System Design

System design is the **process of defining the architecture, interfaces, and data for a system** that satisfies specific requirements. System design meets the needs of your business or organization through coherent and efficient systems. Once your business or organization determines its requirements, you can begin to build them into a physical system design that addresses the needs of your customers. The way you design your system will depend on whether you want to go for custom development, commercial solutions, or a combination of the two.

System design requires a systematic approach to building and engineering systems. A good system design requires you to think about everything in an infrastructure, from the hardware and software, all the way down to the data and how it’s stored.

Components

In system design, there are two key components that play distinct roles: logical components and tangential components. These components help define and shape the characteristics and functionality of a system.

Logical components:

Logical components refer to the abstract and conceptual elements of a system. They are concerned with the logical organization and structure of the system rather than its physical implementation.

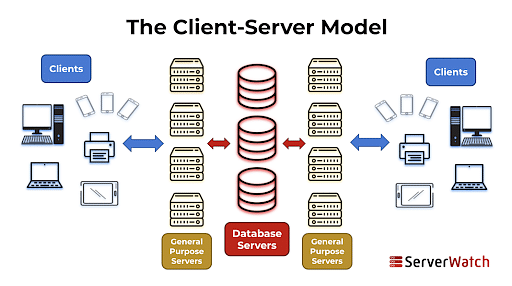
Tangential components:

Tangential components, on the other hand, deal with aspects that are not directly related to the core logic of the system but are essential for its successful implementation and operation.



Client Server Architecture

Client-server architecture is a common design pattern in system architecture where tasks or workloads are divided between the "client" and the "server." This model is used in various computing applications, from web services to database systems. The client and server are separate entities that communicate over a network, and each has specific roles and responsibilities.



Thick client vs thin client

The terms "thick client" and "thin client" refer to two different types of client devices in client-server computing. These terms describe the amount of processing and functionality that is handled by the client-side of a client-server architecture.

Thick Client:

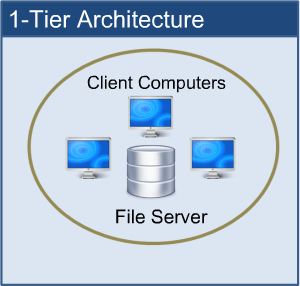
* Also known as a "fat client" or "rich client."
* It has a significant amount of processing and application logic running on the client device itself.
* Thick clients are often more feature-rich and can operate independently of the server to some extent.
* They typically require more resources (CPU, memory) on the client device.
* Examples of thick clients include traditional desktop applications that are installed on a user's computer, such as word processors, graphic design software, or video editing tools.

Thin Client:

* Also known as a "lean client."
* It relies heavily on the server for processing and application logic.
* Thin clients have minimal software installed locally, with most of the application processing happening on the server.
* These devices often have less powerful hardware requirements since they offload most of the processing to the server.
* Examples of thin clients include web browsers (when used to access web applications), simple terminal-based applications, or devices used in virtual desktop infrastructure (VDI) setups.

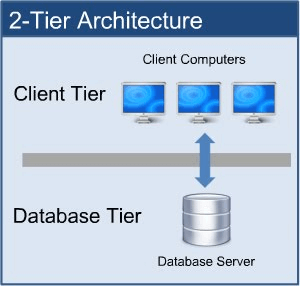
**Tier Architecture**

* One-tier architecture has a Presentation layer, Business layer and Data layers at the same tier i.e. at the Client Tier. As the name suggested, all the layers and components are available on the same machine.
* MP3 players, MS Office etc. are some examples of one-tier architecture.
* To store the data (as a function of the Data Layer) local system or a shared drive is used.



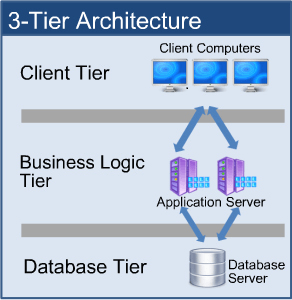
### **2. Two-tier Architecture:**

* In such type of architecture, the client tier handles both Presentation and Application layers and the server handles the Database layer.
* The two-tier architecture is also known as a ‘Client-Server Application’. In two-tier architecture, communication takes place between the Client and the Server.
* The client system sends the request to the server system and the server system processes the request and sends the response back to the client system.



### **3. Three-tier Architecture:**

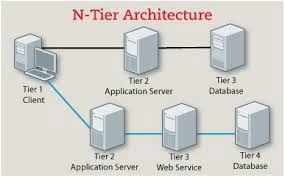
All three major layers are separated from each other. The presentation layer resides at Client Tier, the Application layer acts as middleware and lies at Business Tier and the Data layer is available at Data Tier. This is a very common architecture.



### **4. N-tier Architecture:**

N-tier architecture is also called Distributed Architecture or Multi-tier Architecture.

It is similar to three-tier architecture but the number of the application server is increased and represented in individual tiers in order to distribute the business logic so that the logic can be distributed.



Proxies

Proxies in system design refer to intermediary components or services that sit between clients and servers, acting as intermediaries to facilitate communication. They can be used for various purposes, including improving performance, security, and scalability. Here are some common types of proxies and their roles in system design:

Forward Proxy:

Role: Acts on behalf of clients to access resources on the server.

Use Cases:

Content Filtering: Blocking access to certain websites or content.

Access Control: Restricting access to specific resources based on policies.

Anonymity: Hiding client identities from the server.

Reverse Proxy:

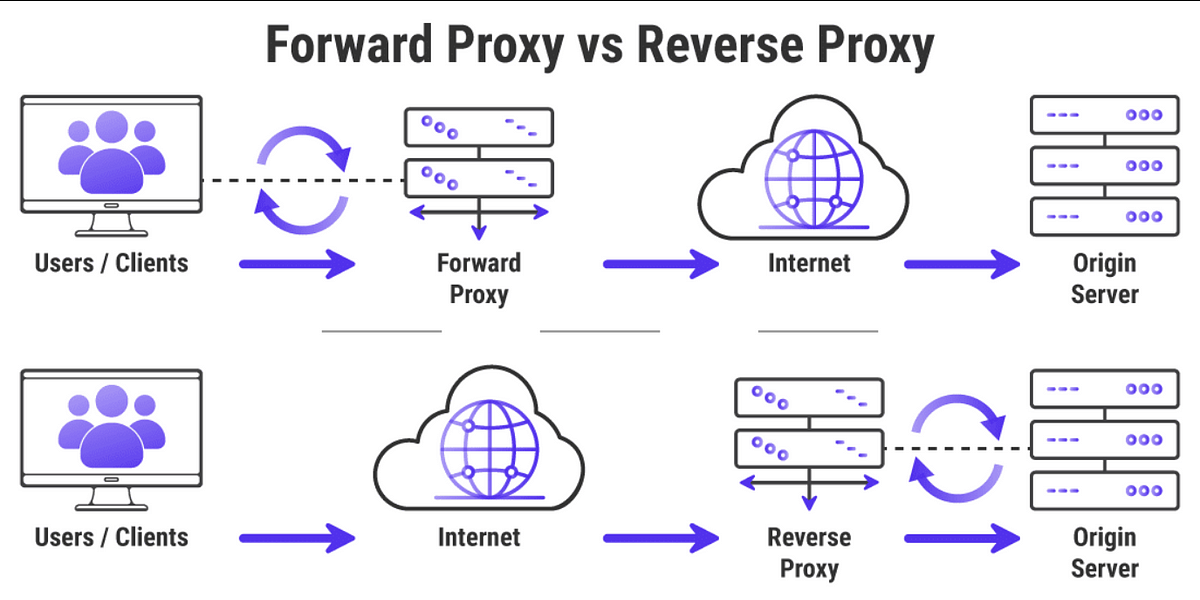
Role: Represents servers to clients, handling requests on behalf of the servers.

Use Cases:

Load Balancing: Distributing incoming traffic across multiple servers to improve performance and reliability.

SSL Termination: Managing SSL/TLS encryption and decryption, offloading this task from backend servers.

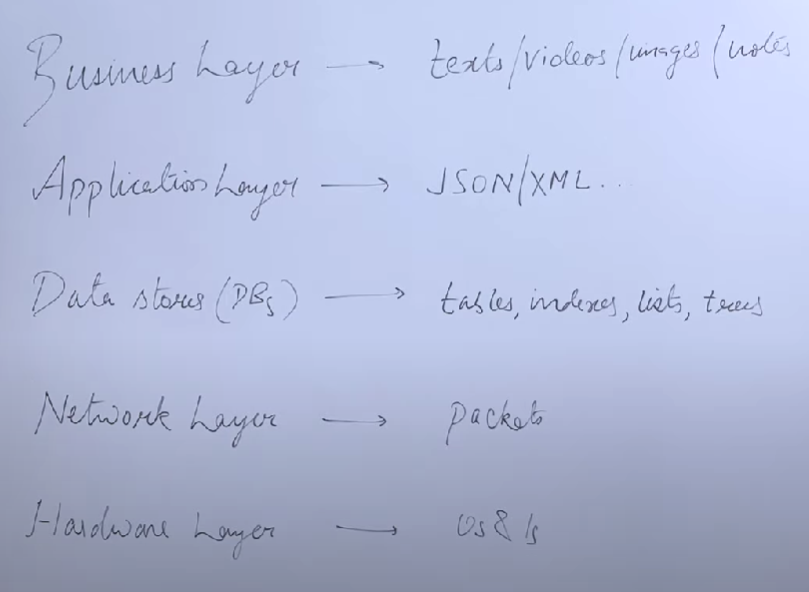
Caching: Storing copies of frequently requested resources to reduce server load and improve response times.



Data and data flow

In system design, "data" refers to the information that the system processes, stores, and manipulates to achieve its intended functionality. Data is a fundamental component of any computing system, and its effective management is critical for the system to perform its tasks accurately and efficiently.

Representation of data



A computer screen shot of a computer

Description automatically generated

Data flow

Data flow in system design refers to the movement and transformation of data within a system.

Data Sources and Sinks:

* Sources: Points where data originates, such as user input, sensors, or external systems.
* Sinks: Destinations or endpoints where data is consumed, stored, or processed.

Data Transformation:

* Definition: The process of converting data from one format or structure to another.
* Examples: Parsing input, aggregating data, converting units, or applying business rules.

Data Storage:

* Definition: Locations where data is persisted for future use.
* Examples: Databases, file systems, caches, and in-memory storage.

Data Processing:

* Definition: Operations performed on data to generate new information.
* Examples: Calculations, filtering, sorting, and aggregations.

Data Transfer:

* Definition: Movement of data between different components or systems.
* Examples: API calls, messaging, file transfers, and network communication.

Data Channels:

* Definition: Paths or conduits through which data flows.
* Examples: Communication channels, queues, pipelines, and network connections.

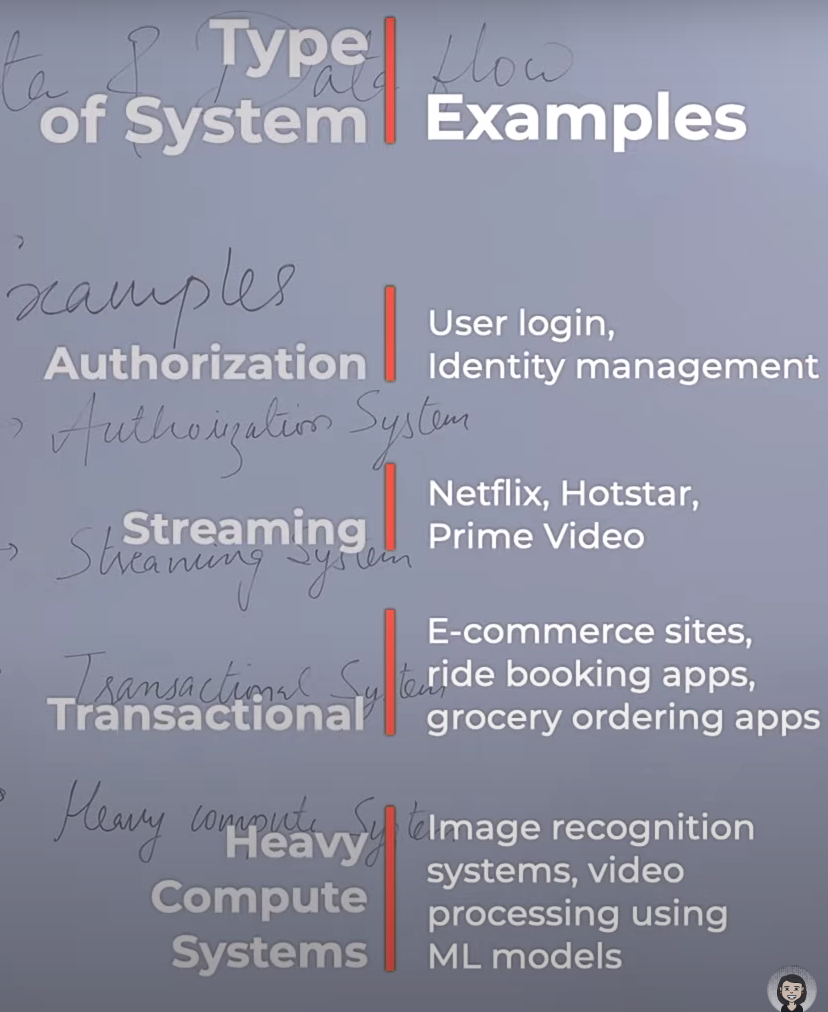
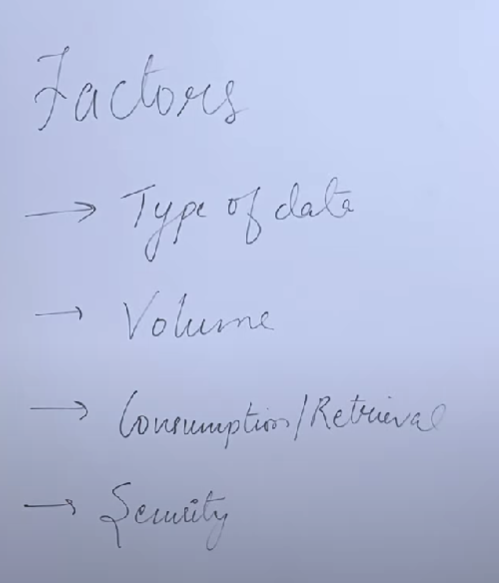
Data Integration:

* Definition: Combining data from multiple sources to create a unified view.
* Examples: ETL (Extract, Transform, Load) processes, data consolidation, and federation.

Data Validation:

* Definition: Ensuring that data meets specified criteria or quality standards.
* Examples: Input validation, error checking, and schema validation.

Factors



Database

In system design, choosing the appropriate type of database is a crucial decision that depends on the specific requirements, characteristics, and constraints of the application.

Relational Databases (RDBMS):

Description: Organizes data into tables with rows and columns, and enforces relationships between tables.

Use Cases:

Well-suited for applications with structured and well-defined data.

Transactional systems, financial applications, and systems with complex queries.

NoSQL Databases:

Description: Encompasses various database types that do not use the traditional relational model.

Types:

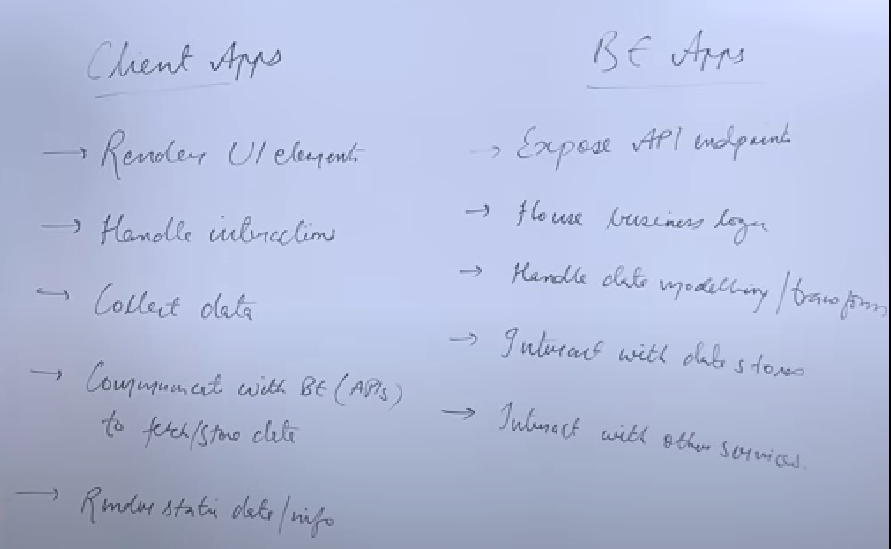
Document-oriented: MongoDB, CouchDB (stores data in JSON-like documents).

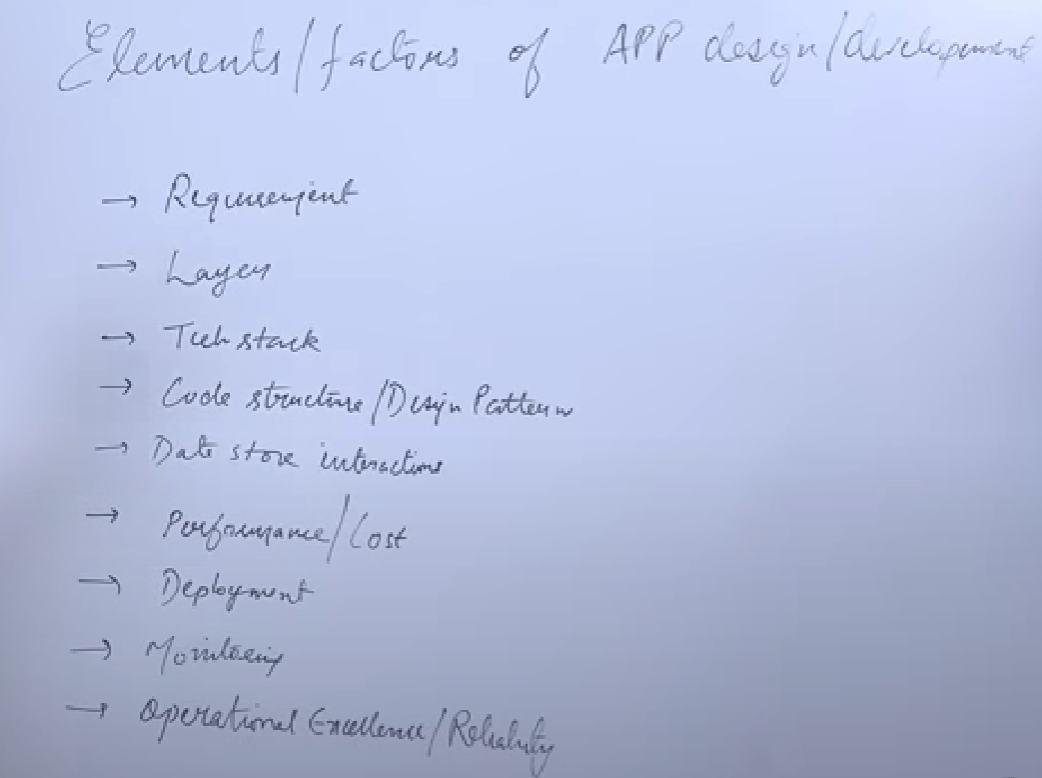
Key-value stores: Redis, DynamoDB (stores data as key-value pairs).

Wide-column stores: Apache Cassandra, HBase (stores data in columns rather than rows).

Application/Services

An application, short for "software application," is a computer program or a set of programs designed to perform specific tasks or functions for end-users, businesses, or other software applications. Applications can vary widely in complexity and purpose, ranging from simple mobile apps to complex enterprise software.





API (Application Programming Interface):

An Application Programming Interface (API) is a set of protocols, routines, and tools for building software and applications.

APIs define how different software components should interact, allowing applications to communicate with each other.

APIs can be used to enable communication between different software systems, services, or libraries.

Key Aspects of APIs:

Endpoints:

APIs expose specific endpoints (URLs or URIs) that represent functionalities or resources.

Requests and Responses:

Clients (applications or systems) make requests to API endpoints, and the API responds with data or performs specific actions.

Data Formats:

APIs define the format in which data is exchanged. Common formats include JSON (JavaScript Object Notation) and XML (eXtensible Markup Language).

Methods:

APIs use HTTP methods (GET, POST, PUT, DELETE, etc.) to specify the type of operation being requested.

Hide Implementation Details:

APIs allow developers to interact with a system or service without needing to understand its internal workings. The complexity of the underlying implementation is abstracted away, and developers can focus on using the provided functionalities.

Expose Essential Functions:

APIs expose a specific set of functions or operations that are essential for a particular task or service. This selective exposure allows developers to access only what they need, promoting simplicity and ease of use.

Encapsulation:

APIs encapsulate the internal logic and data structures of a system. This means that developers interact with a well-defined interface, and the internal complexities are hidden from view. This encapsulation enhances modularity and maintainability.

Standardized Interfaces:

APIs provide standardized interfaces that abstract the details of how specific operations are performed. This standardization allows developers to work with different systems using a consistent set of methods and conventions.

Data Abstraction:

APIs abstract the underlying data structures and formats by defining clear and standardized formats for data exchange. For example, APIs often use JSON or XML to represent data in a structured manner, shielding developers from the intricacies of data storage and serialization.

Advantages of APIs:

Interoperability:

APIs facilitate interoperability by enabling different software systems to communicate and work together.

Modularity:

APIs allow for modularity in software development, where different components or services can be developed and maintained independently.

Reusability:

Once an API is developed, it can be reused in various applications, reducing development time and effort.

Factors Considered While Developing APIs:

Clarity of Purpose:

Clearly define the purpose of the API and the problem it aims to solve.

User Experience:

Design APIs with a focus on user experience, providing clear and concise interfaces for developers.

Scalability:

Consider the potential growth and scalability of the API, ensuring it can handle increased usage.

Security:

Implement robust security measures, including authentication, authorization, and encryption, to protect sensitive data.

Documentation:

Provide comprehensive and easy-to-understand documentation to assist developers in using the API effectively.

Consistency:

Maintain consistency in naming conventions, request/response formats, and overall design to enhance usability.

Cache

a cache refers to a hardware or software component that stores frequently accessed or recently used data to serve future requests more quickly. The primary goal of using a cache is to improve the overall performance and efficiency of a system by reducing the time it takes to retrieve data.

Cache invalidation and cache eviction are two important concepts in the management of caches. They both relate to how a cache handles data that may become outdated or needs to be replaced.

Cache Invalidation:

Definition: Cache invalidation is the process of removing or marking as invalid the cached data that corresponds to a particular piece of information in the underlying data source.

Purpose: The purpose of cache invalidation is to ensure that the cache contains up-to-date information. When the original data changes, the corresponding cache entry becomes outdated or invalid. To maintain data consistency, the cache must be informed that the cached data is no longer valid and needs to be refreshed.

Methods: There are different methods for cache invalidation, such as:

Time-based Invalidation: Set a time limit for how long an item in the cache is considered valid.

Event-based Invalidation: Invalidate the cache entry when a specific event occurs (e.g., data update, deletion, or addition).

Manual Invalidation: Explicitly invalidate a cache entry through manual intervention.

Cache Eviction:

Definition: Cache eviction is the process of removing or replacing items from the cache to make room for new or more relevant data.

Purpose: Caches have a finite capacity, and when the cache is full, decisions must be made about which items to remove. Cache eviction is necessary to free up space for new data and ensure that the cache remains effective in storing the most relevant information.

Strategies: There are various cache eviction strategies, including:

Least Recently Used (LRU): Evict the least recently accessed items from the cache.

First-In-First-Out (FIFO): Evict the oldest items added to the cache.

Random Replacement: Randomly select items for eviction.

Least Frequently Used (LFU): Evict the least frequently accessed items.

Size-based Eviction: Evict items based on the size of the data they occupy in the cache.

Cache patterns:-



REST API

REST, which stands for Representational State Transfer, is an architectural style for designing networked applications. A RESTful API (Application Programming Interface) is an implementation of the principles of REST. It is a set of rules and conventions for building and interacting with web services.

Key principles of REST include:

* Statelessness: Each request from a client to a server must contain all the information needed to understand and fulfill that request. The server should not store any information about the client's state between requests. This makes the system more scalable and easier to maintain.
* Resource-Based: Resources, such as data objects or services, are identified by URIs (Uniform Resource Identifiers). Resources can be manipulated using standard HTTP methods, such as GET, POST, PUT, and DELETE.
* Representation: Resources can have different representations, such as XML, JSON, or HTML. Clients interact with resources through these representations.
* Uniform Interface: The API should have a consistent and uniform interface, making it predictable and easy to use. This includes the use of standard HTTP methods and status codes.
* A RESTful API typically exposes a set of endpoints, each corresponding to a resource, and clients interact with these endpoints using HTTP methods. For example, to retrieve information about a resource, a client might send an HTTP GET request to the resource's URI. To create or update a resource, a client might use HTTP POST or PUT requests.
* RESTful APIs are widely used in web development for building scalable and maintainable web services. They are often preferred for their simplicity, scalability, and compatibility with the HTTP protocol.

State transfer vs stateless

Stateless:

* Definition: A system is stateless if each request from a client to a server contains all the information needed to understand and fulfill that request. In other words, the server does not store any information about the client's state between requests.

State Transfer:

* Definition: State transfer refers to the idea that the client and server communicate by transferring representations of resource states. When a client makes a request, it transfers the state needed to understand and process that request.
* Representation: Resources in RESTful APIs can have multiple representations (e.g., XML, JSON). The client interacts with these representations to create, update, retrieve, or delete resources.

Http vs rest

HTTP:

* Definition: HTTP is a protocol used for communication on the World Wide Web. It is the foundation of any data exchange on the Web and is an application layer protocol. It provides a standardized way for computers to communicate over the internet.
* Usage: HTTP defines a set of rules for how messages are formatted and transmitted, and how web servers and browsers should respond to various commands. It is primarily used for the transfer of hypertext (HTML) and other media files over the internet.

REST:

* Definition: REST, on the other hand, is an architectural style for designing networked applications. It is not a protocol like HTTP but a set of principles that can be applied to create scalable, loosely coupled, and stateless web services.
* Usage: RESTful APIs use HTTP as a communication protocol, but they adhere to the principles of REST. These principles include statelessness, resource-based representation, a uniform interface, and the use of standard HTTP methods.

Protocol vs guidelines

A protocol refers to a set of rules or standards that define how data is transmitted over a network or how different systems communicate. It is a formalized set of procedures and conventions that govern the behavior of entities involved in a communication process.

Guidelines are more flexible and offer general advice or recommendations without strict enforcement. They offer advice and best practices but leave room for interpretation and adaptation.

Protocols are crucial for ensuring standardization and interoperability, while guidelines provide a framework for making informed decisions within a broader context.

Path vs Query parameters

Path Parameter:

Location: Path parameters are included in the path of the URL.

Format: They are usually denoted by a placeholder in the path, such as /users/{id}.

Purpose: Path parameters are used to identify a specific resource or endpoint. They are often used for things like resource retrieval, where the parameter in the path indicates the unique identifier of the resource being accessed.

Example:

GET /users/123

Query Parameter:

Location: Query parameters are included in the URL after a question mark (?) and are in the form of key-value pairs, separated by ampersands (&).

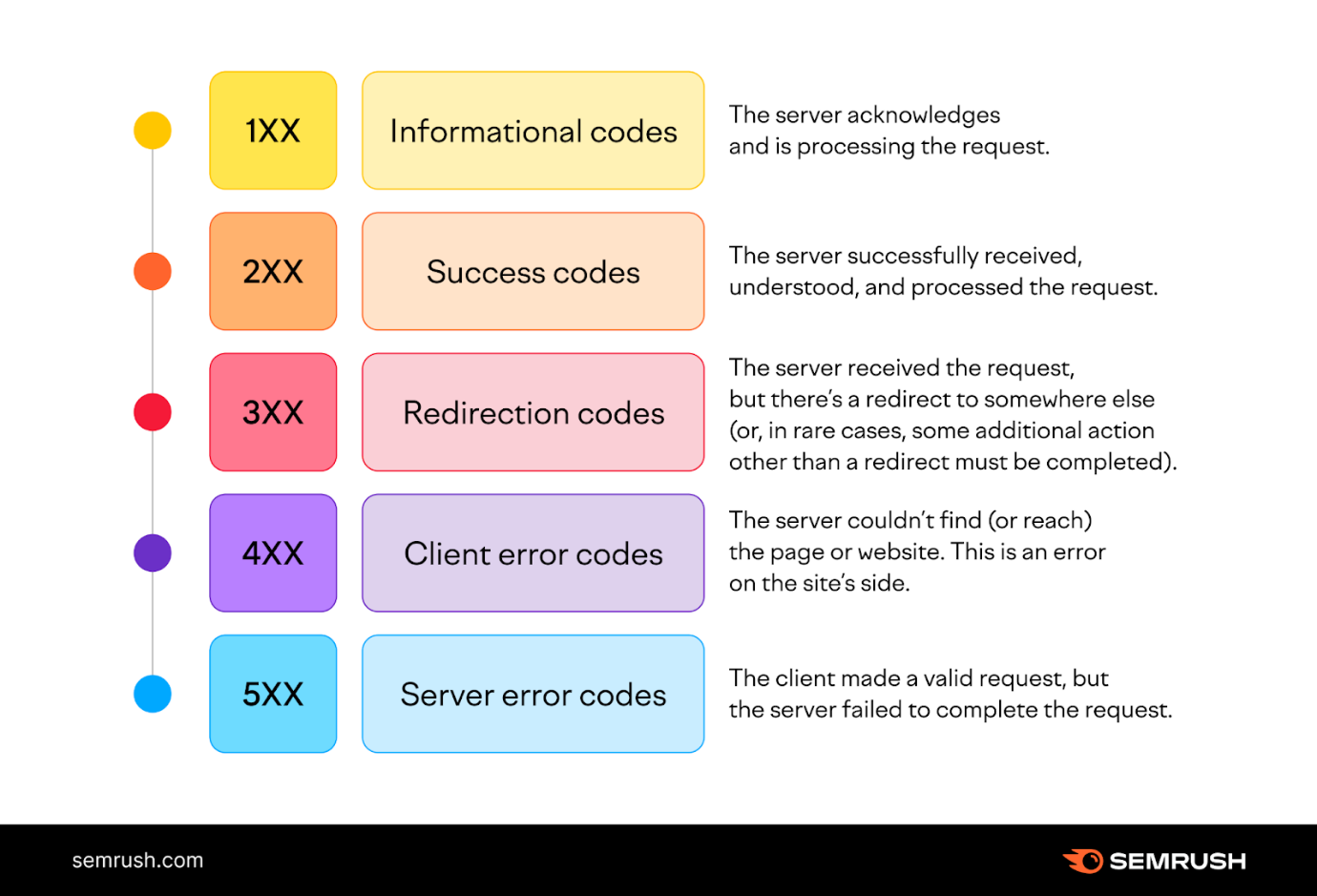
Format: They follow the key=value pattern, and you can have multiple query parameters separated by &.

Purpose: Query parameters are used to provide additional information to a server. They are often used for filtering, sorting, or specifying optional parameters for an operation.

Example:

GET /search?query=example&sort=date&limit=10

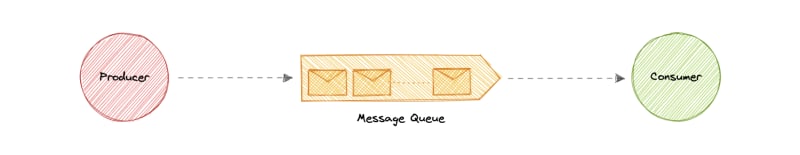
HTTP status codes



# Message Queues

A message queue is a form of service-to-service communication that facilitates asynchronous communication. It asynchronously receives messages from producers and sends them to consumers.

Queues are used to effectively manage requests in large-scale distributed systems. In small systems with minimal processing loads and small databases, writes can be predictably fast. However, in more complex and large systems writes can take an almost non-deterministic amount of time.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--_jJoQnjF--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_800/https:/raw.githubusercontent.com/karanpratapsingh/portfolio/master/public/static/courses/system-design/chapter-III/message-queues/message-queue.png)

**Working**

Messages are stored in the queue until they are processed and deleted. Each message is processed only once by a single consumer. Here's how it works:

* A producer publishes a job to the queue, then notifies the user of the job status.
* A consumer picks up the job from the queue, processes it, then signals that the job is complete.

**Advantages**

Let's discuss some advantages of using a message queue:

* **Scalability**: Message queues make it possible to scale precisely where we need to. When workloads peak, multiple instances of our application can all add requests to the queue without the risk of collision
* **Decoupling**: Message queues remove dependencies between components and significantly simplify the implementation of decoupled applications.
* **Performance**: Message queues enable asynchronous communication, which means that the endpoints that are producing and consuming messages interact with the queue, not each other. Producers can add requests to the queue without waiting for them to be processed.
* **Reliability**: Queues make our data persistent, and reduce the errors that happen when different parts of our system go offline.

## Features

### Push or Pull Delivery

Most message queues provide both push and pull options for retrieving messages. Pull means continuously querying the queue for new messages. Push means that a consumer is notified when a message is available.

### FIFO (First-In-First-Out) Queues

In these queues, the oldest (or first) entry, sometimes called the "head" of the queue, is processed first.

### Schedule or Delay Delivery

Many message queues support setting a specific delivery time for a message. If we need to have a common delay for all messages, we can set up a delay queue.

### At-Least-Once Delivery

Message queues may store multiple copies of messages for redundancy and high availability, and resend messages in the event of communication failures or errors to ensure they are delivered at least once.

### Exactly-Once Delivery

When duplicates can't be tolerated, FIFO (first-in-first-out) message queues will make sure that each message is delivered exactly once (and only once) by filtering out duplicates automatically.

### Dead-letter Queues

A dead-letter queue is a queue to which other queues can send messages that can't be processed successfully. This makes it easy to set them aside for further inspection without blocking the queue processing or spending CPU cycles on a message that might never be consumed successfully.

### Ordering

Most message queues provide best-effort ordering which ensures that messages are generally delivered in the same order as they're sent and that a message is delivered at least once.

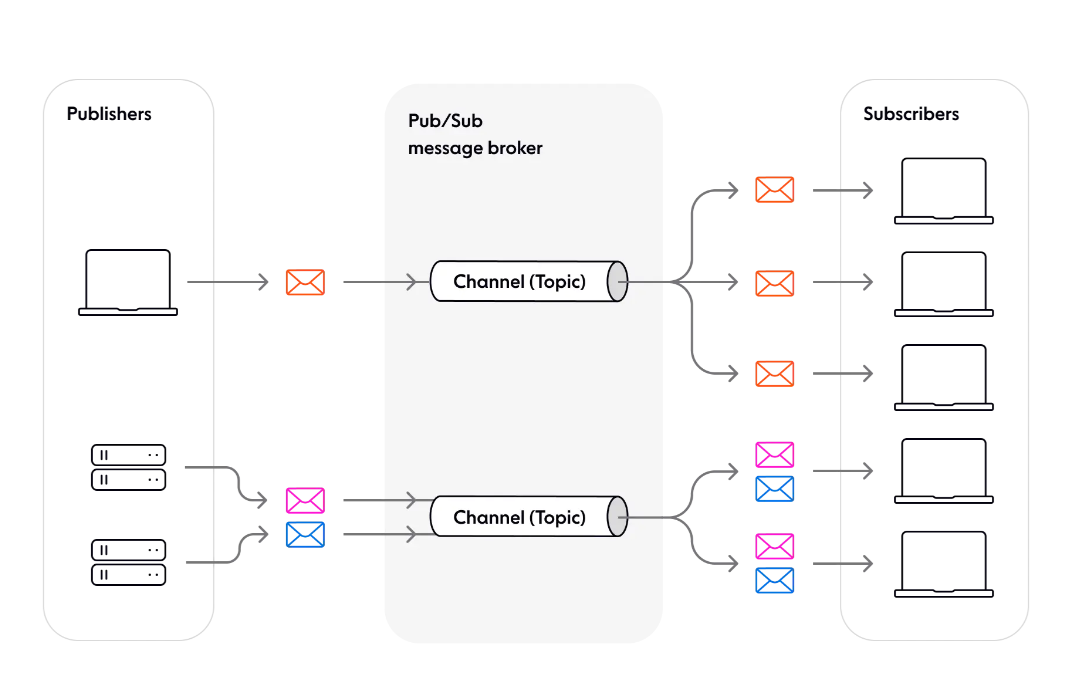
Pub-Sub Architecture

Pub/Sub provides a framework for exchanging messages between publishers (components that create and send messages) and subscribers (components that receive and consume messages).

Note that publishers don’t send messages to specific subscribers in a point-to-point manner. Instead, an intermediary is used - a Pub/Sub message broker, which groups messages into entities called channels (or topics).

Pub/Sub’s loose coupling, asynchronous nature, and inherent scalability make it an excellent solution for distributed systems with a high and fluctuating number of publishers and subscribers. You can use Pub/Sub for many different purposes, such as:

* Sending event notifications.
* Distributed caching.
* Distributed logging.
* Working with multiple data sources.
* Broadcasting updates (one-to-many messaging).
* Building responsive, low-latency end-user experiences like live chat and multiplayer collaboration functionality



Performance metrices

Throughput

Throughput generally refers to the rate at which a system or process can complete a task or deliver a service within a given time period.

It is a measure of the quantity of work performed in a specific amount of time and is often expressed in terms of operations per second, requests per second, transactions per second, or a similar unit.

Bandwidth

It indicates how much data can be transmitted over a communication channel in a given amount of time.

Measured in bits per second (bps), kilobits per second (Kbps), megabits per second (Mbps), or gigabits per second (Gbps)

Latency

Latency is a critical performance metric in computing and networking, representing the time delay between the initiation of a request and the receipt of the corresponding response. It is the time it takes for data to travel from the source to the destination.

Response time

It is the total time it takes for a system to respond to a user request. It includes both the processing time (time taken to perform the requested operation) and any waiting time (time spent in queues or delays).

Latency focuses on the time delay within specific components of a system (network, storage, processing), while response time encompasses the entire duration of request processing, including both processing and waiting times.

Faults and Failures

* A fault is a defect or error in the system that may or may not cause a failure. It is a deviation or mistake in the system's behavior or functionality.
* A fault might be a software bug that leads to incorrect data processing or a hardware malfunction that causes intermittent connectivity issues.
* A failure occurs when a fault leads to a deviation from the system's expected behavior, resulting in a loss of service or functionality. In other words, a failure is the manifestation of a fault that impacts the system's ability to perform its intended functions.
* A failure might involve a server crashing due to a software bug, resulting in the unavailability of a critical service until the issue is resolved.

Types of Faults

Transient Faults:

* Definition: Temporary and short-lived faults that occur but resolve on their own.
* Example: A momentary power fluctuation causing a glitch in a system.

Intermittent Faults:

* Definition: Occur periodically or under specific conditions, making them challenging to predict and reproduce consistently.
* Example: A network connection that drops intermittently.

Permanent Faults:

* Definition: Persistent and long-lasting faults that persist until addressed.
* Example: A defective hardware component that requires replacement.

Fault tolerance and fail-safe systems

Fault tolerance and fail-safe systems are design concepts employed to enhance the reliability and resilience of systems, particularly in critical applications where system failures can have severe consequences. Both concepts aim to minimize the impact of faults and failures, but they approach the problem in different ways.

Fault Tolerance

Fault Tolerance in Software refers to the system's ability to continue operating correctly, or at least in a degraded state, in the presence of software faults. The goal is to minimize the impact of software failures and provide uninterrupted service.

Key Features:

Redundancy: Software components or services may be replicated, and redundant instances can take over if one fails.

Error Handling: Comprehensive error handling mechanisms are implemented to gracefully handle unexpected situations or errors.

Automatic Recovery: Systems may automatically detect faults and initiate recovery processes, such as restarting failed components or rerouting requests.

Example:

In a distributed system, if a service fails to respond or returns an error, fault-tolerant mechanisms may redirect the request to another healthy instance of the service.

Fail-Safe Mechanisms

Fail-Safe Mechanisms in Software focus on preventing or mitigating the consequences of software failures to ensure safety and prevent irreversible damage.

Key Features:

Safe Default State: Systems are designed to default to a safe state in the event of a failure. This may involve stopping critical operations or reverting to a known stable configuration.

Emergency Shutdown: Critical components or processes may have mechanisms for emergency shutdown or isolation to prevent further harm.

Data Integrity Checks: Systems implement checks to ensure data integrity, and fail-safe mechanisms may involve rolling back transactions or using backup data in case of data corruption.

Example:

In a financial software system, if a critical calculation module encounters an error or produces unexpected results, the system may fail-safe by preventing further financial transactions until the issue is resolved.

Relationship and Integration:

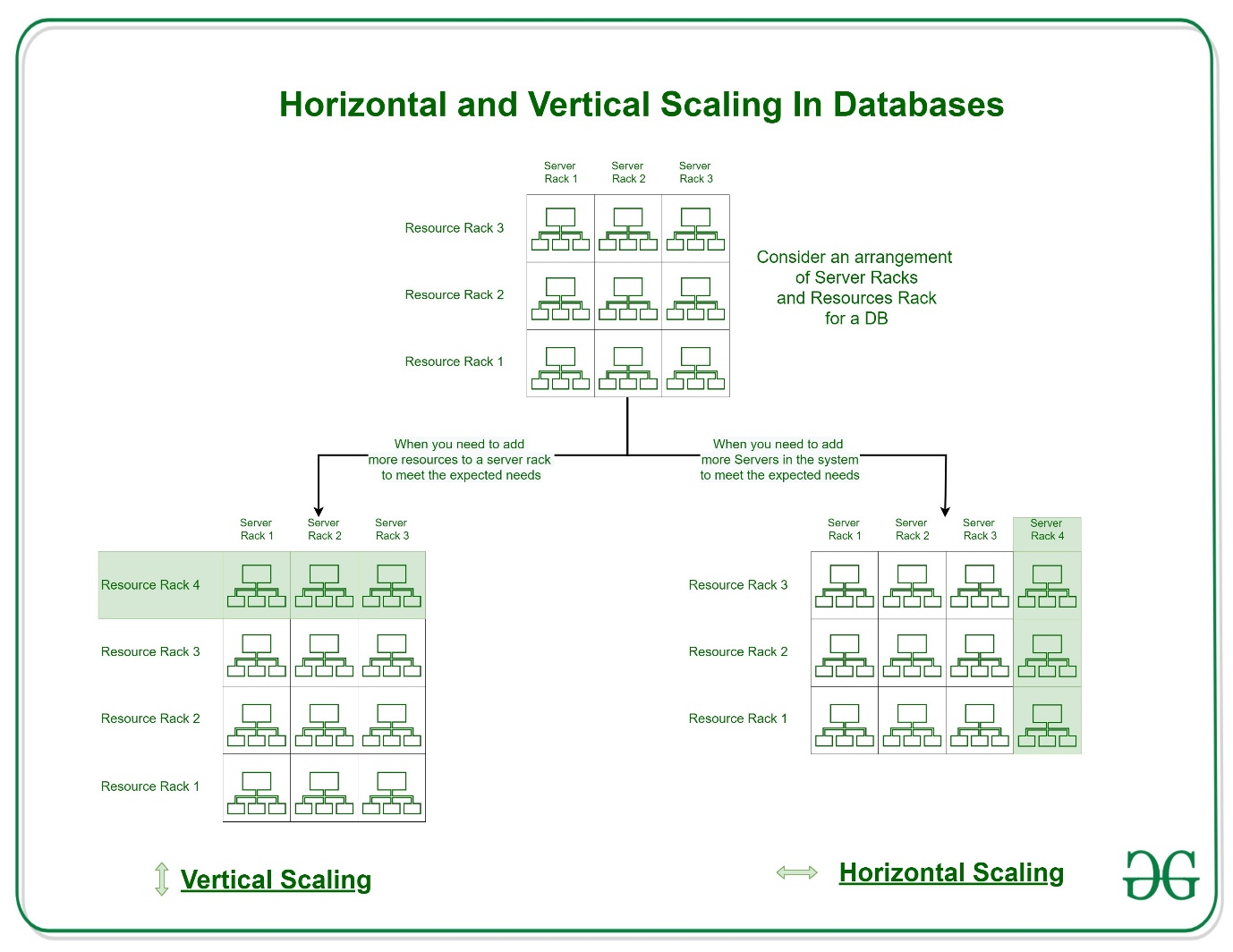
In practice, software systems often integrate both fault-tolerant and fail-safe strategies. For example:

* A fault-tolerant web service may have redundant servers and automatic recovery mechanisms to handle transient faults.
* A fail-safe database system may implement transaction rollbacks and integrity checks to prevent data corruption.

Scaling

Scaling in software systems refers to the process of adapting a system to handle increased workload or growth effectively. This can involve expanding the system's capacity, performance, or resources to ensure that it continues to perform well as demands increase.

Types of scaling



Database Replication

Database replication is a process that involves creating and maintaining copies of a database across multiple servers or locations. The primary goal of database replication is to improve system reliability, performance, and availability. Replication allows for redundancy, load balancing, and fault tolerance.

Replication lag

Replication lag in a database refers to the delay or latency between the time a change is made to the data on the source database (master) and the time it is replicated and applied to the destination databases (slaves or replicas). This lag can be caused by various factors and can impact the consistency and timeliness of data across the replicated databases.

Synchronous and asynchronous database replication

Synchronous

In synchronous replication, transactions are not considered complete until the changes are replicated to all replica databases, and acknowledgment is received by the master database.

Characteristics:

* Transaction Consistency: Synchronous replication ensures that all replicas are consistent with the master database at the end of each transaction.
* Blocking Nature: The master database waits for acknowledgment from all replicas before confirming the completion of a transaction. This can introduce latency and potentially impact the performance of the master.
* High Reliability: Since all replicas are guaranteed to have the same data, synchronous replication provides high data integrity and reliability.

Advantages:

* Consistency: Guarantees data consistency across all replicas.
* Reliability: Ensures high reliability and data integrity.

Disadvantages:

* Latency: The blocking nature of synchronous replication can introduce latency, especially if replicas are located far from the master.
* Performance Impact: The master database's performance can be impacted as it has to wait for acknowledgments from all replicas.

Use Cases:

* Critical Transactions: Applications where consistency and reliability are critical, such as financial transactions.
* High Data Integrity Requirements: Environments where ensuring data integrity is of utmost importance.

Asynchronous

In asynchronous replication, transactions are considered complete as soon as they are committed on the master database. Replication to the replicas occurs independently and asynchronously.

Characteristics:

* Transaction Independence: The master database does not wait for acknowledgment from replicas, allowing it to continue processing transactions without being delayed.
* Potential Lag: There may be a delay (lag) between changes being made on the master and those changes being replicated to the replicas.
* Lower Impact on Master: Since the master does not wait for replicas, there is less impact on its performance.

Advantages:

* Reduced Latency: Asynchronous replication can reduce latency on the master, allowing it to process transactions more quickly.
* Higher Throughput: The master is not blocked by the replication process, enabling higher throughput.

Disadvantages:

* Potential Data Inconsistency: Asynchronous replication may lead to temporary inconsistencies between the master and replicas.
* Limited Real-time Consistency: Applications relying on real-time consistency may find asynchronous replication less suitable.

Use Cases:

* Read Scaling: Environments where read scalability is crucial, and the occasional data inconsistency is acceptable.
* Systems with Geographic Distribution: When replicas are located in different geographical regions, and network latency is a concern.

CAP theorem

The CAP theorem, also known as Brewer's theorem, is a fundamental principle in distributed systems that describes the trade-offs between three key properties: Consistency, Availability, and Partition Tolerance. The CAP theorem states that in a distributed system, it is impossible to simultaneously achieve all three of these properties. A distributed system refers to a collection of interconnected nodes or components that work together to achieve a common goal.

Consistency (C): All nodes in the system see the same data at the same time. In other words, a read operation will return the most recent write.

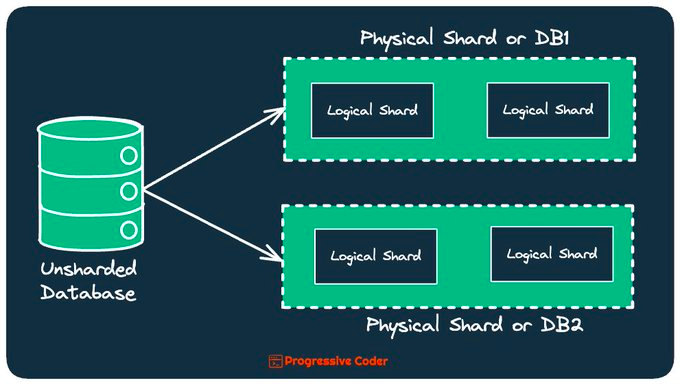
Availability (A): Every request to the system receives a response, without the guarantee that it contains the most recent version of the data. Availability implies that every node (or a majority of nodes) in the system is accessible and responsive.

Partition Tolerance (P): The system continues to operate even when network partitions occur, meaning communication between nodes is disrupted or delayed.

* CA systems (Consistent and Available): These systems sacrifice Partition Tolerance. They ensure that all nodes in the system have the same data and are available for read and write operations as long as there is no network partition.
* CP systems (Consistent and Partition Tolerant): These systems sacrifice Availability. They prioritize consistency and partition tolerance, meaning they may not be available for read and write operations during network partitions.
* AP systems (Available and Partition Tolerant): These systems sacrifice Consistency. They prioritize availability and partition tolerance, meaning that each node may have a different view of the data, and there might be eventual consistency.

Database sharding

Database sharding is a technique used in distributed database systems to improve scalability and performance by horizontally partitioning data across multiple servers. In the context of databases, a shard refers to a horizontal partition of data, and each shard is stored on a separate database server or node. This allows the distributed database system to distribute the load of queries and transactions across multiple servers, enabling it to handle larger volumes of data and a higher number of transactions.



Key-based sharding (Hash-based)

Key-based sharding is a technique used in distributed database systems to horizontally partition data based on a chosen key or attribute, referred to as the shard key. The shard key is critical because it determines how data is distributed across multiple shards or database servers. This approach helps distribute the workload and storage requirements, enabling the system to scale horizontally.

Eg : If a social media application uses user IDs as the shard key, each shard may be responsible for storing the data related to a specific range of user IDs. For example, shard 1 may handle user IDs 1-1000, shard 2 may handle user IDs 1001-2000, and so on.

Range-based sharding

Range-based sharding is a technique in distributed database systems where data is horizontally partitioned based on ranges of a selected attribute or key. This approach involves dividing the dataset into non-overlapping ranges, and each range is assigned to a specific shard or database server. Range-based sharding is particularly useful when the data has a natural ordering, and queries often involve retrieving data within specific ranges.

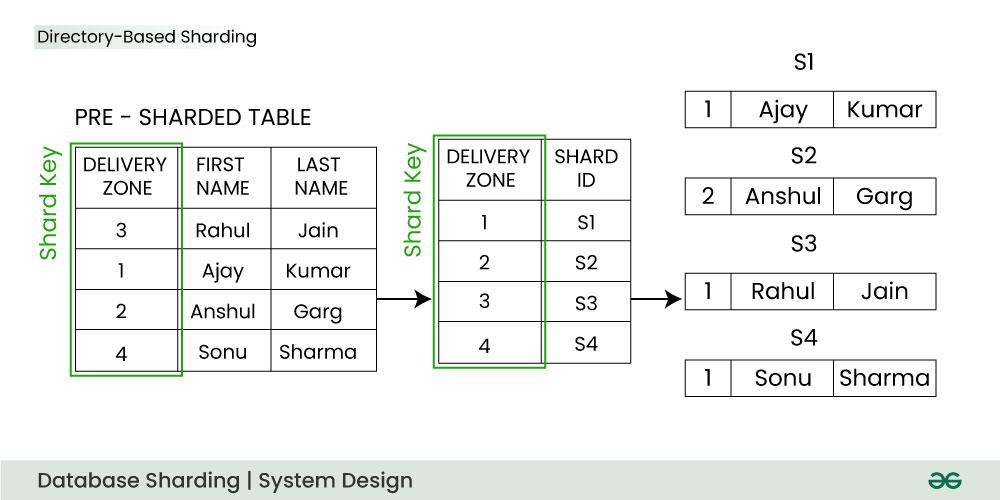
Eg : In a time-series database, range-based sharding might involve partitioning the data based on timestamps. Each shard could be responsible for a specific time range (e.g., Shard 1 for data from 2022-01-01 to 2022-01-31, Shard 2 for data from 2022-02-01 to 2022-02-28, and so on).

Directory-based Sharding

In this method, we create and maintain a lookup service or lookup table for the original database.

Basically we use a shard key for lookup table and we do mapping for each entity that exists in the database.

This way we keep track of which database shards hold which data.

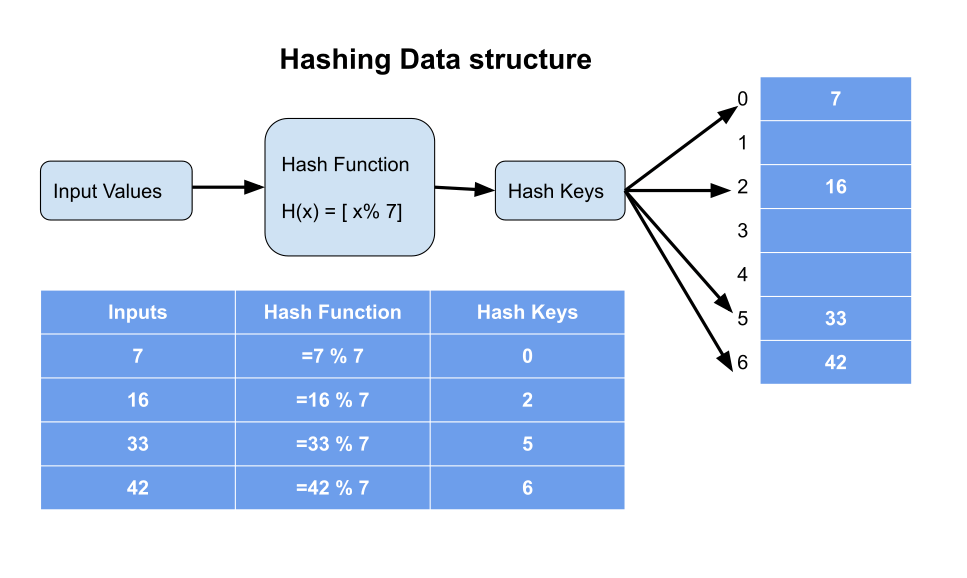


The lookup table holds a static set of information about where specific data can be found. In the above image, you can see that we have used the delivery zone as a shard key:

* Firstly the client application queries the lookup service to find out the shard (database partition) on which the data is placed.
* When the lookup service returns the shard it queries/updates that shard.

Hashing

Hashing is a fundamental concept in system design that involves the use of hash functions to map data of arbitrary size to fixed-size values, typically for the purpose of indexing and retrieval. In system design, hashing is employed in various scenarios to achieve efficient data storage, retrieval, and manipulation.



Consistent hashing

Consistent hashing is a technique used in distributed systems to efficiently distribute data across a changing set of nodes. It was introduced to address the challenges associated with adding or removing nodes in a distributed hash table or key-value store, minimizing the need for rehashing and redistributing data when the number of nodes changes.

<https://ably.com/blog/implementing-efficient-consistent-hashing>

FUNCTIONAL AND NON-FUNCTIONAL REQUIREMENTS

FUNCTIONAL REQUIREMENTS

Definition: Functional requirements describe the specific features, capabilities, and functionalities that the system must provide to meet the needs of its users.

Examples:

User authentication: The system should allow users to log in using their username and password.

Data storage: The system should store user data, including profiles, preferences, and transaction history.

Search functionality: The system should provide users with the ability to search for products, articles, or information.

Key Characteristics:

What the System Does: Functional requirements focus on the actions and behaviors of the system.

Measurable: Functional requirements are often verifiable through testing to ensure that the system functions as intended.

Non-Functional Requirements:

Definition: Non-functional requirements specify the qualities or characteristics that define how well the system performs its functions. They are not concerned with specific behaviors but rather with attributes such as performance, scalability, security, and usability.

Examples:

Performance: The system should respond to user requests within 2 seconds.

Scalability: The system should handle a 20% increase in user traffic without significant performance degradation.

Security: User passwords should be stored securely using industry-standard encryption algorithms.

Usability: The system should have an intuitive user interface with clear navigation.

Key Characteristics:

How Well the System Performs: Non-functional requirements focus on system attributes and qualities.

Qualitative: Non-functional requirements are often qualitative and subjective, describing the system's characteristics rather than specific features.