System Design

System : Architecture or collection of technologies that interact with each other to serve a set of users to fulfill set of requirements. It is build up of different components.

computing system

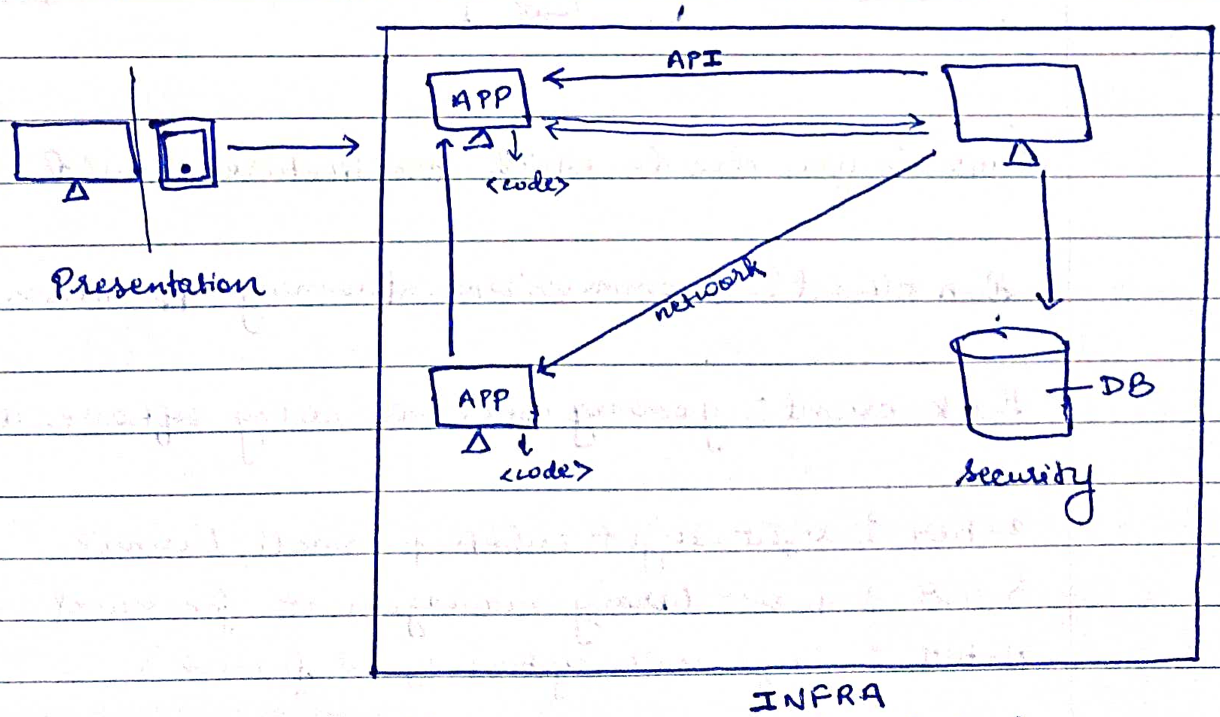
server, dB, caches, application etc. } insta, whatsapp, netflix, hotstar

real world systems

building, school, hospital, temple { walls, floors, ceiling, etc

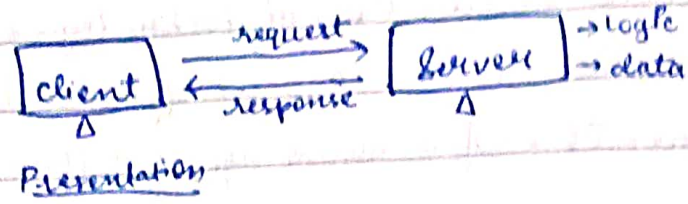
components of system design

① logical entities	② Tangible entities
data	Text, images, video
database	MongoDB, MySQL, Cassandra
Applications	Java, Golang, Python, Amber, React
Cache	Redis, MemeCache
message queues	Kafka, RabbitMQ
infra	AWS, GCP, Azure
communication	APIs, RPCs, messages..



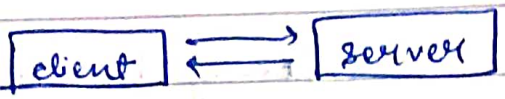
Basic Building Block of components of system

client server architecture

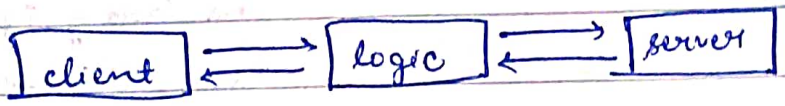


when logic & processing sits on client side = thick client
when logic & processing sits on server side = thin client

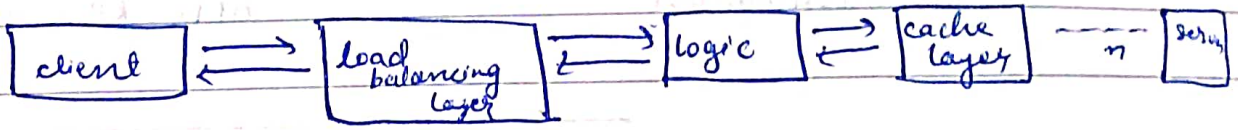
2-tier



3-tier



n-tier



how do you decide which architecture should be used?

- thin client: e-commerce sites, streaming application, etc.

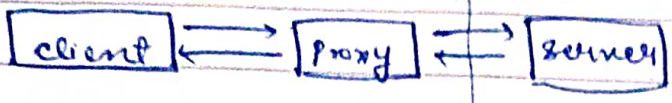
thick client: gaming apps, video editing software, etc.

2-tier : light weight website for small business
3-tier : basic library management for school
n-tier : large scale system (gmail, facebook)

Proxies:

on behalf

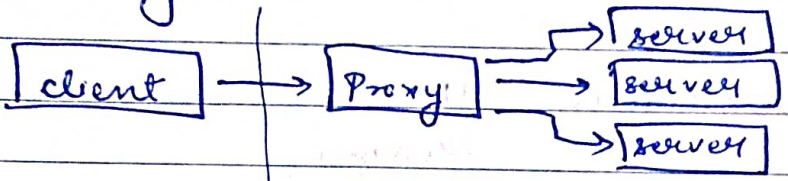
forward proxy: (on client side and talks to server on behalf of client)



uses:

1. disguises a client's IP address (anonimity of client)
2. Block malicious traffic from reaching an origin web server.
3. improve user's experience by caching external site content.

reverse proxy (on server side & talks to client on behalf of server)



anonimity of server

uses:

1. scrubs all incoming traffic before it's sent to our backend servers.
2. can efficiently distribute the load for applications using multiple servers
3. Deflates overall impact of distributed Denial of Services attacks
4. provides a single configuration point to manage SSL / TSL.

> Note: proxy servers can be used to bypass and get access to the blocked site (blocked sites by companies or institutions)
> ex: we cannot access spotify on IGDTUW wifi. It can be accesses by using proxy servers.

saves server from outer world

If reverse proxy fails, it becomes a reason for single point of failure

Data and Data Flow

Business layer : text / videos / images / notes
Application layer : JSON / XML - - -
DataStores (DBs) : tables, indexes, lists, trees
Network layer : packets
Hardware layer : 0s / 1s

DataStores :

		examples
Databases		[username, user's phone number, city, address]
Queues		[send sms request, send email request]
Caches		[request : Response]
Indexes		[most searched items, items searches in last 1 hr]

Dataflow Methods : APIs
Messages
Events

Data generation : → users
→ Internal
→ Insights

factors to be considered about data before considering system design

→ type of data
→ Volume
→ Consumption / retrieval
→ security

Type of system

1. Authorization
   (ex: user login, identity management)
2. Streaming (ex: netflix, hotstar, prime video, etc)
3. Transactional (ex: e-commerce sites, ride booking apps, grocery ordering apps)
4. Heavy compute systems: (ex: Image recognition system, video processing using ML models)

Data bases types: SQL, NoSQL, Column, Search, key value

types of databases:
- → relational
- → Non-relational
  - → key value stores
  - → column based DBs
  - → document based DBs
  - → search DBs
  - → etc.
- → file
- → Network
- → etc

factors used to decide whether to use relational dBs or not
1. schema (how your data is going to be structured)
2. ACID

If your data can be represented in the form of tables and rows while satisfying the property of relational db's like if your is complex and it could be represented using relational tables easily ⇒ relational dBs

A (Atomicity) (either transaction happens completely or doesn't happen at all)

C (consistent) (before query & after the query the data is in the same state)

I (isolation) (one query is not aware of other)

D (durability) (guarantees that transaction completed will persist permanently)

Relational dbs can be vertically scaled (→ increasing storage of one machine)

Non relational dbs (or NoSQL dbs)

- schema not fixed

① KV stores (key, values stores) (fast & provide quick access)

K	V
feature	true
discount	20%
enable city	true
request	response

② Document based

- when not sure of schema or how the data & the field different field of data are going to evolve over time.
- support heavy reads & writes

Collections (as tables)

documents (as rows)

ex: to store product details for an e-commerce website

disadvantage of document dbs

- don't have schema ∴ might have null or empty values so, you have to handle that in your application
- do not provide great transactions ∴ sometimes updates could become complex.

advantages of document db:

- highly scalable
- sharding
- dynamic data flexibility
- special query operations / aggregation

③ Column-wide stores

(= combination of relational db & document db)

- fixed schema with tables and columns
- do not support asset transactions
- heavy writes, special reads
- distributed

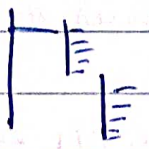
ex: storing health tracking data

storing data for iot device where diff sensors are deployed & sending data continuously

storages interactions in music app

eg: cassandra, HBase, scylla, etc.

④ Search DBs



(data stored in advance index similar to index of a book)

other use cases

→ images/videos (S3, Buckets)

→ large data sets

→ time series

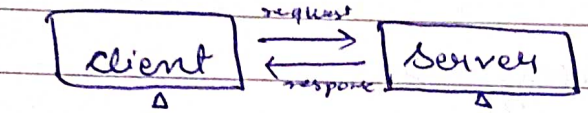
Databases

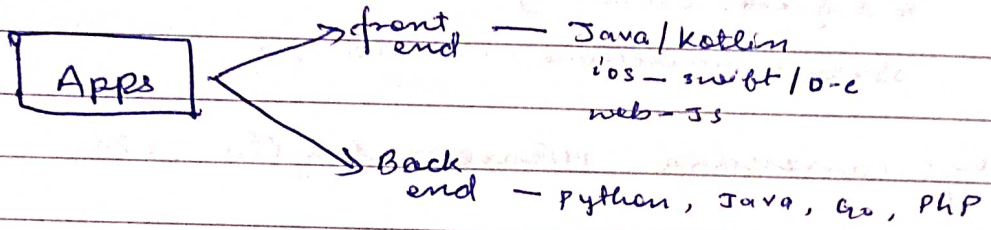
Relational
- fixed schema
- follows ACID properties

Non relational
- no fixed schema
- no ACID
- diff types of relational DBs are used for diff purposes

Key value stores | document based DB | columnar DB | graph DB

Application / Services





Most common features :

Client Apps

→ render UI elements
→ Handle instructions
→ collect data
→ Communicate with BE (APIs) to fetch/store data
→ Render static data/info

Backend Apps

→ Expose API and endpoints
→ House business logo
→ Handle data modelling / transfer
→ Interact with data stores

Elements / factors of APP design / development :

- → requirement
- → layer
- → Tech Stack
- → code structure / Design pattern
- → Data store interactions
- → Performance / cost
- → Deployment
- → Monitoring
- → Operational excellence / reliability

Monolithic architecture

managing notes
handling payments & transactions
user data & login authentication

} single application manages all functionalities

Microservice architecture

managing notes
Handling payments & transactions
User data & logic / authentication

} separate apps to manage separate functionalities

Application Programming Interface (APIs)

when applications have to interact with each other, they do so with APIs

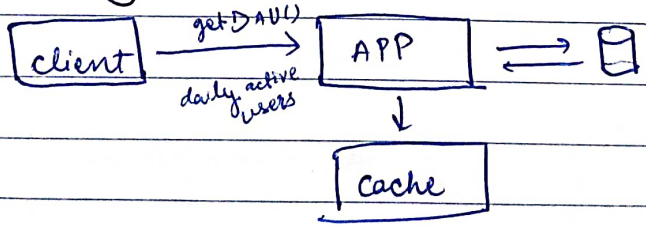
Adv :
- communication
- abstraction
- platform agnostic

Example :
- Private APIs
- Public APIs
- Web APIs (app running on cloud interact through API)
- SDK / library APIs (lock, fork, join, releaselock, etc)

factors :
- API contracts
- Documentation
- Data format
- Security

Standard :
- RPC
- SOAP
- REST

Caching Invalidation



process of updating cache is a cache invalidation

FIFO
LRU
LFU

cache-aside strategy / pattern
read through strategy / pattern
write through strategy / pattern
write around strategy / pattern
write back strategy / pattern

Rest API

REST = Representational State Transfer
(5 or 6 guidelines for data to be exchanged b/w client and server)

guidelines :

1. client server
2. cacheable
3. layered
4. Stateless
5. Uniform Interface
6. Code On demand (server can send code to client that it should execute on runtime (like Java Applets or JavaScript))

example :

Book Catalog (fetch, add, delete, update info of book)

Book
B1
B2
B3
:
B4

- get list of all book
- add a book
- remove a book
- update book information

CRUD : create, read, update & delete.

protocol | domain name | resource path

https://api.allaboutbooks.com/mystore/books

Representational State of Resource

GET <server domain>/mystore/books — JSON response

{B1, ... B4}

Ex: GET (HTTP method) https://api.allaboutbooks.com/mystore/books (URI)

response data [{
name : <book_name_1>,
author : <author_name_1>,
price : <price_1>
},
{
name : <book_name_1>,
author : <author_name_1>,
price : <price_1>
},
{...},
{...}
]

POST ---- similar

POST (HTTP method) https://api.allaboutbooks.com/mystore/books (URI)

payload { id : B5
name : <book_name_5>,
author : <author_name_5>,
price : <price_5>,
publication : <publisher_5>,
count : 100
}

response { status: 200,
id : B5
}

PUT (HTTP method) https://api.allaboutbooks.com/mystore/books/B1 (URI)

payload { count : 95 }

response {
status : 200,
id : B1
}

DELETE (HTTP method) https://api.allaboutbooks.com/mystore/books/B4 (URI)

response { status: 200,
id : B4
}

(of data) State transfer : state (of data) is being transferred

(of REST API) stateless : one server should not know about multiple clients

URI : <server domain> / mystore / books / id

/ ? limit = 20 & offset = 0

/ ? tags = fiction

HTTP Response

200 = success

201 = created

~~301~~

2xx = success

3xx = redirection

4xx = client error

5xx = Server error

1xx = informational

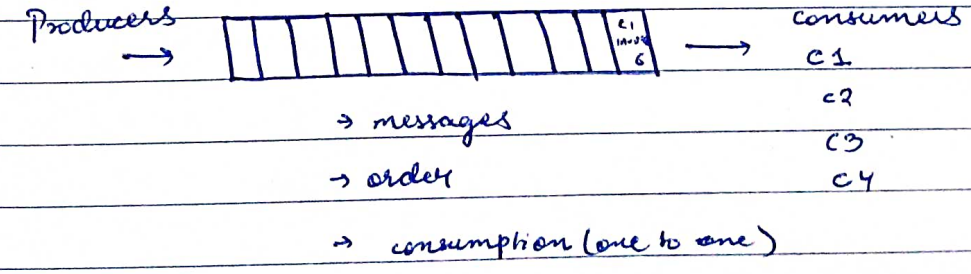
Message Queues

Synchronous vs Asynchronous Communication

Synchronous
(both parties are continuously exchanging information)
(ex: call)

Asynchronous
(continuous exchange of information is not necessary)
(ex: text message)

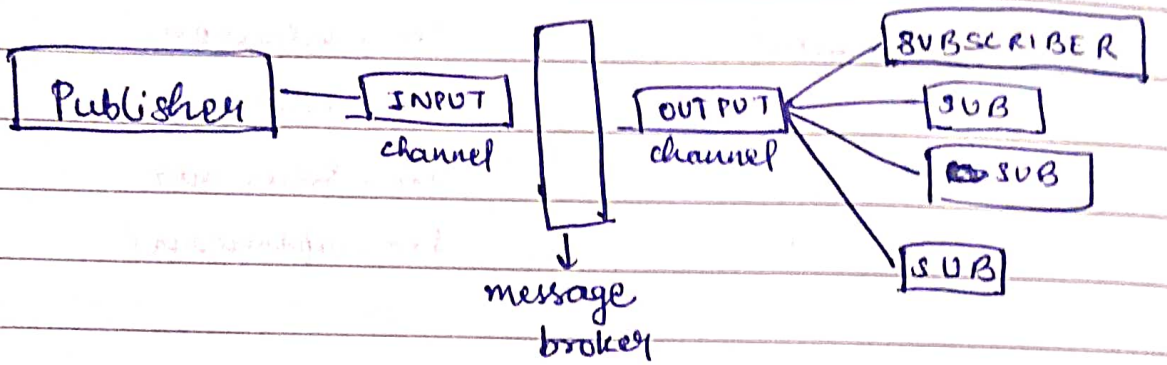
Asynchronous Communication



Pub-Sub Messaging

Since there is a single consumer for any particular queue, hence this can be viewed as Producer/Consumer Model.

Producer/consumer model : Only one staff member consumes a messag
Pub-Sub model : A chunk of staff members subscribes to a msg.



pub-Sub messaging in distributed systems

Performance Metrics

- throughput
- Bandwidth
- latency
- response time.

throughput : no of API calls served per unit time.

Bandwidth : (if you have bandwidth & resources used then throughput increases)

latency :

response time : (if you have response time then throughput increases taking care of the cost)

Performance of Components

(instances)

- Applications
- DBs
- Caches
- Workers
- Message Queues

performance metrics of an application

1. API response time
2. throughput of APIs
3. error occurances
4. Bug / Defect in the code

performance metrics of a database

1. ~~tab~~ time taken by various database queries
2. No of queries executed per unit time (or throughput)

performance metrics of a cache
① latency of writing to cache
② no of cache eviction and invalidation
③ Memory of cache instance

performance metrics of message queues
① Rate of production & consumption
② fraction of stale or unprocessed messages
③ no of consumers affects bandwidth & throughput.

performance metrics of workers
① time taken for job completion
② resources used in processing

performance metrics of Server instances
① Basic requirement
→ memory / RAM
→ CPU

performance metrics tools

Fault vs failures
(ex: bakery & the cake)

fault is the cause and failure is its effect
- understanding type of faults
- tolerating faults
- making systems fail safe

Types of faults → Transient fault (occur for a very small duration hard to locate)
→ Permanent fault (continues until fixed easily identifiable)

Scaling

→ increased load (handle it)
→ not complex
→ performance should not take a hit / increase.

Database replication

→ replication
→ replication lag

adv of synchronous replication (All replicas have to be updated before not acknowledged.)
- replication lag is zero
- data is always consistent in a system

Asynchronous replication
(Host is acknowledged after primary DB is updated. Replicas update asynchronously)

CAP Theorem

Distributed system (a system consisting of grp of machines working in coordination so as to appear as a single coherent system to the end-user)

Consistency (any read that is happening after a latest write, all the nodes should return the latest value of that write)
(data remains consistent across all nodes)

Availability (every available node in the system should respond in a non error format to any read request without the guarantee of returning the latest write)

(no guarantee of latest value)

Partition Tolerance (system will be responding to all read and write even if the communication channel (or middleware) b/w nodes is broken (or partitioned)

CAP Theorem also known as Brewer's theorem.

Data Sharding

Shards < logical / physical

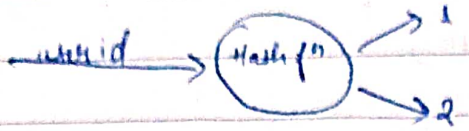
Sharding < algorithmic / dynamic

Vertical partioning (on the basis of columns)
Horizontal partioning (on the basis of rows) (Sharding)

adv of sharding
1. can large data can also be stored.
2. query has to scan only one part of data rather than searching whole data.
3. if you have one DB and it crashes. You will loose all its data

disadv of sharding
1. choosing sharding strategies (if not correct, whole data will not be evenly distributed)
2. hard to come back to non sharded structure.

Key based Sharding (algorithmic sharding)



every time hash value gives the same result for the same user

adv of key based ~~sharing~~ sharding
① data is evenly distributed

disadv of key based sharding
① when data inc & you have to add a new shards then there is problem cuz hashing f^n have to change & you have to move the data.

Range based sharding

adv:
- can have same db schema for all logical & physical shards
- (∵ no hashing f^n) can add more machines, divide machines and not have to do that much of moving around of data

disadv:
- lot of data may go in one range & other range may be empty.

use case:
- monthly data store
- alphabetically data stored.

Directory based Sharding (dynamic sharding)

lookup table holds information whether any row is present in shard.
(similar to a directory)

~~Sharding~~

System Design interview

Q: design twitter

figure out functional requirements & non-function requirements

functionalities provided by the system to fulfill user requirements

system level / component level needs / features that users are supposed to have

Ex: post tweet
delete tweet
fav. a tweet
follow people.

- if user post a tweet, it should take less than a second to be posted.
(latency of posting a tweet)
- system is available to handle certain no of DAU

f" req. APIs
Workers
Events
Messaging

non f" req:
choice of resource → capacity estimation
no. of resources → capacity estimation
availability.

1 million = 10^6
1 billion = 10^9
1 trillion = 10^{12}

Outcome of f" req: system design diagram
component & architecture diagram

Outcome of non-f" req: tech choices
resource utilization
data storages
& servers or hardwares

Capacity estimation

depends on
- no of transactions
- amount of data to be stored
- Availability

approximate always
- power of 10s
- power of 2s

1 million/day = 12/sec (transactions)
= 700/min
= 4200/hour

Capacity estimation of what?
- no of reads & writes
- no of transactions
- huge data transfer
- network bandwidth
- no of requests a server has to serve
- no of servers needed
- how a load balancer can handle these many requests