**Concepts – HTTPS / SSL Certificates**

1. Case 1: Client sends plain text request, Server sends response (HTTP)
   1. Problem: Man in the middle attack. Hacker can sniff the sent data
2. Case 2: Let’s encrypt the sent data.
   1. Let’s use client’s key to encrypt. But, this key needs to be sent to server as well with encrypted data, then only server will be able to decrypt (Symmetric Encryption – same key used for encryption / decryption)
   2. Problem: Hacker can sniff the sent key, hence, can decrypt the data as well.
3. Case 3: Encrypt the key as well.
   1. Asymmetric encryption: use server’s public key to encrypt my own key, Server will decrypt using its private key. (Asymmetric Encryption – Public key – encryption, Private key – decryption)
   2. Problem: Hacker can act as proxy, impersonate server. Intercept server’s public key, send own public key to client. Client uses Hacker’s public key to encrypt its own key. This can again be intercepted by Hacker and decrypted by its own private key. To complete communication, again encrypt using server’s actual public key and send to server
4. Case 4: Need something (SSL certificate) accompanying the server’s public key, to ensure that it actually is server’s public key
   1. SSL certificate: Trusted Certification Authority(CA) uses its own private key and server’s public key to create a signature. Adds this and domain name in certificate and gives to server.
   2. Server shares Public key + Certificate to client. Client uses Server’s public key + CA’s public key to verify the signature.
   3. No sniffing/impersonating this time as to get CA’s public key for verification of certificate, we don’t make a network call, we only use public keys present in browser’s built-in list of trusted root CA certificates (contains their public key).
5. Notes
   1. SSL : Secure Sockets Layer
      1. Old, now deprecated
   2. TLS : Transport Layer Security
      1. New, better security and performance

**Concepts – Authentication & Authorization**

1. Authentication vs Authorization
   1. Authentication :
      1. Who are you?
   2. Authorization :
      1. What can you do?
   3. E.g. : Login to git (Authentication). Can only read repository, can’t edit, modify the code (Authorization/Permission)
2. Authentication
   1. Session vs Token Authentication(JWT)
      1. Session Authentication :
         1. How it works :
            1. Client -> Login (send HTTP POST request with id and password) ->
            2. Server -> authenticate -> create a session ID, store in its memory, and share Session ID in response ->
            3. Client stores Session ID in browser Cookies -> For subsequent requests, sends Session ID cookies alongside request ->
            4. Server checks Session ID, if found in its memory/database, serves data
         2. Problems :
            1. Stateful i.e. session information stored on server, hard to scale. Another server instance won’t have the session data, will have to re-login.
      2. Token based Authentication
         1. How it works
            1. Client -> Login (send HTTP POST request with id and password) ->
            2. Server -> authenticate -> creates a JWT token (JSON Web Token) by encoding user’s data with its Secret key and share in response.
            3. Client stores JWT in maybe cookies, sends it with subsequent requests
            4. Server validates if JWT is correct using its Secret key, if it is correct, servers data.
         2. Notes :
            1. JWT is self-reliant. It stores all the user’s info (e.g. user’s id, name, when was token created, expiry time, etc) –
            2. 3 parts of JWT :

Header (hash algorithm used, etc.),

payload (user data),

Hash (encode header.payload using secret key) : used for validation – this should match after recreating hash(header.payload) using secret key

* + - 1. Benefits
         1. Stateless

Can scale the servers easily, as no need to store any session data, as JWT is self relaint and is stored on client’s end.

All related servers can authenticate using same JWT as long as each of them are using same secret key. (e.g. After Adobe Inside page, no need to re-authenticate when redirecting to payroll site)

* + 1. A diagram of a server and a client

       AI-generated content may be incorrect.
    2. Links
       1. <https://www.youtube.com/watch?v=7Q17ubqLfaM&ab_channel=WebDevSimplified>
  1. OAuth / OAuth2 (Open Authorization) Authentication
     1. Problem it solves:
        1. Register : I don’t want to fill Sign-up form for each site for registering
        2. Login : I don’t want to sign in manually on each site. I don’t want to remember so many passwords for each site.
     2. Solution:
        1. Authorizing a third party(websites) to access **some of our data** like name, email, DOB, profile pic, etc
        2. E.g. :
           1. Sign-in with google, Sign-in with facebook
  2. SSO (Single Sign-on) Authentication
     1. One login for multiple services.
        1. E.g. in corporates – one login for slack, outlook, teams, ms office, tools, etc.

**Concepts – Base 62 encoding, MD5, SHA256 Hash**

1. Base 62 **encoding**
   1. Use either of 62 char set for each char
      1. 0-9, A-Z, a-z
   2. Deterministic, easy
2. MD5 **hashing**
   1. Returns 128 bit output (128 / 4 = 32 char long hexadecimal string)
   2. Prone to collision
   3. Very fast, hence original value can be found by brute force
3. SHA265 **hashing**
   1. Returns 256 bits (256 / 4 = 64 char long hexadecimal string)
   2. Lesser prone to collision (as longer)
   3. Slower, can’t find original value (Preimage resistance)

**Concepts – Rate Limiting Algorithms**

1. Fixed Window Counter
   1. E.g. 2 GB mobile data / day, **renews at 12 am** everyday (i.e. calendar day (e.g., midnight to midnight))
   2. Fixed Interval Windows, counter resets at beginning of new interval. New requests **dropped** if counter reaches the limit in a window.
   3. Problem: At window edges, can use 2 times the max capacity (i.e. burst of requests at windows are not handled properly)
2. Sliding Window Log
   1. Each completed request stored in a queue with timestamp. New request coming at time ‘t’ -> check all requests from time ‘t-windowSize’ to time ‘t’ (i.e. in the current window). If the no. of requests more than limit, **drop** the new request.
   2. During checking in queue, we pop out all the requests whose timestamp is less than the window lower boundary time.
   3. E.g. I wanna date a girl now, however I’ll see how many people have dated her in last 2 months **from now**. If 10 people have dated her, she is already at her limit, I’ll drop the idea, let her rest. Maybe after some time I or somebody else will check again.
   4. Problem :
      1. Extra memory to store completed requests.
      2. Extra overhead of checking and emptying queue might become a slowness bottleness in peak requests time.
3. Sliding Window Counter
   1. To solve problem of storing request + timestamp (memory usage), We still consider prev window, along winth current window(i.e. look back) to overcome fixed window problem
   2. just calculate it like this :
      1. New request -> check I’m at how much percent of current fixed window (e.g. if time 6 sec, I’m at 20% of 5 sec-10 sec window), if X% then we’ll take 100-X% from last/prev window.
      2. Formula :
         1. Completed Request = (100-X%) \* requests in prev window + all requests in current window until now
         2. Needs 2 variables only counter\_prev, counter\_cur;
         3. If completedRequests + 1 > limit, drop
      3. This is something between Fixed Window and Sliding Window log. Unlike sliding window log, it can still have bursts at edge of Last window, and if we are at almost end of current window, we’ll consider only a minimal % of that burst, thinking we still have lot to go.
4. Token Bucket (Most used)
   1. N buckets I token -> anybody can make a request if token is left for them in bucket.
   2. Token refill
      1. Option 1: Refill 1 token on completion of each ongoing request.
      2. Option 2 : Refill X tokens each time after a given time interval
   3. Uncatered requests can be queued or dropped
   4. Benefits
      1. Handles short term burst, as long as tokens are there in bucket (tokens can accumulate till bucket size)
      2. This is the most used rate limiting algorithm for APIs
   5. Problem :
      1. Choosing right parameters (bucket size, token refill rate) can be tricky, and impacts the performance quite a lot.
      2. Overhead of state management (token management). Implementing a globally consistent token bucket across multiple servers (distributed env) can be challenging.
      3. “Greedy client” issues : Some client can end up taking all the tokens, others will keep waiting
5. Leaky Bucket
   1. Uniform flow of requests
      1. Only as per the size of bucket leak, irrespective of how many requests entered the bucket
   2. Queue till bucket capacity, drop requests on bucket overflow
   3. Benefits:
      1. Used in streaming services for uniform, smooth flow
   4. Problem:
      1. Doesn’t allow even short term bursts

**Concepts – Forward & Reverse Proxy**

1. Forward vs Reverse Proxy
   1. Forward Proxy
      1. Hides client – sits in front of client
      2. E.g. Proxy app/sites that we use in mobile to hide our IP (bypass blocked sites)
   2. Reverse proxy
      1. Hides server – sits in front of server
      2. Other tasks : abstraction, traffic distribution, SSL termination, Caching, Authentication, Load balancing, Rate limiting
      3. E.g. NGINX, HAProxy

**Concepts – Cache**

* + - 1. Write Strategies
         1. Write Through Cache

Application server -> Write to Cache->Cache writes to DB

Not that good for write heavy systems

Benefits:

Faster Reads (data already present in cache, first time is not a cache miss)

No Stale Data

Cons:

Slow writes (needs to write to both cache and DB)

Dependency on Cache : Data loss / Unavailability if cache goes down

* + - * 1. Cache Aside

Application server -> Only write to DB, Application writes to Cache on cache miss during read

Good for Write heavy systems

Benefits:

Faster writes (only need to write in DB)

Cache independence (more Fault Tolerant)

Cons:

Slow Reads (first one is cache miss)

Stale data in cache

* + - * 1. Write Behind

Best, good of both worlds

Application server -> Write to Cache -> Cache updates in DB later **Asynchronously**

Benefits:

Fast Read

No Stale Data

Fast Write

Cons:

Cache Dependence (less Fault tolerant)

* + - 1. Read Strategies
         1. Read Through

Application server -> Read from Cache -> If found, return -> If not found, cache beings data from DB and updates itself, and return.

Often paired with Write through strategy

Good for read heavy systems.

Benefits:

Same as of write through strategy

Cons:

Less flexibility on data model/schema. (Needs to maintain same data schema in DB and cache)

* + - * 1. Cache Aside

Application server -> Read from cache -> If found, return -> If not found, **Application server** reads from DB -> updates cache -> returns

Benefits

Flexible schema (Cache and DB can have different data schema, as application controls write)

Cons

Slow reads

Stale data in cache (handled through cache invalidation techniques like TTL)

* + - 1. Links:
         1. Write strategies: <https://www.youtube.com/watch?v=aacJn6sczIY>
         2. Read strategies: <https://www.youtube.com/watch?v=F4Qy4nR_guw>

**Concepts – API**

1. HTTP
   1. Used for Client Server communication
   2. Client initiated
   3. Request -> Response
   4. Types :
      1. GET : Get data
      2. POST : Add new data
      3. PUT / PATCH : Update an existing data
      4. DELETE : Delete an existing data
2. HTTPS
   1. HTTP + Encryption (TLS/SSL)
3. gRPC
   1. By : Google
   2. Used for inter service communications
   3. Fast as data payload size less due to Protocol Buffer (Probuf : Shorthand / shortforms to reduce data size)
4. GraphQL
   1. By : Facebook
   2. Used to customized requests (fine grain control over what resources are needed, can get all in a single call)