🔐 Lecture 10 — Authentication & Authorization System

System Design Masterclass ·

🧠 First Principles — Ground Zero

Jab bhi hum **koi system design** karte hain — chahe *Instagram*, *Banking App*, *College ERP* ya *SaaS product* — **Security sabse pehla concern hota hai.**

Security ko samajhne ke liye sirf **2 fundamental sawal** hote hain.
👉 In dono me confusion hua, to **poora system design galat direction me chala jaata hai.**

🔑 Authentication — Pehchaan

Fundamental Sawal:
Who are you?
(Aap kaun ho? Aapki identity kya hai?)

Real-Life Analogy:
✈️ Airport gate par security guard **ID Card + Ticket** check karta hai.
Iska matlab sirf ek hota hai:
➡️ *"Tum wahi ho jo tum claim kar rahe ho."*

Digital World:
• Login
• Username + Password

⚠️ **Boundary Rule (Bahut Important):**
Authentication ka kaam sirf **identity verify karna** hota hai.
Ye kabhi decide nahi karta ki user kya-kya kar sakta hai.

🛡️ Authorization — Adhikaar

Fundamental Sawal:
What can you do?
(Aap kya-kya access kar sakte ho?)

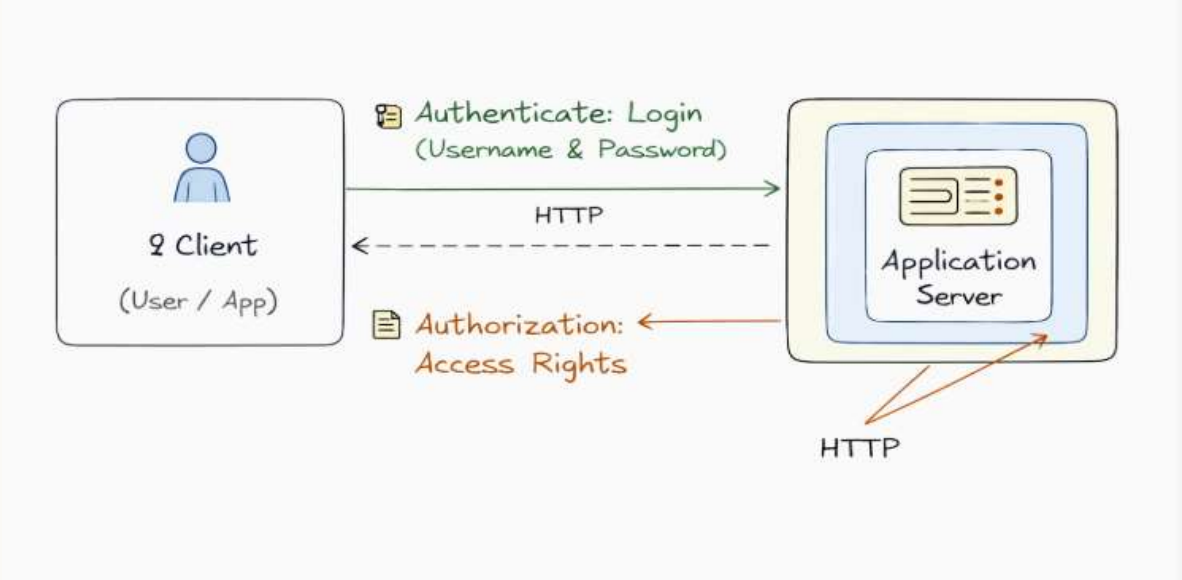
Real-Life Analogy:
🏢 ID verify hone ke baad:
• Business Class Lounge ❓
• Ya sirf Waiting Area ❓

Digital World:
• Access Control
• Admin vs User

🔥 **Golden Rule (Exam + Interview):**
Authorization **kabhi bhi** Authentication se pehle nahi hota.
➡️ Pehle **Pehchaan**, phir **Adhikaar.**

📐 Basic System Interaction :

System ka **sabse basic aur fundamental view**:



Logical Order:
• Step 1 → Authenticate → Login (Username & Password)
• Step 2 → Authorize → Access Rights check

🧠 **Hidden Reality:**
HTTP ek **Stateless protocol** hai.
Server ko yaad nahi rehta user kaun hai.
➡ Isi ek problem se **poora lecture aage build hota hai.**

📋 System Requirements — Humein Kya Banana Hai?

Ek **real-world secure system** ke liye ye **6 cheezein mandatory** hoti hain:

1. **User Registration (Sign-Up)**
   User apni necessary information deta hai.

2. **Login**
   User credentials (ID / Password) ke through verify hota hai.

3. **Multi-Factor Authentication (MFA)**
   Sirf password kaafi nahi — OTP bhi chahiye.

4. **Password Recovery**
   User password bhool sakta hai, isliye secure reset flow zaroori hai.

5. **Session Management**
   HTTP Stateless hai, server ko memory chahiye taaki user ko baar-baar login na karna pade.

6. **Access Control**
   User roles define karna (Admin, User, etc.).

🧮 Back-of-the-Envelope Calculations

Traffic Scenario:
• 100,000 (1 Lakh) users per day

QPS (Queries Per Second):

```
100,000 / 86,400 ≈ 1.15 req/sec
```

📌 **Observation:**
Load bahut kam hai, easily manageable.

💾 Storage Estimation (Authentication Data)

Assumptions:
• 5 KB per user
• 5 years data retention

Formula:

```
Total = Users × Days × Years × Size
```

Calculation:

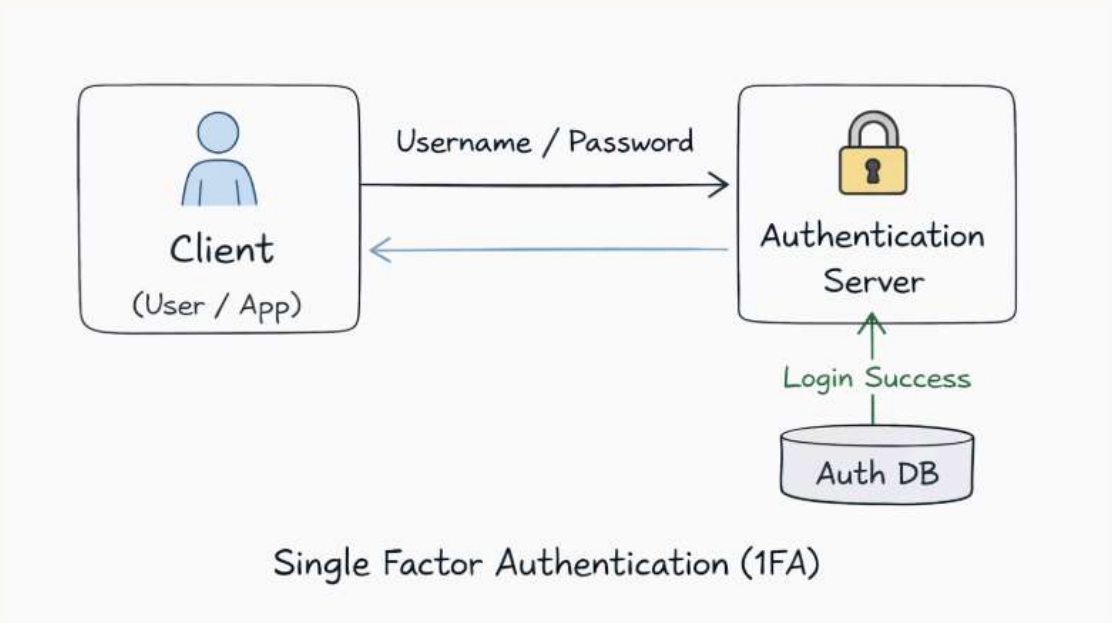
```
100,000 × 365 × 5 × (5 × 2¹⁰) ≈ 914 GB
```

💡 **Conclusion:**
Authentication database ke liye approx **1 TB storage (5 years)** chahiye.

🔄 Authentication Flow — From Basic to Advanced

Single-Factor Authentication :

(Sirf Username + Password)

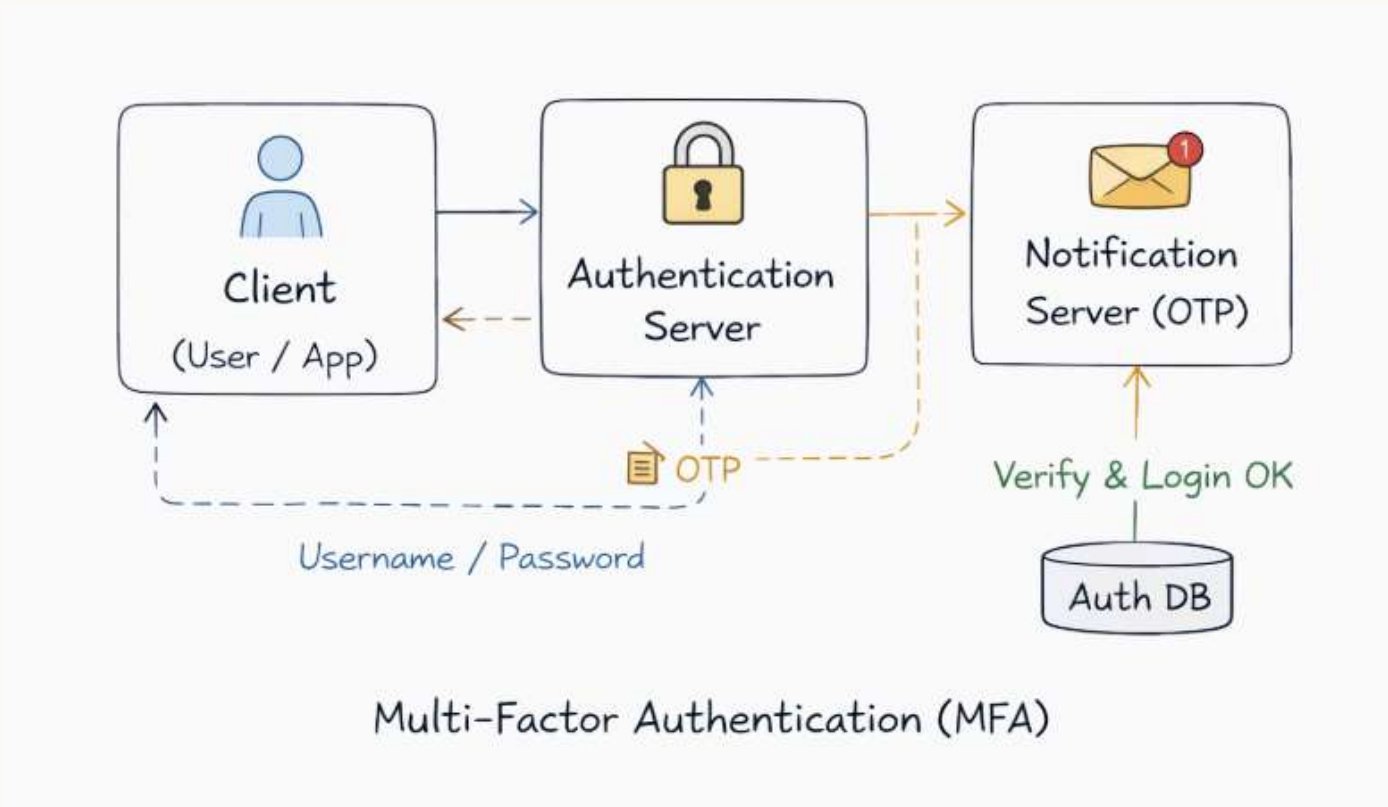


Single Factor Authentication (1FA)

⚠️ **Limitation:**
Password leak hua → account compromise.

Multi-Factor Authentication :

(Password + OTP)



🧠 **Security Insight:**
Agar attacker ke paas password bhi ho, tab bhi bina OTP ke login possible nahi hota.

🧠 Mental Model — Yahan Tak Lock Kar Lo

- Authentication = **Pehchaan**
- Authorization = **Adhikaar**
- HTTP Stateless = **Memory Problem**
- MFA = **Real-world necessity**
- Numbers = **Scale samajhne ka base**

🧠 Session Management — The Core Problem

Yahan se lecture ka **real system design part** start hota hai.

❓ Actual Problem

HTTP ek **Stateless protocol** hai.
Matlab:

- Har request **independent** hoti hai
- Server ko **yaad nahi rehta**:
    - pichli request kisne bheji
    - user login hai ya nahi

👉 Simple sawal:
“Agar server bhool jata hai ki tum kaun ho, to login ka fayda kya?”

Is problem ka solution hi kehlata hai 👉 **Session Management**

🧠 State Kaise Maintain Karein?

Web applications me **sirf 2 tareeke** hote hain:

1️⃣ Server khud yaad rakhe user ko
2️⃣ User khud apni pehchaan saath me le aaye

Yahin se do approaches nikalti hain 👇

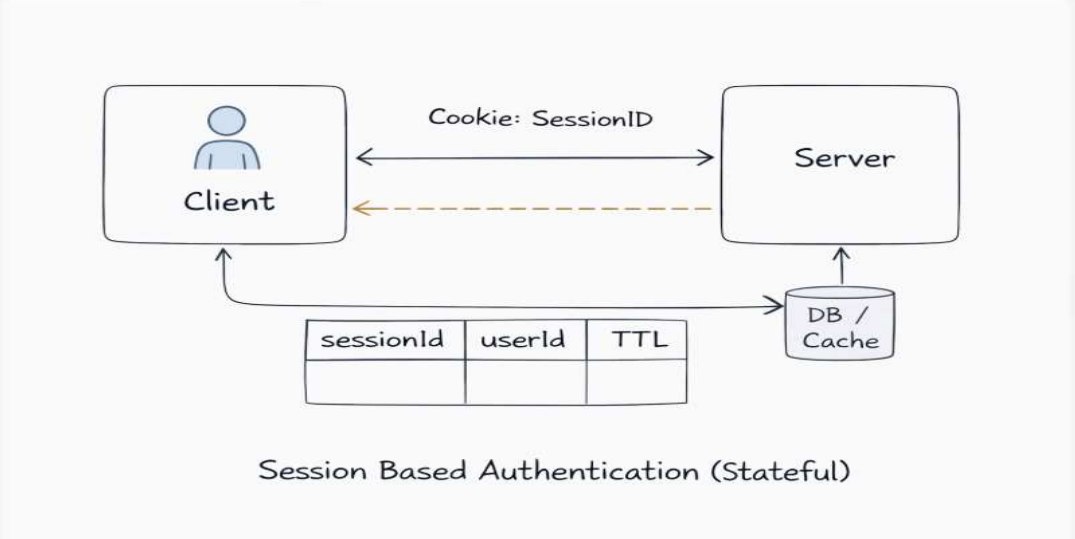
🅰️ Session-Based Authentication (Stateful)

Is approach me **Server + DB/Cache** dono user ko yaad rakhte hain.

🔁 Step-by-Step Flow

1️⃣ Client login request bhejta hai (username + password)
2️⃣ Server credentials verify karta hai
3️⃣ Server ek **SessionID** generate karta hai
4️⃣ SessionID ko **DB / Cache (Redis)** me store karta hai
5️⃣ Client ko SessionID **Cookie** me milta hai
6️⃣ Har next request me client wahi SessionID bhejta hai
7️⃣ Server DB me check karta hai → session valid hai ya nahi

📐 Session-Based Architecture 🙂



⚠️ Major Drawback (Very Important)

Socho:

- Login **Server A** par hua
- Next request **Server B** par chali gayi

👉 Server B ke paas **SessionID ka record hi nahi**

Result:

- User achanak logged-out jaisa behave karega

🧠 Why Scalability Issue Aata Hai

- Session data server-specific hota hai
- Multiple servers → session sync problem

Industry Workarounds:

- Sticky Sessions ❌ (not scalable)
- Centralized Redis Cache ✅

📌 **Reality Check:**
Session-based auth **small / monolithic systems** ke liye theek hai,
lekin **large-scale distributed systems** me pain ban jata hai.

🅱 Token-Based Authentication (Stateles — JWT)

Yeh **modern scalable systems ka default choice** hai.

🧠 First-Principle Difference

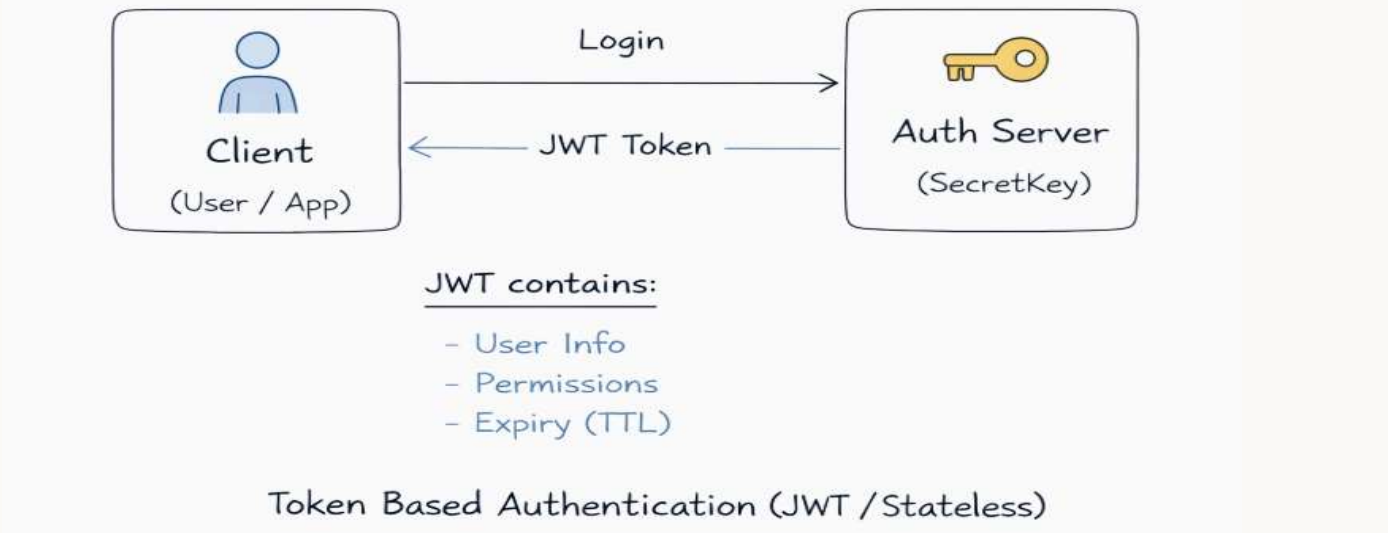
Yahan server **kuch bhi yaad nahi rakhta.**

🔁 Flow Logic

1. Client login request bhejta hai
2. Server credentials verify karta hai
3. Server ek **JWT Token** generate karta hai
4. Token client ko milta hai (usually HTTPOnly cookie)
5. Har request ke saath client token bhejta hai
6. Server token ko **Secret Key** se verify karta hai

❌ **No DB call required**

📐 Token-Based Architecture



🔐 JWT Ke Andar Kya Hota Hai?

Token ek **signed object** hota hai jisme hota hai:

- User information (id, email)
- Permissions / roles
- TTL (expiry time)

👉 Server sirf token verify karta hai, **kuch store nahi karta**

🚀 Why JWT Scale Hota Hai

- Stateless architecture
- No session sync
- Microservices friendly
- Horizontal scaling easy

📌 **Key Line (Interview Gold):**
JWT me user apni pehchaan khud token me le aata hai.

🧠 Session vs Token — Mental Model

- Session-Based:
  👉 Server bole: *“Main yaad rakhunga tum kaun ho.”*
- Token-Based:
  👉 Server bole: *“Tum khud proof leke aao.”*

🧠 Yahan Tak Lock Kar Lo

- HTTP Stateless = root problem
- Session-based = Stateful, scaling pain
- Token-based (JWT) = Stateless, scalable
- Modern systems **JWT prefer karte hain**

🔑 The Two-Token System — Access Token & Refresh Token

Yahan se JWT ka **most important & most misunderstood part** start hota hai.
Interviews + real projects dono me **maximum confusion yahin hota hai.**

❓ Core Question

Agar JWT itna powerful hai,
to **sirf ek token kaafi kyun nahi hota?**

👉 Answer: **Security vs User Experience ka balance**

🧠 Token Types — Clear Separation

🔷 Access Token

- ⏱️ Short-lived (example: **1 hour**)
- 🔁 **Har API request** ke saath bheja jata hai
- 🎯 Actual data access ke liye use hota hai

👉 Ye token **network par sabse zyada travel karta hai**

🔷 Refresh Token

- ⏱️ Long-lived (example: **1 day / 1 month**)
- 🔄 Sirf **naya Access Token l**ene ke liye
- 🔐 Network par **baar-baar travel nahi karta**

⚠️ Risk Analysis — Why One Token Is Dangerous

Socho agar:

- Sirf **ek hi token** ho
- Aur uski life **1 din** ho
- Aur hacker ne token chura liya
  👉 To attacker ke paas **1 poora din** hoga system exploit karne ke liye.

❌ **Unacceptable risk**

🛡️ Defense Strategy — Two Token Logic

🧠 First-Principle Thinking

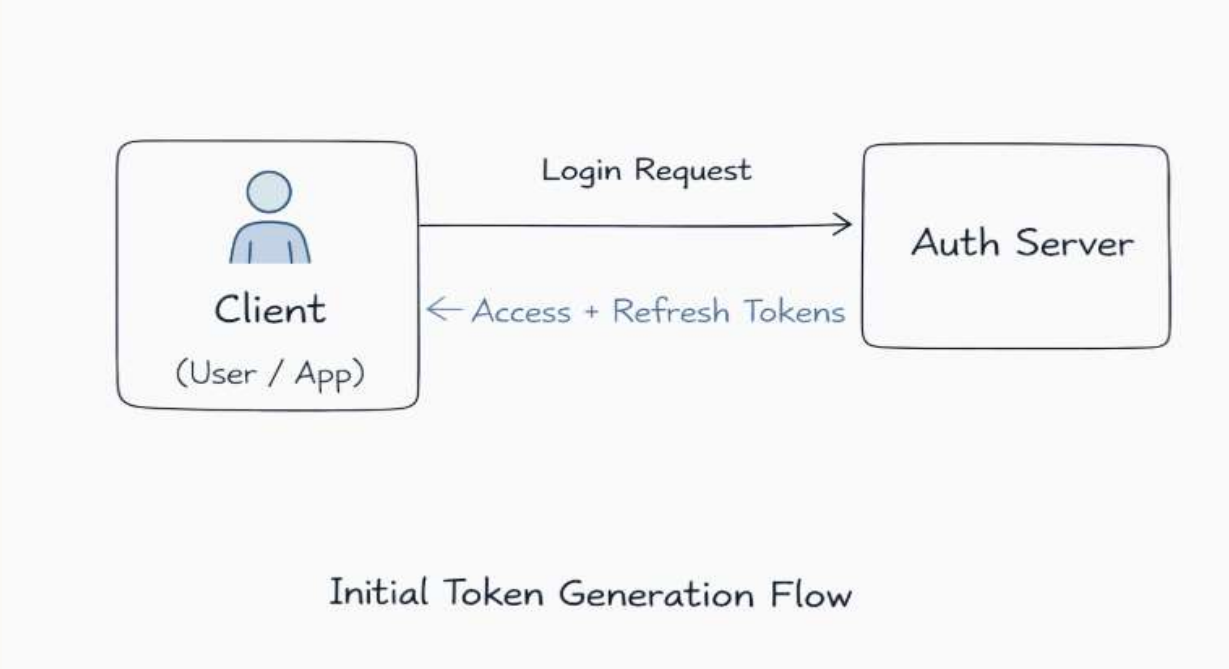
- **Access Token**
  → Chori ho bhi gaya
  → Damage sirf **1 hour** tak

- **Refresh Token**
  → Rarely network par jata hai
  → Isliye **comparatively safe**

👉 **Limited damage + smooth UX**

🔁 Initial Login Flow

User jab **pehli baar login** karta hai:

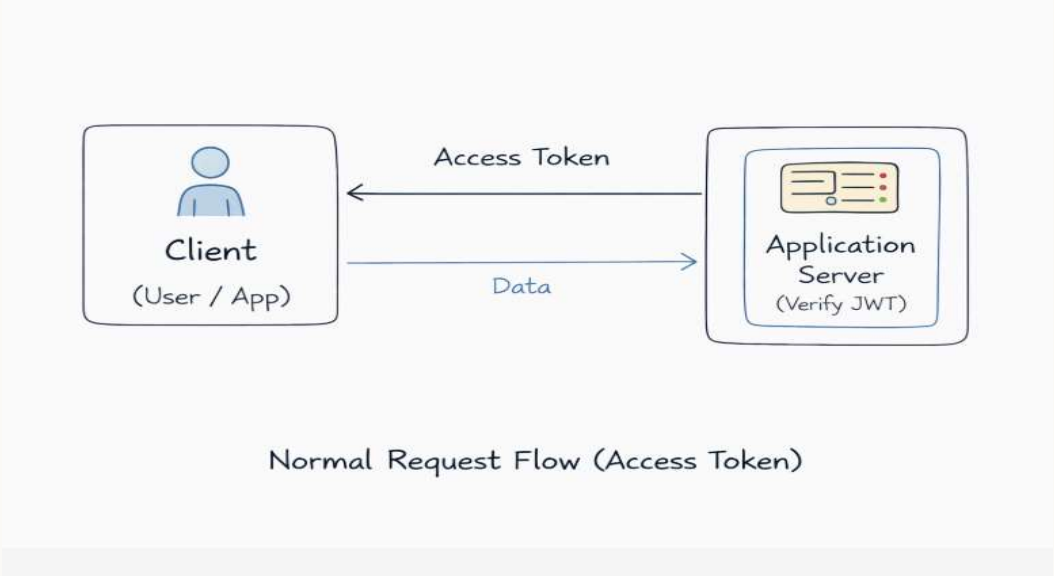


Initial Token Generation Flow

📌 Tokens usually **HTTPOnly cookies** me store hote hain.

🔁 Normal Request Flow

Har normal API request ke time:



Normal Request Flow (Access Token)

🧠 Server side checks:

- Access Token **verify**
- **TTL (expiry)** check
- **Permissions / roles** check

❌ **No DB call required**

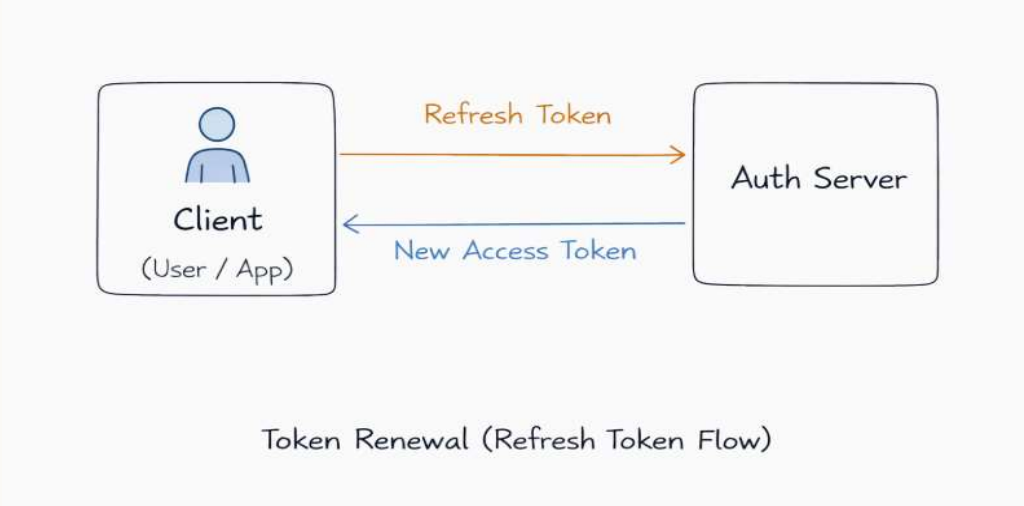
⏱️ Token Expiry — What Really Happens?

Access Token ki life **limited hoti hai.**
Jaise hi TTL expire hota hai:

👉 Normal API request **fail** ho jaati hai
👉 Par user ko logout karna zaroori nahi

🔄 Token Renewal Flow

Browser / client silently ye flow chala deta hai:



Token Renewal (Refresh Token Flow)

🔁 Flow samjho:

- Client Refresh Token bhejta hai
- Server Refresh Token verify karta hai
- **Naya Access Token** issue karta hai

📧 Gmail Example — Real World Mapping

Socho:

- Tum Gmail login karke chale gaye
- 3 ghante baad wapas aaye

Background me kya hota hai:

- Access Token expired
- Browser automatically Refresh Token bhejta hai
- Naya Access Token mil jata hai

👉 **User ko dobara login nahi karna padta**

🧠 What Server Really Does

- Access Token → **Har request** me verify
- Refresh Token → **Sirf renewal time** par verify
- User session ka experience → **smooth & secure**

🧠 Mental Model — Two Token System

- Access Token = **Temporary gate pass**
- Refresh Token = **Permanent ID at security office**

Gate pass expire ho jata hai,
par ID valid hoti hai → naya pass mil jata hai.

🧠 Yahan Tak Lock Kar Lo

- Single token = high risk
- Two tokens = controlled damage
- Access Token = short life
- Refresh Token = long life
- JWT UX + Security dono handle karta hai

🛡️ Password Storage — The Most Critical Part

Yahin par **real security engineering** start hoti hai.
Maximum systems yahin par **silently fail** ho jaate hain.

❓ Core Question

Agar attacker **database hack** kar le,
to kya wo users ke passwords padh sakta hai?

👉 **Goal:**
Database leak hone ke baad bhi passwords safe rehne chahiye.

❌ What NOT To Do — Plain Text Storage

Example:
User = Aditya
Password = `aditya123`

Database me store:

```
aditya | aditya123
```

🚨 **Disaster Scenario:**

- DB leak ho gayi
- Har user ka password openly visible

👉 **Plain text storage = system suicide**

⚠️ Step 1 — Hashing (Alone Is Not Enough)

Next obvious idea:
Password ko **hash** kar do.

```
hash("aditya123") → qwe1234
```

Database me store:

```
aditya | qwe1234
```

❌ Problem: Rainbow Table Attack

Attackers ke paas already hota hai:

- Common passwords
- Unke pre-computed hashes

👉 Wo bas match karke password nikaal lete hain.

Conclusion:
👉 Sirf hashing **enough nahi** hai.

🧂 Step 2 — Salting (Better Defense)

🧠 Idea

Har user ke password ke saath
ek **random string (Salt)** jod do.

Example:

```
Password = aditya123
```

```
Salt = tyu78
```

Hashing input:

```
hash("aditya123tyu78")
```

📌 Important Points

- Har user ka **unique salt** hota hai
- Same password hone par bhi hash **different** hoga
- Salt **database me store hota hai**

👉 Rainbow tables useless ho jaate hain.

🌶️ Step 3 — Pepper (Industry Gold Standard)

🧠 Pepper Kya Hai?

- Ek **secret value**
- Sab users ke liye same
- **Database me kabhi store nahi hoti**

Pepper rehta hai:

- Server config
- Environment variables
- Secure vault

🔐 Final Hash Formula

```
Hash = hash( Pepper + Password + Salt )
```

Example:

```
Pepper = asdf123
```

```
Password = aditya123
```

```
Salt = tyu78
```

Final input:

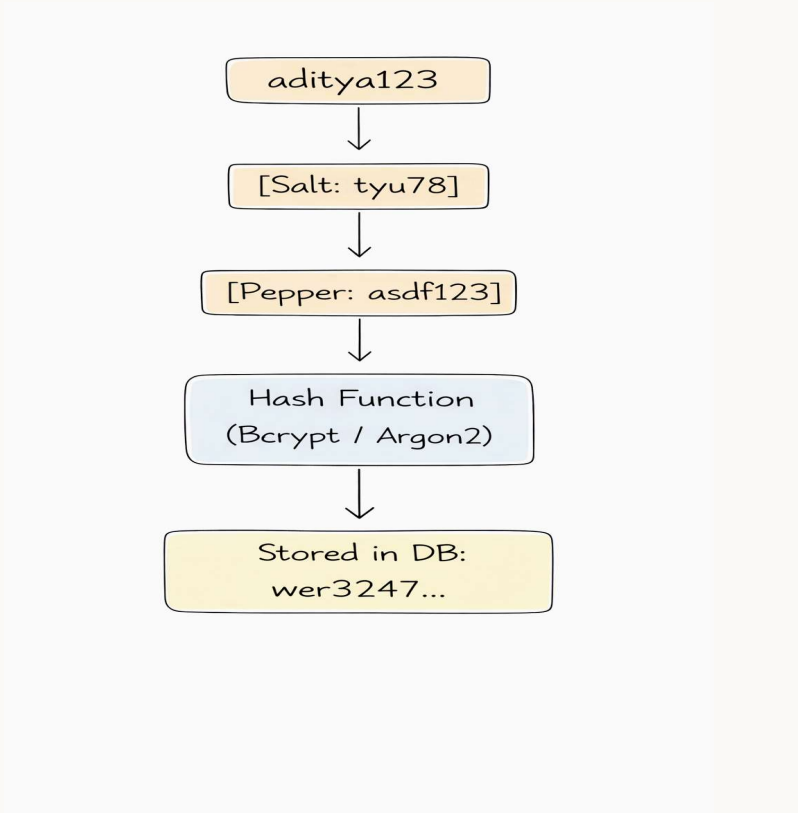
```
hash("asdf123aditya123tyu78")
```

Database me store:

```
aditya | wer3247...
```

📐 Password Hashing Logic

```
User Input: aditya123
```



⚡ Fast Hash vs Slow Hash (VERY IMPORTANT)

❌ Fast Hash Algorithms

- MD5
- SHA-1
- SHA-256

⚠️ Ye **seconds me billions** of guesses allow kar dete hain.
👉 Brute force easy ho jaata hai.

✅ Slow Hash Algorithms (Recommended)

- Bcrypt
- Argon2
- Scrypt

🧠 Ye algorithms **jaan-bujhkar slow** hote hain (milliseconds).
Iska matlab:

- Single guess = expensive
- Billion guesses = practically impossible

👉 **Security ka real wall yahin banta hai.**

🧠 What Server Does at Login

1️⃣ User password enter karta hai
2️⃣ Server:

- Same salt fetch karta hai
- Same pepper add karta hai
- Hash generate karta hai
  3️⃣ DB ke stored hash se compare karta hai
  4️⃣ Match hua → login success

❌ **Password kabhi decrypt nahi hota**

🧠 Mental Model — Password Security

- Plain text = disaster
- Hash only = insufficient
- Salt = rainbow table defense
- Pepper = DB breach protection
- Slow hash = brute force killer

🧠 Yahan Tak Lock Kar Lo

- Passwords **kabhi decrypt nahi hote**
- DB leak hone ke baad bhi passwords safe rehne chahiye
- **Bcrypt + Salt + Pepper = Gold Standard**

👮 Authorization — Permissions & Access Control

Yahan se hum **Authentication ke baad wali duniya** me enter karte hain.
User system ke andar aa chuka hai.
Ab **real question** ye nahi hai ki *user kaun hai* —
real question ye hai:

👉 **“User kya-kya kar sakta hai?”**

🧠 Authentication vs Authorization (One-Line Truth)

- Authentication → **Identity proof**
- Authorization → **Permission check**

👉 Authentication ek baar hota hai
👉 Authorization **har request** par hota hai

⚠️ **Yahin log real systems me galti karte hain.**

🔑 RBAC — Role Based Access Control

Sabse **common aur widely used** authorization model.

🧠 Core Idea

User ko ek **Role** assign karo.
Permissions role se inherit hoti hain.

🏷️ Typical Roles

- Admin
- Member
- Viewer

📐 RBAC Logic (Conceptual Flow)

```
User ──▶ Role ──▶ Permissions
```

Example:

- Admin → Create / Update / Delete
- Member → Read / Comment
- Viewer → Read only

📌 RBAC Real-Life Mapping

Office system:

- HR → Employee data edit
- Manager → Approve leaves
- Employee → View profile

⚠️ RBAC Limitation

RBAC **static hota hai.**
Complex conditions handle karna mushkil hota hai.

Example jo RBAC me mushkil hai:

> “User tabhi access kare jab subscription active ho”

📜 PBAC — Policy Based Access Control

RBAC se ek step aage.

🧠 Core Idea

Access **rules / policies** ke basis par milta hai,
sirf role ke basis par nahi.

📐 PBAC Logic (Conceptual Flow)

```
User + Condition ──▶ Policy Engine ──▶ Allow / Deny
```

📜 Example Policies

- “User paid content tabhi dekhega agar subscription = active”
- “User sirf office hours me dashboard access kare”
- “Country = India ho tabhi feature enable ho”

🚀 PBAC Use Cases

- SaaS platforms
- Subscription-based apps
- FinTech & Enterprise systems

📋 ACL — Access Control List

Ye **sabse granular control** deta hai.

🧠 Core Idea

Har resource ke liye
explicit list hoti hai:
Kaun kya kar sakta hai

📐 ACL Logic (Conceptual Flow)

```
Resource
 ├─ User A → Read
 ├─ User B → Write
 └─ User C → No Access
```

📁 Real-Life Example

Google Drive:

- File A → View only
- File B → Edit
- File C → No access

⚠️ ACL Drawback

- Manage karna mushkil
- Large systems me complex ho jaata hai

🚦 Rate Limiting — An Important Authorization Check

Authorization sirf *kya access* tak limited nahi hota.
Kabhi-kabhi question hota hai:

👉 **“Kitni baar access allowed hai?”**

📐 Rate Limiting Logic

```
User ──▶ Requests Counter ──▶ Allow / Block
```

Example:

- Max 3 requests / second
- Zyada hua → temporarily block

🛡️ Why Rate Limiting Matters

- Brute force attacks
- API abuse
- DDoS protection

🧠 Authorization Mental Model (Lock This)

- RBAC → **Role decides power**
- PBAC → **Policy decides permission**
- ACL → **User-specific control**
- Rate Limiting → **Usage control**

🧠 Yahan Tak Lock Kar Lo

- Authentication ≠ Authorization
- Authorization har request par hota hai
- Large systems me **RBAC + PBAC mix** use hota hai
- Rate limiting security ka part hai, optimization nahi

🏗️ Complete System Architecture — Big Picture

Ab tak humne **individual concepts** dekhe:
Authentication → Sessions → JWT → Password Security → Authorization

Ab in sab ko **ek single flow** me jodte hain.
👉 **Real-world systems aise hi kaam karte hain.**

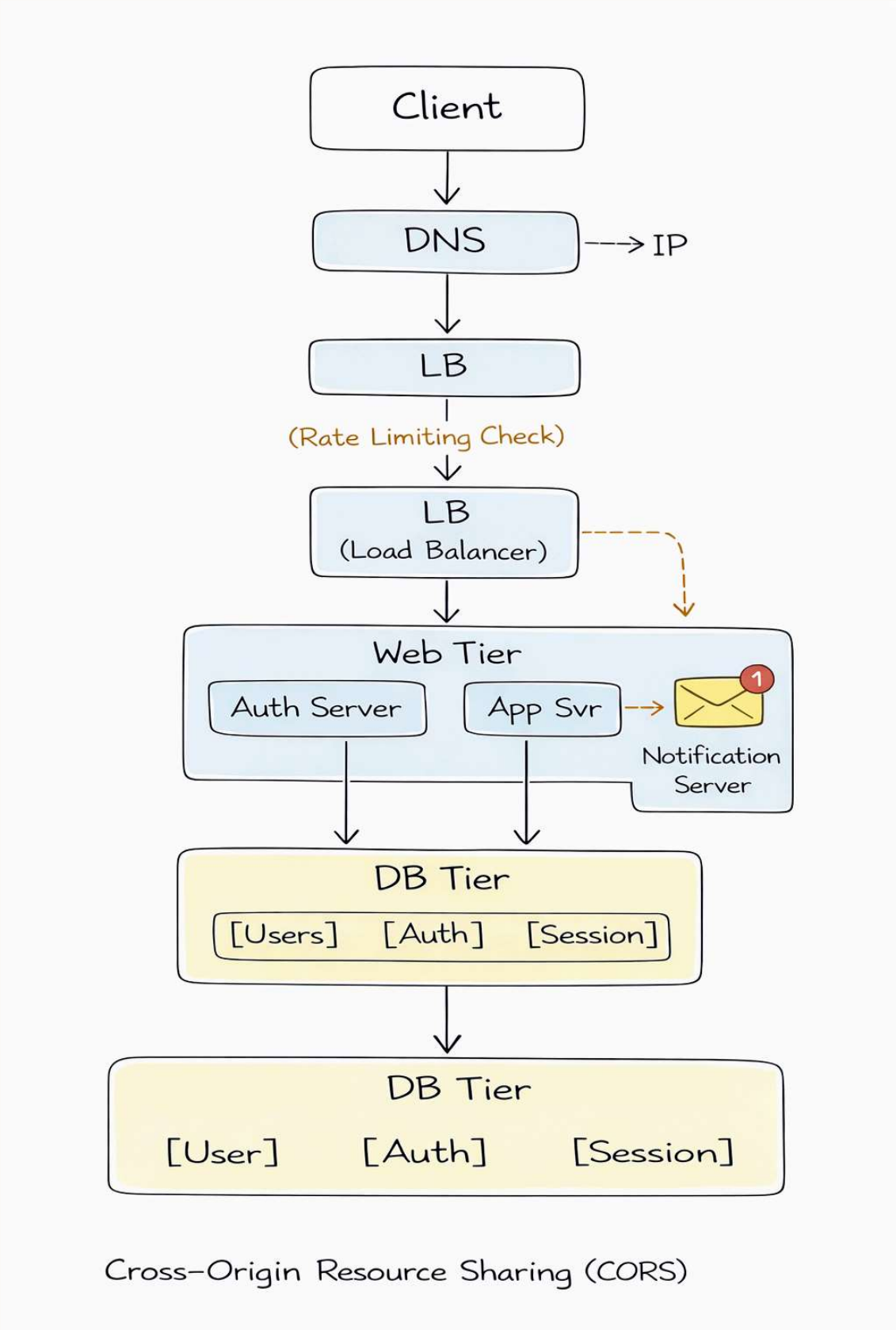
🌐 User Journey — End to End

User jab browser me URL enter karta hai:

`www.instagramclone.com`

📐 Complete Architecture

```
User Action:  Open App / Login / Use Features
```



🧠 Each Component Ka Role

💻 Client

- Browser / Mobile App
- Tokens store karta hai (HTTPOnly cookies)

🌐 DNS

- Domain → IP resolve karta hai

⚖️ Load Balancer (LB)

- Requests ko multiple servers me distribute karta hai
- High availability + scalability

🚦 Rate Limiter

- Request abuse rokta hai
- Brute force & DDoS protection

🔐 Auth Server

- Login / Register handle karta hai
- Password verify karta hai
- Access + Refresh Tokens generate karta hai
- Session / token revoke karta hai

🧠 App Server

- Business logic
- Authorization checks
- Access Token verify karta hai
- Data serve karta hai

📩 Notification Server

- OTP
- Email
- SMS
- Alerts

🗄️ Database Tier

User Table

```
userId | name | email | role | createdAt
```

Credential / Auth Table

```
email | hashed_password | salt
```

Session Table (optional / hybrid)

```
sessionId | email | last_logged_in | status
```

🔑 Specific Flow — User Login

🔁 Login Flow (Step-by-Step)

1️⃣ Client `/login` hit karta hai
2️⃣ Auth Server:

- Password verify karta hai (bcrypt + salt + pepper)
- Access Token generate karta hai
- Refresh Token generate karta hai
  3️⃣ Tokens client ko milte hain (HTTPOnly cookies)
  4️⃣ Optional: session entry DB me log hoti hai

```
Client → /login → Auth Server → Tokens → Client
```

🔄 Normal Data Acces Flow

1️⃣ Client request ke saath **Access Token** bhejta hai
2️⃣ App Server:

- Token verify karta hai
- TTL check karta hai
- Role / permission check karta hai
  3️⃣ Data return hota hai

```
Client → App Server → (Verify Token) → Data
```

❌ **No DB call for auth**

⏱️ Token Expiry Flow

Access Token expire hota hai.

1️⃣ Client `/generate/token` endpoint hit karta hai
2️⃣ Refresh Token bhejta hai
3️⃣ Auth Server Refresh Token verify karta hai
4️⃣ New Access Token issue karta hai

```
Client → Refresh Token → Auth Server → New Access Token
```

🧠 **User ko pata bhi nahi chalta** (Gmail-style experience)

🚪 Logout Flow

🔁 Logout Logic

1️⃣ Client `/logout` hit karta hai
2️⃣ Auth Server:

- Session / Refresh Token revoke karta hai
- DB me token invalid mark karta hai
  3️⃣ Client local tokens delete karta hai

```
Client → /logout → Auth Server → Session Revoke
```

👉 Token invalid hone ke baad koi request allow nahi hoti.

🧠 Full End-to-End Mental Model

- Authentication → *Who are you*
- Authorization → *What can you do*
- JWT → *Stateless identity*
- Access Token → *Short-lived power*
- Refresh Token → *Session continuity*
- Password hashing → *DB breach defense*
- Rate limiting → *Abuse protection*

🧠 Lecture 10 — Final Revision (One Screen)

- AuthN ≠ AuthZ
- HTTP Stateless = root problem
- Sessions = stateful, scaling pain
- JWT = stateless, scalable
- Two-token system = security + UX
- Bcrypt + Salt + Pepper = gold standard
- RBAC / PBAC / ACL = permission models
- Rate limiting = security layer
- Complete architecture = real-world system

✨ **Made By @Harshal Chauhan** ✨