Foundational topics 1

19 September 2024 21:39

Problem → Build System for showing online & offline Users

User 1 •
User 5 •
User 4 •
User 3 •

(Not thinking about Scaling)

→ Ways to Approach a System design Problem

→ Spiral

decide the core & Build around that

ex: Start Schema & Build around it, get Bigger

✓ we are using Spiral for the current Problem

Incremental

when we are unsure & how to implement

① Start w/ dog zero archi
② See how each component would behave → under load → at scale
③ identify the bottleneck
④ re architect
⑤ repeat

→ When the System is Simple enough → go w/ Spiral

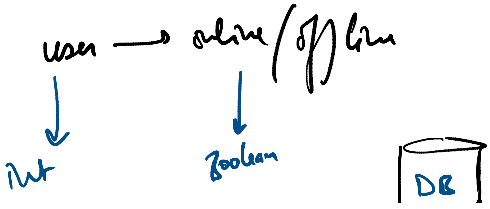
→ complex Systems require incremental → Startups work like this

Remember

→ understand the core property & access pattern
→ don't stick to a particular tech aka don't be rigid
ex: I'll always use Postgres
I'll always use graphQL

→ Build an intuition towards Building Systems

Storage



Int Boolean

Access : Key Value
↳ Access Pattern

DB
(Not decided)

Why?
↳ we can fetch details for particular users
↳ So each id can be a Key
hence Key Value

Interfacing API

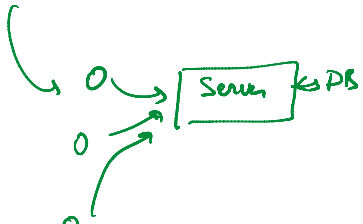
GET /Status/Users ? ids = u1, u2, u3 ...

↳ design decision
→ it is better to have a endpoint that can handle Batch requests
→ individual Request per user is a poor experience for the frontend

→ how to know if user is online

Pull Method
↳ Server asks every client if they are alive
→ HTTP is stateless which means NO persistent connection
So, Pull Method is Not possible

Push Method
↳ client tell Server if they are alive



→ each client will periodically send a "heartbeat" to the Server

POST /heartbeat

when we

We can decide per use case when we want to declare the user dead or the server.

like ⇒ No heartbeat in 30 sec, then Kill

↳ this means we need to store when we received the last heartbeat of every user

Pulse

user_id	last_hb
u_1	1000
u_2	1010
u_3	1500

when you receive last heartbeat

```
UPDATE pulse
SET last_hb = now()
WHERE user_id = u1
```

get status API

so our GET status becomes more complex

GET /status/<user_id>

→ ⇒ No entry in DS → offline

→ ⇒ entry, but last_hb is >30 secs old → offline

Let's estimate the scale

100 users → 100 entries
1000 users → 1000 entries
1m users → 1m entries
1B users → 1B entries

Each entry has 2 columns

user_id , last_hb
↓ ↓
int (4B) int (4B)

Size of each entry = 8B

Total storage required for 1B entries = ~~8 GB~~ 8 GB

Can we do better on storage?

Requirement : user 1, user 2, user 3 — all we care is online/offline

what if absence == offline ?

Idea: if user not present in the DB, we return offline.

So, let's expire the entries after 30 seconds
↓
delete

if we delete entries → we save a bunch of space
by not storing data of inactive users

Total entries = Active users

if 1B total users and 100k active then total entries = 100k
total size = 800KB

How to auto delete ?

Approach 1: Write a CRON job that deletes expired entries

- Not a robust solution
- We need to handle edge case in the business logic

Approach 2: Can we not offload this to our datastore ?

Never re-invent the wheel !

DB with KV + expiration → Redis
(Key value) → DynamoDB

Upon receiving an heartbeat

- update entry in Redis / DynamoDB with ttl = 30 sec

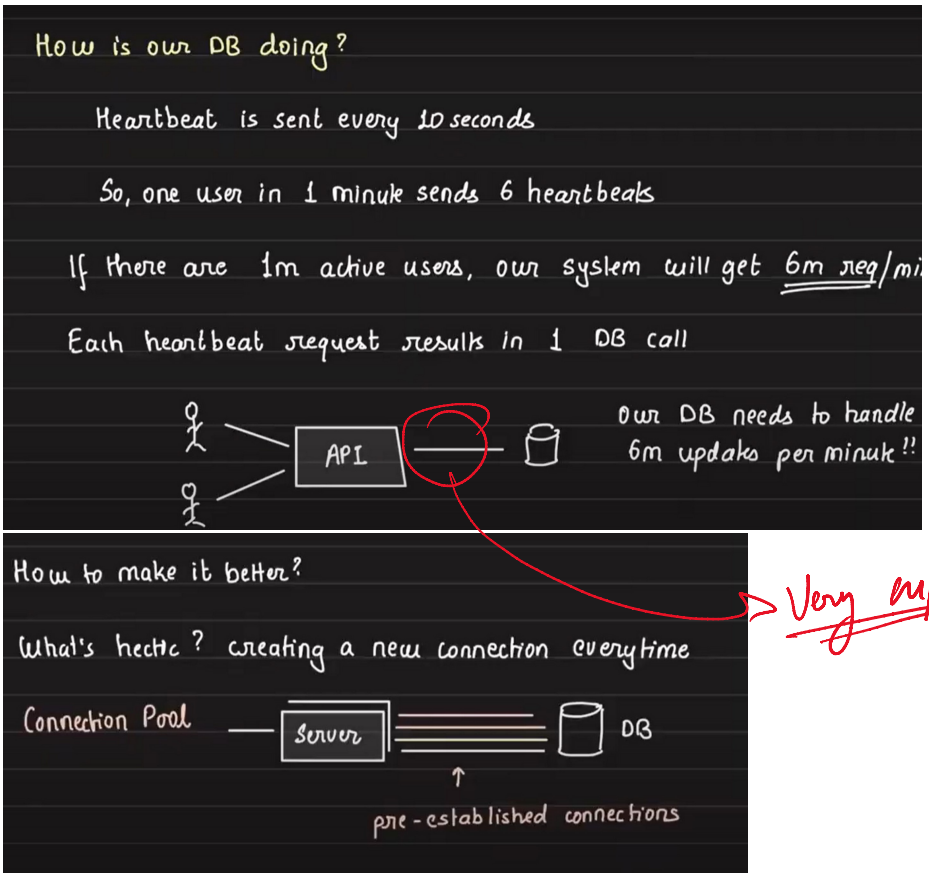
Every heartbeat move the expiration time forward !

Which one would you pick & why ? $ Cost Estimation

Which one would you pick & why ? $ Cost Estimation

Redis	DynamoDB	
✓	✓	persistence
	✓	managed (* team size) ? expertise
✓		vendor lockin (cost / issue)
✓		future extensibility
✓		time sensitivity

Always know your DB well
→ features
→ drawbacks etc



The microupdates that we were about to do for each online/offline event would require a new tcp connection to be established between server and DB, and then a teardown would be required (3 way handshake + 2 step teardown = 5 network calls for each update)

Setting up this connection is expensive, tearing it is also expensive.

Instead, we will use connection pool
Ex: we will setup some tcp connections with DB when server starts up, and maintain these connections. We will use these connections for the updates, and wont care about tear downs. This will help reduce setup+teardown, we will just use existing connections if available. This will reduce computation resources on both server and DB. Also, latency will be low as connection is pre-established

ISSUES

We need to set a minimum and maximum connection pool size. Lets say minimum is 3. if 3 are occupied and one more comes then two things can happen

1 - the new update waits till a connection is available
2 - a new connection is established

In case 2, the new connection established will not be terminated, but rather added to the connection pool, which basically means we are stretching the pool on demand.

We should limit this stretching, as the DB can only support so many tcp connections.

What if there are multiple servers, all with pooling? Will the DB be able to handle it?
For this, we should ensure unused connections are terminated after some time.

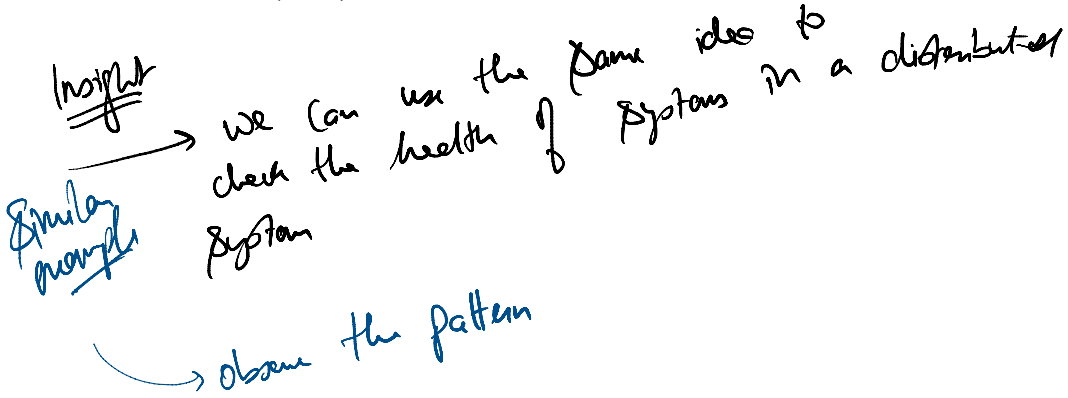
Remember, connection pooling is not a service, but a library/ way of doing things.

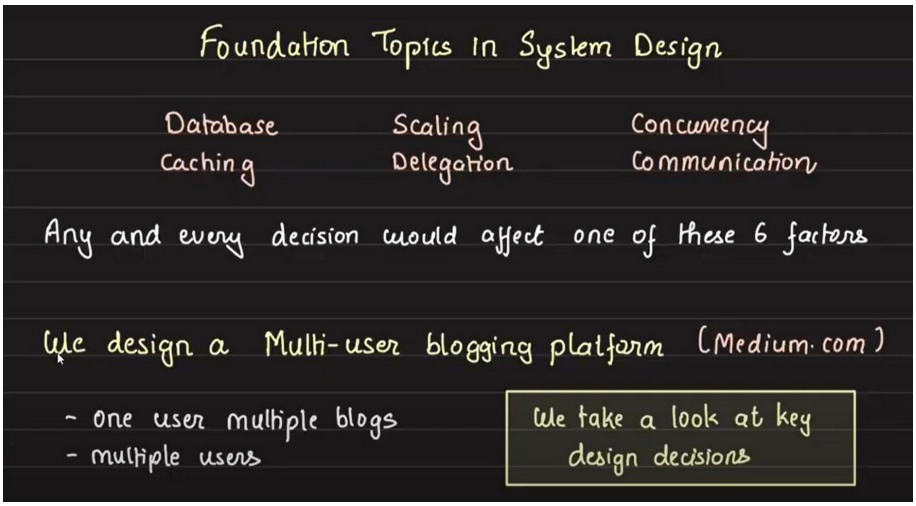
We have a blocking queue + multiple connections.

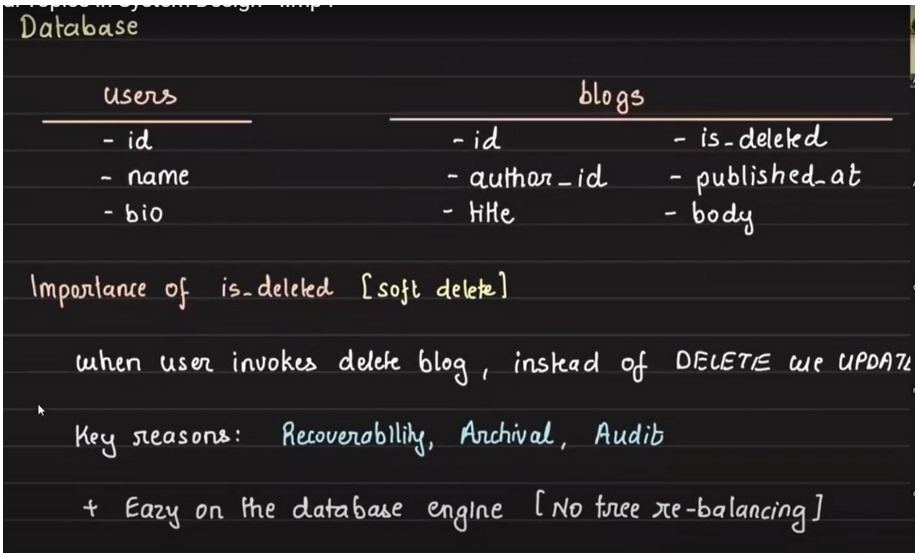
We can imagine this as

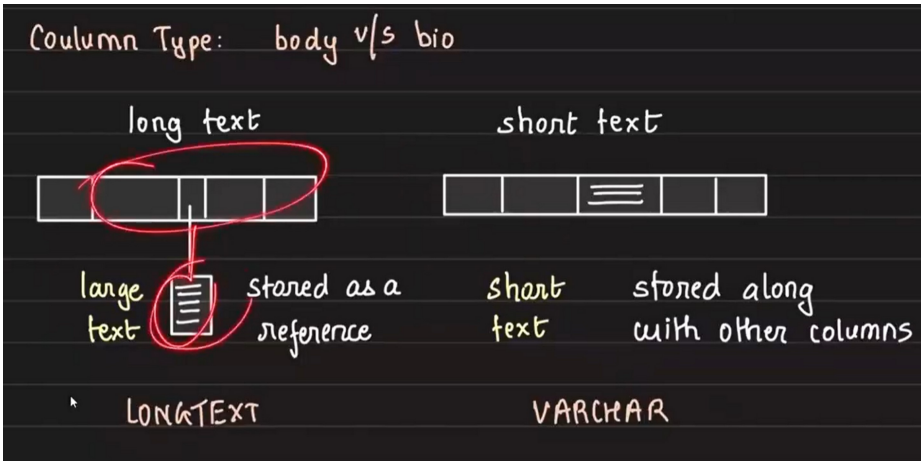
```
Const connection1 = DB.connect()
Const connection2 = DB.connect()
...

Queue.enqueue(connection1, connection2,...)

While(overCapacity) { wait() }
Process(queue.top()); queue.deque();
```

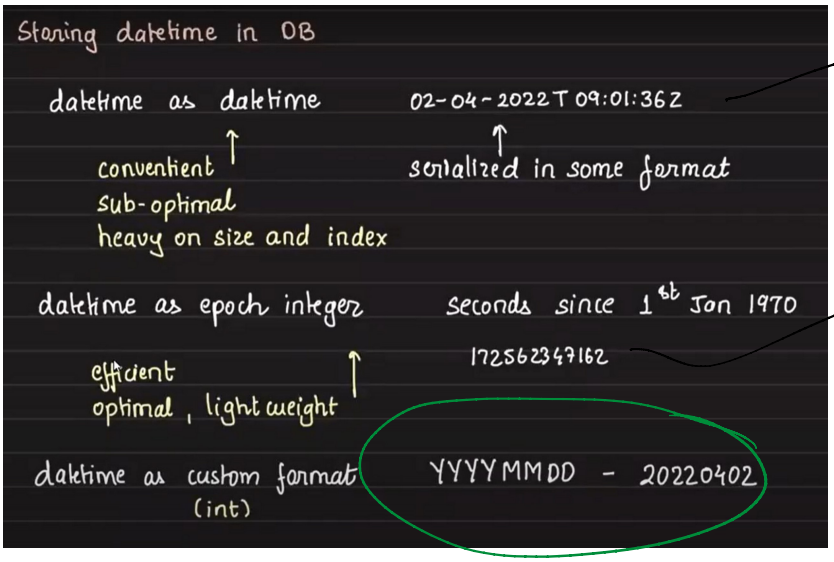
^ tukka maara hai but probably this is what it looks like











takes 20B to store

difference

takes 4B to store

RedBus initially had this, But ran into performance issues

didn't switch to epoch because they wanted readability.

→ So they implemented their own thing

→ They only needed date, so they ignored time & stored date as int

↓

fast & small

Caching

any

- Reduces response times by saving heavy computation

  * Cache are not only RAM based

Typical use: reduce disk I/O or network I/O or compute

Caches are just glorified Hash Tables

↓

with some advanced data structures