@learnwithrockybhatia

Save For Later

System Design Interview Cheat Sheet

Interview Framework

Step 1
Understand the Problem

Identify and list down the functional requirements (what the system should do) and non-functional requirements (how the system should perform).

Step 2
Data Model and Storage:

Design the data model comprehensively, including defining the data schema, selecting the appropriate database type (SQL, NoSQL, distributed stores, graph databases), and planning how data will be stored, retrieved, and updated efficiently.

Step 3
API DESIGN

Define the API endpoints and methods for communication within the system and with external systems. Pay attention to the API contracts, request/response formats, authentication mechanisms, and communication protocols (REST, SOAP, GraphQL).

Step 4
High Level Architecture

Create a high-level architectural diagram that illustrates the components and their interactions with core services. Understand how data flows through the system, and how services communicate.

Step 5
Low Level Design

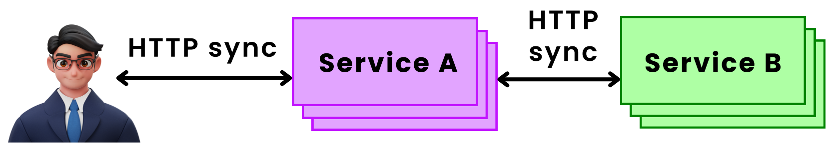
Dive into the low-level design details for each major component. Define data structures, algorithms, and implementation specifics. Consider optimization techniques, tradeoffs, and any potential bottlenecks. Address security and data consistency at this level.

API Design Choices

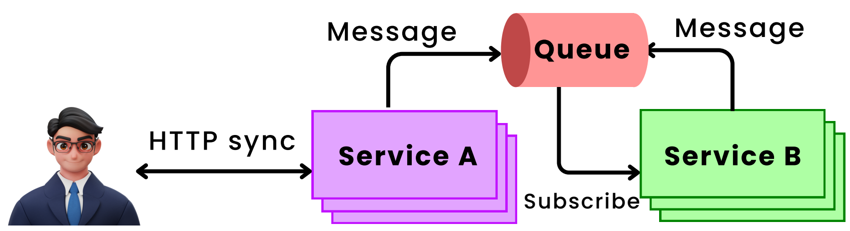
Explain how each part of the system works together. Start by defining APIs and the overall design patterns that your application will use.

	REST	RPC	GraphQL
	Resource	abc 0101 abc	
Properties	• Resource-oriented • Data-driven • Flexible	• action-oriented • high performance	• sinlge-endpoint • strongly-typed requests • no data overfetching • self-documenting
Data	JSON, XML, YAML, HTML, plain text	JSON, XML, Thrift, Protobuf, FlatButters	JSON
Use cases	• web-based apps • cloud apps • client-server apps • cloud computing services • developer APIs	• complexes microservices system • IoT application	• high-performance mobile apps • complex systems and microservice-based architecture

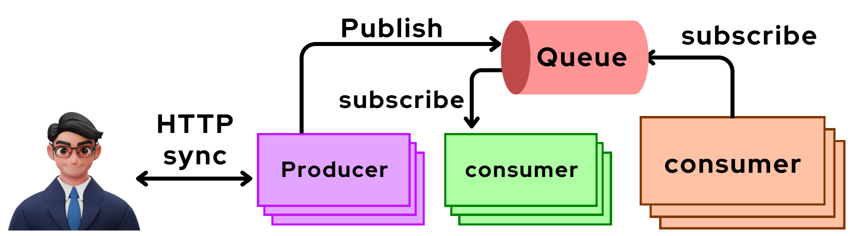
Synchronous



Async Messaging



Publish-Subscribe



Swipe next →

@learnwithrockybhatia

Save For Later

Which Database To Choose ?

Relational

	Good For	Use Case
SQL MySQL, Oracle PostgreSQL, SQL Server, Cloud Sql ,RDS, Cockroach DB, Yugabyte etc.	General Purpose SQL DB	Web Frameworks, ERP, CRM, Ecommerce and web, SaaS Application
New Sql Spanner Cockroach DB YugaByte Amazon Aurora OCI Azure Cosmos etc.	RDBMS+ scale, HA, HTAP	Gaming, Global Financial Ledger, Supply chain/inventory management
DataWarehouse SnowFlake , Redshift, Oracle, Synopsis, Sql Server Bigquery,Hive, Databricks, Teradata, Druid etc	OLAP, Analytics	Data-Mining Analytics, Reporting, BI

Non-Relational(no SQL)

	Good For	Use Case
Document Oriented Mongo DB , Couch DB, FireStore , Oracle , Azure Cosmos etc.	Large scale, complex hierarchical data	Mobile/web/ IoT application, Real-time sync, Offline sync, Personalized apps
Column Oriented Big Table Azure Cosmos Cassandra ScyllaDB	Heavy read + write, events	Personalization, Adtech, Recommendation engines, Fraud detection

In Memory

	Good For	Use Case
Memory Store Redis Memcached Hazelcast Ecache	In-memory and Key-value store	Caching, Gaming, Session store, Leaderboard, Social chat or news feed, Personalization, Adtech

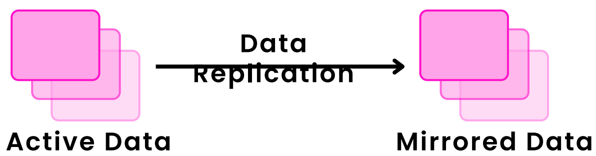
@learnwithrockybhatia

Scalability

Consider the scale of your system. How many users and requests will the server support? What happens with increased demand?

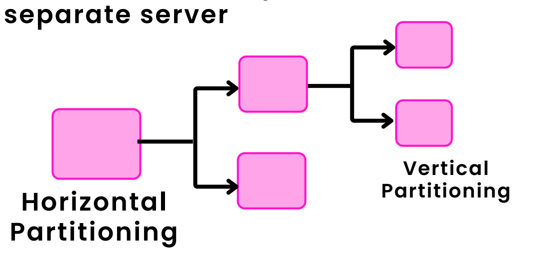
Replication

Is the data important enough to make copies? How important is it to keep all copies the same



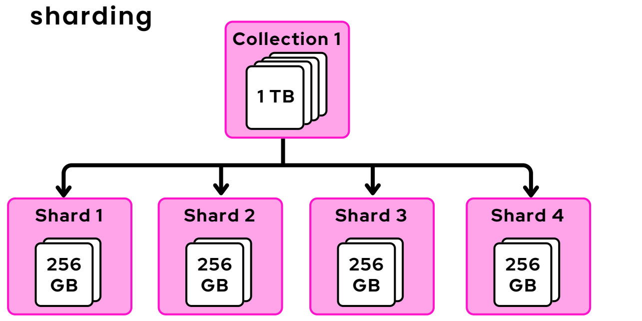
Partitioning

Partitions contain a subset of the whole table. Each partition is stored a separate server



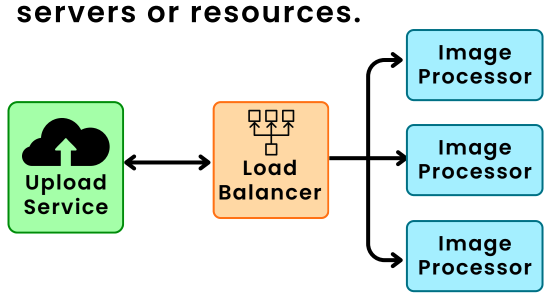
Sharding

Sharding allows a system to create as data increases, but not all data is suitable for sharding

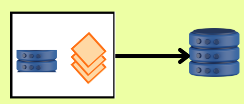
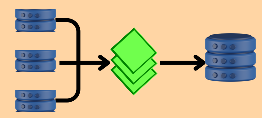


Load Balancing

Load balancing distributes incoming traffic across multiple servers or resources.



Caching

In-memory Cache	Distributed Cache
	
Latency- In-memory cache is faster doesn't require a network request like distributed.	**sharing data/ Consistency**- data can be shared across machines with a distributed cache. **Availability**- distributed cache is not affected by individual server failures

- No. Items
- Cache Miss & Hit
- Disk & Memory Usage

- Write-through
- Read-through
- Write-Around
- Write-Back

Popular caches:

- In-memory
- Redis
- Memcached
- AWS Elasticache
- GCP Memorystore

Eviction:

- LRU (Least Recently Used)
- LFU (Least Freq. used)
- FIFO
- MRU
- Random Eviction
- Least Used
- On-Demand Expiration
- Garbage Collection

- Storing user sessions
- Communication between microservices
- Caching frequent database lookups