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.

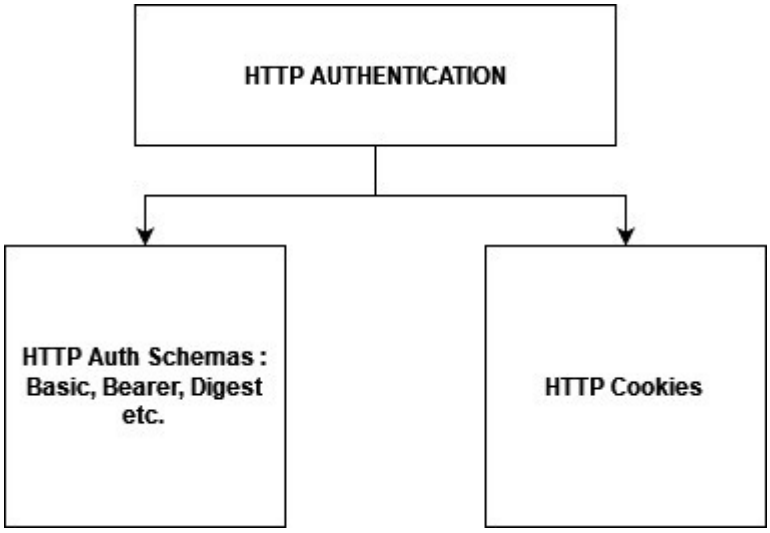
CGI/WSGI/FastCGI/Web Service/Web Server Vs API :

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.



S3 : State | Storage | Session

1. State Management (Stateful vs. Stateless)

State :

state is memory of previous events. It is useful for handling data that changes over time or that comes from user interaction. HTTP is stateless not sessionless, bcz with cookies implemenation it could be made stateful/Sessionful.There can be complex interactions between stateful and stateless protocols among different protocol layers. For example, HTTP Cookies are stateful based on HTTP, a stateless protocol, is layered on top of TCP, a stateful protocol, which is layered on top of IP, another stateless protocol, which is routed on a network that employs BGP, another stateful protocol, to direct the IP packets riding on the network.

- Eg. : In React.js State is similar to Props but it is private, fully controlled by the component and triggers a rendering. State in React refers to local state mainly. It is like a component's personal data storage. As http+cookies = sessionful/stateful, similary Components + State (local/global/useState/useEffect) = Stateful Component. State gives components (Class/Functional) memory. Class comp. have local state & functional comp have hooks (useState & useEffect). React does not persist the data between page load. You should persist your state locally or use a state management library like redux (RRR : react/redux/router). Higher Order Component is a function that takes a component and returns a new component. It's not really a component, more of a design pattern known as decorator. In React apps, whether a component is stateful or stateless is considered an implementation detail of the component that may change over time. You can use stateless components inside stateful components, and vice versa.

Types of States -

- **App./Model State** - db-cache/session/backend/server-side/Sec. memory-HDD [Eg. Node/Django]
- **View State** - frontend/client-side (cache/cookies/session/localstorage/websql/indexDB)/JS Obj./In-process/pri. memory-RAM [Eg. React]

Stateful Service : Stateful ~ Session-based -> You store data on server & client just hold SID/opaque_token(session cookie).

- Stateful is something that has a concept of a session or persistence, like a chat service. For a stateful service, you hand out a new short-lived, single-use token(here, session cookie) for each service - which is then exchanged on the service itself, for a session on that specific service. i.e. server sends back SID to client which is stored at client side in session cookies as SID and sent back to server on each call via HTTP Auth. header or cookies.
- **Cons :** Server had to make a DB Call after verification to see details about user info ie. authorization, which is a burden but can be handled by fast db lookup(redis db). You never use the token itself as the session (See below stateless topic for clarity).

Stateless Service : Stateless ~ Tokens ~ JWT-Auth0/PASETO/Fernet/Branca etc.

- Something that does not have a concept of a session, but rather performs individual self-contained tasks, like a video transcoding service. For a stateless service, there's no session at all, so you simply have the app. server hand out short-lived, single-use tokens for each individual authorized operation. stateless auth. is where user state is never saved in DB memory/Server/RAM/HDD because all data is embedded into token, client hold token and send to server for auth. The DB Call are drastically reduced in stateless approach by using JWT tokens.
- **Cons :**
  - **Token Revocation :** Biggest drawback of token/jwt is that it comes with expiry and as it is self-contained/sufficient, hence not dependent upon server, so it's difficult to revoke-logout/invalidate/update it from server.
    - **Workaround/hack :** One hack is to store "revoked tokens" list in db & check for every call. If the token is part of that list, then block user for the next action. But then now you are making that extra call to the DB to check if the token is revoked and so deceives the purpose of JWT altogether. So you see, ultimately it needs to stateful combo. i.e. fast db calls.+ tokens.
  - **Security :** JWT's are often not encrypted so anyone able to perform a MITM attack and sniff the JWT now has your auth creds. This is made easier because the MITM attack only needs to be completed on the connection between the server and the client.
  - **Scalablity :** In many complex real-world apps, you may need to store a ton of different information. And storing it in the JWT tokens could exceed the allowed URL length or cookie lengths causing problems. Also, you are now potentially sending a large volume of data on every request.
- **Conclusion :** In many real-world apps, servers have to maintain the user's IP and track APIs for rate-limiting and IP-whitelisting. So you'll need to use a blazing fast DB/Stateful/Session/Redis_Cache anyway. To think somehow your app becomes stateless with JWT is just not realistic. The solution is to not use JWT at all for session purposes. But instead, do the traditional, but battle-tested way more efficiently. In simple words, hybrid approach(Stateful+stateless) by making the DB lookups so blazing fast(Redis DB) that the additional calls won't matter.
  - This is similar to many newer developers learning how to build SPAs with React before server-rendered HTML. Experienced devs would likely feel that server-rendered HTML should probably be your default choice, and building an SPA when needed, but this is not what new developers are typically taught.
- **Best usecase :** There are scenarios where you are doing server-to-server (or microservice-to-microservice) communication in the backend and one service could generate a JWT token to send it to a different service for authorization purposes. Example in Oauth/OIDC (Which is used for authorization not authentication. OIDC - OpenID Connect builds on OAuth 2.0) yourwebsite.com lets you login via facebook.com. Another eg. - Password reset, where you can send a JWT token as a one-time short-lived token to verify the user's email. Btw medium.com uses this same approach For Login by sending you a token on email, they call it magic link 😉.

---

2. Storage mechanism (client/browser/web storage) & Security measures :

Note : Server side storage is nothing but storing on database/HDD. So will not discuss it here & Session, Tokens are not storage mechanism but uses cookies as storage mechanism. so will discuss them here with cookies.

1. **Cookies (HTTP/browser/web/internet/client Cookie) [Key/Value] :**
    - **Definition** : It's a Client side storage mechanism having key/value pair as its data structure. It has some associated metadata/attributes for different purpose. HTTP (Stateless) + cookies = Stateful HTTP.
    - **Create/Delete Cookie :**
        - **Request Header** : The "Cookie" HTTP request header contains stored HTTP cookies associated with the server (i.e. previously sent by the server with the "Set-Cookie" header or set in JS using "Document.cookie").
        - **Response Header** : The "Set-Cookie" HTTP response header is used to send a cookie from the server to the user agent via HTTP protocol, so that the user agent can send it back to the server later. To send multiple cookies, multiple Set-Cookie headers should be sent in the same response. It can set by a web server (ngnix,apache) or by the application's code (python,node,php), it doesn't matter much for the browser. What matters is the domain the cookie is coming from.
            - Eg. 1 - `Set-Cookie : id=a3fWa; Expires=Thu, 21 Oct 2021 07:28:00 GMT; Secure; HttpOnly; SameSite=Lax`
            - Eg. 2 - `Set-Cookie : id=a3fWa; Expires=Thu, 01 Jan 1970 00:00:01 GMT /* Removes cookie by setting backdate */`
            - Eg. 3 - `<META> tag in the returned HTML document. Eg. : <meta http-equiv=Set-Cookie content="sessionid=123">`
    - **Type of Cookies (Fundamentally only two) :**
        - **Persistent Cookie (Manual/Longer expiry) :** A persistent cookie expires at specific date/after specific length of time.
            - **Usage** - Remember Password for longer time/Keep logged in over multipb tabs/browser/devices sessions, Tracking/advt (3rd Party) etc.
        - **Session Cookie (shorter/browser expiry - Default/unspecified) :** in-memory/transient/non-persistent cookie. session_id(SID) itself sometime called as server side cookie.
            - Eg. server responds with 2 cookies as follows :
                - `Set-Cookie: theme=light /* Session cookie - No expiry or browser expiry */`
                - `Set-Cookie: sessionTokenID=abc123; Expires=Wed,9 Jun 2021 10:18:14 GMT /* Persistent cookie - Manual Expiry */`
            - **Usage** : A/B Testing, load balancing, robot protection etc. helps to improve page load times, since amount of info in session cookie is small & requires little bandwidth. Session Cookie concept is used as Token (JWT) for Stateless Authentication. Unlike cookies, which are automatically attached to each HTTP request by the browser, JWTs must be explicitly attached to each HTTP request by the web application.
            - **Loadbalancing issues** : server 2 has no idea about received cookie as it has not issued this cookie, server 1 has.
                - **Sol 1 : Session Trickling :** Two servers always talk to each other. Whenever server 1 issues a cookie, it also tells server 2 about this new cookie. Server 2 would then store that in its memory. Vice is versa is also applicable. When user logs out, similar thing happens. Cookie on other server is destroyed.
                - **Sol 2 :** First approach is rudimentary. Better approach to share cookie is via shared Database like MySQL. Concept is same. Only thing is, instead of communicating directly, servers co-ordinate via database.
                - **Sol 3 :** Second approach is better but slower as every request would involve database call. To tackle this, high speed in-memory databases like "Redis Cache DB" are used. Redis is NoSQL, Key-Value database which is perfect fit for this scenario.
                - **Sol 4 :** In a different scenario, where we have micro-services, our API servers or web servers are hidden behind a firewall and there is a separate service that is responsible for issuing and validating cookies. Sometimes application gateways/load balancers are responsible for cookies and web servers are hidden from direct access.
            - **Classification based upon Cookie usage for :**
                - **Authentication (Stateful Sessions/First Party/Same-origin Cookies) :** Earlier Auth was done using Basic HTTP Auth Header Mechanism, but due to its insecurity cookie is used.
                    - Algo. :
                        - Client Req. Server
                        - Http server replies to http client with HTML document + state object (Cookie/SID) & store SID in DB for Ref.
                        - For every subsequent req. client now send cookie by default in header, so that server could maintain session.
                    - Usecase (API/AJAX/Fetch) :
                        - user visits "https://www.a-example.dev"
                        - user clicks button or makes some action which triggers Fetch request to "https://api.b-example.dev"
                        - "https://api.b-example.dev" sets a cookie with "Domain=api.b-example.dev"

        - On subsequent Fetch requests to "https://api.b-example.dev" the cookie is sent back
    - **Tracking/advt/preferences :** 3rd Party Cookies (Different/cross origin) - Persistent cookies
        - Alternate to cookies for tracking : uniqID 2.0, Floc, Apple iAd, HTTP Etag (Cache), Browser fingerprint etc.
- **Pros/Cons of Cookie :**
    - **Pros** : Cookies when compared to session or local storage have benefit against XSS attack and doesn't needed to be manually send token/cookie over each request. Session storage only persist for tab & localstorage is hard to maintain, while cookie persist over browser life-cycle and easy to maintain.
    - **Cons** :
        - Storage : limited to 4KB. Restriction is put by browser not HTTP protcol. (Alt : local or sessionStorage)
            - it can support at least 50 cookies per domain (i.e. per website)
            - it can support at least 3,000 cookies in total.
        - Cookies violates RESTfulness :
            - Working of REST API : Rest API might first look in the Authorization header for the authentication data it needs, since that's probably the place where non-browser clients will prefer to put it, but to simplify and streamline browser-based clients, it might also check for a session cookie for server side log in, but only if the regular Authorization header was missing.
            - Let's review 2 key pieces of information :
                - Sessions are stored on the server i.e. the data which SID points to
                - RESTful constraints dictate that sessions should only be stored by client but only SID, a ref. is stored
            - REST is inspired from HTTP, hence stateless. Restfulness comes from statelessness - of the server. Basically sessions/states need historical data and are dependent on past requests so to speak, restful applications ideally aren't. If client want to have a stateful interaction then each request from the client should contain all the information necessary to service the request – including authentication. HTTP Cookie does this very well except session cookies which only store SID as reference, hence violating the statelessness constraint. In this sense, sessions do indeed violate RESTfulness. But that's not necessarily a bad thing. As with most decisions in software development, we're making a trade-off. By sacrificing statelessness, we're enabling more meaningful user experiences. We can add an item to our cart on our phone, and complete the transaction on our laptop. We can log in to a website once and get a customized view of a page that is tailored to our own filters or settings. Any session state that is stored on the server violates one of the key principles of REST, that state only be stored by clients. While this may make our application not RESTful by its original definition, it's a trade-off that may be worth making. Every application is different and will have varying requirements. By forfeiting some of the constraints of REST, we may be enabling different user experiences.
- **Cookie Attributes :** Browsers do not include cookie attributes in requests to the server—they only send the cookie's name and value. Cookie attributes are used by browsers to determine when to delete a cookie, block a cookie or whether to send a cookie to the server.
    - **Secure** : will send over HTTPS only
    - **HttpOnly** : Will be set by server only. i.e you cant set by JS script document.cookie.
    - **Samesite (CORS)** :
        - **Defintions** :
            - **Site** : For example, the "www.web.dev" domain is part of the "web.dev" site.
            - **SameSite** : The SameSite attribute lets servers specify whether/when cookies are sent with cross-site requests (where Site is defined by the registrable domain and the scheme: http or https). This provides some protection against CSRF. If the user is on "www.web.dev" and requests an image from "static.web.dev" then that is a same-site request.
            - **CrossSite** : If user is on "your-project.github.io" and requests an image from "my-project.github.io" that's a cross-site request.
        - **Values** :
            - **Strict :** cookie is only sent to the site where it originated/first party context (ie. samesite). The browser does not send a cookie to other sites when following a link. Good when you have cookies relating to functionality that will always be behind an initial navigation, such as changing a password or making a purchase. Cookies that are used for sensitive info./auth should have a short lifetime, with the SameSite attribute set to Strict or Lax.
            - **Lax (Default/unspecified) :** cookies are sent when the user navigates to the cookie's origin site. For example, by following a link from an external site.
                - Eg. 2 : Let's say your website include a embedded link/img from my website. And logic is such that if you have cookie (login) then only you could view the image. Now if i had set it to "strict" no matter what your website cant view my image even though you get the cookie. If i had set it to "Lax", its similar to "strict" except now when you

click the link to navigate from that embedded link to my website you will be allowed. Remeber the image is still not visible in another site. This kind of approach is used with URL Navigation. LAX is relax.
        - There are a limited set of circumstances in which a browser will send a cookie :
            - Link (HTTP Get) <a href="…">
            - Prerender (HTTP Get) <link rel="prerender" href="…">
            - Form HTTP Get <form method="get" action="…">
        - The situations in which Lax cookies can be sent cross-site must satisfy both of the following:
            - Request must be a top-level navigation. You can think of this as equivalent to when the URL shown in the URL bar changes, e.g. a user clicking on a link to go to another site.
            - The request method must be safe (e.g. GET or HEAD, but not POST/AJAX).
            - When using Lax, you need to ensure that HTTP Get requests do not change the state of your application. Using Get to change the state of your application is a very bad practice, but it becomes even more dangerous when your application uses Lax, and you make the false assumption that you are now protected against CSRF attacks.
    - **None + Secure** : cookies are sent on both samesite & cross-site requests, but only in secure contexts (i.e., if SameSite=None then the Secure attribute must also be set).
- **CORS Settings** :
    - Frontend (React/Fetch) - credentials: "include"
    - Backend (Node/Python) - Access-Control-Allow-Credentials | Access-Control-Allow-Origin
- **Domain (Cookie Scoping/Permission)** : Specifies which hosts can receive a cookie.
    - Default/unspecified : domain defaults to the same host that set the cookie, excluding subdomains.
    - Specified : domain includes domain and subdomains.
        - Eg 1 : `Domain=mozilla.org, cookies are available on subdomains like developer.mozilla.org.`
        - Eg 2 : `example.com & subdomain.example.com OR sub1.example.com & sub2.example.com can share cookies`
    - Conclusion :
        - Browser uses the following heuristics to decide what to do with cookies (Sender host = actual URL you visit):
            - Reject the cookie altogether if either the domain or the subdomain in "Domain" don't match the sender host
            - Reject the cookie if the value of "Domain" is included in the Public suffix list (PSL)
            - Accept the cookie if the domain or the subdomain in "Domain" matches the sender host
        - Once the browsers accepts the cookie, and it's about to make a request it says:
            - Send it back the cookie if the request host matches exactly the value I saw in "Domain"
            - Send it back the cookie if the request host is a subdomain matching exactly the value I saw in "Domain"
            - Send it back the cookie if the request host is subdomain like lol.hi.dev included in a "Domain" like hi.dev
            - Don't send it back the cookie if request host is a main domain like hi.dev and "Domain" was lol.hi.dev
    - **Note** : SSO - Two different domains can never share cookie via plain HTTP. It could be done via some IPC Mechanism. Eg. servers with *.google.com co-ordinate with youtube.com servers.Try login into Google's account page. You will have some round-trip from YouTube as well. It happens very fast and we don't realize those intricate redirects.This idea of sharing authentication with multiple websites is called as Single Sign On.
- **Path (Cookie Scoping/Permission)** : The Path attribute indicates a URL path that must exist in the requested URL in order to send the Cookie header. The %x2F ("/") character is considered a directory separator, and subdirectories match as well. A cookie with a given Path attribute cannot be sent to another, unrelated path, even if both path live on the same domain.
    - Default/Unspecified Path : '/'
    - Specified Path : applies to current path and all its children, not parent path. Eg. : If you set Path = "/docs" then
        - Valid Req. Path : /docs, /docs/, /docs/web, /docs/web/http
        - Invalid Req. Path : /, /docsets, /fr/docs
- **Expires** : Indicates the maximum lifetime of the cookie as an HTTP-date timestamp. If unspecified, the cookie becomes a session cookie. A session finishes when the client shuts down, after which the session cookie is removed. Max-Age indicates the number of seconds until the cookie expires. A zero or negative number will expire the cookie immediately. If both Expires and Max-Age are set, Max-Age has precedence.

- **Cookie Security :**
    - **Detection** : Heuristic Algo. - tracking unusual user behaviour like changes in IP, browser, mobile, fingerprints etc.
    - **Attacks & Prevention** :
        - MITM (use SECURE)
        - XSS (Use HTTP-ONLY)
        - CSRF (Use REFERER HTTP header OR Anti-CSRF token OR SameSite)

2. **Tokens (Web Tokens) :** Tokens are cryptographically signed entity. Token use cookies as storage Mech. All data needed for Authe/Autho can be stored in tokens (JWT), hence no load on server/DB calls and in that sense stateless. Could be made stateful if used as sessions (Opaque_tokens/session_cookie-tokens - which is not recommended for authentication). As JWT is most popular we consider here JWT only for tokens. Alternate to JWT Tokens : JWT-Auth0/PASETO/Fernet/Branca etc.
    - **Types of Tokens (Fundamentally only 2)** : Could be stored in Cookies (Session), Localstorage etc.
        - **JWT Tokens (Stateless)** : A JWT has readable content, as you can see for example on https://jwt.io/. Everyone can decode the token and read the information in it.
            - **Pros** :
                - Cookies are hard to share due to various domain policies put up by HTTP and browsers.
                - Cookies have size restrictions.
                - Difficult to make CORS calls with Cookie based authentication.
                - Tokens if stored in Local or Session Storage, then they do not suffer from CSRF attacks.
                - Scalibility as backend does not need to do DB lookup for every API Call.
                - Easy identity exchange in distributed env : There are scenarios where you are doing server-to-server (or microservice-to-microservice) communication in the backend and one service could generate a JWT token to send it to a different service for authorization purposes. Example in Oauth/OIDC (Which is used for authorization not authentication. OIDC - OpenID Connect builds on OAuth 2.0) yourwebsite.com lets you login via facebook.com. Another eg. - Password reset, where you can send a JWT token as a one-time short-lived token to verify the user's email.
            - **Cons** :
                - **Token Revocation** : Biggest drawback of token/jwt is that it comes with expiry and as it is self-contained/sufficient, hence not dependent upon server, so it's difficult to revoke-logout/invalidate/update it from server.
                    - **Workaround/Hack** : One hack is to store "revoked tokens" list/Blacklist in db & check for every call. If the token is part of that list, then block user for the next action. But then now you are making that extra call to the DB to check if the token is revoked and so deceives the purpose of JWT altogether because scalablity is hurted. However, one can revoke all tokens by changing the signing key (Which is again quite not a good solution). So you see, ultimately it needs to stateful combo. i.e. fast db calls.+ tokens
                - **Storage Scalability** : In many complex real-world apps, you may need to store a ton of different information. And storing it in the JWT tokens could exceed the allowed URL length or cookie lengths causing problems. Also, you are now potentially sending a large volume of data on every request. Solution could be to use local or session storage but it comes with its pros/cons.
                - **Security** :
                    - When tokens are stored in cookies :
                        - MITM Attack (Due to non-encryption of JWT) - Use Secure Flag
                        - XSS Attack (Use HTTP-Only Flag)
                        - CSRF (Use REFERER HTTP header OR Anti-CSRF token OR SameSite)
                    - When tokens are stored in Localstorage - Not Recommended as they are accessible by JS, hence XSS attack & you can't disable them like in cookie using http-only flag. Thus recommended to store jwt in session cookies.
                    - Detection of stolen refresh tokens (revoke them by changing the private key) & to use only short-lived access tokens (JWT or Opaque).
        - **Opaque Tokens (kind of stateful)** : An opaque token on the other hand has a proprietary format (random seq. of alphanumeric char. w/o meaning) that is not intended to be read by you. Only the issuer/server knows the format & are stored in persitent storage like db.
            - **Pros** : A single token can easily be revoked on demand.
            - **Cons** : These require a session store/database/cache lookup each time they are used. i.e. server sends back SID to client which is stored at client side in session cookies as SID/opaque_tokens and sent back to server on each call via HTTP Auth. header or cookies. Server lookups in DB (redis cache) each time after verification.
    - **Conclusion** : In many real-world apps, servers have to maintain the user's IP and track APIs for rate-limiting and IP-whitelisting. So you'll need to use a blazing fast DB/Stateful/Session/Redis_Cache anyway. To think somehow your app becomes stateless with JWT is just not realistic. The solution is to not use JWT at all for session purposes. But instead, do the traditional, but battle-tested way more efficiently. In simple words, hybrid approach (Stateful+stateless) by making the DB lookups so blazing fast(Redis DB) that the additional calls won't matter. This is similar to many newer developers learning how to build SPAs with React before server-rendered HTML. Experienced devs would likely feel that server-rendered HTML should probably be your default choice, and building an SPA when needed, but this is not what new developers are typically taught.

- **Token (Authe/Autho) Classification based upon usage of Protcols :**
  - **OIDC : Authentication - ID Tokens (JWT as per OIDC) :** An ID Token, is the user's identity, also usually in JWT format, but doesn't have to be. An ID token must not contain any authorization information, or any audience information — it is merely an identifier for the user.
  - **OAuth 2.0 : Authorization - Access Token (Opaque_tokens) [OAuth2 -> OIDC] | Authentication - Refresh Token (Opaque)**
    - **Access Tokens :** also called user or page access token. Access tokens are used as bearer token. It's a token you put in the Authorization HTTP Req. header. Generally Access token are in JWT Format. If access token are opaque (As in Oauth 2.0), then they can be easily revoked from db. Otherwise if they are JWT it is also good as they are short lived. Access token are only good for single resource, so we use refresh token to gen. for multiple resources.
      - **Bearer Token** - A bearer token means that the bearer (who holds the access token) can access authorized resources without further identification. Access tokens, ID tokens, and self-signed JWTs are all bearer tokens. Using bearer tokens for authentication relies on the security provided by an encrypted protocol, such as HTTPS; if a bearer token is intercepted, it can be used by a bad actor to gain access.If bearer tokens don't provide sufficient security for your use case, consider adding another layer of encryption or using a mutual Transport Layer Security (mTLS) solution such as BeyondCorp Enterprise, which limits access to only authenticated users on a trusted device.
    - **Refresh Tokens** = Access + ID Token (Use to gen. new access, id token once expired). Refresh token function alot like session tokens (Cookies), hence called Auth/Session tokens. (Opaque)
    - **NOTE** : There are multiple approaches to use refresh and access token in combo. will discuss here only widely used and recommended one.
      - Short-lived access token (JWT, Opaque) + long-lived refresh token (Opaque)
      - Short-lived access token (JWT, Opaque) + short-lived refresh token/Rotation/RTR (Opaque)
  - **Others** :
    - Self-signed JSON Web Tokens (JWTs)
    - Federated tokens - Federated tokens are used as an intermediate step by workload identity federation. Federated tokens are returned by the Security Token Service and cannot be used directly. They must be exchanged for an access token using service account impersonation.

3. **Session :** It's time period b/w client & server which keeps user interaction stateful. Session can use either cookies (Traditional approach) or tokens (JWT - Modern Approach - not recommended) for Authentication/Authorization. we could define roughly 2 types of session management :
    - **Server side sessions :** Server sets SID in cookie(Only SID is exposed, rest data stored in db - widely used), App state, Session state.
    - **Client side sessions :** Session info. is stored in cookie (user data are openly exposed), URL querystring, hidden form field, View state
4. **SessionStorage (Key/Value) :** It is tab specific, and scoped to the lifetime of the tab. It may be useful for storing small amounts of session specific information, for example an IndexedDB key. It should be used with caution because it is synchronous and will block the main thread. It is limited to about 5MB and can contain only strings. Because it is tab specific, it is not accessible from web workers or service workers.Data stored in sessionStorage is specific to the protocol of the page. In particular, data stored by a script on a site accessed with HTTP (e.g., http://example.com) is put in a different sessionStorage object from the same site accessed with HTTPS (e.g., https://example.com).
    - Whenever a document is loaded in a particular tab in the browser, a unique page session gets created and assigned to that particular tab. That page session is valid only for that particular tab.
    - A page session lasts as long as the tab or the browser is open, and survives over page reloads and restores.
    - Opening a page in a new tab or window creates a new session with the value of the top-level browsing context, which differs from how session cookies work.
        - **Session Storage (Page/Tab Session) vs session cookies (Browser session)** - while session storage behaves similarly to session cookies, except that session storage is tied to an individual tab/window's lifetime (AKA a page session), not to a whole browser session like session cookies. A page session lasts for as long as the browser is open and survives over page reloads and restores. Opening a page in a new tab or window will cause a new session to be initiated, which differs from how session cookies work. session cookies are deleted when the client/browser shuts down. Web browsers may use session restoring, which makes most session cookies permanent, as if the browser was never closed.
    - Opening multiple tabs/windows with the same URL creates sessionStorage for each tab/window.
    - Duplicating a tab copies the tab's sessionStorage into the new tab.
    - Closing a tab/window ends the session and clears objects in sessionStorage.
5. **Local Storage (Key/Value) :** The only difference is that while data stored in localStorage has no expiration set, data stored in SessionStorage gets cleared when the page session ends. Local storage behaves similarly to persistent cookies. It is limited to about 5MB and can contain only strings. LocalStorage is not accessible from web workers or service workers. LocalStorage data is specific to the protocol of the document. In particular, for a site loaded over HTTP (e.g., http://example.com), localStorage returns a different object than localStorage for the corresponding site loaded over HTTPS (e.g., https://example.com).
6. **Cache (Browser/web - Key/Value - Async) :**
    - **Browser Cache (Cache Storage - Web API) :** kind of client-side cache, which means it’s also a type of site caching. It works in the same way and it’s a cache system that’s built into a browser. CacheStorage is a storage mechanism in browsers for storing and retrieving network requests and response. It stores a pair of Request and Response objects. The Request as the key and Response as the value. Cache Storage is where Service Workers can download and store a complete version of web app. CacheStorage is not a Service Worker API, but it enables SW to cache network responses so that they can provide offline capabilities when the user is disconnected from the network. You can use other storage mechanisms in SW. For example, IndexedDB, a SQL database in the browser, can be used to store network responses. The browser cache can also be used to store information that can be used to track individual users. This technique takes advantage of the fact that the web browser will use resources stored within the cache instead of downloading them from the website when it determines that the cache already has the most up-to-date version of the resource. For example, a website could serve a JavaScript file with code that sets a unique identifier for the user (for example, var userId = 3243242;). After the user's initial visit, every time the user accesses the page, this file will be loaded from the cache instead of downloaded from the server. Thus, its content will never change. The ETag (or Entity Tag) is a string that serves as a cache validation token. This is usually a hash of the file contents. The server can include an ETag in its response, which the browser can then use this in a future request (after the file has expired) to determine if the cache contains a stale copy. If the hash is the same, then the resource hasn’t changed and the server responds with a 304 response code (Not Modified) with an empty body. This lets the browser know it’s still safe to use the cached copy.
    - **Site/HTTP/Page Cache :** system that temporarily stores data such as web pages, images, and similar media content when a web page is loaded for the first time. It can be private, shared, proxy, managed, heuristic cache etc.
        - **HTTP Header for Caching :** ETag, Cache-Control, Expires, Last modified etc.
    - **Server Cache :** Object caching, CDN caching, Opcode caching
7. **Indexed DB (Non-Relational - hybrid/Async) :** The IndexedDB (IDB) is a complete database system available in the browser in which you can store complex related data, the types of which aren't limited to simple values like strings or numbers. You can store videos,

images, and pretty much anything else in an IndexedDB instance. However, this does come at a cost: IndexedDB is much more complex to use than the Web Storage API. Earlier WebSQL(Relational) was used earlier but is now deprecated.

8. **Others :** File system API - based on byte stream concept. Async. Stored in sandboxed section of the user's local file system.

---

3. Session management (usecase : Authentication/Authorization)

Session Management : Could be done via stateful (opaque tokens/SID/Session cookie) or stateless (JWT). Both have its pros and cons. stateful gives revocation feature at cost of DB Calls/Scalability and stateless give Scalability at cost of Revocation. So both are Contradicting each other.

Session Management flow via :

- **HTML Hidden Field** - Will discuss in Authentication Section
- **URL Rewriting** - Will discuss in Authentication Section
- **Session Management API** - built in framework like ASP.NET, J2EE, PHP, Django, Node etc. provide methods/libraries.
- **User Authentication** :
  - **Using http cookies (Storage - Cookie Storage)** :
    - Eg. The contents of a user's shopping cart are usually stored in a database on the server, rather than in a cookie on the client. To keep track of which user is assigned to which shopping cart, the server sends a cookie to the client that contains a unique SID (typically, a long string of random letters and numbers). Because cookies are sent to the server with every request the client makes, that session identifier (SID) will be sent back to the server every time the user visits a new page on the website, which lets the server know which shopping cart to display to the user.
  - **Using Tokens (Storage - Session cookies)** : JWT should not be used as sessions.(See tokens for detail)
    - **If Token Stored in Local Storage** - JWT is stored in local storage and header of the token is sent with every new request in HTTP Authentication request header manually by web application.
    - **Combo. of Access Token + Refresh Token** :
      - Unpopular/unwidely used :
        - Long - lived access token
        - Short - Medium term lived access token used to get a new access token.
        - Short - Medium term access token whose usage extends its expiry.
        - Short - lived access token
      - **Widely Used/Recommended** :
        - **Short-lived access token + long-lived refresh token (Recommended : Priority Low)**
          - When a new refresh token is obtained, the old refresh and access tokens are invalidated on the backend and removed from the frontend. Doing this properly is not straightforward. If the user voluntarily logs out, the access and refresh tokens are revoked and cleared from the frontend
        - **Rotating refresh tokens/RTR/short-lived + short-lived access tokens (Recommended : Priority High)**
          - Refresh token rotation helps a public client to securely rotate refresh tokens after each use. used in critical apps like banking, stock market etc. When a new refresh token is obtained, the old refresh and access tokens are invalidated on the backend and removed from the frontend.

A2 : Authentication & Authorization

Authentication (User Authe.)

Note :- InfoSec : CIA -> Parkerian Hexad (Google it)

Authe. Factors : KOI

- **K**nowledge factors : password, PIN, security que. etc.
- **O**wnership factors : wrist band, ID card, HW/SW token etc.
- **I**nherence factors : fingerprint, retina, DNA, sign., face, voice etc,

Types of Authe. : (will discuss bold one's)

- **Password based** - simple cookie based hashing user/pass etc.
- Passwordless - nothing but a hype term.
- MFA (Multifactor auth)
- Biometric, Captchas
- Certificate Based (Digital Sign.) - uses SSO
- Single Sign-On (SSO) - based on cookies
- **Token Based :** HTTP Basic, API Key, OAuth/OIDC (JWT)

Types of Authe. Protocol : Based on stateful & stateless authe. (will discuss bold ones)

- **HTTP Autho. Header (Basic Schema)** (Other Schema - Bearer, Digest, HMAC, etc.) - require no cookies, SID, login page etc.
- **OpenID/OIDC** (on top of OAuth 2.0) - will discuss in Authorization
- **SAML** (Closely used as alternate to OIDC)
- **JWT** : JWS/JWE/JWK/JWA
- Kerberos
- LDAP
- PAP
- CHAP
- EAP
- P2P
- FIDO2 - UAF, U2F
- SSL/TLS - Uses public key cryptography - digital certificates
- AAA architecture protocols :
  - TACACS, XTACACS and TACACS+
  - RADIUS (LoginRadius)
  - DIAMETER
- Others : Hawk authe, AWS Sign., NTLM, Etc... (See wiki for more)

User Authentication choice/ways :

NOTE on Session Management :

- **Client Side** - Session info. is stored in cookie (user data are openly exposed), URL querystring, hidden form field, View state
- **Server side** - Server sets SID in cookie(Only SID is exposed, rest data stored in db - widely used), App state, Session state.

- **Old/Obslete Methods (Rarely used) :**
  - **IP address** : The server knows the IP address of the computer running the browser and could theoretically link a user's session to this IP address. However, IP addresses are generally not a reliable way to track a session or identify a user.
  - URL query string : A querystring is information added to the end of a page's URL using "?" as the separator between the URL and the querystring and "&" to separate each querystring pair. Eg: http://www.examplesite.com?sid=Hu33Ymd923Js1ULsd45
    - Pros :
      - Lightweight and easy to employ.
      - Useful when client browser cookie function is disabled.
      - Transferable through a copy of the URL.
    - Cons :
      - URL length has limitations: usually 255-character
      - Visible and easy to tampered.
      - If a user visits a page by coming from a page internal to the site the first time, and then visits the same page by coming from an external search engine the second time, the query strings would likely be different. If cookies were used in this situation, the cookies would be the same.
  - **Hidden form fields** : A hidden form field is not visible in the browser, but the value can be sent to the server along with the other form fields. Example: <input type=hidden name=sid value=Hu33Ymd923Js1ULsd45>
    - GET - Similar to URL query strings (add form field to url).
    - POST - form information & hidden fields sent in HTTP request body (Neither url nor cookie)
    - Pros :
      - Not as visible as the querystring.
      - session info is not copied when the user copies the URL (to bookmark the page or send it via email, for example)
    - Cons :

        - docs need to embedd data inside, waste bandwidth. have to embed the result from the previous page on the next page.
        - Everyone can see the embedded data by viewing the original source code.
        - We cannot store all kinds of objects in hidden boxes except text/string values.
        - Hidden boxes travel over the network along with the request and response, indicates more network traffic.
    - **"window.name" DOM property** : All current web browsers can store a fairly large amount of data (2–32 MB) via JavaScript using the DOM property window.name. This data can be used instead of session cookies and is also cross-domain. The technique can be coupled with JSON/JavaScript objects to store complex sets of session variables on the client side.The downside is that every separate window or tab will initially have an empty window.name property when opened. Furthermore, the property can be used for tracking visitors across different websites, making it of concern for Internet privacy. In some respects, this can be more secure than cookies due to the fact that its contents are not automatically sent to the server on every request like cookies are, so it is not vulnerable to network cookie sniffing attacks. However, if special measures are not taken to protect the data, it is vulnerable to other attacks because the data is available across different websites opened in the same window or tab.
    - **HTTP Authentication & Authorization Framework :**
        - **Basic Schema** - Oldest method, only to be used with https, not used widely. No login page required, username/password will be by default prompted to User by browser. It transmits credentials as user ID/password pairs, encoded using base64.
            - The **server** responds to client with `401` (Unauthorized) response status & provides info. on how to authorize with `www-Authenticate` **response header** containing at least one challenge. Eg. **www-Authenticate: <Basic> realm=<access to site>**
            - A **client** that wants to authenticate itself with the server can then do so by including an `Authorization` **request header** with the credentials - Key/value, JWTokens Etc. Eg. **Authorization: <Basic> <Hu33Ymd923Js1ULsd45>**
            - If All goes well, page/response is displayed. Session established.
        - Other Schemas - Bearer, Digest, HOBA, AWS3, Etc.
        - **Pros** : Implementation is simplest bcz it does not require cookies, SID/login pages, rather uses standard fields in HTTP header (Server : www-authenticate & Client : authorization).
        - **Cons :** Basic Schema has no way to logout/revoke user & is unsafe (without https).

- **Traditional Method (used presently) : Stateful session management**
  - **Cookie based authentication (server side)** - SID stored into cookies (1st party session cookies) & sent back to server on each http req. header automatically.
    - **Note** : **See Cookie topic for details (HTTP + Cookies = Stateful)**
    - Lib. : Passport-local strategy (uses express.session internally)
    - State Store : DB, Redis Cache DB, Filesystem etc.
    - Web FW : ASP.NET, django, nodejs etc. support via session management API for session based authentication.
- **Modern Method : Stateless session management (JWT) & Hybrid Approach**
  - **Glossary** : RT - refresh token | RTR - refresh token rotation | OT - Opaque token | AT - access Token | IT - identity token
  - **NOTE on modern web apps (Evolution) :** The key thing to remember is that the data, which software application is processing, is known as The Model or The Application State. Few brave souls might call it Domain Model/Business Logic of App. An App could be desktop or web app. Struggle is to understand reasonable ways of presenting this application state to user (of web front-ends).
    - Desktop MVC -> AM-MVC -> MVP -> MVVM/MVB(Base of react/angular/vue) -> Web MVC(sub-category of distbd app) -> Server Side MVC(Model 2/SSR - ASP.NET,Struts,Django,ROR,CodeIgniter), browser's internal MVC, front-end MVC(CSR) -> Microservices/Backend/Fullstack.
    - JS tech. wise : jQuery -> Backbone(MV*) -> Knockout.js (MVVM) -> Angular 1/Emberjs/Auerlia (MVVM) -> Angular 2+/React/vue/CSR->SSR (Component->class->functional/MVVM/MV*) -> svelte/Compiler/SSG/CSM/pre-rendering -> Monolith/stimulusjs/strudlejs -> Microfrontends+microservices/backend/fullstack
  - **NOTE - WHY Tokens ? :** Web applications have evolved from simple static websites (two-tiered architecture) into complex multi-layered SPA and SSR driven API first systems. CMS systems have grown into Headless content-first systems. Modern web applications are all about decoupled UI and API. Front-end community has changed rapidly in recent times. It started with DOM infused algorithms introduced by jQuery, which was quickly succeeded by MVC based Backbone.js. And, in no time, we found ourselves in the jungle of bidirectional and unidirectional data flow architecture. Most web apps are turning into SPA — Single Page Applications. Add to that is the mix of Server-side rendering, pre-rendering, etc. Core function of cookie was to preserve identity of the user between multiple page requests. With SPA, it is absolutely not required to maintain cookie to identify user. Once the application is loaded, user doesn't need to refresh the page. HTML5 provides us with new Storage API — LocalStorage and SessionStorage. All that a UI needs is a token to make API calls. They may be same domain, sub-domain or on a different domain (CORS). With fetch API, it becomes very simple to make CORS API calls. REST API have evolved into stateless API. These APIs do not need session like the way cookie provides. All it needs is valid auth token and API returns appropriate response.

- **Token Based Authentication (General/Stateless/cookie-less)** :
  - Library : jwtokens (jwt.io), passport-jwt strategy
  - JWT are stateless bcz server doesn't need to maintain state as JWT uses RSA. Token itself is all that is needed to verify a token bearer's/Client authorization. Hence no need to store user info at server side. Authorization is also based on token which we will discuss in OAuth/OIDC.
  - **JWT Token Storage (Client-side) :** Tokens sent to server via HTTP Authorization Request Header for every subsequent request.
    - **Cookies** (Using JWT as sessions - **not recommended**) [discussed in stateful JWT]
    - **SessionStorage, LocalStorage** (mostly LS is used, although both SS/LS are **not recommended** due to security)
    - **RT is recommended over cookie, SS, LS.**
  - **JWT Problem (Token revocation) :** Since JWT is stateless & comes with their own expiry, there is no way to revoke (Logout) the user's session once the server signs a valid token. i.e Server can't control tokens after sending.
    - **Solution :**
      - Maintaing token revocation/blacklist **(recommended here as temp. solution - better approach is JWT+RT/RTR)**
      - Change Private Key/Digital Sign. (**not recommended** as it will invalidate across all devices)
      - using jwt with http cookies (**not recommended** - discussed in stateful JWT)
      - **Recommended** to use JWT with RT to keep user logged in (OT is session cookie) [See - Stateless JWT with RT]
        - Short-lived AT (JWT, OT) + long-lived RT (OT)
        - Short-lived AT (JWT, OT) + short-lived RTR (OT)
  - **Algorithm (JWT - Stateless - General) :**
    - User navigates to web app at https://ui.example.com
    - Initial API call is made. If it returns status as 401 means user doesn’t have token and a login view is shown.
    - User enter creds. Server verifies by hashing password match in DB.
    - After verification, server send back token by adding into HTTP Bearer Authentication response header (could also send tokens via - URL query string/post req body/etc.) with expiry which is stored client side - Session/localStorage. If Client is :
      - **REST-API/curl/postman** : token should be send/stored via the HTTP Auth header makes sense. (**recommended**)
      - **Browser** : token (store : local/session) & send token via HTTP Auth header (**both below approach not recommended**)

        - If token need for **tabs/window** - **sessionstorage**, if req. to persist across **multiple tabs/window** then **localstorage**
        - SPA : token (Store : memory/server side/DB) i.e. stateful JWT approach with cookies.
    - Unlike Cookies, for every subsequent req. token needs to be passed manually via authorization header (alongwith any API public key if any). Server match public_api_key/token again private key/DB. If token has expired new token is attached.
    - For logout/revoke after expiry token is deleted from server. **token revocation problem is solved by maintaing** `revoked token blacklist`**.** user is shown login view again.
- **Stateful JWT (JWT + session cookie)** - Using JWT's as Session **(Not recommended)**
    - **NOTE** : we get statelessness benefit of jwt (scalibility) & token revocation problems is solved either with token revocation list or using cookies. Both comes with its pros/cons. A thrid approach (stateless JWT + RTR/RT) is recommended. However, we'll discuss the cookie approach here for the sake of knowing.
    - **Token Storage (Cookies)** : Use cookies with (Httonly, secure, samesite) flags. Following are the **Drawbacks** :
        - JWT token size should not exceed than that of cookie's allowed size.
        - samesite = strict (Protects from CSRF, but invalidate the whole point of using tokens as it can't be used for cross origin eg. OAuth). Temporary workaround is samesite=lax.
        - samesite = lax (Allows - GET, HEAD, OPTION, TRACE | Disallow - POST) : Could use Authorization Bearer Header though.
    - **Algorithm (Stateful JWT : JWT + session cookie) :**
        - First time, User enter creds, Server verifies it, create a token/SID/session_cookie & maps it to user DB ID and store in redis cache db.
        - Server sends token/SID/session_cookie (http-only) as response to browser.
        - Now for every subsequent req. token gets send to server, which it verifies and allows access if valid. Token will be valid until expiry, however server can revoke/logout on demand (delete the SID/session_cookie from server side, hence invalidating). Also it could update the token revocation blacklist etc.
        - **NOTE** : **This approach is similar to JWT stateless, however only difference is use of cookies to make it stateful. Still not recommended, better use RT approach.**

- **Stateless JWT with RT** : **Recommended** & Widely used
  - 3rd party lib - AWS cognito, Firebase etc.
  - **NOTE** : OT is like session cookie (Httponly, secure, samesite). Remeber there are only two type of tokens : JWT & OT. Refresh token and access token use either or any of them based upon which protocol is used (OAuth/OIDC).
  - **Algorithm** (Used in OAuth2.0 -> OIDC) : mentioned in detail in authorization section.
    - **Short-lived AT (JWT, OT) + long-lived RT (OT)**
    - **Short-lived AT (JWT, OT) + short-lived RT/RTR (OT)**

Authorization (User Autho./Access Control)

NOTE : There is no specific type/protocol for authorization. Some are HTTP Autho. & OAuth 2.0/OIDC. We will discuss here only **OAuth 2.0/OIDC**. To delegate Autho. use - Authorization as a Service (Eg. Cerbos)

Access Control System (Autho.) Factors :

- AAA : Authentication, Authorization, Accountability/Auditing
  - Identity & Access Mgmt. (IAM)
- Other keywords :
  - Access control List, Access control matrix
  - Authorization OSID (Similar to Authentication OSID)

Authorization/Access Control Model :

NOTE : As per OWASP, prefer ABAC & ReBAC over RBAC.

- **RBAC** (Role-Based-Access-Control) : RBAC is a model of access control in which access is granted or denied based upon the roles assigned to a user.A role is nothing but a collection of permissions. Permissions are not directly assigned to an entity; rather, permissions are associated with a role and the entity inherits the permissions of any roles assigned to it. Generally, the relationship between roles and users can be many-to-many, and roles may be hierarchical in nature. RBAC treats authorization as permissions associated with roles and not directly with users attribute. For example, imagine that you work as a department manager in an organization. In this situation, you should have permissions that reflect your role, for example, the ability to approve vacation requests and expense requests, assign tasks, and so on. To grant these permissions, a system manager would first create a role called "Manager" (or similar). Then, they would assign these permissions to this role and would associate you with the "Manager" role. Of course, other users that need the same set of permissions can be associated with that role.
  - **Pros** : Managing authorization privileges becomes easier because sysadmin can deal with users and permissions in bulk instead of having to deal with them one by one. consider RBAC if :
    - You’re in a small- to medium-sized enterprise
    - Your access control policies are broad
    - You have few external users, and your organization roles are clearly defined.
  - **Cons** : RBAC simplicity is also a big disadvantage in systems as they scale. You can’t assign granular permissions like project and team-level administrative access without

expanding your roles table. This makes maintaining RBAC a headache in a SaaS application with lots of users and permission levels.

- **ABAC** (Attribute-Based-Access-Control) : In ABAC "subject requests to perform operations on objects are granted or denied based on assigned attributes of the subject, assigned attributes of the object, environment conditions, and a set of policies that are specified in terms of those attributes and conditions". Attributes are simply characteristics that be represented as name-value pairs and assigned to a subject, object, or the environment. When using ABAC, a computer system defines whether a user has sufficient access privileges to execute an action based on a trait (attribute or claim) associated with that user.
  - Eg. online store that sells alcoholic beverages provided proof of their age : Alcoholic beverage = resource | Online store = resource owner | Age of consumer validated during registration process is claim, that is proof of user's age attribute. Presenting the age claim allows the store to process access requests to buy alcohol. So, in this case, the decision to grant access to the resource is made upon the user attribute.
  - **Pros** : Flexibility - ABAC allows admins to implement granular, policy-based access control, using different combinations of attributes to create conditions of access that are as specific or broad as the situation calls for. In short, choose ABAC if:
    - You're in a large organization with many users
    - You want deep, specific access control capabilities
    - You have time to invest in a model that goes the distance
    - You need to ensure privacy and security compliance
  - **Cons** : Implementation complexity - Admins need to manually define attributes, assign them to every component, and create a central policy engine that determines what attributes are allowed to do, based on various conditions ("if X, then Y").
- **ReBAC** (Relationship Based Access Control) : "Does this user have a sufficient relationship to this object or action such that they can access it?" ReBAC grants access based on the relationships between resources. For instance, allowing only the user who created a post to edit it. This is especially necessary in social network applications, like Twitter or Facebook, where users want to limit access to their data (tweets or posts) to people they choose (friends, family, followers).
  - **Pros** : Supports Multi-Tenancy and Cross-Organizational Requests, Ease of Management. Support fine-grained, complex Boolean logic, Robustness, Speed.
- Others :
  - Graph-Based Access Control (GBAC)
  - Discretionary Access Control (DAC)
  - History-Based Access Control (HBAC)
  - History-of-Presence Based Access Control (HPBAC)
  - Identity-Based Access Control (IBAC)
  - Lattice-Based Access Control (LBAC)
  - Mandatory Access Control (MAC)

- Organization-Based Access Control (OrBAC)
- Rule-Based Access Control (RAC)/Rule-Based Role-Based Access Control (RB-RBAC)
- Responsibility Based Access Control (RespBAC)

User Authorization Choice [OAuth/OAuth 2.0 -> OpenID/OIDC]

NOTE : Library - passport.js OAuth strategy | Identity providers - Auth0, okta, etc.
Keywords :

- Permissions - declaration of action that can be executed on resource.
- Privileges - privileges are assigned permissions to user.
- Scopes - Scopes enable a mechanism to define what an application can do on behalf of the user.
- Attributes - characteristic of users.
- ACL - access control list. kind of matrix of what scope is given to service.
- Groups - collection of user.
- Role - Eg. group 1 has role of manager. so all user in that group are managers.
- Consent - Consent is the act of letting the user participate in deciding what data to share with a third party.
- Claims - consent screen presented to the user contains all the Claim Names with corresponding descriptions for each claim.

- **OAuth/OAuth 2.0 :** Authorization Protocol (Psuedo Authentication)
  Definition : Facilitates the authorization of one site to access and use information related to the user's account on another site. Although OAuth is not an authentication protocol, it can be used as part of one. Eg. Google, FB logins. OAuth 2.0 is used to grant authorization. An example is a to-do application, that lets you log in using your Google account, and can push your to-do items as calendar entries, at your Google Calendar. The part where you authenticate your identity is implemented via OpenID Connect, while the part where you authorize the to-do application to modify your calendar by adding entries, is implemented via OAuth 2.0. It enables you to authorize the Web App A to access your information from Web App B, without sharing your credentials. It was built with only authorization in mind and doesn't include any authentication mechanisms (in other words, it doesn't give the Authorization Server any way of verifying who the user is), but people started using it as authentication protcol hence it was called pseudo authentication and thus OpendID connect was created for authentication.
  Generally, OAuth provides clients a "secure delegated access" to server resources on behalf of a resource owner. It specifies a process for resource owners to authorize third-party access to their server resources without providing credentials. Designed specifically to work with Hypertext Transfer Protocol (HTTP), OAuth essentially allows access tokens to be issued to third-party clients by an authorization server, with the approval of the resource owner. The third party then uses the access token to access the protected resources hosted by the resource server. In particular, OAuth 2.0 provides specific authorization flows for web applications, desktop applications, mobile phones, and smart devices.

- **Tokens :**
  - Part 1
  - Part 2
  - **Access Tokens (OT - Session_Cookies) - Authorization** - Could be used to send API Calls for Resource Access.
    - Access Tokens should be treated as opaque strings by clients. They are only meant for the API. Your client should not attempt to decode them or depend on a particular access_token format. An access_token cannot be used for authentication. It holds no information about the user. It cannot tell us if the user has authenticated and when.
  - **Refresh Token (OT) - Authentication** -> Access + ID Token (Use to gen. new access, id token once expired)
- OAuth2 grant types/flows - To request & get an access token
  - Authorization Code
  - PKCE
  - Client Credentials
  - Device Code
  - Refresh Token

- **OpenID/OIDC & SAML :** Identity Authentication Protocol build on top of OAuth [Used for SSO - IPC]
  NOTE : 2 main protocols for authentication over networks - OIDC (OpenID Connect) and SAML (Security Assertion Markup Language)
  OIDC Definitions : An authentication method for the web (Token based - JWT). OpenID is a way to use a single set of user credentials to access multiple sites. OAuth was originally dreamt up as a solution to something known as “delegated access,” or allowing applications to talk to one another and share information on the user’s behalf. OIDC adds the authentication feature to OAuth’s powerful authorization utility.
  Tokens (OIDC) : ID Token (JWT) - Authentication
  - An id_token cannot be used for API access. Each token contains information on the intended audience (recipient). According to the OpenID Connect specification, the audience (claim aud) of each id_token must be the client_id of the client making the authentication request. If it isn't you shouldn't trust the token. An API, on the other hand, expects a token with the audience set to the API's unique identifier. So unless you are in control of both the client and the API, sending an id_token to an API will not work. Furthermore, the id_token is signed with a secret that is known to the client (since it is issued to a particular client). This means that if an API were to accept such token, it would have no way of knowing if the client has modified the token (to add more scopes) and then signed it again.

- In the example we used earlier, when you authenticate using Google, an id_token is sent from Google to the to-do application, that says who you are. The to-do application can parse the token's contents and use this information, like your name and your profile picture, to customize the user experience.The access_token can be any type of token (not necessarily a JWT) and is meant for the API. Its purpose is to inform the API that the bearer of this token has been authorized to access the API and perform specific actions (as specified by the scope that has been granted). In the example we used earlier, after you authenticate, and provide your consent that the to-do application can have read/write access to your calendar, an access_token is sent from Google to the to-do application. Each time the to-do application wants to access your Google Calendar it will make a request to the Google Calendar API, using this access_token in an HTTP Authorization header.

SAML - Authentication protcol just like OIDC :

- **Federated Identity :** We are searching for a concept called "federated identity." This is the ability to link a user's digital identity across separate security domains. In other words, when two applications are "federated," a user can use one application by authenticating their identity with the other, without needing to create separate usernames/passwords for both. The application responsible for performing this authentication is known as the identity provider (IDP). If you work for enterprise companies, you are likely familiar with IDPs like Okta, Google OAuth, Microsoft Active Directory, or Auth0. Common OSS alternatives include Keycloak and Gluu. When you sign in with your Google account (IDP), Firebase does not create a new account using the traditional username/password authentication pattern. In fact, if you create an account through Google and separately attempt to log in using that very same Gmail address, Firebase will not be able to locate your account. This is because Firebase and Google are federated - to gain access to Firebase, you have to authenticate your identity through Google, not directly with Firebase. To discuss federated identity without mentioning SAML would be disingenuous. Recall that federated identity is the ability to use the same digital identity across multiple domains, meaning we are primarily concerned with the act of authentication, which involves proving who you are.
- **OIDC Vs. SAML :**
  - OIDC is an authentication layer built on top of an authorization protocol, whereas SAML is its own self-contained authentication and authorization protocol.
  - SAML uses XML-based federation while OIDC uses JSON/REST.
  - SAML requires individual actors within the SAML architecture to be pre-configured. By keeping a tight leash on who is part of the system, SAML identity providers can tightly control who accesses data. OIDC work with any identity provider that supports the protocol. This form of authentication is more popular with consumer and native mobile applications, like gaming or productivity apps. Exporting identity management to companies like Google, Amazon, and Microsoft, these app developers can significantly reduce the friction of user sign-up. It is important to note that OIDC is extensible and can be configured to meet the security demands that enterprises require.

- **Identity Providers :** Inhouse (Google, MS etc), SaaS (Auth0, Okta), Others/Adhoc (Custom 3rd party login pages)

Conclusion : Authentication & Authorization

- Authentication : Excluding obsolete methods
  - Cookie based/Session based
  - Token based (JWT)
    - Normal Stateless JWT (Local/SessionStorage)
    - Stateful JWT (JWT with Cookies)
    - Stateless JWT + Refresh Token [Mostly recommended] => Used in OAuth2.0/OIDC
