NOTE :

- There are 3 Main People/Group of people/Org responsible of modern day web dev :
  - **Tim berner lee** : URI, WWW, HTTP, HTML
  - **Lou Matriouli** : Cookies
  - **CERN/NCSA/Ari Luotonen/Robert Mccool** : http web server, CGI/REST API

About HTTP :

- Based on Client Server (Req-Res) Model
- Mostly uses Reliable/TCP-IP Connection over port 80 (HTTP3/QUIC uses UDP)
- Stateless But not Sessionless : HTTP Cookies provide session Mechanism/Stateful sessions.
- Proxies/Middleware (B/W Client & Server) : Cache, Filter, Log, Load balance, Auth./Cookies, CORS, Tunnel, etc.
- API - XMLHttpRequest, server-sent events etc. [REST, SOAP etc.]
- HTTP cookies violate the REST architectural style because even without referencing a session state stored on the server, they are independent of session state (they affect previous pages of the same website in the browser history) and they have no defined semantics.
- Cache - private, Shared (Proxy, Managed)
- Status Code : 1xx - Info | 2xx - Success | 3xx - Redirect | 4xx - Client Error | 5xx - Server Error
- Authorization ~ Request Header (Client) <=> Authentication ~ Response Header (Server)
- HTTP Auth (General Syntax) :
  - **Authentication** - WWW-Authenticate/Proxy-Auth: <type/Auth-Scheme> realm/charset/token68/etc.
  - **Authorization** - Authorization/Proxy-Auth : <type> <creds>

HTTP Authentication :

- **HTTP Auth Schema/Methods** :
  - **Basic Schema (Insecure/Used with HTTPS)**
    - Resource/ww-Authe OR Proxy Authe. : Basic realm="Access 2 site", charset="UTF-8" (Server res./HTTP header)
    - Autho/Proxy-Autho : Basic <UTF-8 Base64 Encoded key> (Client req/HTTP header)
  - **Bearer Schema** : Used to send info back from client <-> server in http header
    - Resource/WWW-Authe OR Proxy Authe. : Basic realm="Access 2 site", charset="UTF-8" (Server res./HTTP header)
    - Autho/Proxy-Autho : Bearer (Client req/HTTP header)
  - **Other Schemas** : Digest, HOBA, Mutual, Negotiate/NTLM, VAPID, SCRAM, AWS4-HMAC-SHA256 etc.
- **HTTP Cookies** : Stateful/Storage/etc.
  - HTTP Header : **Set-Cookie** (creates a session/link req. with state of server thus making http stateful)
  - Disadvantage (Cons) : Server needs to store sessionID for every user which increase overhead on server, hence other Auth mechanisms like JWT, Oauth etc. used for Auth purpose. Also it's vunerable to CSRF Attack, has size limitation, less scalable depending upon the implementation.
  - Algorithm (General) : SID is also a cookie (session cookie).
    - Server creates session/SID & stores in DB (After Creds Verified) & Sentback cookie+sessionID (i.e 2 cookies)
    - Cookie(user/pass) + SID (session cookie) stored in browser.
    - For every next req. SID is verified/hashed against DB.
    - For logout/session expiry - delete cookie+SID from client (By setting time as backdate) & SID from DB (backend).

HTTP Auth (Basic) Vs. Other Auth Mechanism (Cookies/SessionID) :
In basic Auth there is no way to Revoke/logout user & is also insecure. Hence, other Auth mechanisms like cookie based, Session/Token/JWT based approach is used where server after authentication.
General flow is like server sets a cookie, sessionID at client side using which it makes connection stateful and delete cookie or session at time of revoking.

App Server vs Web server :

Apache/Ngnix HTTP/Httpd web servers are written in C/C++, so they cannot execute Python/node/php code directly, a bridge is needed b/w the web server and the program. These bridges/interfaces/CGI/mod_wsgi, define how programs interact with the web server. This Bridge is Application server.

```
Program (py) <-> App server (py mod_wsgi module) <-> web server (apache/ngnix http/httpd)
```

The dynamically generated urls are passed from the WebServer to the Application server. The application servers matches to url and runs the script for that route. It then returns the response to the WebServer which formulates an HTTP Response and returns it to the client. Earlier, Programs using CGI to communicate with their web server need to be started by the server for every request. So, every request starts a new Python interpreter – which takes some time to start up – thus making the whole interface only usable for low load situations. Hence WSGI, FastCGI all comes into the scene lateron.

In RedHat, the httpd itself configures and give access to use CGI. Any code written in the folder /var/www/cgi-bin is treated as CGI. Here the client can only run the program but cannot change the code. While in /var/www/html the file is treated as web page and httpd provides the facility of viewing the page only. To access the cgi code the client need to provide the URL on web browser. So every CGI program is nothing but API’s that can be accessed by any user. This is how API works.

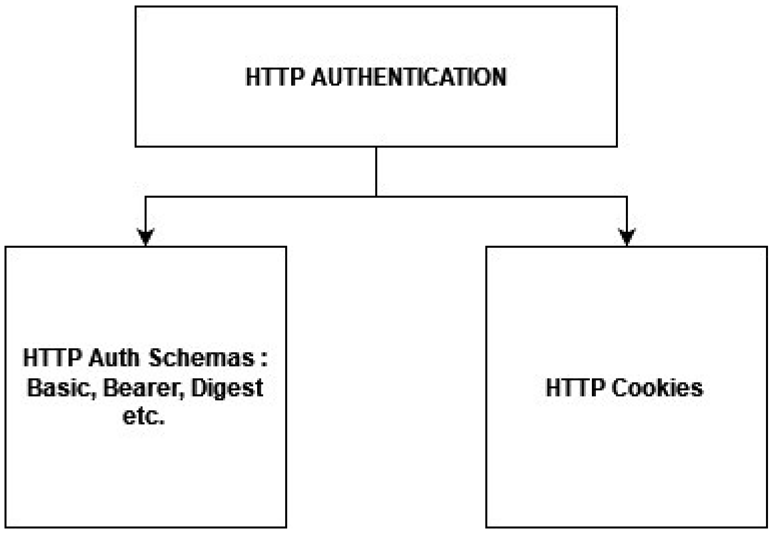
CGI/WSGI/FastCGI/Web Service/Web Server Vs API :

API work because of CGI. CGI is like a gate which allows any user to go in the OS and run the program. Earlier CGI used to create dynamic pages/web forms to execute a script/command/run some logic in backend and give the output back to frontend. Dynamic web sites are not based on files in the file system, but rather on programs which are run by the web server when a request comes in, and which generate the content that is returned to the user. Every web-service or web programs work on the concept of CGI. Whenever we want to run a program in another OS without networking/Remote Login/SSH, make the code as CGI code in the web server. API also works on the concept of CGI. In the URL, we provide a PROGRAM name, so the client can interfere in the server by using program known as INTERFACE/WSGI and run APPLICATION typically known as API/send json response/send text or html/execute py script file etc. CGI helps the client to come inside the server and after the client enters it all depends on API, which program to run.

HTTP Security : Access Control (Req./Res. Header based Mech.)

- **CSP (Content Security Policy)** - CSP is an HTTP header that allows site operators fine-grained control over where resources on their site can be loaded from. It is used to detect and mitigate certain types of website related attacks like XSS, click-jacking, etc. It controls resources the client is allowed to load for a given page
- **Same origin & Cross origin Policy (HTTP Response Headers)**
  Origin - two URLs have the same origin if the protocol/scheme, port (if specified), and host/domain are the same for both. You may see this referenced as the "scheme/host/port tuple", or just "tuple". To Allow cross origin use CORS (with http cookies samesite flag) and to block them use CSRF/Anti-CSRF tokens.
  - **CORS (Cross-origin resource sharing) :** CORS isn’t actually enforced by the server, but rather the browser. The server simply states the sites that are allowed cross origin access through the Access-Control-Allow-Origin header in all its responses. It is up to the browser to respect this policy. Ajax, XMLhttpRequest, fetch() etc.
    - **Response Header**
      - Access-Control-Allow-Origin
      - Access-Control-Allow-Methods
      - Access-Control-Allow-Headers
      - Access-Control-Allow-Credentials
      - Access-Control-Max-Age
      - Access-Control-Expose-Headers
    - **Request Header**
      - Origin
      - Access-Control-Request-Method
      - Access-Control-Request-Headers
  - **CORP (Cross origin resource policy) :** Prevents other domains from reading response of the resources to which this header is applied. CORP only affects cross-origin requests that can already be made without requiring CORS to relax the same-origin policy - i.e. requests which are already allowed by the same-origin policy, such as cross-origin requests for images, CSS stylesheets, and for JavaScript scripts. conveys a desire that the browser blocks no-cors cross-origin/cross-site requests to the given resource. CORP complements CORB. The COEP when used upon a document, can be used to require subresources to either be same-origin with the document, or come with CORP to indicate they are okay with being embedded. This is why the cross-origin value exists.
  - **COEP (Cross origin embedder policy) :** Allows a server to declare an embedder policy for a given document. It prevents a document from loading any cross-origin resources that don't explicitly grant the document permission (using CORP or CORS).
  - **COOP (Cross origin opener policy) :** Prevents other domains from opening/controlling a window. It allows you to ensure a top-level document does not share a browsing context

group with cross-origin documents.
    - **CORB (Cross origin read blocking) :** Mechanism to prevent some cross-origin reads by default.
- **Others** - HPKP, HSTS/HTTPS, Cookie security, X-Content-Type, X-Frame, etc.


HTTP AUTHENTICATION
HTTP Auth Schemas : Basic, Bearer, Digest etc.
HTTP Cookies