Websockets

DAY 2

- Web Sockets enable real-time communication by creating a persistent two-way connection b/w client & server.
- They are needed because HTTP can't send instant updates without repeated requests.

Working

1. HTTP handshake :- Client sends HTTP request. (For upgrading)
2. Protocol upgraded after server agrees. (TO web socket protocol)
3. Persistent connection
   (connection stays open, messages can flow both ways Continuously)

Used in :- Chat Apps, Live dashboard, Multiplayer games.

* Why not use just polling

- Polling means client repeatedly asks server for updates.
- Problems
  - wastes bandwidth, creates unnecessary requests, adds delay.

Web Sockets require :-

- Connection management
- Scaling strategies
- Load balancing support
- stateful session handling.

Authentication & Security

- Authentication – Login, identity verification (OAuth)
- Authorization – Access permissions. (OIDC)

* **OAuth 2.0**

Allows an app to access user data from another app 'without' knowing the user's password.

Flow :- * User clicks Login with Google
↓
App redirects user to Google → User approves access
↓
Google sends access tokens → App uses token to access Google APIs

* **OIDC** (Open ID Connect)

- OIDC adds Identity on top of OAuth.
- OIDC adds :- 1) ID Token (JWT) → contains user identity
  2) user info (email, name, picture)

Backend login flow

1. Frontend redirects user to Google → User logs in
2. Google returns Authorization code
3. Backend exchanges code for Access token & ID token.
4. Backend verifies ID token and then:
   - Creates user in DB (if new)
   - Issues own JWT / session
5. Frontend uses your backend token not Google's.

- ID Token is for Login, Access token is for APIs
- Always issue your own backend JWT.

JWT vs Sessions

working of both

Session

- User logs in.
- server creates session record
- server sends session_id (usually as a cookie)
- client sends cookie on every request
- Server looks up session → user info
- session data lives:
  - RAM
  - Redis
  - Database
- Auth. state is stored on server.

JWT

- User logs in → server issues JWT
- client stores JWT (cookie)
- client sends JWT in headers
- Server verifies token sign.
- Perfect for microservices, API Gateways and Horizontal Scaling.

Best Practice for backend architecture

1) OIDC login (Google, etc)
2) Backend verifies ID token
3) Backend issues short-lived JWT
4) Refresh token stored securely → API validates JWT

One liners

- Sessions are stateful, JWT is stateless
- JWT improves scalability, not security.
- Sessions are easier to revoke.
- JWT is ideal for APIs & microservices.
- Use short lived JWT with refresh token.

Hashing & Salting

Hashing :- • Converts data into a fixed length, irreversible value.
- One-way (cannot be reversed)
- same input → same output
- Backend never stores password — only hash.

Why hashing alone is not enough?
- If two users have same password ⇒ same hash.
- Attackers can detect common passwords, use precomputed hash.

Salting : Random value added to the password before hashing.
- hash (password + salt) (• Each user gets unique salt)
- So, if two users have same password, different hashes will be generated.
- salt is stored alongside hash.
- Security comes from strong hashing algorithm.
- Passwords must be hashed, NOT encrypted.

Never use :- MD5, SHA-1, SHA-256 (Alone)
why :- Too fast → easier brute force

Use :-
- bcrypt (built in salt, slow)
- argon2 (memory-hard, modern)

• • Slow hashing limits guesses/sec and makes brute force expensive.

- **Pepper** (advanced) → Secret value, same for all users, stored in env variable. If DB leaks, attacker still need pepper.

(Use in high security system)