

Authentication and Authorization

1. What is Authentication?

Authentication means verifying who you are.

It checks whether you are the same person you claim to be.

Examples:

- Login with username and password
- OTP sent to your phone
- Fingerprint or Face ID

Real-life example:

- ATM card + PIN verifies that the bank account is yours.
- Authentication is the process of confirming the identity of a user or system.

Authentication vs Authorization:

- Authentication: Who are you?
- Authorization: What are you allowed to do?

Authenticate : Login (username & passwords)

Flow :

user comes to our platform

user go to register himself/herself (Sign In)

User go to Login. (Authentication)

Now you can access application server, based on your policies/rules. (Authorization)

Requirement

1. User Registration : user can register on our platform (basic details --> name, mobilen. etc)
2. User Login : UserName & password (Authenticat
3. 1 FA va MFA (Multi factor Authentication)
Secure applications --> banking/Gmail etc.
4. Password Recovery : (Forgot password)
Notification --> mobile / email (Already registered
5. Session Management : Stateless vs Stateful A

6. Access control (Authorization)

Leetcode clone --> premium feature

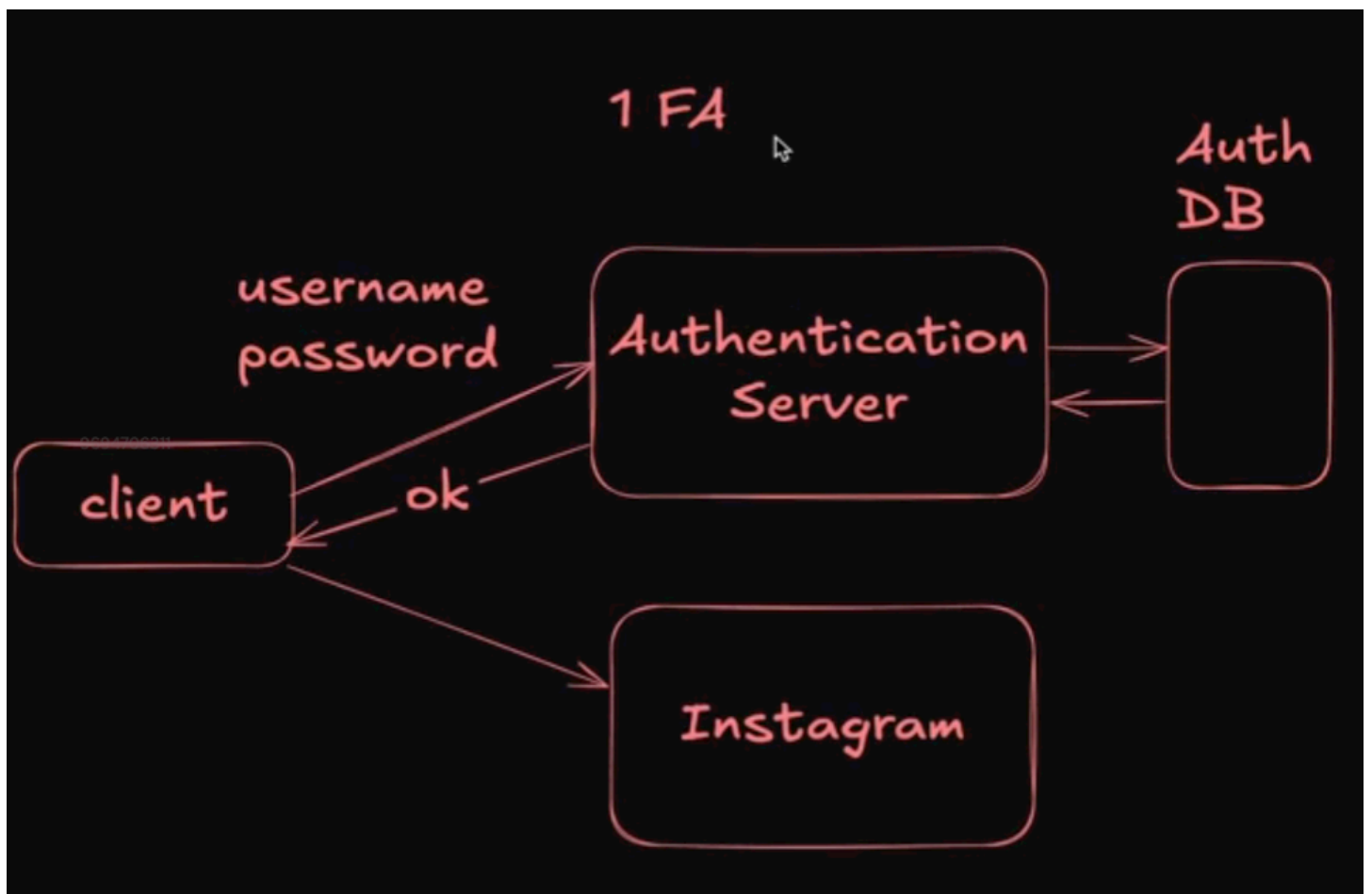
`/solve/question/{{questionId}}`

Response : Question

User level --> key : isPremiumUser

API level --> key

2.1FA Authentication



→ You prove who you are using only ONE thing.

→ 1FA is an authentication method where the user is verified using only one factor, usually a password.

System checks just one proof to let you in.

1FA Example:

Username + Password

- You enter username
- You enter password
- That's it Login done

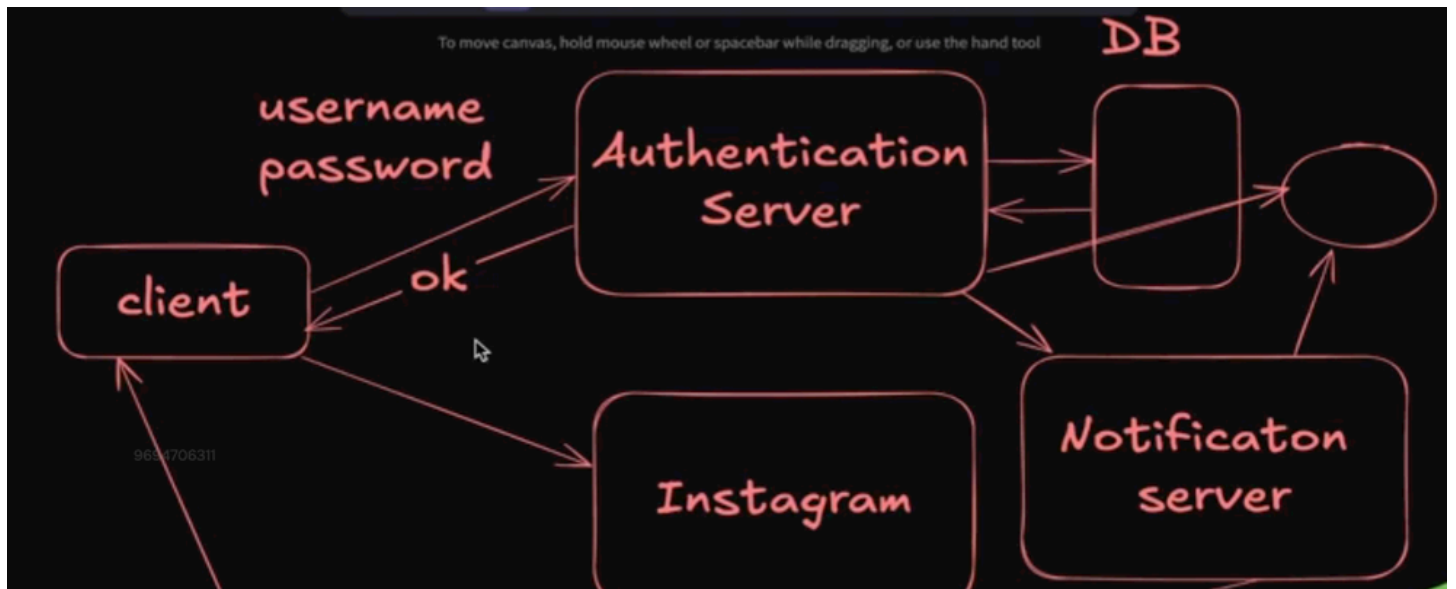
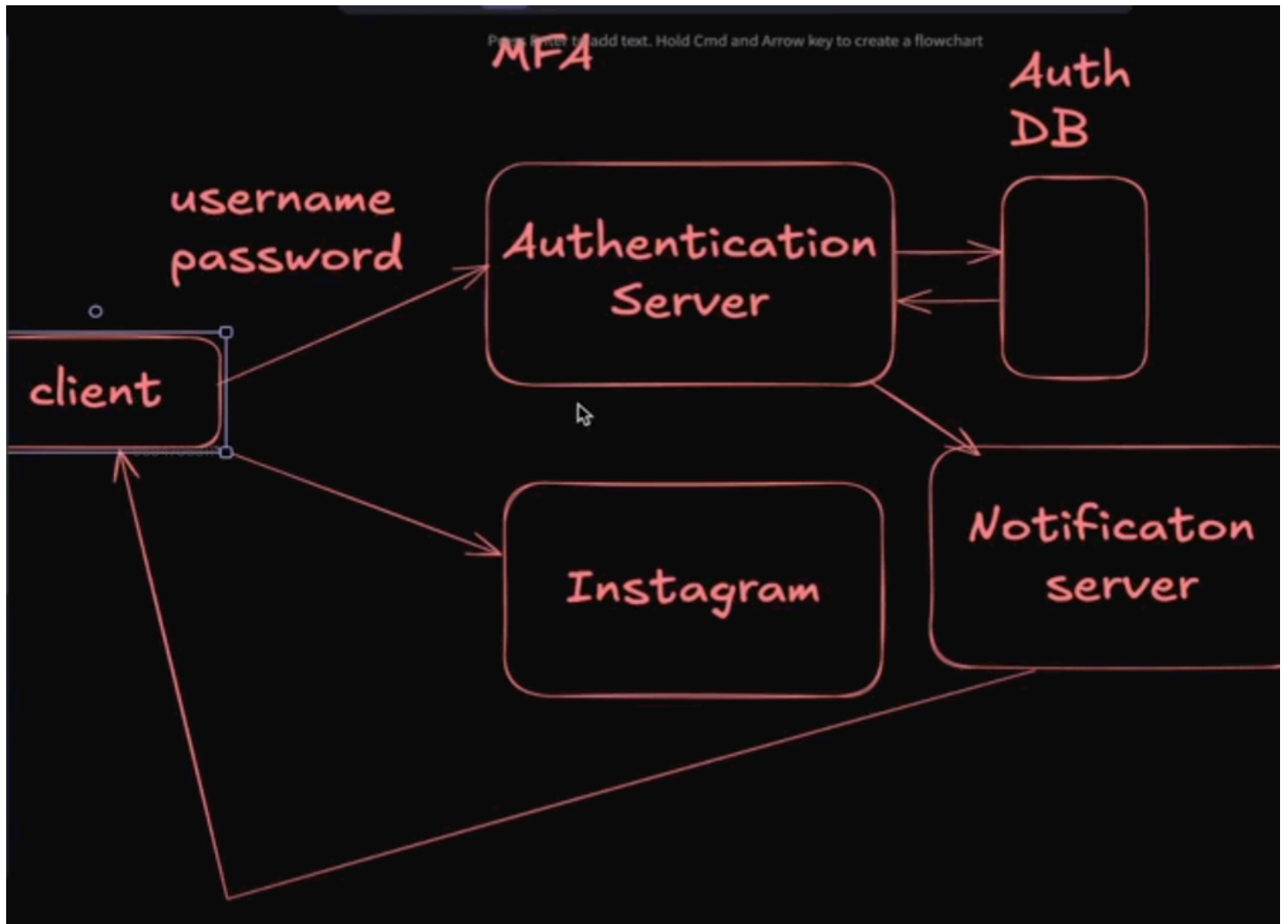
This is 1FA because:

- Only one factor is used → password

More real-life examples:

- Email login with only password
- ATM using only PIN
- Phone lock using only pattern

3.MFA Authentication



You prove who you are using MORE THAN ONE thing.

System says:

“Password alone is not enough. Show one more proof.”

MFA Example (most common):

- 1.Password (something you know)
- 2.OTP on phone (something you have)

Login done

This is MFA because 2 factors are used.

Real-life examples:

- Gmail login
 - Password
 - OTP / phone notification
- Bank app
 - MPIN
 - Fingerprint / OTP
- Office laptop
 - Password
 - Smart card / OTP

Types of factors

- Something you know → password, PIN
- Something you have → phone, OTP, card
- Something you are → fingerprint, face

MFA = any 2 or more of these.

MFA is an authentication method where a user is verified using two or more different factors for extra security.

4.Session-Based Authentication

Server remembers you after login using a session.

What is a Session?

A session is a temporary memory created on the server for a logged-in user.

Think like this

Server says:

“Okay, Vijay is logged in. I’ll remember him for some time.”

Session-Based Authentication

After you login once,
you don’t need to send username & password again and again.

Instead → session id is used.

How it works (Step by Step)

1.Login Request

You send:

username + password

2.Server Verifies

- Server checks details
- If correct

3.Session Created

Server does:

- Creates a session
- Generates a Session ID (random unique string)
- Stores session in server memory / database

Example:

Session ID: abcd1234

User: Vijay

4.Session ID sent to Browser

Server sends:

- Session ID inside a cookie

Set-Cookie: sessionId=abcd1234

Browser saves it

5.Next Requests

Now every request automatically sends:

Cookie: sessionId=abcd1234

6.Server Checks Session

Server:

- Reads sessionId
- Finds user data

- Allows access (no login again)

Logout (Important)

When you logout:

- Server destroys the session
- Session ID becomes invalid

Real-Life Example

Hotel reception:

- You check in (login)
- They give you a room key (session id)
- You show key to enter room
- Key expires after checkout (logout)

Where Sessions are Stored?

- Server RAM
- Database
- Redis (very common)

Session Expiry

Session ends when:

- Time expires (30 min idle)
- User logs out
- Server restarts

Advantages

- Very secure
- Easy to implement
- Server fully controls session

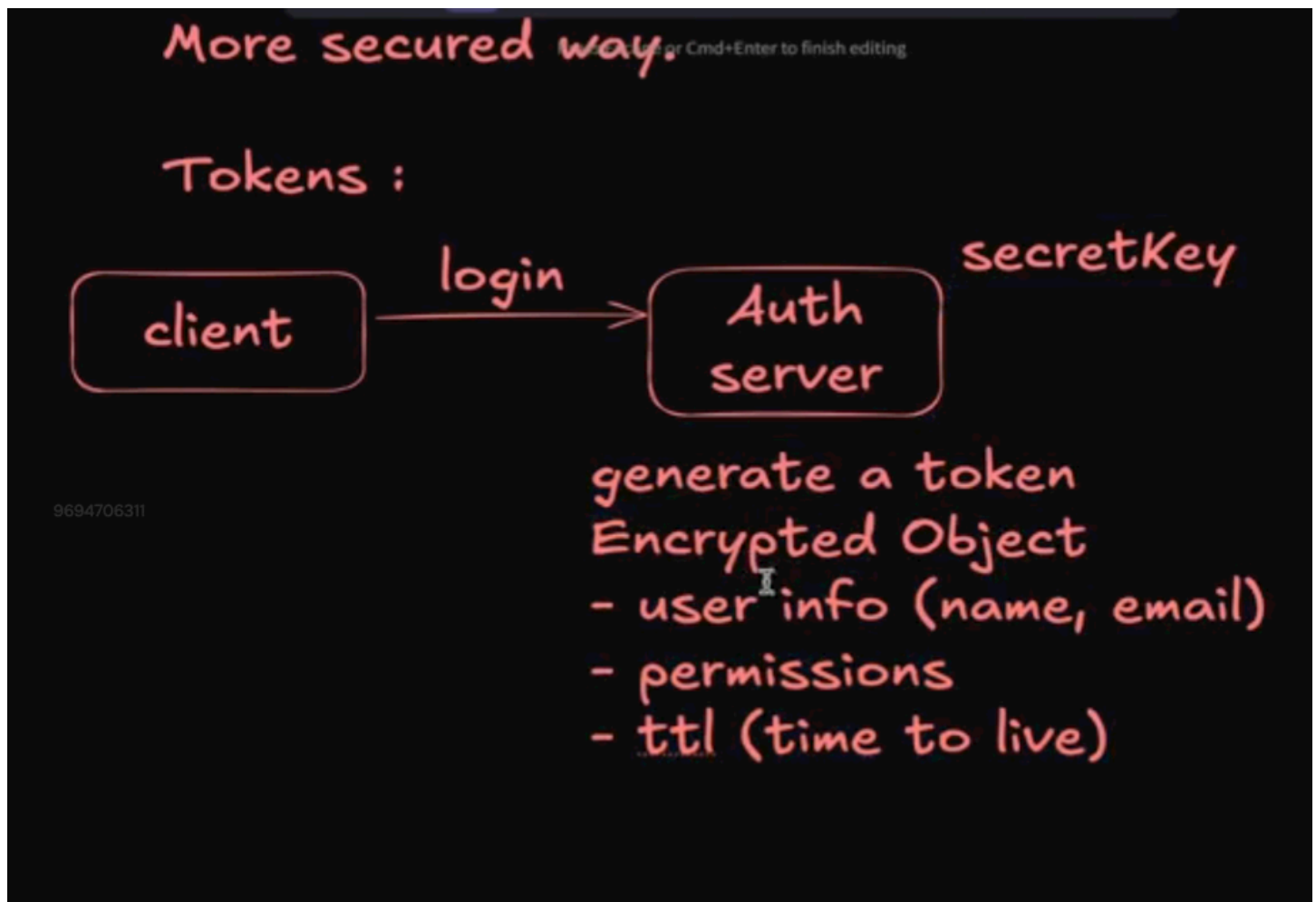
Disadvantages

- Not scalable for large systems
- Server memory increases
- Hard with multiple servers (needs Redis)

Session	Token (JWT)
Stored on server	Stored on client
Uses session id	Uses token
Stateful	Stateless

Session-based authentication is a method where the server creates a session after login and uses a session ID to authenticate the user for future requests.

Session-based authentication is not stateless because the server maintains session state for each user.



5.What is Token-Based Authentication?

Token-based authentication is a method where the server gives a token to the user after login, and the user sends this token with every request to prove their identity.

The server does not store session data, so it is STATELESS.

What is a Token?

A token is a long encoded string.

Example:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

A token usually contains:

- User ID
- Expiry time
- User role
- Signature (for security)

Most commonly used token: JWT (JSON Web Token)

How Token-Based Authentication Works (Step by Step)

1. User Logs In

The user sends:

username + password

2. Server Verifies Credentials

- If details are correct
- Server generates a token

3. Server Sends Token

The token is sent to the client:

- In response body
- OR in header
- OR inside a cookie

Example:

Authorization: Bearer <token>

4. Client Stores the Token

The client stores the token in:

- Memory
- LocalStorage
- Cookie

5. Token Sent with Every Request

For every protected API call, the client sends:

Authorization: Bearer <token>

6. Server Validates Token

The server:

- Verifies the token signature
- Checks expiry time
- Allows or denies access

Why Token-Based Authentication is Stateless?

Because:

- The server does not remember the user
- All required information is inside the token
- Each request is self-contained

Token Expiry

Tokens have an expiry time to:

- Improve security
- Limit damage if a token is stolen

Advantages

- Stateless and scalable
- Works well with APIs and mobile apps
- No server memory usage
- Easy to use with multiple servers

Disadvantages

- If token is stolen, it can be misused
- Token revocation is difficult
- Token size is larger than session ID
- Needs protection from XSS attacks

Real-Life Example

A movie ticket:

- You show the ticket at the gate
- The gate does not remember you
- If the ticket is valid, entry is allowed

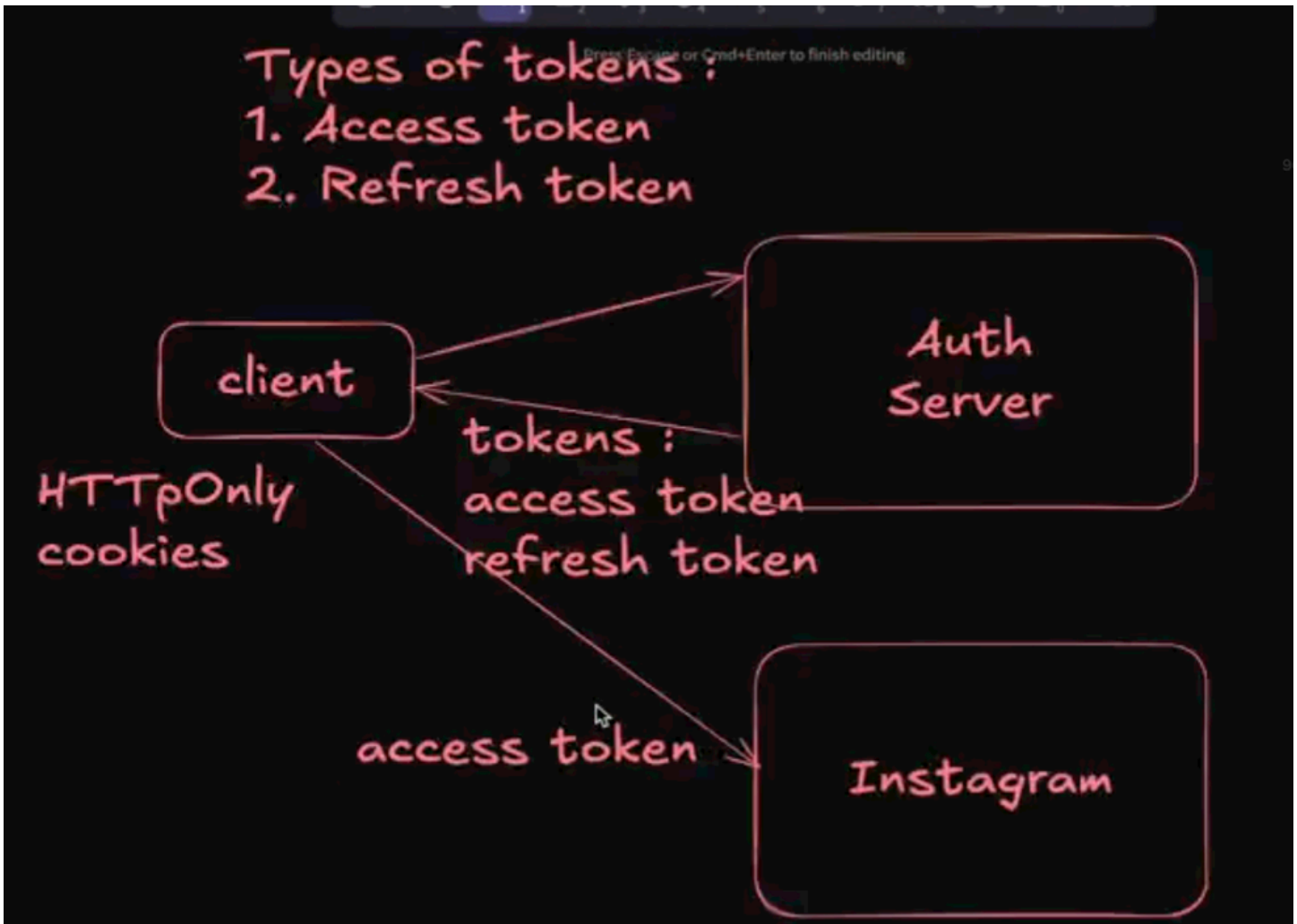
Session vs Token

Session-Based	Token-Based
Stateful	Stateless
Session stored on server	No server storage
Uses session ID	Uses token
Mostly web apps	APIs & mobile apps

Token-based authentication is a stateless authentication mechanism where the client uses a token to authenticate each request instead of a server-stored session.

Point	Session-Based Authentication	Token-Based Authentication
Basic idea	Server creates a session and remembers the user	Server gives a token, client sends it every time
State	Stateful	Stateless
Where user data is stored	On the server	Inside the token (client side)
What client sends	Session ID	Token (JWT / access token)
Storage on client	Cookie (almost always)	Header / Cookie / LocalStorage
Server memory usage	High (session stored for every user)	Very low (no session storage)
Scalability	Hard to scale	Easy to scale
Multiple servers	Needs Redis / shared DB	Works easily
Server restart	Sessions are lost	No impact
Best use case	Traditional web apps	APIs, mobile apps, SPAs
Security control	Server can invalidate session anytime	Hard to revoke token before expiry
Expiry handling	Server controls expiry	Expiry inside token
CSRF risk	High (cookie-based)	Low (when using headers)
XSS risk	Lower	Higher if stored wrongly
Network size	Very small (only session ID)	Larger (token contains data)
Logout	Easy (destroy session)	Hard (token still valid)

Performance	Slightly slower	Faster
Example	Bank website login	REST API authentication



6. Access Token

What is an Access Token?

An access token is a short-lived token used to access protected APIs/resources.

It proves that the user is authenticated and authorized.

Why is it called “Access” Token?

Because it gives access to:

- APIs
- User data
- Protected routes

If access token is valid → request allowed

If not → request rejected

How Access Token Works (Step by Step)

1. User logs in

- Sends username + password

2. Server verifies credentials

3. Server creates Access Token

Token contains:

- User ID
- Role (user/admin)
- Expiry time
- Signature

4. Server sends access token to client

5. Client stores access token

- Memory / Cookie / LocalStorage

6. Client sends token with every request:

Authorization: Bearer <access_token>

7. Server verifies token

- Signature
- Expiry
- Permissions

Access Token Expiry

- Access tokens are short-lived
- Usually: 5–15 minutes (sometimes 1 hour)

Reason:

- If token is stolen → damage is limited

Where is Access Token Used?

- REST APIs
- Mobile apps
- Single Page Applications (React, Angular)
- Microservices

Advantages of Access Token

- Stateless authentication
- Fast API access
- Works well in distributed systems

Disadvantages of Access Token

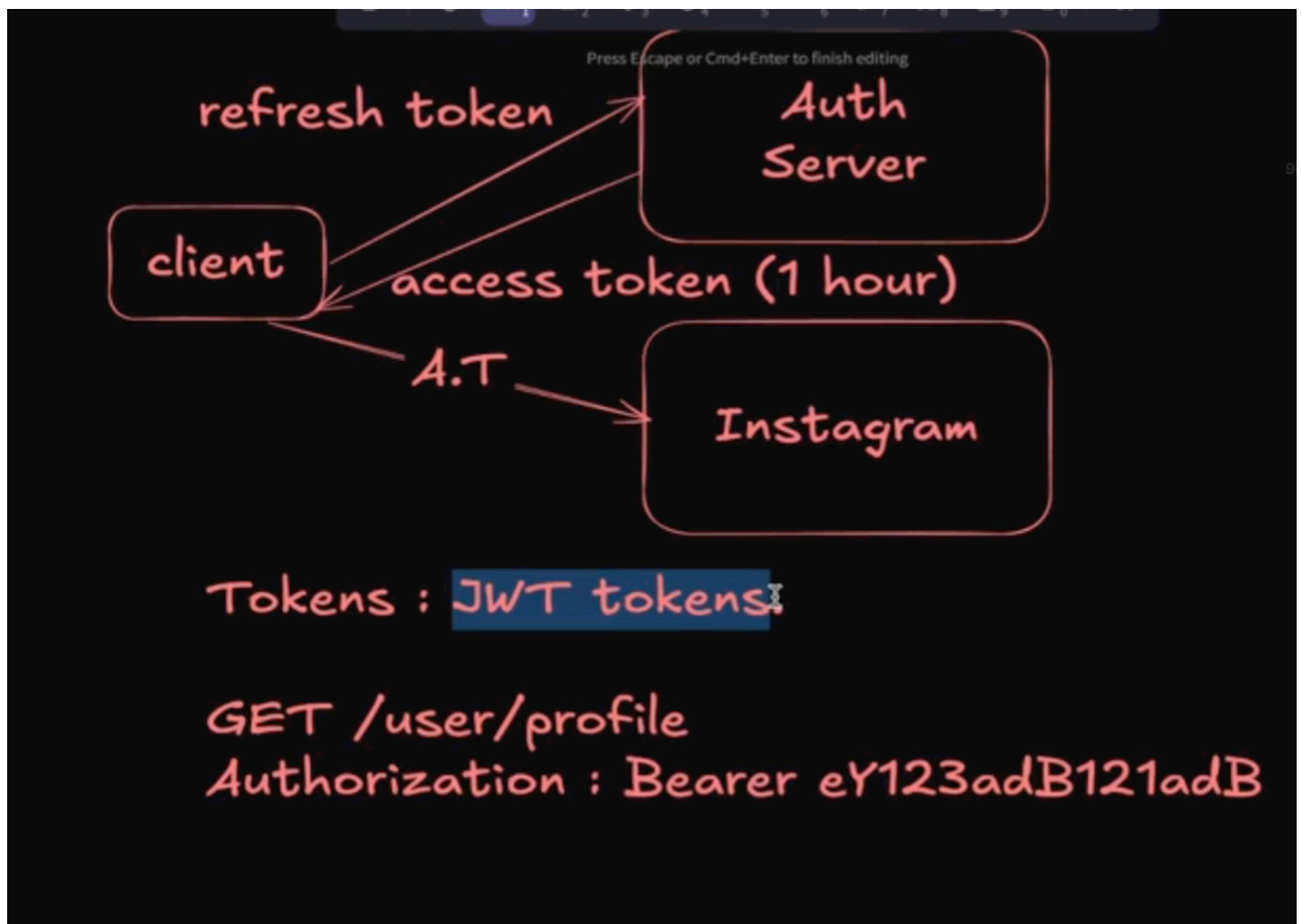
- If stolen → attacker can use it
- Hard to revoke immediately
- Needs secure storage

Real-Life Example

Concert wristband:

- Shows you paid
- Lets you enter areas
- Expires after event

An access token is a short-lived token used by the client to access protected resources by sending it with each request.



7.Refresh Token

What is a Refresh Token?

A refresh token is a long-lived token used to get a new access token when the old one expires.

It is NOT used to call APIs directly.

Why Refresh Token is Needed?

Access tokens are short-lived (for security).

When access token expires → user should not login again every time.

Refresh token helps to:

- Keep user logged in
- Improve user experience
- Still stay secure

How Refresh Token Works (Step by Step)

1. User logs in

- Server generates:
 - Access Token (short-lived)
 - Refresh Token (long-lived)

2. Client stores:

- Access token → memory / short-term
- Refresh token → secure storage (HttpOnly cookie)

3. Access token expires

4. Client sends refresh token to:

/refresh-token

5. Server verifies refresh token:

- Valid
- Not expired
- Not revoked

6. Server sends:

- New access token
- (Optional) new refresh token

Refresh Token Expiry

- Long-lived
- Usually:

- Days / Weeks / Months

Example:

- Access token → 15 minutes
- Refresh token → 7 days

Where Refresh Token is Stored?

Best practice

- HttpOnly + Secure Cookie

Avoid:

- LocalStorage
- JS-accessible storage

Advantages of Refresh Token

- User doesn't need to login again
- Better security than long access tokens
- Good user experience

Disadvantages of Refresh Token

- If stolen → attacker can generate new access tokens
- Needs proper revocation logic
- Slightly more complex to implement

Real-Life Example

Metro card:

- Card valid for months
- Each ride uses small ticket
- When ticket expires → card generates new one

A refresh token is a long-lived token used to obtain a new access token without requiring the user to log in again.

Point	Access Token	Refresh Token
Purpose	Access protected APIs/resources	Generate new access tokens
Token life	Short-lived (minutes)	Long-lived (days / weeks)
Used for	Every API request	Only when access token expires
Sent frequency	Sent with every request	Sent rarely
Where sent	Authorization: Bearer header	Refresh endpoint (/refresh)
Storage location	Memory / Cookie / LocalStorage	HttpOnly Secure Cookie (best)
Security risk if stolen	High (direct API access)	Very high (can mint new tokens)
Revocation	Hard to revoke instantly	Easier (store & blacklist)
Server storage	Not required (stateless)	Often stored / tracked
Contains user data	Yes (claims, roles)	Usually minimal info
CSRF risk	Low (when in header)	High (cookie-based)
XSS risk	High if stored in JS	Low (HttpOnly cookie)
Performance impact	Very fast	Slight overhead
Rotation	Optional	Recommended
Example expiry	10–15 minutes	7–30 days
Best practice	Keep lifetime short	Protect strongly

8.1 How Passwords Are Stored in Application DB

There are 3 main ways (from worst to best):

1. Plain Password (VERY BAD)

What is it?

Password is stored exactly as user types it.

Example DB record:

username: vijay

password: mypassword123

How it works

1. User signs up → enters password
2. App saves password directly in database
3. During login → password is compared as-is

Problems (Why it's BAD)

- If DB is hacked → all passwords leaked
- Admin can see user passwords 🙄
- Same password reused on other sites → huge risk
- Never allowed in real applications

Verdict

Do NOT use plain passwords

Even for small projects

2. Hashed Password (GOOD)

What is Hashing?

Hashing converts password into a fixed-length unreadable string using a hash function.

Example:

password: mypassword123

hash: 482c811da5d5b4bc6d497ffa98491e38

☞ Hashing is one-way

Original password cannot be reversed

How it works (Step by Step)

Signup:

1. User enters password
2. App hashes password using algorithm:
 - bcrypt
 - SHA-1,256
 - Argon2
 - PBKDF2
3. Store hash in DB

Login:

1. User enters password
2. App hashes entered password
3. Compares hash with DB hash
4. If match → login success

Why hashing is better

- DB leak ≠ password leak
- Original password never stored
- Secure if strong algorithm used

Weakness (still exists)

- Same password → same hash
- Vulnerable to:
 - Rainbow table attacks
 - Brute force (if hash is weak)

That's why Salt & Pepper is used.

3. Salt and Pepper Technique

What is Salt?

Salt = random string added to password before hashing

Example:

password: mypassword123

salt: XyZ@9!

Combined:

mypassword123XyZ@9!

Then hashed.

What is Pepper?

Pepper = secret value stored outside the database

- Same for all users
- Stored in:
 - Environment variable
 - Server config
- NEVER in DB

How Salt + Pepper Works (Step by Step)

Signup:

1. User enters password
2. App generates unique salt
3. App adds pepper (secret)
4. Hash:

hash(password + salt + pepper)

1. Store:

hash + salt

(pepper NOT stored)

Login:

1. User enters password
2. App gets salt from DB
3. App adds pepper from server
4. Hash again
5. Compare hashes

Why Salt + Pepper is Powerful

Salt protects against:

- Rainbow table attacks
- Same password → different hash

Pepper protects against:

- Database breach
- Attacker still needs server secret

Method	Stored in DB	Reversible	Security Level
Plain Password	Password	Yes	Very Low
Hashed Password	Hash	No	Medium
Salt + Pepper	Hash + Salt	No	Very High

Use bcrypt / Argon2

Use unique salt per user

Use pepper as env variable

Never store plain passwords

Never use MD5 / SHA1 alone

Passwords should never be stored in plain text. Applications store hashed passwords, and for strong security, use unique salts and a secret pepper to protect against database breaches and attacks.

9.What is a Rainbow Table? (Simple English)

A Rainbow Table is a precomputed list of passwords and their hashes used by attackers to crack hashed passwords quickly.

It is a ready-made table that stores:

plain password → hash

Example:

password123 → 482c811da5d5b4bc6d497ffa98491e38

admin123 → e99a18c428cb38d5f260853678922e03

If an attacker gets your password hash from a database, they just look it up in the rainbow table.

Match found = password cracked

Why is it dangerous?

Because:

- No need to try passwords one by one

- Very fast
- Millions of hashes already stored

How Attack Works (Step by Step)

- 1.Attacker hacks a database
- 2.Gets hashed passwords
- 3.Uses rainbow table
- 4.Finds matching hash
- 5.Gets original password

When Rainbow Table Works Best

Rainbow tables work only when:

- Same password → same hash
- No salt is used
- Weak hash algorithms used (MD5, SHA1)

Example (Without Salt)

password = "hello123"

hash = abc123

Same password for everyone → same hash

Rainbow table can crack all users at once 🤖

Why Salt Stops Rainbow Table

With salt:

password = hello123

salt1 = X9@

hash1 = hash(hello123X9@)

salt2 = Q7#

hash2 = hash(hello123Q7#)

Same password different hashes

Rainbow table becomes useless

Real-Life Analogy

Rainbow table =

A dictionary where:

- Lock number → key

Salt =

Change lock design for every house

How to Protect Against Rainbow Table Attacks

Use unique salt per user

Use slow hashing algorithms (bcrypt, Argon2)

Avoid MD5 / SHA1

Never store plain passwords

A rainbow table is a precomputed table of password hashes used to reverse hashed passwords when no salt is used.

10. Why slow hash functions are GOOD for passwords?

First: What is a “slow” hash function?

A slow hash function is designed to:

- Take more time to compute (milliseconds, not microseconds)
- Use more CPU and memory

Examples:

- bcrypt
- Argon2
- PBKDF2
- scrypt

Fast hashes (BAD for passwords):

- MD5
- SHA-1
- SHA-256 (alone)

Why slow hash functions are GOOD

1. They Stop Brute-Force Attacks

Brute force = trying millions of passwords.

- **Fast hash:**
 - Millions of guesses per second
- **Slow hash:**
 - Few hundred guesses per second

Attacker becomes very slow and expensive.

2.They Kill Rainbow Table Attacks

Slow hash + salt =

- Attacker cannot precompute hashes
- Rainbow tables become useless

3.They Protect Against GPU & ASIC Attacks

Hackers use:

- GPUs
- Special hardware (ASICs)

Slow hash functions:

- Are memory-hard
- GPUs become inefficient
- Attacks become costly

4.Time Cost = Security Knob

Slow hashes allow you to control:

- How slow hashing is

Example:

- bcrypt cost = 12
- Increase cost as computers get faster

Security improves over time without changing logic.

5.One Password Guess = High Cost

For users:

- Hashing once during login → no problem

For attackers:

- Billions of guesses → impossible

Small pain for user, huge pain for attacker

Hash Type	Speed	Password Safety
MD5	Very fast	Unsafe
SHA-256	Very fast	Unsafe
bcrypt	Slow	Safe
Argon2	Slow + memory heavy	Best

Real-Life Analogy

- Fast hash = cheap lock
- Slow hash = heavy steel lock

Breaking 1 lock is okay

Breaking 1 million locks = impossible

What Happens If DB Is Hacked?

- Attacker gets hashes
- Slow hash makes cracking:
 - Very slow
 - Very expensive
 - Often not worth it

You should use this

1. Use Argon2 (best)
2. Or bcrypt (cost ≥ 12)
3. Always use unique salt
4. Optionally use pepper

Slow hash functions are used for password storage because they significantly reduce the speed of brute-force and rainbow table attacks, making password cracking impractical even if the database is compromised.

11. Authorization & Access Rights

Authorization answers the question:

“What is a user allowed to do?”

It happens after authentication (login).

Example:

- Can user read a file?
- Can user delete an account?
- Can user access admin panel?

1.RBAC – Role-Based Access Control

What is RBAC?

In RBAC, access is given based on the role assigned to a user.

Users → Roles → Permissions

How RBAC Works (Step by Step)

1. **System defines roles:**
 - Admin
 - Manager
 - User
2. **Each role has permissions:**
 - Admin → read, write, delete
 - User → read only
3. **User is assigned a role:**
 - Vijay → User
4. **System checks role before action:**
 - Vijay tries delete → denied

Example (Real World)

Company:

- Employee → view data
- HR → edit employee data
- Admin → full access

Advantages of RBAC

- 1.Easy to understand
- 2.Easy to manage
- 3.Scales well for organizations

Disadvantages of RBAC

- 1.Too rigid
2. Hard for complex rules
3. Role explosion (too many roles)

Best Use Case

- Corporate apps
- Admin panels
- Bank systems

2.PBAC – Policy-Based Access Control

What is PBAC?

In PBAC, access is decided using policies (rules), not fixed roles.

Decision is based on:

- User attributes
- Resource attributes
- Environment conditions

How PBAC Works

System checks a policy, for example:

IF user.department == "HR"

AND request.time < 6 PM

AND resource.type == "EmployeeFile"

THEN allow

Example

- HR can access files
- Only during office hours
- Only from office network

Advantages of PBAC

1. Very flexible
2. Handles complex conditions
3. Fine-grained control

Disadvantages of PBAC

1. Hard to design
2. Hard to debug
3. More processing required

Best Use Case

- Enterprise systems
- Government systems
- Cloud platforms

3.ACL – Access Control List

What is ACL?

An ACL is a list attached to a resource that says:
Who can do what on this resource

How ACL Works

Each resource has a list:

File A:

- Vijay → read
- Admin → read, write
- Guest → no access

Example (File System)

Linux file permissions:

`rwxr-x---`

Advantages of ACL

- 1.Very specific control
- 2.Easy for small systems

Disadvantages of ACL

- 1.Hard to manage at scale
- 2.Large lists for many users
- 3.Not flexible for conditions

Best Use Case

- File systems
- Small apps
- Resource-specific permissions

Feature	RBAC	PBAC	ACL
Based on	Roles	Policies & rules	Resource lists
Flexibility	Medium	Very high	Low
Complexity	Low	High	Medium
Scalability	Good	Very good	Poor
Fine-grained control	Medium	Very high	High
Easy to manage	Yes	No	No
Common examples	Admin/User	Cloud IAM	File permissions

- RBAC → access based on role
- PBAC → access based on rules and conditions
- ACL → access based on resource-specific lists

12.What is an API Endpoint?

An API endpoint is a specific URL where a client (browser, app, frontend) sends a request to get or send data.

Endpoint = URL + HTTP method

An API endpoint is a communication point where the client interacts with the server to perform an action.

Structure of an API Endpoint

Example:

`https://api.example.com/users/123`

It has 4 parts:

1.Base URL

`https://api.example.com`

2.Path / Resource

/users

3.Parameter (optional)

/123

4.HTTP Method

GET / POST / PUT / DELETE

HTTP Methods Used in API Endpoints

Method	Purpose	Example
GET	Read data	Get user
POST	Create data	Create user
PUT	Update full data	Update user
PATCH	Update partial data	Update user email
DELETE	Remove data	Delete user

Types of API Endpoints (with examples)

1.GET Endpoint (Read Data)

Purpose:

Fetch data from server

Example:

GET /users

GET /users/101

Use case:

- Get all users
- Get one user

2.POST Endpoint (Create Data)

Purpose:

Send data to server to create a new resource

Example:

POST /users

Body:

```
{  
  
  "name": "Vijay",  
  
  "email": "vijay@gmail.com"  
}
```

3.PUT Endpoint (Update Full Resource)

Purpose:

Replace existing data

Example:

PUT /users/101

4.PATCH Endpoint (Update Partial Resource)

Purpose:

Update only specific fields

Example:

PATCH /users/101

5.DELETE Endpoint (Remove Data)

Purpose:

Delete a resource

Example:

DELETE /users/101

Most endpoints are protected.

Example:

Authorization: Bearer <access_token>

Without valid token → 401 Unauthorized

Authorization in API Endpoints

Checks:

- Is user logged in?
- Does user have permission?

Example:

- Admin → DELETE allowed
- User → DELETE denied

Request Parts in an API Endpoint

1.Headers

Authorization

Content-Type

2.Query Parameters

GET /users?page=1&limit=10

3.Path Parameters

/users/{id}

4.Body

JSON data (POST, PUT)

Response from API Endpoint

Example:

```
{  
  
  "status": "success",  
  
  "data": {
```

```
"id": 101,  
  
"name": "Vijay"  
  
}  
  
}
```

HTTP Status Codes (Important)

Code	Meaning
200	OK
201	Created
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
500	Server Error

An API endpoint is a specific URL combined with an HTTP method that allows a client to interact with server resources securely and efficiently.

