# System Architecture:

C A P S A R O U S E D  
  
Consistency, Availability, Performance, Scalability

Agility, Resiliency, Observability, Usability, Security, Extensibility, Durable

L E

Latency, Elasticity,

# Feed Design:

* To generate a new feed, the system must fetch the metadata (likes, comments, time, location and etc) of those photos and pass it to the ranking algorithm to determine which photos should be arranged in the newsfeed based on metadata.
* From the backend, it will require querying lots of tables simultaneously and then ranking them using predefined parameters, hence this approach will result in higher latency. It will take a lot of time to generate a newsfeed.
* So, the pre-generating news feed is adopted. We create a server dedicated to generating the newsfeed unique to each user and storing it in a separate newsfeed table. With this approach, when the user clicks the update, the newsfeed from the DB will be displayed to the user.

## Serving the feed:

* **Push** — when a new photo/video is uploaded by a user then, will update all of his/her followers. Long-pooling is used. If a user follows a lot of people or celebrities, the server has to keep push updates to the user quite frequently.
* **Pull** — Users will refresh their newsfeed (make a pull request to the server). The new post will not be visible until users don’t refresh.
* **Hybrid Approach**— Apply the **pull-based approach** for all the users with lots of followers or celebrities. Apply **push-based approach** for the normal users.

Important concepts while implementing any system:

* Traceability
* Distributed IDs
* Log aggregation
* API metrics
* Centralised configuration
* Localisation

**Traceability:**

* **Request ID** — a unique ID for the user’s request, generated at the point of user interaction. Pass this throughout the system in every flow of execution that has spawned from that original user request.
* **Requestor ID** — a unique ID for the user or system that initiated the request.
* **Correlation ID** — if a business process is composed of multiple requests, then aggregating these events under one correlation ID can be useful.
* **Session ID** — if an end-user is interacting with your system in the context of a “session”, then aggregating these events under one session ID can be useful. A session need not just be an HTTP session on a website but could be the running of a local desktop or mobile app.

Object Storage – To store photos and documents.

UML terminology

grpc

top down == function driven

how to identify microservices:

event driven

strict bounded context

each service owns its own data

SOA - Does not consider opex, Operations, Not for cloud

microservices came into picture for cloud

SOA - coarse grained services which communicate via Enterprise Service Bus.

Microservices communicate via language agnostic protocols via network.

###############################################################################

Inversion of Control

Separate what-to-do part from when-to-do part.

Ensure that when part knows as little as possible about what part; and vice versa.

There are several techniques possible for each of these steps based on the technology/language you are using for your implementation.

The inversion part of the Inversion of Control (IoC) is the confusing thing; because inversion is the relative term. The best way to understand IoC is to forget about that word!

--

Examples

Event Handling. Event Handlers (what-to-do part) -- Raising Events (when-to-do part)

Dependency Injection. Code that constructs a dependency (what-to-do part) -- instantiating and injecting that dependency for the clients when needed, which is usually taken care of by the DI tools such as Dagger (when-to-do-part).

Interfaces. Component client (when-to-do part) -- Component Interface implementation (what-to-do part)

xUnit fixture. Setup and TearDown (what-to-do part) -- xUnit frameworks calls to Setup at the beginning and TearDown at the end (when-to-do part)

###############################################################################

Points to be followed:

1. Every fuction should have single responsibility

SOA vs Microservice

Granularity: Course grained services Fine-grained services

Ease of Deployment: Requires recreating and redeploying entire application Each service can be built and deployed independently

Remote Call Overhead: Low communication overhead High communication overhead due to an increase in remote calls

Speed of Deployment: Slow deployment speeds Rapid, continuous, and automated deployment

Persistence: All services in SOA share data storage Each service is free to choose its own data storage

Ease of On-Boarding: Semi-difficult to onboard new developers as the scope of the entire application may need to be understood Easy to onboard new developers, as there is no need to understand the scope of the entire application

Communication Method: Communicates through an enterprise service bus Communicates via API layer with lightweight protocols like REST

Scalability: Can be challenging to scale Extremely scalable through the use of containers.

# How to store passwords in DB:

<https://www.vaadata.com/blog/how-to-securely-store-passwords-in-database/>  
password = 𝘩𝘢𝘴𝘩( 𝘱𝘢𝘴𝘴𝘸𝘰𝘳𝘥 + 𝘴𝘢𝘭𝘵).

Salt can be stored as plain text.

# Why to use JWTs:

JWTs are a good way of securely transmitting information between parties because they can be signed, which means you can be sure that the senders are who they say they are. Additionally, the structure of a JWT allows you to verify that the content hasn't been tampered with. JSON is used everywhere, so JSON parsers are available in all languages. JWT is signed, not encrypted.

# How does HTTPS work:

[HTTP](https://www.cloudflare.com/learning/ddos/glossary/hypertext-transfer-protocol-http/), is the primary protocol used to send data between a web browser and a website.

HTTPS is encrypted in order to increase security of data transfer. The encryption protocol is Transport Layer Security (TLS.)

It uses two different keys to encrypt communications between two parties:

The private key – owned by the owner of a website and it’s kept private. This key lives on a web server and is used to decrypt information encrypted by the public key.

The public key - this key is available to everyone who wants to interact with the server in a way that’s secure. Information that’s encrypted by the public key can only be decrypted by the private key.

Client 🡪 Webpage, Webpage sends back SSL certificate which contains public key back to the user. Then TLS handshake happens.

Then the client validates the certificate, then generates a session key and encrypts using the public key. The webpage server decrypts this key with its private key. Now the webpage also knows about the session key. Since the session key exists with both, this type of encryption is symmetric encryption.

# TODO Database Schema:

Foreign key is mapping an object reference to another object

# TODO SQL Basics:

<https://www.w3schools.com/sql/>

# API Gateway:

Client 🡪 HTTP Request 🡪 API Gateway  
  
API gateway then performs these checks:

API gateway validates HTTP request attributes.  
Checks for whitelist denylist  
Talks to identity provider for authentication/authorization  
Rate limiting  
After these checks, routes to the relevant service by path matching  
It also handles errors

# Databases:

CAP Theorem

Consistency Availability Partition tolerance

Not all the three would be true at the same time.

CP, AP, AC - are the only possible combinations

Availability: Every request received by a non-failing node in the system must

return a response, be it success or failure. Achieved by replicating data across different servers.

Consistency: In a consistent system, once a client writes a value to any server

and gets a response, it expects to get that value (or a fresher value) back

from any server it reads from. Achieved by updating several nodes before any read. Hence all nodes see same data at a given point of time.

Partition tolerance: In case of network failure, there should be no impact.

If a node fails and it is partitioned from the clusters, that is partition.

For a distributed system, the system will continue to work unless there is a

total network failure. A few nodes can fail and the system keeps going. Achieved by replicating data across combination of nodes/networks.

CP, AP, CA:

When a network partition occurs, either of the two hold true but not both.

Either the DB would be consistent, i.e., reads and writes are consistent,

but the nodes are not available to take new requests

or the nodes are available but the messages are not consistent.

If the system is not distributed, then it need not have partition tolerance, because there is a single node and no concept of partition.

so both C and A are valid.

CP - mongoDB, redis

AP – Cassandra, couch db

CA - MSSql, mariaDB

Database system designed with ACID guarantees (RDBMS) usually chooses consistency over availability

whereas system Designed with BASE guarantees, availability.

Relational databases strongly follow ACID principles:

Atomic: Everything in a transaction succeeds or the entire transaction is

rolled back.

Consistent: A transaction cannot leave the database in an inconsistent state.

Isolated: Transactions cannot interfere with each other.

Durable: Completed transactions persist, even when servers restart etc.

NoSQL databases follow BASE principles:

Basically Available, Soft State, Eventual consistency

Couch DB is eventually consistent in exchange for high availability.

CouchDB prioritizes availability, while MongoDB prioritizes consistency

https://medium.com/system-design-blog/cap-theorem-1455ce5fc0a0

Graphical user interface, application

Description automatically generated

Document – Horizontal scaling

Relational – Vertical scaling

Because of all sorts of table relations

* For **cache**— use a **key-value DB**.
* For **graph-**like data — use a **graph DB**.
* If you tend to query on **subsets of columns** /features — use **column DB.**
* For all other use cases — **Relational** or **Document DB**.

If the data is huge, querying the data would be a bottleneck. NoSQL suits well for huge data.  
No SQL is cost saving as it can be horizontally scaled and no requirement of dedicated machines

It effectively uses cloud.

# Data Partitioning(Sharding):

Divide large dataset into small partitions in different machines.  
Why do we need partitioning:

More manageability  
Increased query performance – instead of querying whole database, we query the partitions  
Scalability is easy  
Availability – as different partitions can be backed up one at a time

Partitioning criteria:  
Horizontal – Range based

Rows upto million in one table, and the next in another and so on.  
If the range is not chosen carefully the database size may be unbalanced.

Vertical  
One feature in one table, say User Info in one table.

# Cache:

## Cache invalidation:

If data is modified in the DB, cache needs to be invalidated.  
3 types:

Write through  
Write Around  
Write Back

Write through – Same time data is written to db and cache  
+ Data consistency  
+ Fault tolerance in case of failure  
- High latency as there are two writes to be done

Write Around – Data is written only to DB  
+ Not flooding the cache  
New data may not be available in the cache.

Write Back – Data is written only to cache, then written to DB after some time  
+ low latency, high throughput  
- chances of data loss

# Miscellaneous:

1. Writing keeps connection open for a long time.

Microservices When to Use:

1. Scalability
2. When the business boundaries are clear.
3. High speed delivery

# Resources:

https://blog.devgenius.io/top-10-architecture-characteristics-non-functional-requirements-with-cheatsheat-7ad14bbb0a9b