System Design

What is System Design?

System design is the process of designing the architecture of a software system, which involves making decisions about its structure, components, connections, data, and how it will scale. The goal of system design is to build a stable, scalable, accessible, and efficient system that can meet the needs of users well.

Why System Design is Important?

Imagine you are a software engineer tasked with designing a system for managing 10 million players, grouping them into teams of 10 based on their skill level. Given that an average of 10,000 players are online at any given time, querying the database each time a player comes online or goes offline to change it status would be inefficient. Additionally, real-time changes in player status could lead to inaccuracies in matchmaking.

Optimized Approach

To ensure both speed and accuracy, we can implement a **queue-based matchmaking system** using an **in-memory database like Redis**. Here's how it works:

- Players are placed into **skill-based queues** stored in Redis when they start matchmaking.
- A **matchmaking service** continuously pulls players from these queues to form teams of 10.
- If a queue lacks enough players, a **timeout mechanism** expands the skill range slightly to allow near-skill-level players to be grouped together.
- **Redis Pub/Sub or WebSockets** can be used to track real-time player status changes, reducing database load and improving accuracy.

Comparison Table

Feature	Direct Database Querying	Queue-Based Matching with Redis
Performance	Slow, due to frequent DB queries	Fast, as Redis operates in-memory
Accuracy	Can lose accuracy due to real-time status changes	More accurate with Pub/Sub updates
Scalability	Limited, as database queries increase with players	Highly scalable with distributed Redis instances
Latency	High, as database operations take time	Low, as Redis fetches data in milliseconds
Flexibility	Hard to adjust matchmaking criteria dynamically	Supports dynamic skill-range adjustments with timeouts
Resource Usage	High database load	Efficient memory usage with minimal DB writes

By leveraging Redis, we improve matchmaking efficiency while reducing strain on the database. This ensures a fast, scalable, and accurate player grouping system.

author : Yasin Karbasi

Chapter 01 - Basics of Distributed Systems

What is Distributed Systems?

in distributed systems, software is divided into independent and scalable components. These components and services collaborate to function as an integrated system. Typically, they are networked and communicate with each other to provide a unified service

Features of Distributed Systems

- **Transparency** – Users should not be aware that the system consists of multiple nodes. Transparency applies to access, location, migration, concurrency, and failure handling.
- **Scalability** – The system should efficiently accommodate an increasing number of nodes without performance degradation.
- **Fault Tolerance** – The system must continue functioning even if some nodes fail, ensuring high availability and reliability.
- **Concurrency** – Multiple processes or users should be able to interact with the system simultaneously without conflicts.
- **Coordination** – Nodes must synchronize and manage data and processing efficiently to maintain system integrity.
- **Security** – Authentication, authorization, and data protection mechanisms must be in place to prevent attacks and ensure secure communication.

Common Architectures in Distributed Systems

- **Client-Server Architecture** – Clients send requests to a central server, which processes them and returns responses.
- **Peer-to-Peer (P2P) Architecture** – Each node acts as both a client and a server, sharing resources directly without a central authority.
- **Microservices Architecture** – The system is divided into independent services, each handling a specific function, communicating via APIs.
- **Event-Driven Architecture** – Components communicate asynchronously by sending and responding to events rather than direct calls.
- **Service-Oriented Architecture (SOA)** – Services interact over a network using standardized protocols, promoting reusability and interoperability.
- **Serverless Architecture** – Applications are executed in a cloud-based environment where resources are allocated dynamically, reducing infrastructure management overhead.

Monolithic vs. Distributed Systems Comparison

Aspect	Monolithic System	Distributed System
Architecture	Single, unified application	Multiple independent services
Scalability	Limited; scaling requires duplicating the whole system	High; can scale individual components separately
Deployment	Easier, as everything is in one place	More complex, requiring orchestration tools (e.g., Kubernetes)
Development Speed	Faster for small projects	Slower due to complexity and inter-service communication
Maintenance	Can become difficult as the codebase grows	Easier to maintain individual components
		More resilient, failures in one service don't affect others (if

Aspect	Monolithic System	Distributed System
Fault Tolerance	A failure in one part may crash the entire system	properly designed)
Performance	Can be efficient for small applications	Better suited for large-scale applications with high traffic
Flexibility	Less flexible; changing one part affects the whole system	More flexible; services can be modified independently
Technology Stack	Typically uses a single technology stack	Allows different services to use different technologies
Security	Easier to secure since everything is in one place	More complex security due to multiple entry points
Cost	Lower initial cost but higher long-term maintenance cost	Higher initial investment but more cost-effective at scale
Time to Market	Faster for simple applications	Slower due to the need for managing multiple components
Examples	Traditional enterprise applications, CMS, ERP	Cloud-native applications, e-commerce, microservices-based platforms

CAP Theorem (Consistency, Availability, Partition Tolerance)

The CAP Theorem, introduced by Eric Brewer, states that in a distributed system, it is impossible to guarantee all three of the following properties simultaneously. A system can only achieve two out of the three:

- **Consistency (C)** – Every node sees the same data at the same time. When an update occurs, all subsequent reads will return the latest data.
- **Availability (A)** – The system always responds to requests, even if some nodes fail or some data is outdated.
- **Partition Tolerance (P)** – The system can continue operating even if network failures cause some nodes to become unreachable.

Key Considerations in CAP Theorem

- **Trade-off in distributed systems** – You must choose between Consistency, Availability, and Partition Tolerance, as achieving all three simultaneously is impossible.
- **System classifications based on CAP**
  - **CP (Consistency + Partition Tolerance)** – Ensures data consistency but may reject some requests during network failures. (e.g., MongoDB, HBase)
  - **AP (Availability + Partition Tolerance)** – Guarantees responses but may return outdated or inconsistent data. (e.g., Cassandra, DynamoDB)
  - **CA (Consistency + Availability)** – This combination is impossible in distributed systems since network failures (Partitions) are inevitable. However, it is achievable in non-distributed systems like traditional SQL databases (PostgreSQL, SQL Server).
- **Impact on system design**
  - If Consistency is a priority (e.g., banking systems), Availability may be reduced to prevent data inconsistency.
  - If Availability is more important (e.g., social networks), Consistency may be sacrificed temporarily.
- **Real-world implementations**
  - Many NoSQL databases prioritize AP or CP over full consistency.
  - SQL databases typically focus on Consistency rather than Availability.

Consistency Models (Strong, Eventual, Weak Consistency)

In distributed systems, consistency models define how updates to data are propagated across different nodes and what guarantees the system provides regarding data visibility. The main types of consistency models are:

- **Strong Consistency** – Every read operation always returns the most recent write. Once data is updated, all nodes immediately reflect the latest change.
- **Eventual Consistency** – The system does not guarantee immediate consistency, but all nodes will eventually converge to the same state.
- **Weak Consistency** – There is no guarantee that all nodes will see the latest update, even after some time.

Consistency Model	Guarantee	Performance	Use Case Example	System Examples
Strong Consistency	Always the latest data	High latency	Banking, stock trading	Spanner, SQL (ACID-compliant)
Eventual Consistency	Data converges over time	Medium latency	Social media, DNS, CDN	DynamoDB, Cassandra
Weak Consistency	No guarantee of up-to-date data	Low latency	Video streaming, gaming	UDP-based systems

Scalability (Vertical vs. Horizontal Scaling)

Scalability in system design refers to the ability of a system to handle increased loads or to be expanded to accommodate growth. There are two main types of scalability: Vertical Scaling and Horizontal Scaling. Both have their strengths and weaknesses, and choosing between them depends on factors such as the architecture of the system, the type of load, and cost considerations.

- **Vertical Scaling (Scaling Up)** Vertical scaling involves adding more power (CPU, RAM, storage) to an existing machine or server. In this approach, you scale by upgrading the existing hardware to handle a larger workload. Vertical scaling is often used for applications that are difficult to distribute or for workloads where a single machine can handle the task. This includes databases that require a single point of coordination or applications that rely on complex single-threaded operations.
- **Horizontal Scaling (Scaling Out)** Horizontal scaling involves adding more machines or servers to distribute the load across multiple systems. Instead of upgrading a single server, you increase the number of servers in the system. This is typically used for systems designed for distributed computing or systems that need to handle large numbers of users concurrently, such as web applications, microservices, or cloud-based systems.

Comparison of Vertical Scaling vs Horizontal Scaling

Feature	Vertical Scaling (Scaling Up)	Horizontal Scaling (Scaling Out)
Definition	Adding more resources (CPU, RAM, storage) to an existing server.	Adding more machines or servers to distribute the load.
Implementation Simplicity	Easier, typically involves just upgrading hardware.	More complex, requires designing a distributed architecture.
Scalability	Limited by the physical hardware capacity.	Nearly unlimited, can scale by adding more servers.
Use Case	Suitable for applications needing powerful hardware.	Suitable for large, distributed systems handling high traffic.
Cost	Costs increase as hardware is upgraded.	Can be more cost-effective by using cheaper, smaller servers.
Drawback	Limited by hardware upgrades.	Complexity in managing data consistency and network latency.
Fault Tolerance	Single point of failure—if the server fails, the system goes down.	Better fault tolerance with distributed systems.
Database Considerations	Suitable for centralized databases that require high performance.	Requires data partitioning (sharding) and distributed systems.
Development Simplicity	Less complex, requires fewer changes to software and architecture.	Requires more complex design and distributed system management.

On-Demand Virtualization Automation

On-demand virtualization automation refers to the process of dynamically creating, managing, and terminating virtual machines (VMs) or containers based on real-time demand. This approach optimizes resource utilization, reduces costs, and enhances scalability without manual intervention. In this approach, we create or delete machines and allocate resources to them on demand using scripts.

Key Benefits

- **Optimized Resource Utilization** Automatically allocates computing resources (CPU, RAM, storage) only when needed, preventing wastage.
- **Cost Efficiency** Reduces operational costs by eliminating the need for over-provisioning.
- **Improved Scalability** Dynamically scales up or down based on workload demands, ensuring seamless performance.
- **Simplified Infrastructure Management** Centralized monitoring and automated provisioning simplify administration.
- **Reduced Deployment Time** Enables faster deployment of applications, VMs, and containers with minimal manual configuration.

Load Balancing

Load Balancing is a crucial concept in designing scalable and distributed systems. The primary goal of a Load Balancer is to distribute incoming requests across multiple servers or services to ensure better performance, higher availability, and prevent overloading a single server.

Why Use a Load Balancer?

- **Scalability** – Distributes traffic to multiple servers, increasing system capacity.
- **Fault Tolerance** – If one server fails, the Load Balancer redirects requests to healthy servers.
- **Performance Optimization** – Ensures faster request processing by balancing the workload.
- **Even Load Distribution** – Prevents a single server from being overloaded while others remain underutilized.

Types of Load Balancing Algorithms

1. **Round Robin** – Requests are distributed sequentially across available servers. For example, with three servers A, B, and C: Request 1 → A Request 2 → B Request 3 → C Request 4 → A ...

- Pros
  - Simple and effective for homogeneous servers.
- Cons
  - If a server is slower, it may still receive the same number of requests, causing imbalances.

2. **Weighted Round Robin** – An improved version of Round Robin where each server is assigned a weight. More powerful servers receive more requests.

- Pros
  - Suitable for environments with servers of different capacities.
- Cons
  - Requires careful weight tuning.

3. **Least Connections** – Requests are routed to the server with the fewest active connections. This is ideal for stateful applications where request processing time varies.

- Pros
  - Ensures fairer load distribution.
  - Works well for applications with long-running requests.
- Cons
  - May increase latency if the system doesn't handle connection time efficiently.

4. **Least Response Time** – Requests are sent to the server with the lowest response time, ensuring faster performance.

- Pros
  - Optimized for speed and responsiveness.
- Cons
  - Requires continuous monitoring of response times.

5. **IP Hashing** – Uses a hashing algorithm on the client's IP address to consistently route a specific client to the same server.

- Pros
  - Useful for session-based applications where users should always connect to the same server.
- Cons
  - If a server fails, redistributing requests can be difficult.

6. **Least Bandwidth** – Routes requests to the server with the least amount of bandwidth usage.

- Pros
  - Ideal for applications that handle large data transfers, such as streaming.
- Cons
  - Requires real-time bandwidth monitoring.

7. **Random** – Requests are distributed randomly among servers.

- Pros
  - Simple to implement.
- Cons
  - May lead to uneven load distribution.

Data Replication & Sharding

In large-scale distributed systems, Data Replication and Sharding are two essential techniques used to improve performance, scalability, and reliability. While Replication focuses on creating copies of data across multiple nodes for redundancy and availability, Sharding partitions data across multiple nodes to distribute the load efficiently.

Data Replication

Data Replication is the process of storing copies of the same data on multiple servers or locations to enhance fault tolerance, read scalability, and disaster recovery.

Types of Data Replication

1. **Master-Slave Replication (Primary-Replica)** – A single master node handles all write operations. One or more slave nodes replicate data from the master and handle read requests. If the master fails, a slave can be promoted to the new master.

- Pros
  - Improves read performance by distributing queries across multiple replicas.
  - Provides data redundancy.
- Cons
  - Write operations are limited to a single master, which can become a bottleneck.
  - Lag between master and replicas can lead to stale reads.

2. **Master-Master Replication (Active-Active)** – Multiple nodes act as masters, allowing both read and write operations. Changes are synchronized across all nodes, either asynchronously or synchronously.

- Pros
  - High availability with no single point of failure.
  - Suitable for geographically distributed applications.
- Cons
  - Handling conflicts when two nodes update the same data simultaneously can be complex.

    - Requires advanced conflict resolution strategies.

3. **Quorum-based Replication (Consensus-driven)** – Used in distributed databases like Apache Cassandra, Amazon DynamoDB, and Raft-based systems. Instead of a single master, multiple nodes participate in a consensus algorithm (e.g., Paxos, Raft) to ensure data consistency.

- Pros
    - No single point of failure.
    - Suitable for distributed and decentralized systems.
- Cons
    - Higher latency due to consensus requirements.

Replication Strategies

Synchronous Replication – Writes are committed to all replicas before acknowledging success.

- Pros – Strong consistency.
- Cons – High latency.

Asynchronous Replication – Writes are acknowledged immediately, and replicas are updated later.

- Pros – Low latency.
- Cons – Risk of data loss in case of failure.

Sharding (Data Partitioning)

Sharding is a technique used to horizontally partition data across multiple database servers to distribute the workload efficiently. Instead of duplicating data, sharding splits it into smaller pieces (shards), each stored on a different node.

Sharding Strategies

1. **Hash-based Sharding** – A hash function is used to distribute records across shards. Example: shard_id = hash(user_id) % number_of_shards.

- Pros
    - Ensures even distribution of data.
    - Reduces hot-spot issues.
- Cons
    - Resharding (adding more shards) is complex because data must be redistributed.

2. **Range-based Sharding** – Data is divided based on value ranges. Example: User IDs 1-1000 → Shard 1, 1001-2000 → Shard 2, etc.

- Pros
    - Simple to implement.
    - Efficient for range queries.
- Cons
    - Uneven distribution can lead to hot-spot problems (e.g., all recent users in one shard).

3. **Directory-based Sharding** – A lookup table (directory) maps records to specific shards.

- Pros
    - Flexible, as new shards can be added easily.
- Cons
    - The lookup table becomes a single point of failure and performance bottleneck.

4. **Geo-based Sharding** – Data is partitioned based on geographical location. Example: Users in Europe → European server, US → US server, etc.

- Pros
    - Reduces latency for users in different regions.
- Cons
    - Some regions may have more load than others, requiring dynamic scaling.

Feature	Replication	Sharding
Goal	Redundancy & availability	Scalability & load distribution
Data Copying	Full copy on multiple nodes	Split data across multiple nodes
Read Scalability	High (multiple read replicas)	Depends on query type
Write Scalability	Limited (single master bottleneck)	High (data is partitioned)
Complexity	Easier to implement	Harder to reshard dynamically

author : Yasin Karbasi

Chapter 02 - Networking

Client-Server Communication & Network Protocols

In networking, client-server communication refers to the exchange of data between a client (e.g., a web browser or mobile app) and a server (e.g., a backend API or database). This communication relies on various protocols, which define how data is transmitted over the network.

Common Network Protocols

TCP/IP (Transmission Control Protocol / Internet Protocol)

- TCP/IP is the fundamental suite of protocols used for communication over the internet, ensuring reliable data transmission.
- TCP manages data segmentation and reassembly, ensuring packets arrive in order, while IP handles addressing and routing.
- TCP is connection-oriented, requiring a handshake before transmission. IP ensures data reaches the correct destination.
- Used in web browsing, email, VoIP, file transfers, all internet communication.

SSL/TLS (Secure Sockets Layer / Transport Layer Security)

- SSL/TLS are cryptographic protocols that encrypt data during transmission, securing communication over the internet.
- TLS uses public-key cryptography for authentication and symmetric encryption for secure data exchange.
- TLS is the modern version of SSL; SSL 2.0 and 3.0 are deprecated.
- Used in secure websites (HTTPS), email encryption, VPN connections, online banking.

DNS (Domain Name System)

- DNS translates human-readable domain names (e.g., google.com) into IP addresses.
- A recursive query is sent to a DNS resolver, which then queries authoritative servers to resolve the domain to an IP.
- Supports caching to speed up lookups, and DNSSEC enhances security by preventing spoofing.
- Used in website address resolution, email routing, distributed networks.

SSH (Secure Shell)

- SSH is a secure protocol for remote system access and file transfers.
- Uses public-key authentication and strong encryption to prevent eavesdropping.
- Commonly used for server management supports features like SSH tunneling and SCP/SFTP for file transfers.
- Uses port 22
- Used in secure remote login, system administration, encrypted file transfers.

HTTP/HTTPS (Hypertext Transfer Protocol)

- HTTP is the protocol for web communication, and HTTPS is its secure version with encryption.
- Follows a request-response model, where the client sends a request, and the server responds with data (HTML, JSON, etc.).
- HTTPS uses TLS encryption for security, preventing MITM attacks.
- HTTP uses port 80 and https uses port 443.
- Used in web browsing, API communication, secure transactions.

WebSocket

- WebSocket is a protocol for persistent, full-duplex communication between a client and server.
- After an initial handshake over HTTP, WebSocket keeps the connection open for real-time data exchange.
- More efficient than HTTP polling for low-latency applications.
- ws users port 80 and wss uses port 443.
- Used in live chat, stock market updates, real-time gaming.

RPC (Remote Procedure Call)

- RPC allows a client to execute a function on a remote server as if it were local.
- The client sends a request with the function name and parameters, and the server executes it and returns a response.
- Can be synchronous (blocking) or asynchronous (non-blocking), and gRPC (a modern RPC framework) supports TLS encryption.
- Used in microservices, distributed computing, blockchain interactions.

FTP/FTPS (File Transfer Protocol)

- FTP is a protocol for transferring files between a client and a server, while FTPS adds SSL/TLS encryption for security.
- Clients connect to an FTP server, authenticate (if required), and transfer files.
- FTP operates in active and passive modes, and FTPS ensures secure authentication and encrypted transfers.
- Used in website file uploads, data backups, enterprise file sharing.

SMTP (Simple Mail Transfer Protocol)

- SMTP is the standard protocol for sending emails between mail servers.
- An email client sends messages to an SMTP server, which then relays them to the recipient’s mail server.
- SMTP is used for sending only; protocols like IMAP/POP3 handle email retrieval.
- Used in email sending, mail server communication.

Protocol Comparison Table

Protocol	Purpose	Used In
TCP/IP	Core internet communication	Web, email, VoIP, all networking
SSL/TLS	Encrypts network traffic	Secure websites, VPNs, banking
DNS	Resolves domain names	Website access, email routing
SSH	Secure remote access	Server login, secure file transfer
HTTP	Web communication (insecure)	Basic websites, APIs
HTTPS	Secure web communication	Secure websites, APIs
WebSocket	Real-time communication	Chat apps, gaming, live updates
RPC	Remote procedure calls	Microservices, distributed systems
FTP	File transfer (insecure)	Transferring files
FTPS	Secure file transfer	Secure file transfers
SMTP	Email sending protocol	Sending emails, mail server communication

Routing & Proxies in Networking

In distributed systems and modern networking, routing and proxies play a crucial role in efficiently directing traffic between clients and backend services. Different routing strategies and proxy types help improve scalability, reliability, and load balancing.

Types of Proxies

Reverse Proxy

A reverse proxy sits in front of backend servers and handles client requests on their behalf.

- **Use Cases**
  - Load balancing across multiple servers
  - Caching responses to improve performance
  - SSL termination to offload encryption from backend servers
  - Security by hiding internal server details
- Examples:
  - NGINX
  - HAProxy
  - Envoy Proxy (used in Service Meshes)
  - Traefik (dynamic routing for microservices)

Forward Proxy

A forward proxy sits between clients and the internet, acting as an intermediary for outbound requests.

- **Use Cases**
  - Hiding client IP addresses (for security or anonymity)
  - Filtering requests (e.g., blocking access to certain websites)
  - Caching static content to reduce bandwidth usage
- **Examples**
  - Squid Proxy
  - Shadowsocks (used for bypassing censorship)

API Gateway

An API gateway is a reverse proxy designed specifically for API requests. It manages authentication, rate limiting, and routing.

- **Use Cases**
  - Centralized API management (authentication, security)
  - Rate limiting & throttling to prevent abuse
  - Load balancing & failover for microservices
- **Examples**
  - Kong
  - Apigee (Google Cloud API Gateway)
  - AWS API Gateway

Content Delivery Network (CDN) & Edge Networking

Content Delivery Network (CDN)

A CDN is a distributed network of servers designed to cache and deliver content (images, videos, scripts, etc.) closer to users.

Why Use a CDN?

- Lower Latency – Content is served from the closest server to the user.
- Reduced Server Load – Caching reduces requests to the origin server.
- Better Reliability – If one server goes down, traffic is rerouted.
- Security Benefits – DDoS protection and Web Application Firewalls (WAF).

How it Works?

1. A user requests a webpage (e.g., example.com).
2. DNS routes the request to the nearest CDN server instead of the origin server.
3. The CDN serves cached static assets (e.g., CSS, JavaScript, images).
4. If the requested content is not cached, the CDN fetches it from the origin server.

Popular CDN Providers

- Cloudflare
- Akamai
- Amazon CloudFront
- Google Cloud CDN
- Fastly

Edge Networking

Edge networking moves computing and processing closer to end-users instead of relying on a central data center.

Why Use Edge Computing?

- Ultra-low Latency – Processing happens near the user, reducing round-trip time.
- Better Scalability – Local processing reduces the load on centralized servers.
- Real-Time Processing – Ideal for IoT, AI, and interactive applications.

How it Works?

1. A request is processed at the edge server instead of traveling to a central data center.
2. The edge server can perform computations, caching, and filtering before sending data back.
3. If additional processing is needed, only critical data is sent to the origin.

Edge Networking vs. CDN

Feature	CDN	Edge Networking
Purpose	Caching & fast content delivery	Localized processing & computing
Focus	Reducing latency for static assets	Processing real-time data near users
Examples	Cloudflare CDN, AWS CloudFront	AWS Lambda@Edge, Cloudflare Workers

Key Differences: CDN vs. Edge Networking

Feature	CDN	Edge Networking
Main Function	Caches & delivers content	Processes data at the edge
Best For	Static content (images, JS, CSS)	Real-time apps & IoT
Latency Reduction	Moderate	Extremely low
Computing Power	No computation	Edge servers process data
Example Providers	Cloudflare, Akamai, AWS CloudFront	Cloudflare Workers, AWS Lambda@Edge

author : Yasin Karbasi

Chapter 03 - Storage & databases

What is ACID in Databases?

ACID is a set of properties that ensure reliable transactions in a database. It stands for:

1. A - Atomicity
2. C - Consistency
3. I - Isolation
4. D - Durability

These properties are critical for relational databases (SQL) to maintain data integrity, especially in applications like banking, finance, and e-commerce.

1. Atomicity (All or Nothing)

Ensures that a transaction is either fully completed or fully rolled back if something goes wrong.

Example: Transferring money between bank accounts

- Step 1: Deduct $500 from Account A
- Step 2: Add $500 to Account B
- If Step 2 fails, Step 1 is rolled back, ensuring no partial updates.

No incomplete transactions—either everything happens or nothing happens.

2. Consistency (Valid State)

Ensures that the database remains in a valid state before and after a transaction by enforcing rules and constraints.

Example: A bank transaction should never create negative balances if a rule prevents overdrafts. If a withdrawal request would cause a negative balance, the transaction fails.

The database follows integrity rules and never ends up in an invalid state.

3. Isolation (Transactions Don't Interfere)

Ensures that multiple transactions can occur simultaneously without affecting each other.

Example: Two customers buying the last ticket at the same time

- Transaction 1: Checks available tickets (1 left)
- Transaction 2: Also checks available tickets (1 left) Without Isolation, both could buy the last ticket, causing overbooking.

Transactions are processed independently and in isolation, preventing data corruption.

4. Durability (Data is Permanently Saved)

Ensures that once a transaction is committed, it remains stored—even in case of power failure or crash.

Example: A completed bank transfer should remain saved, even if the server crashes after confirmation. Data remains safe after a successful transaction, even in case of system failure.

SQL vs. NoSQL

When designing a system, choosing between SQL (relational databases) and NoSQL (non-relational databases) depends on factors like scalability, flexibility, and consistency. Below is a comparison of both in terms of advantages and disadvantages in system design.

SQL (Relational Databases)

SQL databases store data in structured tables with fixed schemas and use SQL (Structured Query Language) for querying.

Advantages

- ACID Compliance – Ensures Atomicity, Consistency, Isolation, and Durability, making SQL databases ideal for systems requiring data integrity (e.g., banking, financial transactions).
- Structured Data – Best suited for applications with clearly defined relationships between entities (e.g., a CRM system managing customers and orders).
- Powerful Querying – SQL provides advanced querying capabilities, including JOINs, transactions, and complex aggregations.
- Standardized Language – SQL is widely adopted, making it easier to find developers and integrate with various tools.

Disadvantages

- Scalability Issues – Scaling SQL databases horizontally (sharding) is complex; they are primarily designed for vertical scaling (adding more power to a single server).

- Fixed Schema – Changing the database schema requires migrations, which can be time-consuming and risky for rapidly evolving applications.
- Performance Bottlenecks – High-traffic applications with many read/write operations may suffer from performance degradation due to strict consistency enforcement.

Use Cases

- Banking & Finance (due to ACID compliance)
- ERP and CRM Systems
- Traditional Web Applications

Common SQL Databases

1. **MySQL**

- Open-source and widely used
- Supports replication and clustering
- Strong ACID compliance with InnoDB engine
- Best for: Web applications, e-commerce, banking

2. **PostgreSQL**

- Advanced open-source database
- Supports JSON data along with relational tables
- High concurrency with MVCC (Multi-Version Concurrency Control)
- Best for: Data analytics, geospatial applications, enterprise apps

3. **Microsoft SQL Server**

- Developed by Microsoft
- Supports high availability and business intelligence tools
- Strong security with encryption and role-based access
- Best for: Enterprise solutions, Windows-based applications

4. **Oracle Database**

- High-performance enterprise-grade database
- Supports partitioning, sharding, and strong security
- Ideal for large-scale applications with complex transactions
- Best for: Banking, large-scale enterprise applications

5. **SQLite**

- Lightweight, serverless database
- Used for mobile and embedded applications
- No need for a separate server process
- Best for: Mobile apps, IoT devices, small-scale apps

NoSQL (Non-Relational Databases)

NoSQL databases are schema-less and can store data in different formats: key-value, document, column-family, and graph databases.

Advantages

- Scalability – Designed for horizontal scaling (adding more servers), making them ideal for distributed systems and high-traffic applications.
- Flexible Schema – Schema-less structure allows easy modifications, making NoSQL databases suitable for dynamic applications.
- High Performance – Optimized for fast reads and writes, especially in distributed environments (e.g., caching layers).
- Best for Unstructured Data – Can handle large volumes of JSON, XML, and semi-structured data, making them great for applications like real-time analytics, social media, and IoT.

Disadvantages

- Lack of ACID Compliance – Most NoSQL databases follow BASE (Basically Available, Soft state, Eventually consistent), which may result in data inconsistency.
- Limited Querying Capabilities – NoSQL databases lack JOINs and complex queries like SQL, requiring additional processing logic in the application.
- Data Redundancy – Since there are no relationships like in SQL, duplicate data is often stored, leading to data inconsistency.
- Less Mature Ecosystem – NoSQL tools and expertise are not as widespread as SQL, which can make development more challenging.

Use Cases

- Real-time analytics and Big Data
- IoT and streaming applications
- Social media and recommendation systems
- Content management systems (CMS)

Common SQL Databases

1. **MongoDB (Document-based)**

- Stores data in JSON-like BSON format
- Supports flexible schema and indexing
- Scales horizontally using sharding
- Best for: CMS, real-time analytics, IoT

2. Cassandra (Column-family)

- Distributed, highly scalable database
- Designed for high availability (No single point of failure)
- Used by companies like Facebook, Netflix
- Best for: Big data, real-time analytics

3. Redis (Key-Value Store)

- In-memory database for ultra-fast performance
- Supports caching, pub/sub messaging, and session storage
- Best for: Caching, gaming leaderboards, real-time data processing

4. Firebase Firestore (Document-based)

- NoSQL database managed by Google
- Real-time syncing for mobile and web apps
- Best for: Mobile apps, real-time chat applications

5. Neo4j (Graph-based)

- Optimized for graph-based relationships
- Uses Cypher Query Language (CQL)
- Best for: Social networks, recommendation systems, fraud detection

SQL vs. NoSQL

Criteria	SQL (Relational)	NoSQL (Non-Relational)
Data Structure	Structured with fixed schema	Unstructured or semi-structured
Scalability	Vertical (Single Server)	Horizontal (Multiple Servers)
ACID Compliance	Strong (for data integrity)	Weak (Eventual consistency)
Performance	Slower for large datasets	Faster for read-heavy workloads
Use Case	Banking, ERP, CRM, E-commerce	Big Data, Social Media, IoT, Caching

author : Yasin Karbasi

Chapter 04 - Caching

What is Caching?

Caching is a technique used to store frequently accessed data in a high-speed storage layer to improve performance and reduce load on backend services. Instead of fetching data from a slow source (e.g., database, API, or disk), caching allows retrieval from a fast, temporary storage (e.g., in-memory).

Why Use Caching?

- Improves Performance – Reduces latency by serving data from a closer, faster source.
- Reduces Database Load – Minimizes repeated queries to the database, reducing overhead.
- Handles High Traffic – Helps scale applications by reducing backend bottlenecks.
- Enhances User Experience – Faster responses lead to better user experience.

Types of Cache

Caching can be implemented at different layers based on the system's needs:

1. Application Cache

- Stored in memory within the application.
- Example: Caching computed values inside a function.

2. Database Cache

- Reduces redundant queries by caching query results.
- Example: MySQL Query Cache, Redis as a caching layer for databases.

3. Content Delivery Network (CDN) Cache

- Stores static assets (CSS, JS, images) at edge locations closer to users.
- Example: Cloudflare, AWS CloudFront, Akamai.

4. Distributed Cache

- A shared cache across multiple servers, useful for large-scale applications.
- Example: Redis, Memcached.

5. Web Cache (Reverse Proxy Cache)

- Caches entire HTTP responses to reduce load on backend servers.
- Example: Nginx, Varnish Cache.

6. Client-Side Cache

- Caches data in the browser (cookies, local storage, service workers).
- Example: Browser caching CSS/JS files.

Cache Type	Description	Examples	Pros	Cons
Application Cache	Stores frequently used data in app memory.	`functools.lru_cache` in Python	Fast access, no extra setup needed.	Limited by application memory.
Database Cache	Caches query results to reduce DB load.	MySQL Query Cache, Redis	Speeds up DB queries, reduces latency.	May serve stale data, requires invalidation.
CDN Cache	Caches static assets at edge locations.	Cloudflare, AWS CloudFront	Low latency, reduces server load.	Best for static content only.
Distributed Cache	Shared cache across multiple servers.	Redis, Memcached	Scales well, high availability.	Needs proper eviction strategy.
Reverse Proxy Cache	Caches HTTP responses to offload backend.	Nginx, Varnish Cache	Improves web performance, reduces load.	Harder to invalidate dynamically.
Client-Side Cache	Caches in browser (cookies, local storage).	Service Workers, IndexedDB	Reduces server calls, improves UX.	Users may need to clear cache manually.

Cache Invalidation Strategies

Since cached data can become outdated, cache invalidation strategies help maintain consistency.

1. Time-to-Live (TTL) Expiry

- Cache items expire after a set time (e.g., 60 seconds).
- Works well for periodically updated data.
- Example: API responses cached for 5 minutes.

2. Write-Through Cache

- Writes data to both the cache and database simultaneously.
- Ensures data consistency but adds write latency.
- Example: A user profile update is saved in both Redis and PostgreSQL.

3. Write-Back (Lazy Write) Cache

- Data is written to the cache first and asynchronously updated in the database.
- Faster writes but risks data loss if the cache crashes before syncing.

4. Cache Eviction (LRU, LFU, FIFO)

- Least Recently Used (LRU) – Removes the least recently accessed item.
- Least Frequently Used (LFU) – Removes the least accessed items over time.
- First-In-First-Out (FIFO) – Removes the oldest cached item first.

5. Manual Invalidation

- Developers explicitly remove or refresh cached data when needed.
- Example: Clearing product cache when inventory is updated.

Strategy	Description	Pros	Cons	Best Use Case
TTL Expiry	Sets a time limit for cached data.	Simple, automatic expiration.	May serve stale data before expiry.	API responses, session data.
Write-Through	Writes data to cache and database at same time.	Ensures consistency, always fresh.	Slower writes due to dual storage.	User profiles, frequently updated data.
Write-Back	Writes to cache first, then updates DB later.	Faster writes, reduces DB load.	Risk of data loss if cache fails.	Analytics, logs, bulk writes.
LRU (Least Recently Used)	Removes least recently accessed items.	Efficient, prevents cache bloat.	May remove needed data if not accessed often.	General-purpose caching.
LFU (Least Frequently Used)	Removes least accessed items over time.	Keeps most popular data in cache.	Harder to implement, needs frequency tracking.	High-traffic APIs.
FIFO (First-In-First-Out)	Removes oldest cached items first.	Simple and predictable.	Not always optimal for performance.	Limited-memory caches.
Manual Invalidation	Explicitly removes or refreshes cache.	Full control over updates.	Requires additional logic and tracking.	Dynamic content like product catalogs.

Chapter 05 - Message Queues & Event-Driven Architecture

Event-Driven Architecture (EDA)

Event-Driven Architecture is a design pattern where components (services or systems) communicate by producing and consuming events. An event is typically a state change or an action that has occurred within the system, such as "User Created," "Order Placed," or "Payment Processed."

In an event-driven system:

1. Event Producers generate events (e.g., an order is placed).
2. Event Consumers listen for and process events (e.g., the inventory system processes the order and updates stock levels).
3. Event Bus (or Event Broker): Carries events from producers to consumers, often in the form of a message queue or a stream-processing system.

Benefits of Event-Driven Architecture:

- Loose coupling: Services don’t directly depend on each other, making the system more flexible.
- Real-time processing: As events are triggered, they are processed immediately.
- Scalability: Services can scale independently based on the volume of events they handle.
- Fault tolerance: Components can continue functioning if others fail, as long as they can process the events later.

Use Cases and Examples

1. **Microservices Communication:** In a microservices-based architecture, services can communicate with each other using event-driven mechanisms, ensuring decoupled and asynchronous communication.For example Order Service might generate an event "OrderPlaced," and other services like Inventory, Payment, and Shipping can listen to this event and perform respective actions like deducting stock, processing payment, or initiating shipping.

2. **Real-Time Systems:** Stock Trading Systems: Stock price updates, trades, or market changes are events that consumers (e.g., client applications, risk managers) react to in real time. Chat Applications: A new message or notification can trigger events that update the status of messages for all participants.

3. **IoT Systems:** Sensors on devices can generate events based on readings (e.g., temperature exceeds a threshold), and consumers like alert systems or monitoring applications can take action on these events.

4. **Payment Systems:** Payment Processing: Once a payment is confirmed, the system can trigger events like "PaymentCompleted" for services like order fulfillment or customer notification.

5. **Log Aggregation:** Logs from different services generate events that are consumed by a centralized log processing service to provide analytics and monitoring.

Message Queues

A Message Queue (MQ) is a communication mechanism used in distributed systems to enable the asynchronous exchange of messages between different services or components. It decouples the sender and receiver, meaning that the sender does not have to wait for the receiver to process the message. Instead, the message is placed in a queue, and the receiver can process it at its own pace.

How It Works

1. A sender (Producer) sends a message to the queue.
2. The message is stored in the queue until the consumer (Receiver) retrieves and processes it.
3. If the consumer is unavailable or busy, the message remains in the queue until it's processed, ensuring reliability.

Key Benefits:

- Asynchronous communication: Allows services to communicate without waiting for an immediate response.
- Scalability: Allows independent scaling of producers and consumers.
- Reliability: Ensures messages are not lost if a service is temporarily unavailable.
- Decoupling: Components don’t need to know the specifics of each other, enabling flexibility.

Common Message Queues

RabbitMQ

- A widely used open-source message broker that supports AMQP (Advanced Message Queuing Protocol).
- It enables reliable message delivery, flexible routing, and supports both point-to-point and publish-subscribe models.
- Often used for background task processing, job queues, and asynchronous communication between services.

Apache Kafka

- A distributed event streaming platform designed for handling high throughput and low-latency messaging.
- Kafka allows for publishing and subscribing to streams of records, similar to message queues, but is more suitable for high-throughput event-driven systems.
- Ideal for handling large-scale real-time data feeds, such as logs, user activity tracking, and event-driven systems.

Amazon SQS (Simple Queue Service)

- A fully managed message queuing service from AWS, which offers easy integration with other AWS services.
- It allows you to decouple and scale microservices, distributed systems, and serverless applications.
- Used for asynchronous communication between services in the AWS ecosystem.

ActiveMQ

- An open-source message broker that supports various messaging protocols (e.g., AMQP, MQTT, and STOMP).
- It’s designed for enterprise-level applications that require message queuing with high availability and fault tolerance.
- Commonly used in enterprise environments where JMS (Java Message Service) is required.

Redis Pub/Sub

- Redis offers lightweight Pub/Sub functionality that can be used to send messages between distributed systems or to notify services of events.
- Common in systems requiring high-performance, real-time messaging and event notification.

Azure Service Bus

- A fully managed message queuing service provided by Microsoft Azure that supports both queues and topics for publish/subscribe.
- Ideal for hybrid cloud solutions where enterprise-grade messaging and event handling are required.

author : Yasin Karbasi

Chapter 06 - API Design

API Design and Its Types

API (Application Programming Interface) design refers to the process of structuring how different software components interact. The goal is to create APIs that are scalable, maintainable, and easy to use. There are several types of API architectures, each serving different use cases.

Types of APIs

REST (Representational State Transfer)

- Uses HTTP methods (GET, POST, PUT, DELETE).
- Works with standard formats like JSON and XML.
- Stateless (each request must contain all necessary information).
- Scalable and widely used for web applications.

GraphQL

- Clients specify the exact data they need.
- Single endpoint (/graphql) handles all requests.
- Reduces over-fetching and under-fetching of data.
- Ideal for complex data relationships and frontend-driven applications.

gRPC (Google Remote Procedure Call)

- Uses HTTP/2 and Protocol Buffers (protobuf) for efficient data exchange.
- Supports bi-directional streaming.
- Faster and more efficient than REST for high-performance applications.
- Best for microservices and low-latency systems.

SOAP (Simple Object Access Protocol)

- Uses XML for message format.
- Stronger security and reliability features (e.g., WS-Security).
- Used in enterprise applications (e.g., banking, healthcare).

WebSockets

- Enables real-time communication with persistent connections.
- Ideal for applications requiring instant data updates (e.g., chat apps, stock market).
-

Feature	REST	GraphQL	gRPC	SOAP	WebSockets
Data Format	JSON, XML	JSON	Protobuf	XML	Custom (Binary/Text)
Communication	Request/Response	Query-based	Request/Response, Streaming	Request/Response	Full Duplex
Performance	Medium	Optimized queries	High (Binary, HTTP/2)	Low (XML overhead)	High
Ease of Use	High	Medium	Low (requires protobuf)	Low	Medium
Use Cases	Web APIs	Frontend-heavy apps	Microservices	Enterprise apps	Real-time apps

Rate Limiting & Throttling

Rate Limiting

Rate limiting controls how many requests a client can make within a specific timeframe. It helps prevent abuse and ensures fair resource distribution.

Techniques for Rate Limiting

- Fixed Window: Limits requests in a fixed time window (e.g., 100 requests per minute).
- Sliding Window: Uses a rolling window to smooth out request distribution.
- Token Bucket: Requests consume tokens from a bucket, which refills at a fixed rate.
- Leaky Bucket: Requests are processed at a fixed rate; excess requests are discarded or queued.

Throttling

Throttling is the process of slowing down or delaying excessive requests from a client. Unlike rate limiting (which blocks requests outright), throttling allows limited requests at a reduced speed.

Feature	Rate Limiting	Throttling
Purpose	Restrict requests	Slow down requests
Response	Rejects requests when limit exceeds	Allows requests but with delay
Example Use Case	API quota enforcement	Avoiding server overload

Service Mesh & API Gateway

Service Mesh

A service mesh manages service-to-service communication in microservices. It handles traffic routing, security, and observability.

Key Features

- Traffic control: Load balancing, retries, circuit breaking.

- Security: Mutual TLS, authentication, and authorization.
- Observability: Logging, monitoring, and tracing.

Common Service Meshes

- Istio
- Linkerd
- Consul

API Gateway

An API Gateway is a reverse proxy that manages API traffic between clients and services. It handles authentication, rate limiting, and request transformation.

Key Features

- Authentication & Security: JWT, OAuth, API keys.
- Traffic Control: Load balancing, caching, rate limiting.
- Protocol Translation: Converts between REST, GraphQL, gRPC, etc.

Common API Gateways

- Kong
- NGINX API Gateway
- AWS API Gateway
- Traefik

Feature	Service Mesh	API Gateway
Scope	Service-to-service communication	Client-to-service communication
Traffic Handling	Internal (within microservices)	External (from clients)
Security	Mutual TLS, service identity	Authentication, authorization
Observability	Service-level monitoring	API-level logging

author : Yasin Karbasi

Chapter 07 - System security

Authentication & Authorization

Authentication and authorization are crucial for verifying user identities and ensuring they have appropriate access.

Authentication (Who are you?)

Authentication ensures that the user is who they claim to be.

Common Authentication Methods

- **OAuth (Open Authorization):** A protocol that allows secure API authorization without exposing credentials. Commonly used for social logins (e.g., "Login with Google").
- **JWT (JSON Web Token):** A compact, self-contained token used for secure information exchange. It's widely used in stateless authentication (e.g., token-based authentication).
- **SSO (Single Sign-On):** Allows users to log in once and access multiple applications without needing to authenticate again (e.g., Google SSO for multiple services).
- **Multi-Factor Authentication (MFA):** Adds extra layers of security beyond passwords, such as OTPs, biometric verification, or security keys.

Authorization (What are you allowed to do?)

Once authenticated, authorization ensures users have the correct permissions to access specific resources.

Common Authorization Methods

- **Role-Based Access Control (RBAC):** Users are assigned roles, and roles determine what actions they can perform.
- **Attribute-Based Access Control (ABAC):** Access control is based on attributes such as user role, device type, or location.
- **OAuth Scopes:** Defines specific permissions that applications can request (e.g., "read-only" access to a user’s profile).

Encryption

Encryption ensures that data remains secure both in transit and at rest.

Encryption in Transit

- **TLS (Transport Layer Security):** Encrypts data sent between clients and servers. Successor to SSL.
- **HTTPS (HyperText Transfer Protocol Secure):** A secure version of HTTP that encrypts data using TLS to prevent eavesdropping and tampering.

Encryption at Rest

- **Database Encryptionf:** Encrypts stored data to prevent unauthorized access (e.g., AES-256 encryption in databases).
- **Disk Encryption:** Protects entire storage devices using encryption techniques like BitLocker (Windows) or FileVault (Mac).

End-to-End Encryption (E2EE)

- Ensures that only the sender and recipient can read messages (e.g., WhatsApp, Signal).
- Uses public-private key cryptography (e.g., RSA, ECC).

Common Vulnerabilities

SQL Injection (SQLi)

Attackers inject malicious SQL queries to manipulate databases. **Prevention:** Use prepared statements and ORM frameworks to prevent direct SQL execution.

Cross-Site Scripting (XSS)

Attackers inject malicious scripts into web pages, allowing them to steal user data. **Prevention:** Sanitize user inputs and use Content Security Policy (CSP).

Cross-Site Request Forgery (CSRF)

Attackers trick users into making unwanted actions (e.g., changing passwords) on trusted sites. **Prevention:** Use CSRF tokens to verify legitimate requests.

Web Shell (WSO)

Attackers upload a malicious shell script to a server, gaining unauthorized access. **Prevention:** Validate file uploads, restrict executable file types, and monitor system files.

DDoS Protection & Firewalls

DDoS Protection

A Distributed Denial-of-Service (DDoS) attack overwhelms a server with excessive traffic, causing downtime.

- **Rate Limiting:** Restrict the number of requests from a single IP.
- **CDN (Content Delivery Network):** Distribute traffic across multiple servers to absorb attacks (e.g., Cloudflare, Akamai).
- **Web Application Firewall (WAF):** Protects web applications from malicious traffic and bot attacks.

Firewalls

Firewalls control incoming and outgoing network traffic based on security rules.

Network Firewalls: Protect the internal network from external threats. **Host-based Firewalls:** Protect individual servers from unauthorized access. **Next-Gen Firewalls (NGFW):** Combine traditional firewalls with additional security features like deep packet inspection and intrusion prevention.

author : Yasin Karbasi

Chapter 08 - Monitoring & Logging and Error Management

Observability

Observability helps in understanding a system’s behavior by collecting and analyzing data related to its operations. It consists of three main pillars:

Monitoring

- Continuous tracking of system metrics like CPU usage, memory, response time, request rate, etc.

- Helps detect anomalies and performance issues.

Logging

- Captures detailed event records that help diagnose problems.
- Logs can be structured (JSON) or unstructured (plain text).

Tracing

Tracks a request’s journey across multiple services (especially in microservices architecture). Helps in debugging slow or failed requests.

Monitoring Tools

Prometheus

- Open-source monitoring tool that collects and stores time-series data.
- Pull-based system with powerful query language (PromQL).
- Works well with Grafana for visualization.

Grafana

- Visualization tool for monitoring metrics from Prometheus, InfluxDB, and more.
- Supports real-time dashboards and alerting.

ELK Stack (Elasticsearch, Logstash, Kibana)

- **Elasticsearch:** Stores and indexes logs for fast searching.
- **Logstash:** Ingests, processes, and transforms logs.
- **Kibana:** Provides UI for searching and analyzing logs.

Fault Tolerance & Circuit Breaker Pattern

Fault Tolerance

The system’s ability to continue functioning despite failures. Achieved using redundancy, replication, retries, failover mechanisms. Example: Load balancers, database replication, auto-scaling.

Circuit Breaker Pattern

- Prevents cascading failures by stopping requests to a failing service.
- Works in three states
  - Closed → Requests flow normally.
  - Open → Requests are blocked after repeated failures.
  - Half-Open → Allows limited requests to check if the service recovers.
- **Example:** Netflix Hystrix, Resilience4j (for Java).

author : Yasin Karbasi