request

1. **– Internal Server Error**

**HTTPS (Secure Hyper Text Transfer Protocol)  
======================================**

**Kubercosion**

* HTTP with a security feature.
* Encrypts the data that is being retrieved by HTTP.
* Uses encryption algorithm to scramble the data that’s being transferred.
* HTTPS secures data by using one of following protocols:
* **SSL** (Secure Sockets Layer)
* **TLS** (Transport Layer Security)

**Secure Sockets Layer (SSL)**

* Protocol that’s used to ensure security on the internet.
* Uses public key encryption to secure data.
* Website using https sends SSL certificate to the client.
* A SSL certificate is a small certificate that is used to authenticate the identity of a https website.

**Transport Layer Security (TLS)**

* Successor to SSL
* Latest industry standard cryptographic protocol.
* Authenticates the server, client, and encrypts the data.

**Public Key Cryptography:**

* Any message encrypted with Bob’s public key can only be decrypted with Bob’s private key.
* Anyone with access to Alice’s public key can verify that a message (signature) could only have been created by someone with access to Alice’s private key.

**How Https Works:**

* **Browser :** Give me https://youtube.com
* **Youtube:** Sure, here’s my certificate, containing my public key. It is also signed by Google CA (Certificate Authority).
* **Browser:** I know Google CA’s public key. I will verify this. So, looks like you’re indeed who you say you are. I have created a new secret key and encrypted it with your public key.
* **Youtube:** Only I have my private key and can decrypt this. So I now have your public key and you have private key.From now on youtube will encrypt all of the communication to you with your public key which you can decrypt with your private key.

**Certificate Authority:**

* How to make sure party you are talking is the one who they claim they are?

**Parties Involved:**

1. **Your Web-Browser**
2. **Google CA -** Consider as trusted Certificate Authority on internet. As any party involved in public key cryptography, Google CA has a private key and public key.
3. **https://youtube.com -** Also needs to create private and public key pair.

**Youtube:** Sends Certificate signing request to Google CA with its key pair.

**Google CA:** Signs Certificate with its private key. Anyone who has Google CA’s public key can verify that it was actually signed by Google CA.

**Browser:** Already has list of trusted certificates and these certificates are issued by known CAs thus browser know public key of CAs.

When https://youtube.com sends its certificate claiming it’s signed by Google CA browser can verify it using Google CA’s public key at already has Google CA’s certificate.

Now browser has youtube’s public key to decrypt messages send by youtube encrypted using its private key.

**Self-Signed Certificate:**

* Sometimes it may happen well known CA doesn’t sign your request.
* Mainly used for testing the HTTPS app deployed in staging environment.
* Create a private and public key pair for the app.
* Create your own Certificate authority (CA) having private and public key pair.
* App creates a certificate signing request to custom CA.
* CA shall make sure you have access to URL you’re claiming to be yours.
* Custom CA sign the certificate and sends to the app.
* Anyone with CA’s public key can verify it’s signed by that CA.
* Your second app wants to interact with your HTTPS app just deployed in the staging environment.
* Second app looks the certificate provided by first app and finds public key. Certificate also tells it signed by custom CA.
* Make second app to trust the custom CA and provide certificate of custom CA to second app so that it can have public key of Custom CA.
* Public key of custom CA can be used by second app to verify certificate provided by first app claiming to be signed by custom CA.
* If app is not having end users on open internet then cost involved in getting certificated signed by trusted CA can be removed by using Self-Signed Certificate procedure.

**Kubernetes can automate the process of using Self-Signed Certificate as well as getting signed by trusted CA.**

**JWT (JSON WEB TOKENS)   
=====================**

**https://www.youtube.com/watch?v=\_XbXkVdoG\_0**

* JWT is very popular way to perform user authorization in web apps.
* JWT is also very popular in context of microservices architecture.
* JWT is commonly used for managing authorization but idea behind JWT is to create a standard way for two parties to communicate securely.
* Open Industry Standards Specification RFC 7519 outlines how JWT should be structured and how to use it for exchanging information.

**Authorization Strategies**

* Session Token
* JSON Web Token
* All authorization mechanism exists due to HTTP being a stateless protocol.
* Every interaction using HTTP needs to contain all the information needed for that interaction. Nothing is remembered from before i.e. no state is maintained over multiple requests.
* Server running a static webapp does not have any issue with http as each response from server is standalone and does not need to know who the user is. In this case user just needs to send URL of the resource to the server.
* When the response from server is dynamic and user based, server needs to know who the user is. In this case user needs to send to server URL + Identitiy.
* Example, a banking application. On first request server authorizes with userid and password while for subsequent requests also like checking balance etc. server needs to know your identity.
* To remember an authorized user, a dynamic webapp needs to maintain session information for that user.
* There are multiple ways in which webapp manages and remembers sessions. Two popular options are using session token and JSON Web token.
* In **Session Token strategy** server creates a session and keeps track of it itself. The Server creates a Session-id associated with the session and also gives that session-id to the client with response object in a cookie.
* Subsequently on every request to the server, client/ browser adds cookie containing session-id in header to request object. The server looks session-id in the reference table and identifies the client.
* **Session ID + Cookies –** Most popular mechanism for authorization.
* Session token strategy holds good for webapp running on single monolithic server. But nowadays single app splits to run on several servers using a load balancer. In this case if ServerA authorizes client and provides a session-id then ServerB will not recognize client with same session-id as ServerB does not have session references maintained by ServerA.
* A use case for above problem may be to provide a Shared Session Cache (Redis Cach e) where every server can look up for session references. But problem is it becomes single point of failure. If this cache goes down each sessions goes down.
* Other use case to follow Sticky Session pattern in which load balancer remembers which server has given user session and it always redirects requests from that user to that specific server. It is also not scalable. It is also a failure in case of microservices architecture.
* In **JSON WEB TOKEN** strategy instead of server saving the client information and state on the server and returning the id as a token it returns entire client information as a token. A JSON payload containing client information returned to client with response. Server doesn’t save any session information.
* On every subsequent request client sends whole JSON token with request object saying this is my id and this is my name and I’m authenticated.
* The token is not id but a JSON object containing all the information about the client session called as JSON WEB TOKENS (json tokens exchanged over the web).
* The issue of security is handled by signing the token that sent by client with each request. When server authenticates a client it sends a signed token to client. So, client sends back this signed token to server on each request which server verifies to be its own signature.
* It is a way for client and the server to communicate information directly without the server to remember information about each client.
* **Session Token** is Reference token (reference to a state). It refers to a state on the server. Session-Id can be exchanged with cookies.
* **JSON Web Token** is value token. It contains value. It can also be exchanged with cookies.
* **JWT** doesn’t look like a simple JSON object. It is in encrypted format.

**Structure of JSON WEB TOKEN (JWT)**