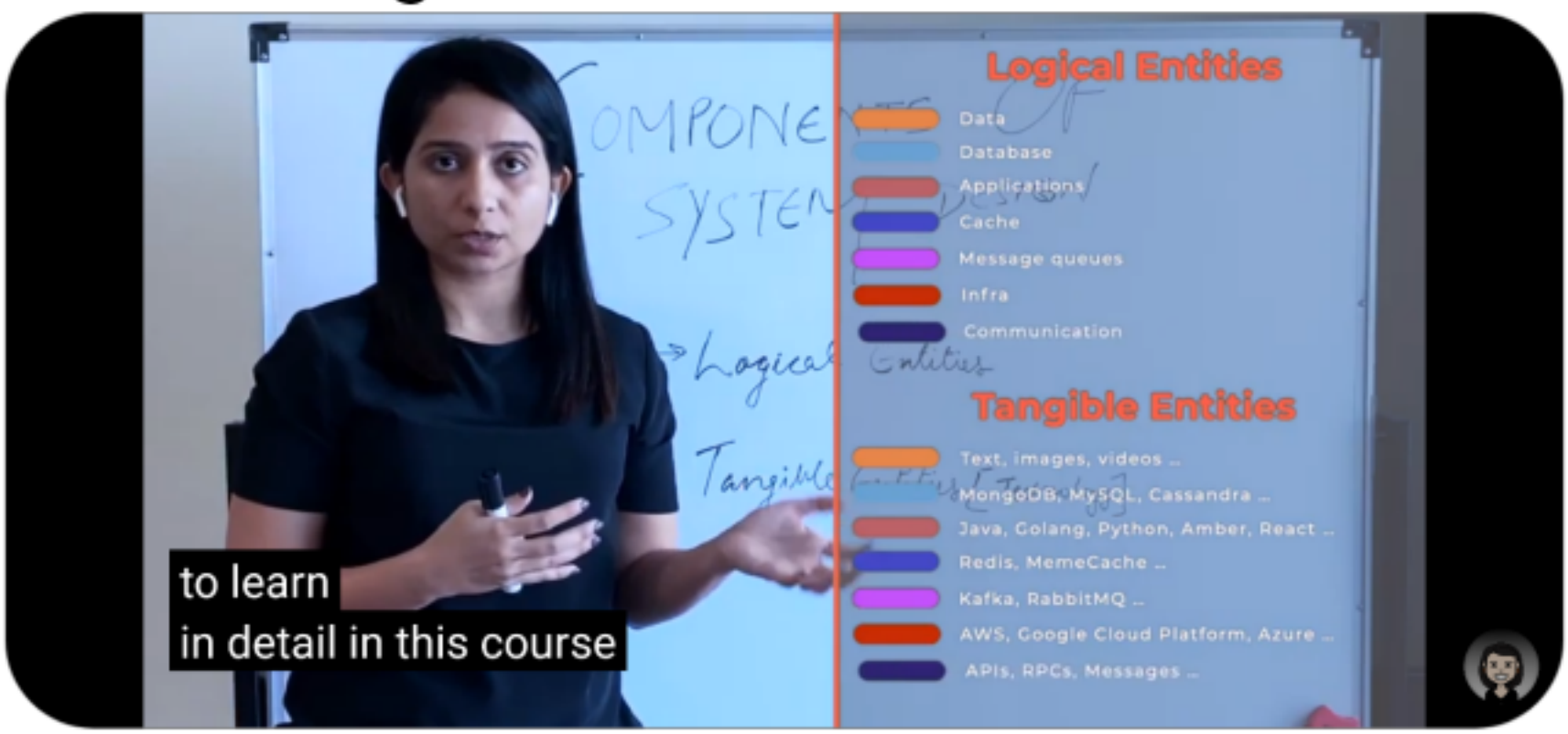
System Design

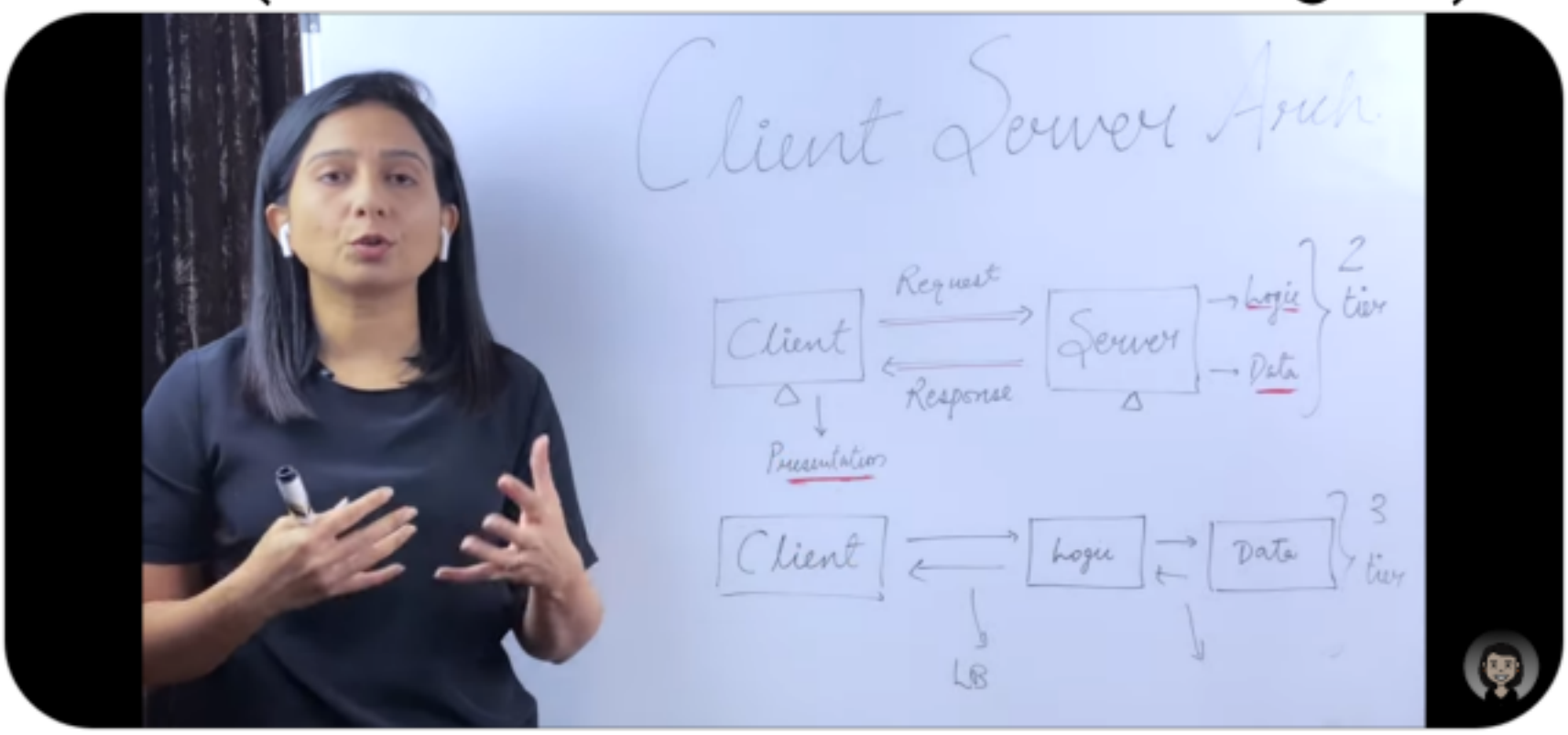
- System is a collection of software components which interacts with each other to serve specific set of requirements for specific set of users. Designing of such systems varies from one system to other.
- There are basically 2 components in a system. Logical entities and Tangible entities.

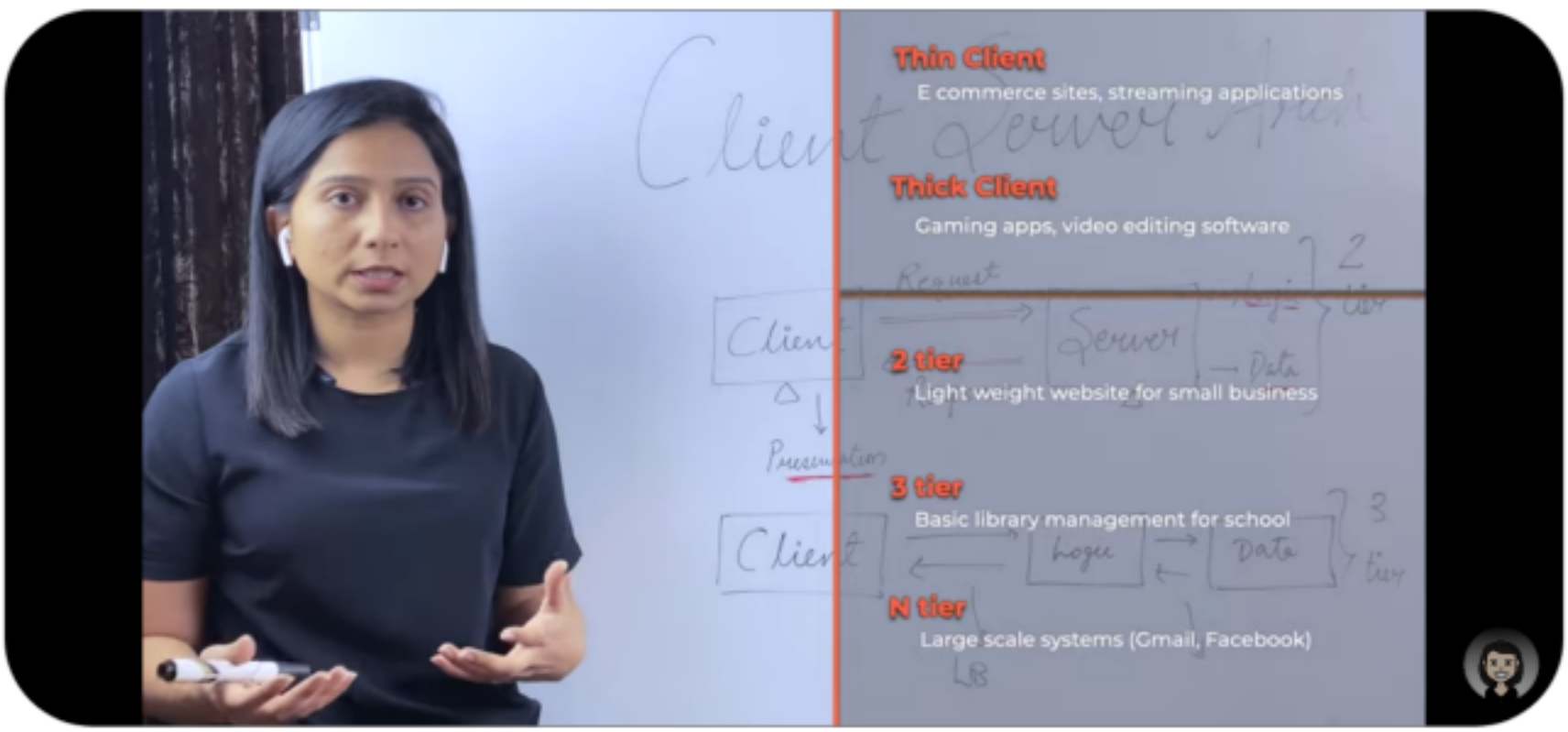


Client Server Architecture

- Client will request a server for

some data and server responds with that data. 2-tier, 3-tier(server with business and data) or n-tier(with LB, Cache layer etc) architecture. Clients can be either thick clients(where it has all business logics) or thin clients (where server has all the logics).





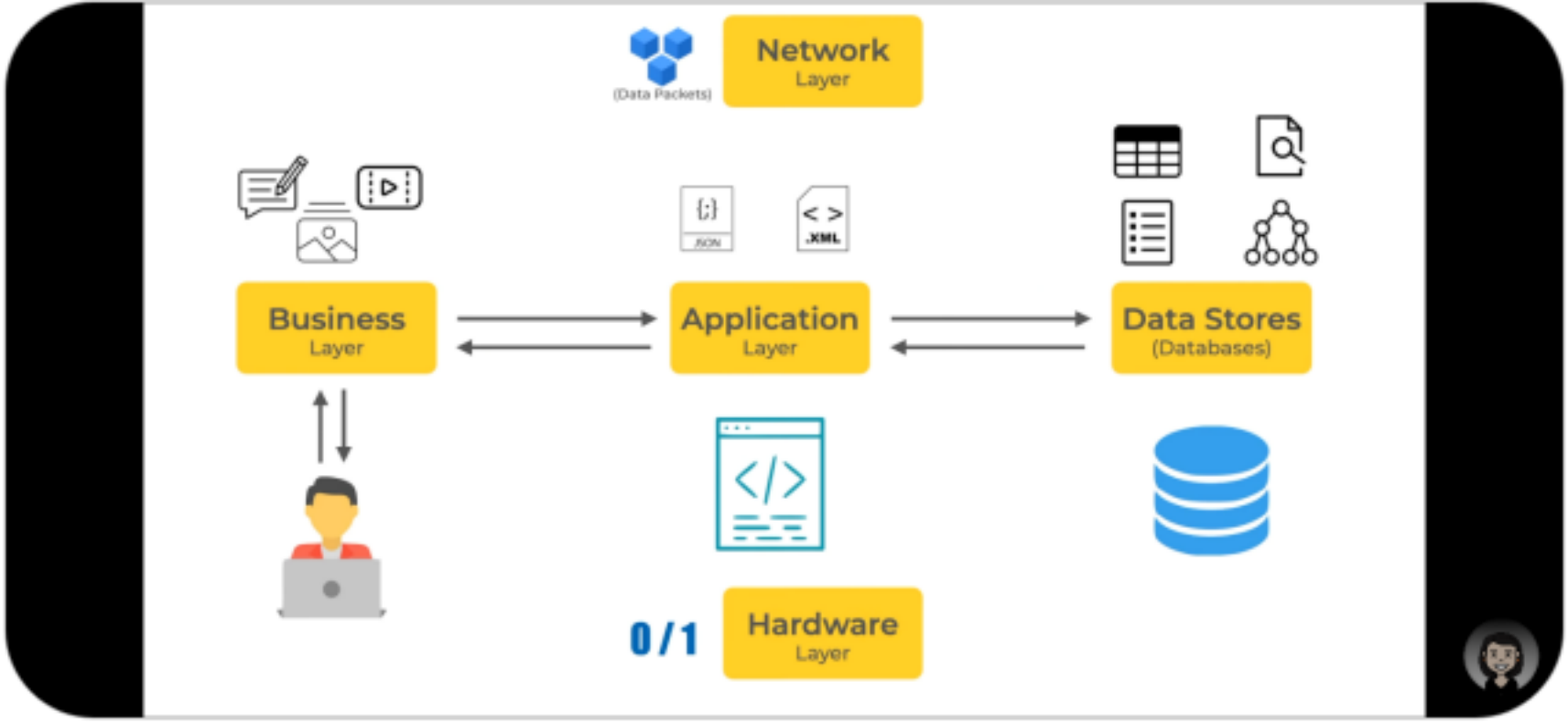
Proxies / Proxy Server

- It's the intermediary hardware or software component which sits between the Clint and server. It's used to filter requests. Like logging request, request transforming, security, privacy, traffic control, blocks set of web sites etc. We can use it for caching purpose which can serve lots of requests.
- **Forward Proxy** is the one which sits between client and server, towards the client side and handles the requests & responses. Here clients does not know IP of server it knows only IP of the proxy. Its used to blocking access to certain sites, traffic control with multiple clients etc. A **Reverse Proxy** is the one which

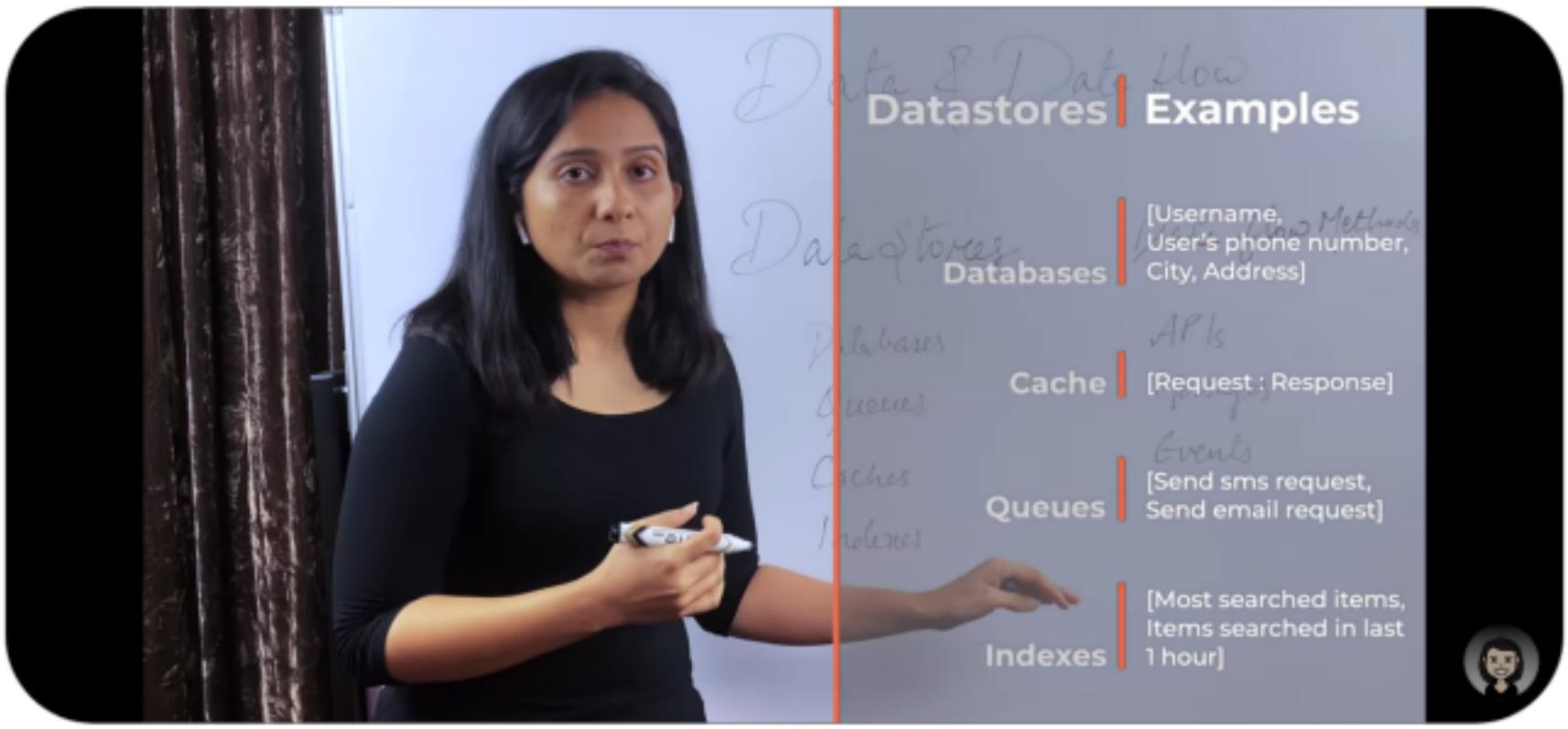
sits between a client & server, towards the server side / in the server side which talks to different servers and send back the response to client. Its useful for traffic control, load balancing, caching server response etc.

Data & Data Flow

- It decides whats our data format at each layer & how data gets transfered between different layers.



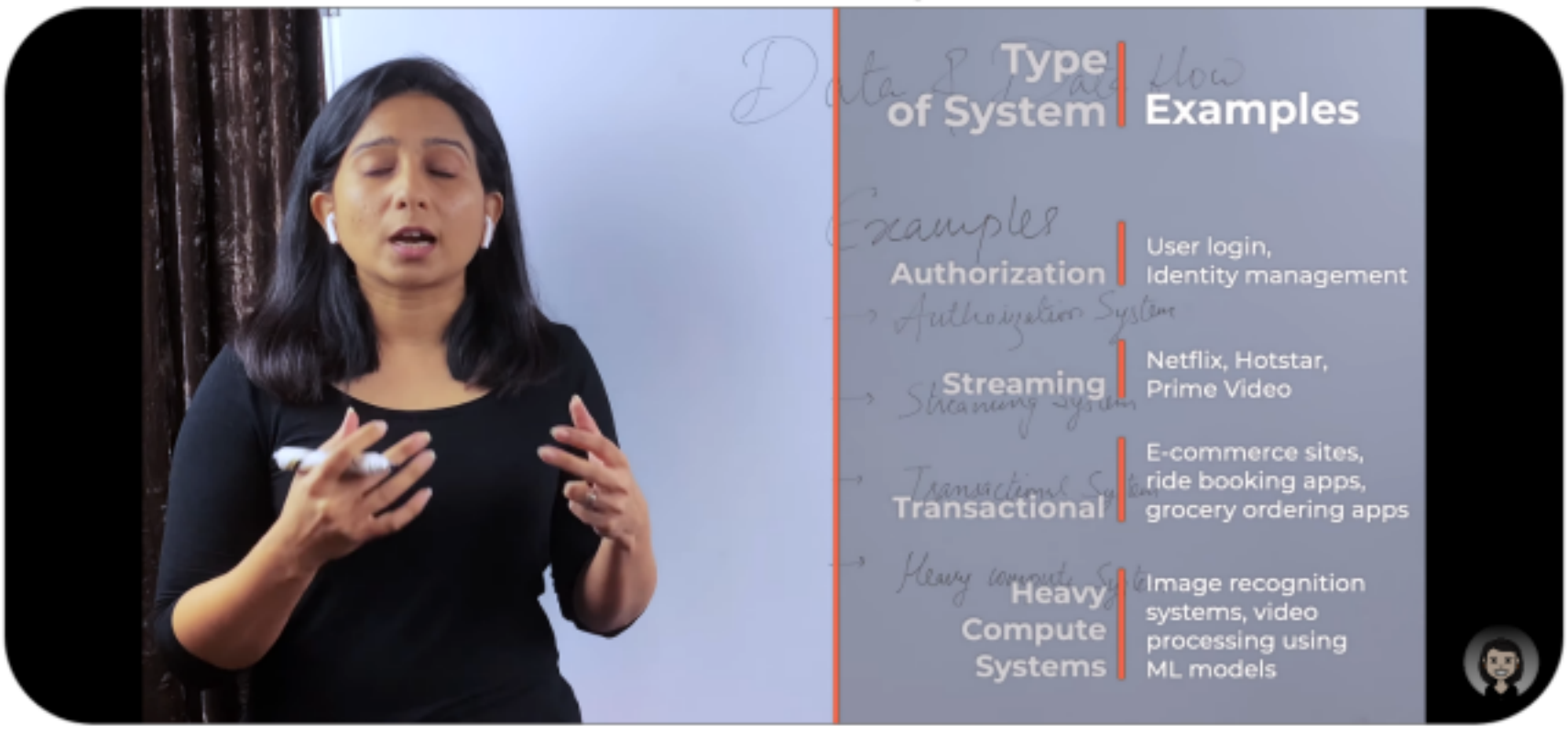
- Some of the data store options are databases (SQL/NoSQL), queues, caches & indexes. Some of the data flow options are APIs, events & messages.



- Common sources of the data can be users who feed data to systems, internal systems generated data like log data,

business data etc & insights like history of user data, payment history, subscription details etc.

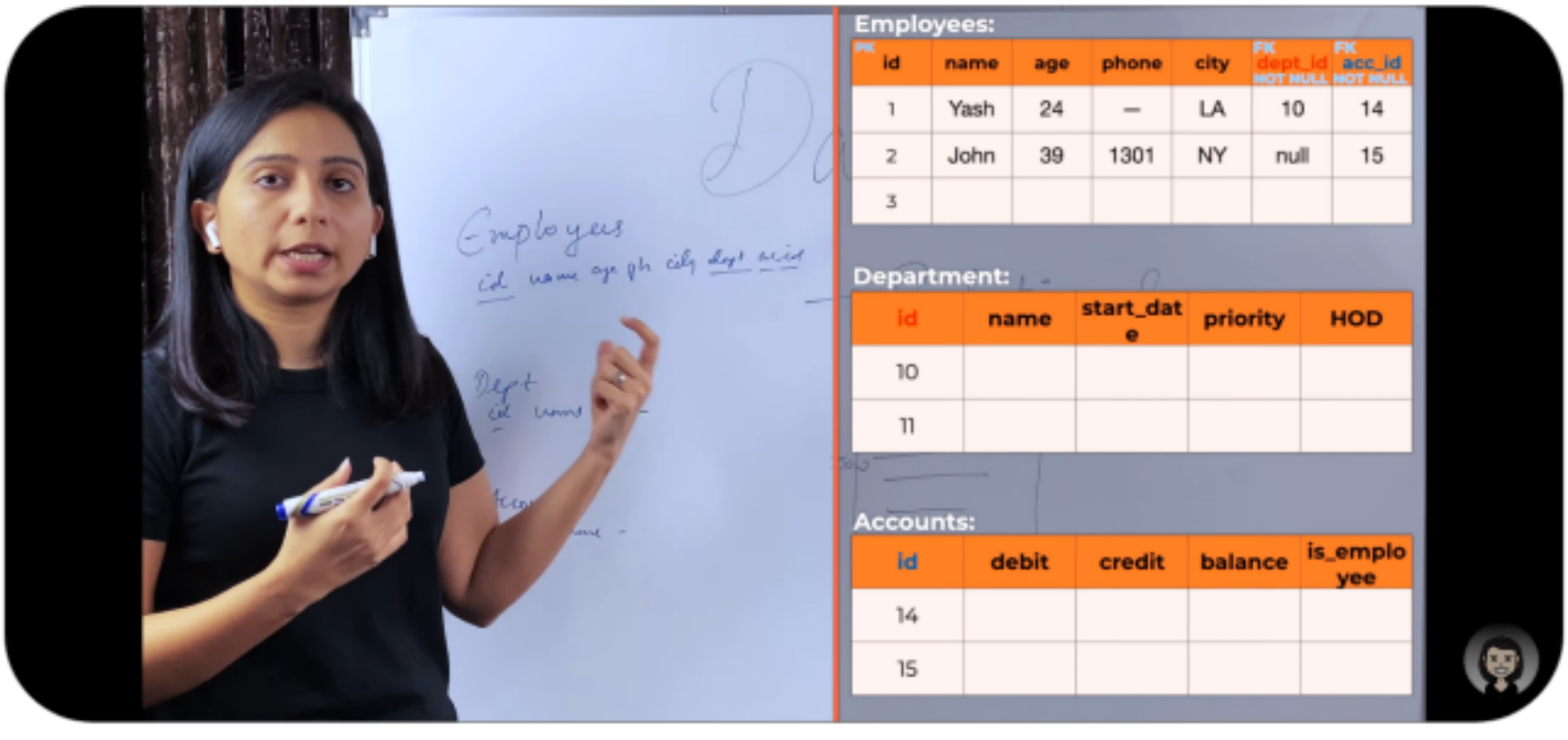
- Factors to consider while designing a system: Data type - text, audio, video etc. Total volume of the data, consumption / retrieval or read/ write, and security of the data.

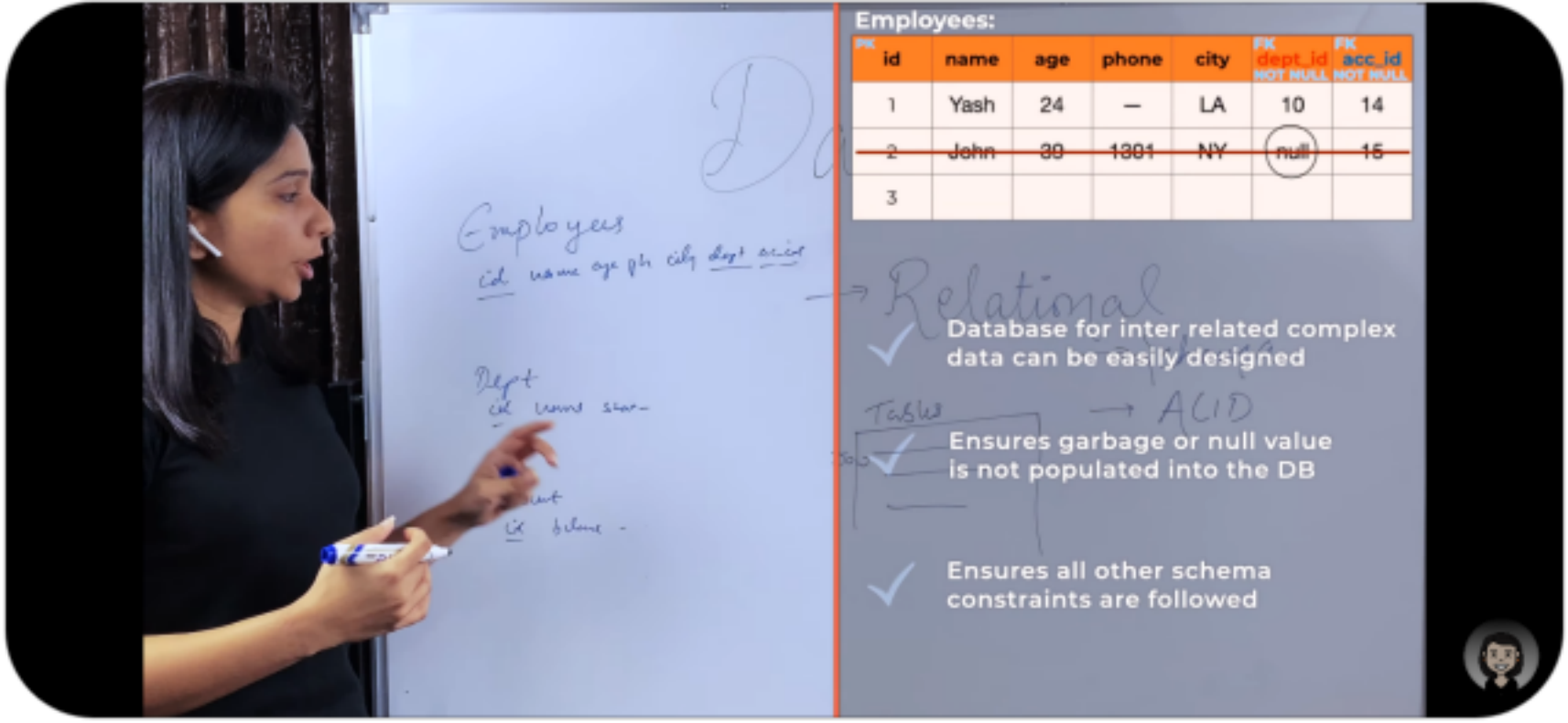


Database Types

- Relational, non-relational (key value store or caching, column based dbs, document based dbs, search based dbs), file, network.

- **Relational databases**: Two most important deciding factor for relational DB are schema & ACID properties. Schema defines how our database to be structured with tables, rows, columns and constraints. It also defines the relationship between entities.
  ACID - Atomicity (either commit as a whole or rollback), Consistency - 2 read operations gives the same data, Isolation - each transaction happens independently, Durability - storing data into disk.

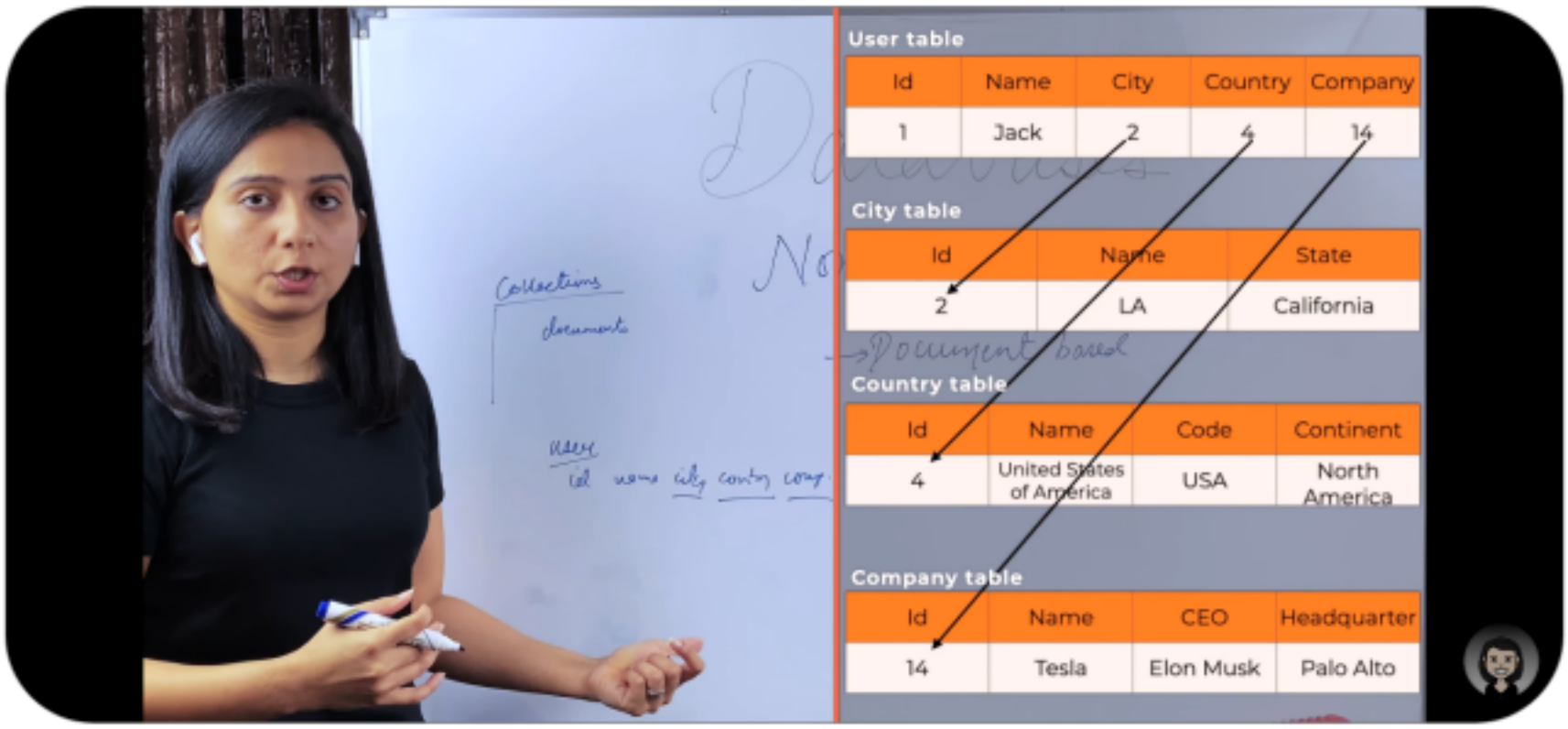


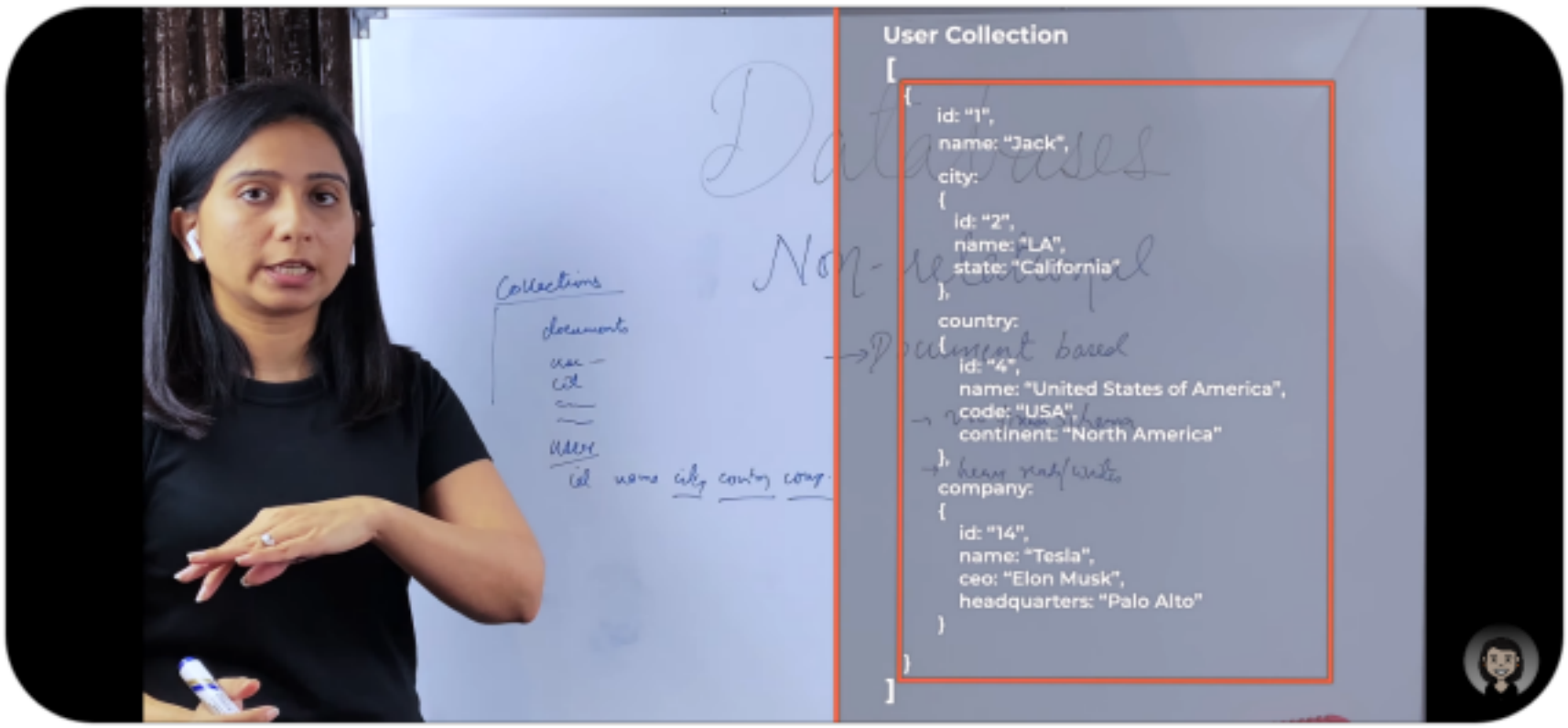


- Relational dbs performance will decrease as data grows. Also, it provides only vertical scaling option where we can increase its storage. Horizontal scaling means divide the table and put it onto multiple machines can not be achieved in relational dbs
- **Non-relational database**: To overcome relational DB issues we can use non-relational dbs. **Key-value** dbs are mainly used for caching purpose, which are quiet fast and stores data in memory. Ex Redis, DynamoDB. **Document** dbs provides no

schema and high read & write. Ex Mongo DB, Couch db. **Column** dbs are sort of midway between relational and document dbs. They have fixed schema with tables, rows and columns. But they do not support ACID transaction. Such dbs are used when we have the requirement of heavy writes. For ex iot sensor data, live stream data etc. It provides high support for distributed databases. Ex of such dbs are Cassandra, HBase. **Search/index** dbs will support full text search queries for the data stored in search dbs. Ex of such dbs are Elastic, Solr etc. The data stored in search dbs are not the primary data store. **Graph** dbs in which data are stored in graph structure with nodes, properties & lines. Example Neo4j, infinite

Grapgh.





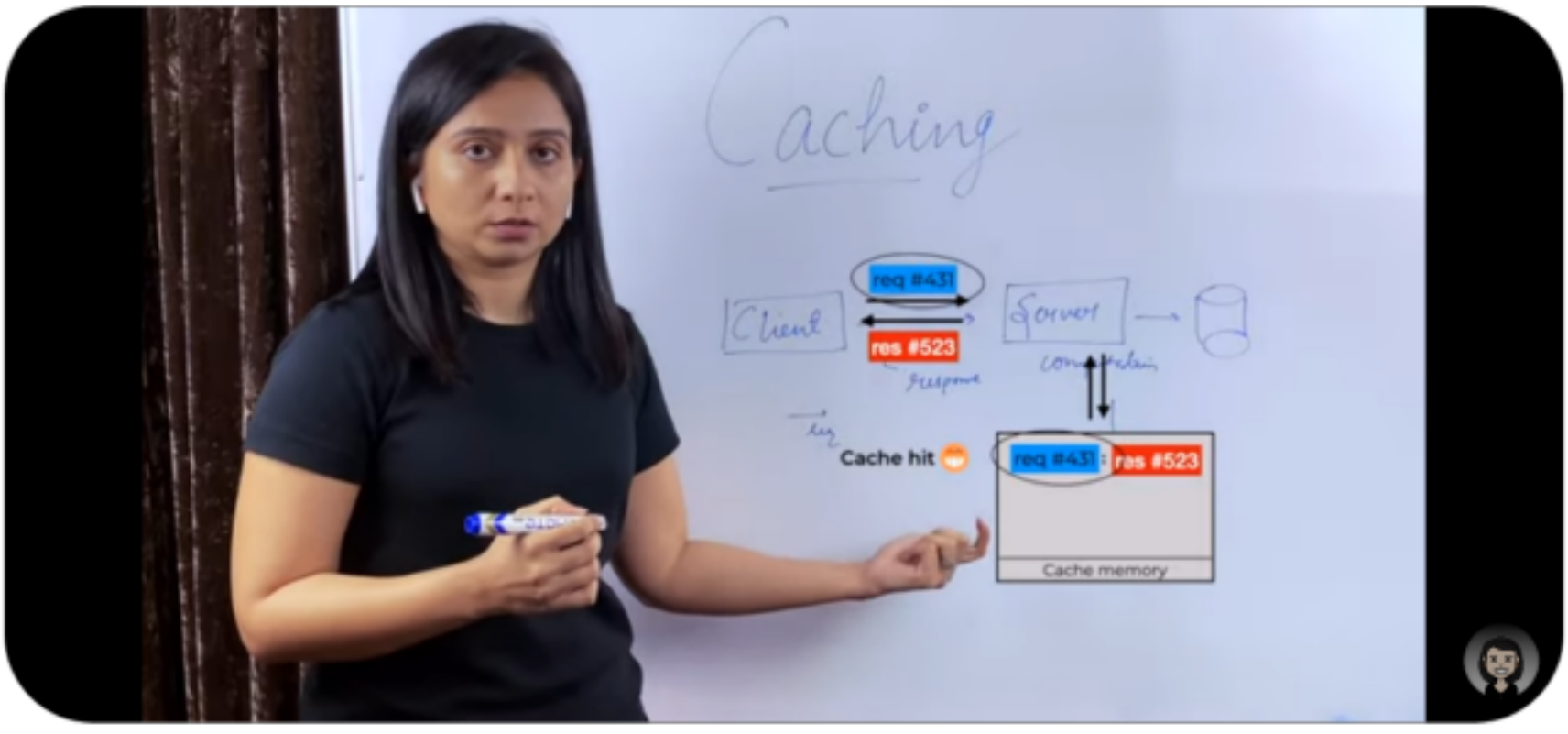
- Other use cases are like to store files, images & videos. Like s3 in amazon, blobs in Azure etc. Time series dbs like Graphite, Prometheus etc.

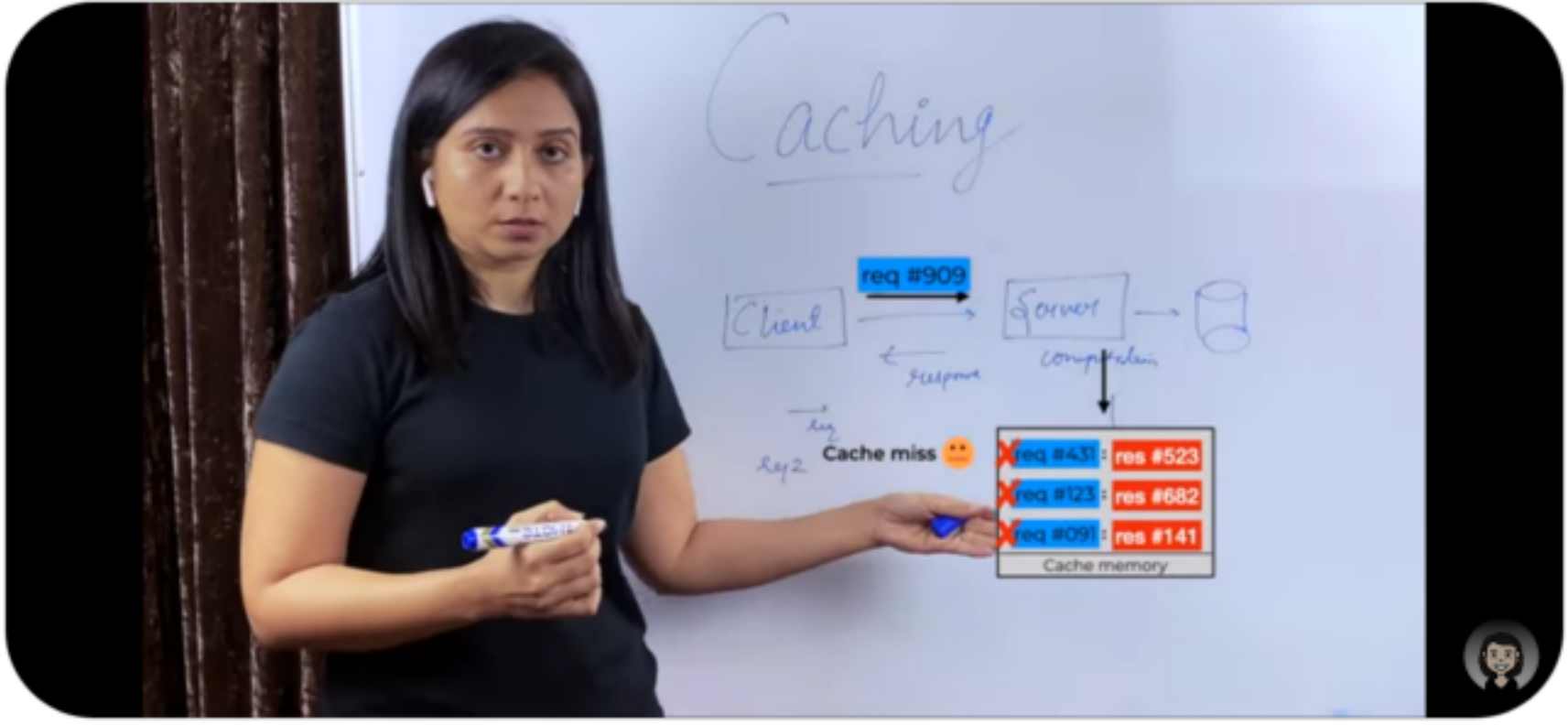
Caching

- It is like short term in-memory (on RAM not on disk) which has limited space but it is faster and contains most recently accessed items. It works on the principle that recently requested data is likely to be requested again.
- It can be used in almost any layer. Hardware, os, web browser, web apps, but are often found nearest to the front end.
- It can be a layer between server and database, which stores the

data for some period of time so that whenever the same client ask for same data, instead of recomputing it and fetching it from DB every time, we fetch it from cache DB and return it to client. We can use either forward proxy server or reverse proxy server for caching purpose.

- If we found the data in cache its called **Cache Hit else Cache miss**





- **Cache Types: Application Server Cache** - Each application instances has its own Cache which works as its local storage. But if we have multiple instances of this app, and we have a load balancer which sends request to any of these instances, which can increase the cache miss. To solve this problem we can use other types of cache.
- **Distribute Cache -** A distributed cache is a system that pools together the random access memory(RAM) of multiple networked computers across

different physical machines across different regions into a single in-memory data store used as a data cache. There will be single cache cluster which will have 2 or more cache servers. Data will be distributed evenly between these available servers using some consistent hashing function. Also, if we configure replication factor then each data will also copied to other servers so that if one server is down we can serve the data from other servers. We can also easily scale up by adding servers to cluster.

- **Global Cache:** All nodes use the same single cache space.
- **Content Distribution Network**(CDN): It is used to store and serve large amount of any static data. First client will ask CDN for static data, and if data is

not available in CDN it query the back end server & then cache it locally.

- **Cache Invalidation:** When data is updated in DB, it should be invalidated in the cache. The process of updating the existing data values or deleting & adding a new value in cache whenever the actual data value is changed in the system is called Invalidation.
- One option for this is to mention the expiry time (such as few seconds, minutes or hours) for each entry in cache through a method called **Time To Leave** (TTL). Other options are like removing the key from cache by Application itself, so there will be a cache miss happens and new fresh data will be fetched. Last option is, whenever there is a data update in DB then

application itself will update its corresponding value in the cache also, so cache hit will happens and the updated data will return. Also, we can have combination like TTL + data update.

- **Cache Eviction:** If the limit of the cache is lets say 1000 and when we want to add 1001th key into this cache, then one of the key has to be evicted so that room for the new key will be available. This process is called cache eviction. There are multiple ways to implement this cache eviction.
- **Cache Eviction Policies**: One of the popular strategy is **FIFO** (The one which is inserted first will be evicted). Other options are Least Recently Used (**LRU** - The one which is not used recently will be evicted), **LFU (**Least Frequently Used - the one which is less

frequently used will be evicted).

- **Cache Patterns:** One of the popular pattern is **Cache-aside** pattern. In this pattern cache never talks to DB. Application talks to cache and db to fetch existing value from cache or to add/remove/update values from db to cache. We use combination of TTL and application logic for the cache invalidation & eviction. Advantages is if the cache is down or failed then application can still serve data.
- Another pattern is **Read through and Write through** pattern. In this pattern cache sits between application and DB. Here application talks to cache which in turn talks to DB. Read through pattern is used when there are heavy read operations and write through is used when there are

heavy write operations. We can also use both of them together. Disadvantage is cache should have same data model as of db as it directly talks to DB.

- **Write around** pattern: Here application write directly to DB but uses a cache between apps and db for the read operation.
- **Write back** pattern: Here all write operations are done to cache which sits between app and db and later some time cache will write all those data in batches to DB. This pattern can handle db failure for some time.

Performance Metrics

- **Throughput**: Amount of work / data which can be done / transferred in a given time. In system it will be equivalent to number of API calls served per unit time. So a food app can serve 50 orders in 30 minute which is its throughput. Another food app can serve 100 orders in 30 minutes which is more throughput than previous.
- **Bandwidth:** Amount of data which can be transfered between different systems across the network. We can have high throughput only when we have high resources and high bandwidth.
- **Response Time:** Time taken by each API to send the response

back to clients. If this response time increases even though we have bandwidth the throughout will decrease.

- **Latency:** Amount of time in ms it takes to transfer data.

Fault vs Failure

- Fault is the cause and failure is its effect. So any component in a system can raise fault or breaks which will propagate that fault to entire system making the entire system to fail. So we should always have a fault tolerance and fail safe system.
- Faults can be at infrastructure level like no cpu, memory, network issue etc or it can be human error like bugs in the software.
- If there are any infrastructure

(hardware) level faults, we can have tolerance mechanism to it by having multiple instances of the same application replicated in different hardware so that if one instance failed to serve request, other instance can serve the same request.

- ○ To tolerate software related bugs developer has to write multiple types of tests. Also, we will show a friendly message in ui.

Cassandra

- ○ Add spring-boot-starter-data-cassandra dependency
- ○ @Table, @PrimaryKeyColumn(name, type=Partitioned/clustered, ordinal=0/1(for clustered type), ordering=ascending/

descending(for clustered type))

- ≡ ○ Application yml file to define key-space, contact-point and port
- ≡ ○ Create a new configuration class to create this key space and base path of the package where entities are exists
- ≡ ○ Create a repository which extends CassnadraRepository

Redis

- ≡ ○ Add the dependency spring-boot-starter-data-redis which internally has a redis java client library called jedis.
- ≡ ○ Create a new configuration class to create JedisConnectionFactory bean with host and port of Redis server and RedisTemplate by passing entity and JedisConnectionFactory

reference to it.

≡ ○ Create a normal repository to define crud methods. Get HashOperations object from RedisTemplate.opsForHash() and use it for the query like hashOperations.put()/get()/ entries()/delete() etc

---