Let's get started the biggest problem with building authentication system initially is that HTTPS is stateless. so we need to figure out how to make HTTPS stateful.

If we have a simple user/pass passed in as input to the login API, this works but,:

* We need this in each of the input request because https is stateless, so subsequent req doesn’t know prev one was authenticated.
* If any other user gets hold of the user/pass they can bob.

A close-up of a black board

AI-generated content may be incorrect.

This can be fixed by using Sessions in HTTPS.**A blackboard with writing on it

AI-generated content may be incorrect.**

We create a session table, keeping unique sessionId for users. This id is created when user successfully logs in and saved in DB. This sessionId is sent to client as well and for every subsequent request we expect the sessionId in the requestHeader as well. This sessionId can then be verified against the one in DB and thus doing AUTHENTICATION.

We must generate the sessionId in a uniqyue cryptic way for security reason.

#### Session Token Storage:

The default option you will see is cookie based storage. A cookie is just like this little extra data that I can attach to a web page. A cookie has information like

* the domain that is associated with
* the duration like how long is going to last and
* the data that is associated with

The problem with cookie is that we introduced some new attacks here cross site scripting attacks and csrf,csrf stands for cross site request forgery.

So I send an email to the person I seek to attack. I'm gonna give them a URL, in this URL is going to be like bank.com/pay and it has data in it that like makes it so it pays me.

Now in cookie method the Browser is going to immediately include the tokens from cookies to the request,which is a security flaw.

Cookies are very easy to implement but it has a lot of security flaws with it.

However if you want to be secured general right we want these we want our session tokens to not immediately be sent and that can be achieved for example doing it through local storage.

A black board with writing on it

AI-generated content may be incorrect.

#### Architectural Concern:

Scaling as and when users grow and how caching may help.

Also we can make the service as a seprate microservice.

#### JWT:

Moving on to one option and how to solve this it is json web tokens also known as JWT.

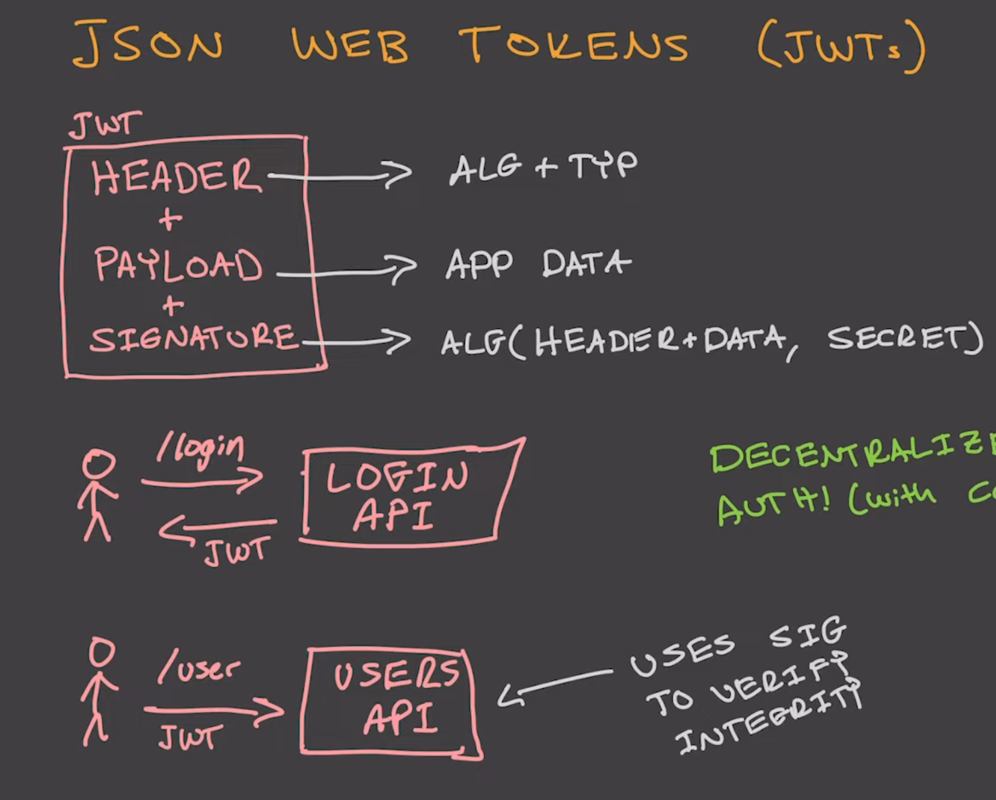
JWTincludes a header, a payload and a signature.

The header is not really relevant to us that much the header just includes the encryption method that we are going to use and then the type of the token type of the token is always jwt.

The payload is the actual application data so like I mentioned we need to somehow keep on the client side what the username is right maybe if there an admin or not.

We also have this signature and the signature is the biggest part of this. if we don't have any signature we have no way to verify the integrity of the data. In order to solve the problem, this signature is going to use the algorithm from above and the algorithm is going to encrypt the header and data. its going to provide some sort of checksumright and this is going to be developed using a secret. this is a secret I want to keep on the server side.

So the login API will generate jwt when it's when it gets a request. this jwt is going to go back to the client and be stored on the client. Now whenever the client is going to make a request to the server, it is going to send the jwt and its going to use the fact that we are using asymmetric encryption.

****

More on JWT Tokens at the end.

**Why Authentication and Authorization matter?**



**How Authentication works?**

To authenticate a person, we can assign a unique phrase to each person, and given the person tells the phrase correctly and their name. We can say that ok, we have identified the person. This is the usual usernames and passwords approach. When the right credentials are given, a system considers the identity valid and grants access. This is known as 1FA or ***Single-factor authentication***(SFA).

SFA is considered fairly insecure. Why? Because users are notoriously bad at keeping their login information secure. Multi-factor authentication (MFA) is a more secure alternative that requires users to prove their identity in more than one way. Some such ways are:

* Single-use PIN numbers / OTP
* Authentication apps run by secure 3rd party (ex. Google/Microsoft Authenticator)
* Biometrics

Once authenticated, the person would keep performing actions freely on the application. And the application is expected to have the person recognized throughout their journey without forgetting them. Ideally, it would be too much to ask the user to provide the password everytime they move to a different page, or they do some activity. So we need a way to keep the user authenticated after they have entered their credentials and they have been authenticated once. This is called **Session Management**.

2 ways to keep the user authenticated:

* **Session-based authentication**: When a user logins to a website on a browser, the server creates a session for that user and assigns a sessionid. This sessionid is stored by the server for reference and is sent back to the user to be stored in a cookie in the browser. Now each time the user would make a request the browser will send the sessionid along with the request. Which will help in authenticating the request. And, thus, preserving authentication while the user is on the site.
* **Token-based authentication**: For this, the server creates a encrypted token which is sent to the user and is saved by the browser only, as HttpOnly Cookies. All required information such as the user information, permissions and expiry of the token are encrypted within the token. The token is sent along for calls to the server. The server simply decrypts the token with the secret key and verifies the user. This token is refreshed at intervals.

The main differences between these two approaches would be that token-based authn is **Stateless**, cause the token need not be stored on the server side. But for session-based authentication, the token are needed to be stored on the server side as well, which makes it **Stateful**. Which brings up complications, when the system is scaled or the number of users grows.

For token-based authentication, we mostly use JWTs (JSON Web Tokens).

**not using jwe**

* JSON Web Encryption (JWE) is a standard for exchanging encrypted data using JSON and Base64. JWE allows you to encrypt a JWT payload so that only the intended recipient can read it. JWE also provides integrity and authentication checks
* JWT-JWE is not supported by all languages and frameworks
* JWT-JWE cannot be invalidated intentionally, without storing the token in a database or cache store, defeating the purpose (stateless session verification) of using JWT in the first place
* workaround would involve shortening expiry time which would increase the load on the database every time one needs to refresh the token
* large payloads in scenario of storing additional details like access-control, role id etc via JWT-JWE would increase request size, which is not ideal for mobile devices with limited bandwidth

**using simple randomly generated hash**

* an arbitrary length string is generated using a cryptographically secure hash function
* seed can be an arbitrary and in demonstration is just the current timestamp
* this string is used as the session token and key for the cache store
* no personal identifying information can be reverse engineered from the session token
* session can be invalidated by removing the key from the cache store
* incoming session token can be verified by checking if the key exists in the cache store along with the expiry time validation
* request payload size is small, which is ideal for mobile devices with limited bandwidth

**redis or memcached for cache store**

* redis or memcached are in-memory databases, that don’t pay the cost of disk access and are very fast relative to disk based databases
* they are simple key-value stores, with a very small set of operations
* they have in-built support for expiry of keys
* complex data structures can be marshalled into strings or byte arrays and stored as values

**How Authorization works?**

Role Based Authorization Control (RBAC) (source: Ajay Shekhawat | Dribble)Role Based Authorization Control (RBAC)

Once the user is authenticated, we would still need to ensure they’re only allowed to access resources that they have permissions to access. Unauthorized access to sensitive data can be a disaster. By the principle of least privileged, companies would usually set up access policies such that by default you have access to what is required for you absolutely. And then in progression to that you have additional access. Common ways to segment access are:

* **Role-based Access Control (RBAC)** : Users are assigned to a certain group/role that comes with set permissions. Examples: admin, member, owner.
* **Policy-based Access Control (PBAC)** : Dynamically determines access privileges during authorization based on policies and rules. Policies are based on user roles, job functions, and organizational requirements.
* **Attribute-based Access Control (ABAC)** : Users are permitted access according to attributes like title, certification, training, and/or environmental factors like location.
* **Access Control Lists (ACLs)** : Each user or entity has individual permissions that can be turned on or off, similar to installing a new app on your phone and deciding which permissions to grant (location services, contacts, etc.)

A screen for Access Control Lists (source: Povio)A screenshot of a computer

AI-generated content may be incorrect.

A screen for Access Control Lists

ACL is frequently used at granular level than either ABAC or RBAC - for example, to grant individual users access to a certain file. ABAC and RBAC are generally instituted as company-wide policies.

**Authentication System Design**

Authentication and Authorization System Design (source: InterviewPen. Modified)A diagram of a system

AI-generated content may be incorrect.

Authentication and Authorization System Design

**functional Requirements**

Lets first start with defining the ***Functional requirements*** of the system:

* **Registration** : Allow users to register by providing necessary information.
* **Login** : Authenticate users based on their credentials.
* **Session Management** : Efficiently manage user sessions to ensure security.
* **Password Recovery** : Provide a secure process for users to recover their passwords.
* **Access Control** : Define roles and permissions for different user types.
* **Audit Trail** : Maintain detailed logs of authentication events for auditing.
* **Performance** : Ensure low latency and quick response times.
* upon successful login, a session token should be generated and passed on to the client
* session token should have an expiry time configurable
* session token should have a grace period configurable, within which the user can refresh the token
* session verification should yield relevant details like user id (typically), role id etc
* state management of the mechanism is left to implementation

**constraints**

* session has to be verified for validity and expiry on every request
* a typical system also has role and a set of permissions associated with that role or user, which is used to authorize the user to perform certain actions. retrieval of this permission occurs on every request and has to be quick.
* session token should be opaque to the client and should not contain any information that can be used to identify the user
* session token should be cryptographically secure and resistant to replay attacks, tampering, brute force attacks and timing attacks
* forceful logout of a user may or may not be possible

A few ***Non-functional requirements*** that we are not going to consider for the scope of this article are:

* **Multi-Factor Authentication (MFA)** : Implement a robust MFA system.
* **Security** : Prioritize data security through encryption, secure storage, and secure communication.
* **Scalability** : Design the system to handle a growing number of users and transactions.
* **Reliability** : Minimize system downtime and ensure high availability.
* **Usability** : Develop an intuitive user interface for a seamless experience.

**Process flow**

1. user identifies themselves with a username and password, oauth token or any other mechanism
2. verify the credentials and generates a session token with an expiry time
3. create payload with at least user id and expiry time, add any other information as needed
4. marshal the payload into a string or byte array as needed
5. store the payload in a cache store with the session token as key
6. set the expiry time plus grace period as the expiry time for the cache store, if supported
7. return the session token to the client
8. on every request, client sends the session token through either header, cookie or in request body
9. if the cache store removes the session token, the user is logged out automatically. send appropriate response to client
10. otherwise, retrieve the payload from the cache store and verify the expiry time
11. if ***current\_time*** is greater than expiry time plus grace period, user is logged out. send appropriate response to the client
12. if ***current\_time*** is greater than expiry time, but less than expiry time plus grace period, — re-identify user, — upon failure: logout, — upon success: regenerate session token and renew expiry
13. if ***current\_time*** is less than expiry time, session is valid. continue with the request

**Capacity Estimation**

**Traffic Estimation**

First lets start with ***Traffic Estimation***. Assuming an average traffic of 100,000 per month.We are estimating a 100k user traffic per month. Which translates to 0.04 request per second. We would need to respond to each request within 500ms 90% of the time, ie we require a p90 latency of 500ms.

assumed\_traffic\_per\_month = 100000 #requests

assumed\_traffic\_per\_day = assumed\_traffic\_per\_month / 30

~= 3350 (assuming on higher end; 3333.33 to be precise)

estimated\_time\_per\_request = 500 #ms; P90 of 500ms

traffic\_per\_second = (assumed\_traffic\_per\_month) / (30\*24\*60\*60)

= 0.04

**Service Level Objective (SLO)** : 500ms (maximal acceptable latency, immaterial of the load on the system) The average capacity 1 instance can take, based on our calculations is approximately 35ms to serve a request, assuming there are no heavy processing happening for the particular request.

Lets generate two more *derived metrics* using the above metrics.

* ***Capacity*** : Acceptable backlog per instance : Maximum number of requests(load) that can be accepted by an instance, without compromising SLO.
* ***Demand*** : Backlog per instance : Total number of requests (load) that flows into a unit/instance based on current traffic.

Thus,

SLO = 500ms

approx\_response\_time\_for\_one\_request = 35 #ms

capacity = SLO/approx\_response\_time\_for\_one\_request

= 500 / 35

~= 20

load\_on\_one\_instance = 0.04

instances\_available = 1

demand = traffic\_per\_second / instances\_available

= 0.04

With the demand and capacity available, lets calculate total number of instances required.

total\_units\_required = demand / capacity

= 0.04 / 20

= 0.002

~= 1

Thus, we would be easily be able to handle 100k requests per month, with 0.04 requestsper second, with 1 instance. Where each unit can handle 20 requests per second without compromising SLO.

**Storage Estimation**

We would ideally need to store the user details for each user for authentication and authorization access. Assuming, 5kb /user

monthly\_new\_users = 500

monthly\_additional\_storage = 500 \* 5kb

= 2500kb

~= 2GB

So every month, assuming we will onboard 500 new users, we will require 2GB more storage. Incase we would like to keep authentication logs. Each authentication request is expected to take 2kb to store.

auth\_request\_size = 2kb #assumption

monthly\_storage = monthly\_visitors \* auth\_request\_size

= 100,000 \* 2KB

~= 200MB

Thus, each month we would require an additional of 200MB, assuming a monthly traffic of 100k.

**Database Design**

Now that we have the capacity estimation done. Lets create the schemas of the database required to support the functional requirements.A black and white diagram with text

AI-generated content may be incorrect.

Authentication and Authotization Database Schema

Authentication and Authotization Database Schema

Lets quickly go over the tables. We are using 6 tables.

1. Users - To store all the user information
2. Credentials - To store the access/refresh credentials once the user has been authorized.
3. Passwords - To store the user encrypted user passwords.
4. PasswordRequests - To store the password change requests that comes for a particular user.
5. Sessions - To store when the user had an active session and when was their last activity.
6. ActivityApproval - To store approval requests for a activity performed by a particular user, that would be verified by admin.

**High-Level Design for the Authentication System**

Authentication and Authotization HLDA diagram of a server

AI-generated content may be incorrect.

Authentication and Authotization HLD

System Endpoints

|  |  |
| --- | --- |
| **Endpoint** | **Description** |
| ***/login*** | Authenticate user credentials. |
| ***/logout*** | End user session and revoke authentication tokens. |
| ***/register*** | Create a new user. |
| ***/update/:userId*** | Update user information. |
| ***/delete/:userId*** | Delete a user account. |
| ***/grant/:userId/:permission*** | Grant specific permissions to a user. |
| ***/revoke/:userId/:permission*** | Revoke permissions from a user. |
| ***/check/:userId/:resource*** | Check user’s access to a specific resource. |
| ***/create/:userId*** | Create a new user session. |
| ***/expire/:sessionId*** | Expire a user session. |
| ***/validate/:sessionId*** | Validate an active user session. |

**Requirements Fulfilment**

Now, with all the things in place lets see how we can complete all the requirements.

**Registration**

* **Requirement** - When a new user visits our application. We need to store user details so we can authorize/identify the user next time they visit.
* **Fulfilled** - When a new user visits the application, and enters user details along with their email and password. That will get captured in the database. The user details will be stored in the User table. And the password will be stored in the credentials table in an encrypted form.

**Login**

* **Requirement** - When an existing user visits our application. We need to identify the user so we can authorize/identify their actions and show them data that belongs to them.
* **Fulfilled** - When an existing user visits the application, and enters their details, email and password. We hash the password and match the hash against the hash stored for the user in the Credentials Table. If it matches then we have been able to identify the user successfully. Else they need to enter the correct password that they entered while registering. This process is called ***Authentication***.

**Session Management**

* **Requirement** - When a user has authenticated themselves, by entering their user and password. We need to make sure that they stay logged in as they perform their future actions / try to see sensitive data, without having them re-entering their password again and again.
* **Fulfilled** - When the user authenticates successfully. The Authentication server will share 2 tokens with the client. A **access\_token** and a **refresh\_token**. The access\_token can have encrypted data with it and it has a short expiry time cause of security reasons. Once the client has the access\_token, it sends the access\_token back along with every request which helps in authenticating the request. When the access\_token expires, the client has to ask for a new access\_token using the refresh\_token provided. This helps in maintaining a session.

**Password Recovery**

* **Requirement** - When a user forgets their password, they would need to be able to reset their password securely.
* **Fulfilled** - When the user forgets their password, they can submit their email address in the forgot password page, and we will generate a one time code and send the link to their email. When the user clicks on this link with the code and email address. We will securely be able to identify that the password recovery request is authentic. And provide the user to set their new password. And thus being able to recover the password.

**Access Control**

* **Requirement** - When a user performs a certain action, we need to make sure that the user is authenticated to perform that action and only then allow the action to happen.
* **Fulfilled** - We will maintain a list of actions, lets say 1-12, for simplicity. For each user we will maintain the authenticated actions for that user. So now lets say the user tries to perform an action #id 4. We will check if the user has the permissions to perform action 4. If yes, we will go ahead and complete the request. Else we show that the request isnt successful cause of lack of permissions.

**Audit Trail**

* **Requirement** - Incase of a security incident, we should be able to have enough logs to look through and have a plausible reason as to what might have happened / or any leads as to what might have been a possible cause of it.
* **Fulfilled** - For such scenarios, we can keep logs for every authentication action that happens on the server. (a). For each login request, we will keep a log as to when the authentication happened, from where, ips and other relevant details. (b). For each password recovery request, we will keep a log as to when it was initiated, from where, ips, whether the request was complete and other relevant details. (c). Additionally, we will keep logs around each time a user is authorized / unauthorized for an action and by who. These logs in all, should be able to indicate what might have happened incase trying to understand some scenario.

**Performance**

* **Requirement** - Requirement for performance as discussed in capacity estimation section, is 0.04 requests/second and 100k requests per month.
* **Fulfilled** - We have already handled the requirement with enough servers in the capacity estimation section.

Implementation in Type Script:

import {createHash} from "crypto";  
import {isAfter} from "date-fns";  
  
class Auth {  
 cache: Cache;  
 db: Database;  
 sessionDuration: 100\_000; // fetch from environment  
 sessionRefreshDuration: 10\_000; // fetch from environment  
  
 async createSession(userId: number): Promise<Payload> {  
 const issuedAt = new Date();  
  
 const expiresAt = new Date(issuedAt);  
  
 expiresAt.setMilliseconds(  
 issuedAt.getMilliseconds() + this.session\_duration,  
 );  
  
 const sessionToken = hash(issuedAt.toISOString(), 15);  
  
 const payload: Payload = {  
 session: sessionToken,  
 userId: userId,  
 issuedAt: issuedAt,  
 expiresAt: expiresAt,  
 };  
  
 const created = await this.cache.set(sessionToken, payload, this.session\_duration / 100);  
  
 if (!created) {  
 throw new Error("unable to create session");  
 }  
  
 return payload;  
 }  
  
 async verifySession(session: string): Promise<Payload> {  
 const payload = await this.cache.get<Payload>(sessionToken);  
  
 if (payload === null) {  
 return new Error("session not found");  
 }  
  
 const now = new Date();  
  
 const {userId, expiresAt} = payload;  
  
 if (!isAfter(now, expiresAt)) {  
 return payload;  
 }  
  
 const refreshExpiry = new Date(expiresAt);  
  
 refreshExpiry.setMilliseconds(  
 refreshExpiry.getMilliseconds() +  
 this.sessionRefreshDuration,  
 );  
  
 if (isAfter(now, refreshExpiry)) {  
 return new Error("session beyond refresh period");  
 }  
  
 const validCredentials = await checkUserCredentials(this.db, userId);  
  
 if (!validCredentials) {  
 throw new Error("user not found");  
 }  
  
 return this.createSession(userId);  
 }  
}  
  
function hash(  
 text: string,  
 length: number,  
): string {  
 const hasher = createHash("sha256").update(text);  
  
 const hash = hasher.digest("hex");  
  
 const sliced = hash.slice(0, length);  
  
 return sliced;  
}  
  
async function checkUserCredentials(  
 db: Database,   
 userId: number,  
): Promise<boolean> {  
 return true;  
}  
  
/\*\*  
 \* inteface over redis or memcached  
 \*/  
class Cache {  
 async get<T = any>(key: string): Promise<T | null> {  
 return null;  
 }  
  
 async set<T = any>(  
 key: string,  
 paylod: T,  
 expiryInMs = 0,  
 ): Promise<bool> {  
 return true;  
 }  
}  
  
/\*\*  
 \* interface over postgresql, mysql or sql  
 \*/  
class Database {}  
  
type Payload = {  
 userId: number;  
 issuedAt: Date;  
 expiresAt: Date;  
 session: string;  
};

JWT:

**JSON Web Token or JWT**, as it is more commonly called, is an open Internet standard (RFC 7519) for securely transmitting trusted information between parties in a compact way. The tokens contain claims that are encoded as a JSON object and are digitally signed using a private secret or a public key/private key pair. They are self-contained and verifiable as they are digitally signed. JWTs can be signed and/or encrypted. The signed tokens verify the integrity of the claims contained in the token, while the encrypted ones hide the claims from other parties.

JWT's can also be used for the exchange of information though they more commonly used for authorization as they offer a lot of advantages over session management using in-memory random tokens. The biggest of them being the enabling the delegation of authentication logic to a third-party server like **AuthO** etc.

A JWT token is divided into 3 parts namely header, payload, and signature in the format of

[Header].[Payload].[Signature]

* **Header** − The Header of a JWT token contains the list cryptographic operations that are applied to the JWT. This can be the signing technique, metadata information about the content-type and so on. The header is presented as a JSON object which is encoded to a base64URL. An example of a valid JWT header would be
* { "alg": "HS256", "typ": "JWT" }

Here, "**alg**" gives us information about the type of algorithm used and "typ gives us the type of the information.

* **Payload** − The payload part of JWT contains the actual data to be transferred using the token. This part is also known as the "claims" part of the JWT token. The claims can be of three types registered, public and private.
* The registered claims are the ones which are recommended but not mandatory claims such as iss(issuer), sub(subject), aud(audience) and others.
* Public claims are those that are defined by those using the JWTs.
* Private claims or custom claims are user-defined claims created for the purpose of sharing the information between the concerned parties.

Example of a payload object could be.

{ "sub": "12345", "name": "Johnny Hill", "admin": false }

The payload object, like the header object is base64Url encoded as well and this string forms the second part of the JWT.

* **Signature**− The signature part of the JWT is used for the verification that the message wasnt changed along the way. If the tokens are signed with private key, it also verifies that the sender is who it says it is. It is created using the encoded header, encoded payload, a secret and the algorithm specified in the header. An example of a signature would be.

HMACSHA256( base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)

If we put the header, payload and signature we get a token as given below.

eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6I

kpvaG4gRG9lIiwiYWRtaW4iOmZhbHNlfQ.gWDlJdpCTIHVYKkJSfAVNUn0ZkAjMxskDDm-5Fhe

WJ7xXgW8k5CllcGk4C9qPrfa1GdqfBrbX\_1x1E39JY8BYLobAfAg1fs\_Ky8Z7U1oCl6HL63yJq\_

wVNBHp49hWzg3-ERxkqiuTv0tIuDOasIdZ5FtBdtIP5LM9Oc1tsuMXQXCGR8GqGf1Hl2qv8MCyn

NZJuVdJKO\_L3WGBJouaTpK1u2SEleVFGI2HFvrX\_jS2ySzDxoO9KjbydK0LNv\_zOI7kWv-gAmA

j-v0mHdJrLbxD7LcZJEGRScCSyITzo6Z59\_jG\_97oNLFgBKJbh12nvvPibHpUYWmZuHkoGvuy5RLUA

Now, this token can be used in the Authorization header using the Bearer schema as **Authorization − Bearer** <token>

If you look closely, you will notice two period (.) symbols in the JWT. These period symbols break up the JWT into three segments — **Header, Payload and Signature**.

The use of JWT token for authorization is the most common of its applications. The token is usually generated in the server and sent to the client where it is stored in the session storage or local storage. To access a protected resource the client would send the JWT in the header as given above.

# JWT Authentication Flow

Below diagram shows the flow of JWT authentication. As you can see in the diagram below, nothing is being stored in the server end.

A diagram of a server

AI-generated content may be incorrect.

There are 2 main types of token to work with:

* Access Token
* Referesh Token

rest API will issue the client application in access token and a refresh token.

the access token is given a short time before it expires for example 5 to 15 minutes and the refresh token has given a longer duration for it expires possibly several hours a day or even days.

no security measures are perfect and we do want to consider the risks of cross site scripting and cross site request forgery

It is recommended for front end client applications to only store access tokens in memory, so they will be automatically lost when the app is closed. They should not be stored in local storage or in a cookie. Essentially, if you can store it somewhere with JavaScript, a hacker can also retrieve it with JavaScript. Just keep access tokens in memory. Which you might also refer to as the current application state.

Our API will issue refresh tokens in an HTTP only cookie. This type of cookie is not accessible with JavaScript.

Refresh tokens do need to have an expiration which will then require users to login again. Refresh tokens should not have the ability to issue new refresh tokens. Because that essentially grants indefinite access if for refresh token falls into the wrong hands.

So the overall access token process involves issuing an access token during user authorization. The users application can then access our REST APIs protected routes with the access token until it expires. Our api will verify the access token with middleware every time the access token is used to make a request when the access token does expire the users application will need to send their refresh token to our apis refresh endpoint to get a new access token. of course our refresh endpoint will verify the token and cross reference the refresh token in our database too. Storing a reference to the refresh token in the database will allow refresh tokens to be terminated early if the user decides to logout. And again refresh tokens need to be allowed to expire, so indefinite access, Cannot be gained.

A screenshot of a phone

AI-generated content may be incorrect.

Access token:

* grants access to resources
* usually short lived (minutes)

Refresh token:

* used to generate a new access token
* usually long lived (hours, days, months)

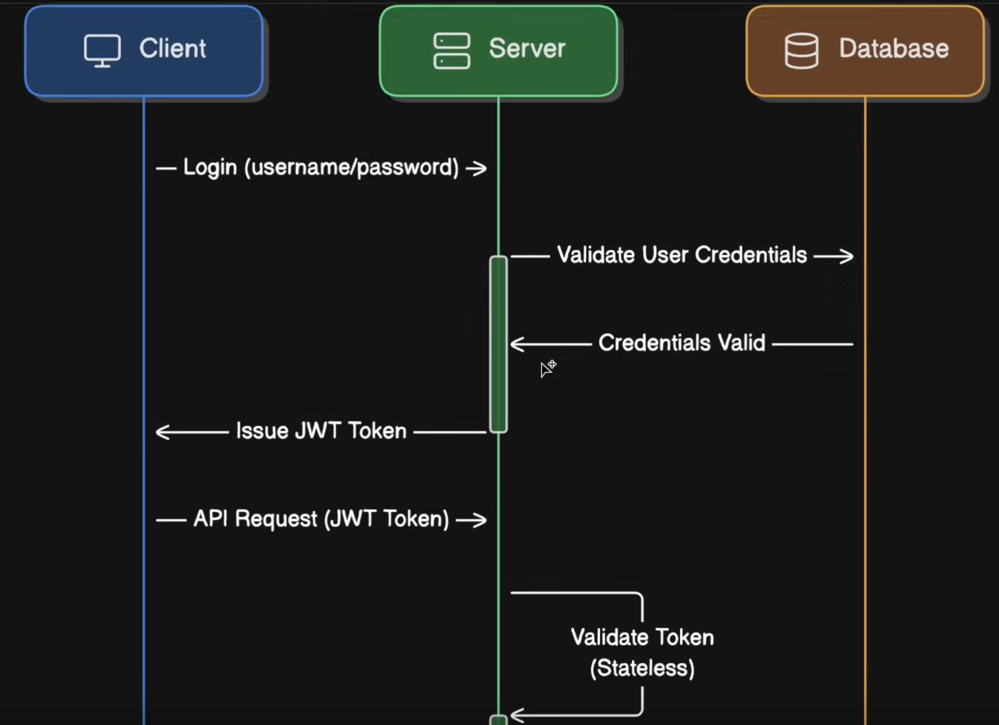
access tokens are mostly for allowing user to access the resources. It is added with every request.

Access tokens are stateless since they aren’t saved on serverside anywhere. They are generated in the fly added to user’s response and reside on client-side in-memory.

Hence, once published they cant be revoked because after sending them in response, server has no knowledge about it.

But now, since access tokens have short expiry date, lets say user was idle for 15min and expire time was 10min. So when a new req will be sent to server with access token in req header, the response would fail saying 401 i.e. access is forbidden.

Now user needs to request a new token, this is achieved using Refresh Token. This token is stateful, i.e. is stored in DB or server files. When a req with expired access token is received, refresh token is used to generate a new access token and send that as part of response.

Now if some bad actor gets hold of refreshtoken, they can forever access the resources since new access tokens can be now generated.thats why, refresh tokens also have expiry time but a little longer. After that, a new refresh token is needed to create. 

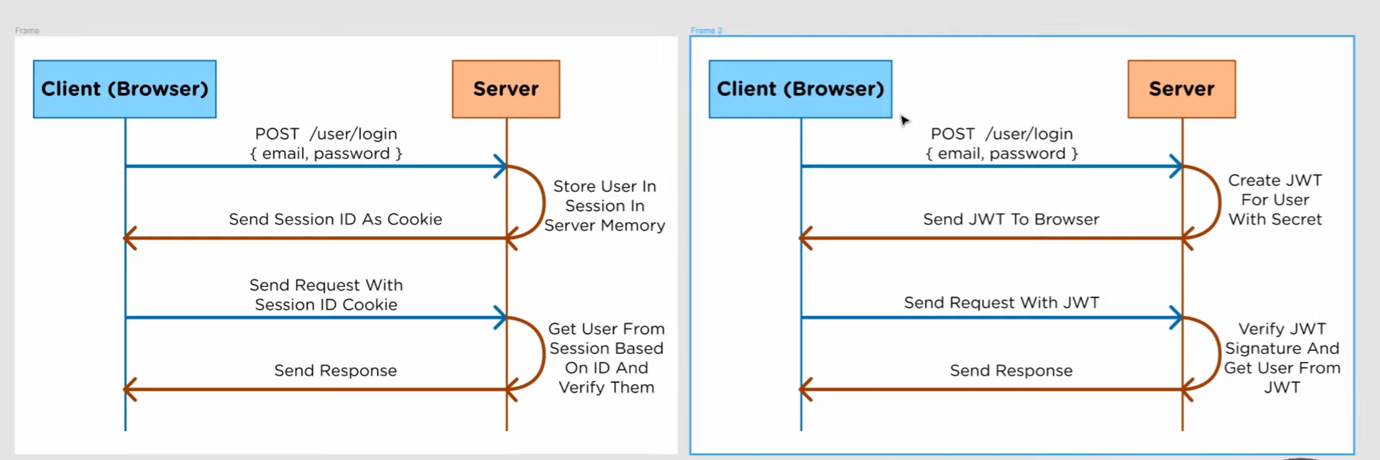
Benefit is here that, once cred are verified (1 DB call), every other request can be validated within server only without having to go to DB everytime to check if creds are valid. JWT verifies encrypted access token from request using its algo and tells if the user making the req is authenticated or not.

This is why access-tokens are made stateless, do avoid DB call.

Now this differs from Sessions, where a unique session Id was generated post login and that ID was sent in response as Cookie and saved in DB as well. Here the problem was that for each call when authentication was needed, server has to go to DB to check if the sessionID from request is matching the sessionId from DB. This was Stateful Approach.

A diagram of a program

AI-generated content may be incorrect.



One more view of Sessions vs JWT above. No storing for JWT since it uses encryption and decryption of tokens to identify the user’s authenticity.

The expiration features enhance security, but they also introduce challenges for long-lasting user sessions. Let’s consider a scenario where users actively engage with the application, and their token expires. Asking the user to log in again can negatively affect the user experience.

## **The Role of Refresh Tokens**

Refresh tokens offer an elegant solution to the challenges posed by token expiration. These long-lived tokens are used to obtain a new access token when the current one expires. Now, when users log in, the server issues both a short-lived JWT (the access token) and a long-lived refresh token. Both tokens are sent to the clients when a user is granted access.

When users attempt to access a resource, they include the JWT access token in each request. Once the JWT expires, the client uses the refresh token to request both a new JWT and a new refresh token. This process is known as refresh token rotation.

## Access and Refresh Token

The primary purpose of an access token is to grant secure access to protected resources on behalf of a user. These tokens are usually short-lived and may have an expiration date embedded in them. On the other hand, refresh tokens are typically long-lived and allow clients to request new access tokens.

The fundamental distinction between access and refresh tokens lies in the ease of independently validating access tokens.

An access token, especially one with a signature like a signed JWT, may be validated by a resource server, eliminating the need for interaction with the authorization server. **The authorization server issues tokens to client applications on behalf of a resource owner to authenticate subsequent API calls to the resource server. The resource server, on the other hand, is responsible for hosting and providing access to protected resources based on the validity of the access token.**

Refresh tokens need to communicate with the authorization server for validation. This separation of validation from authorization server queries allows for improved latency and simplified access patterns.