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.