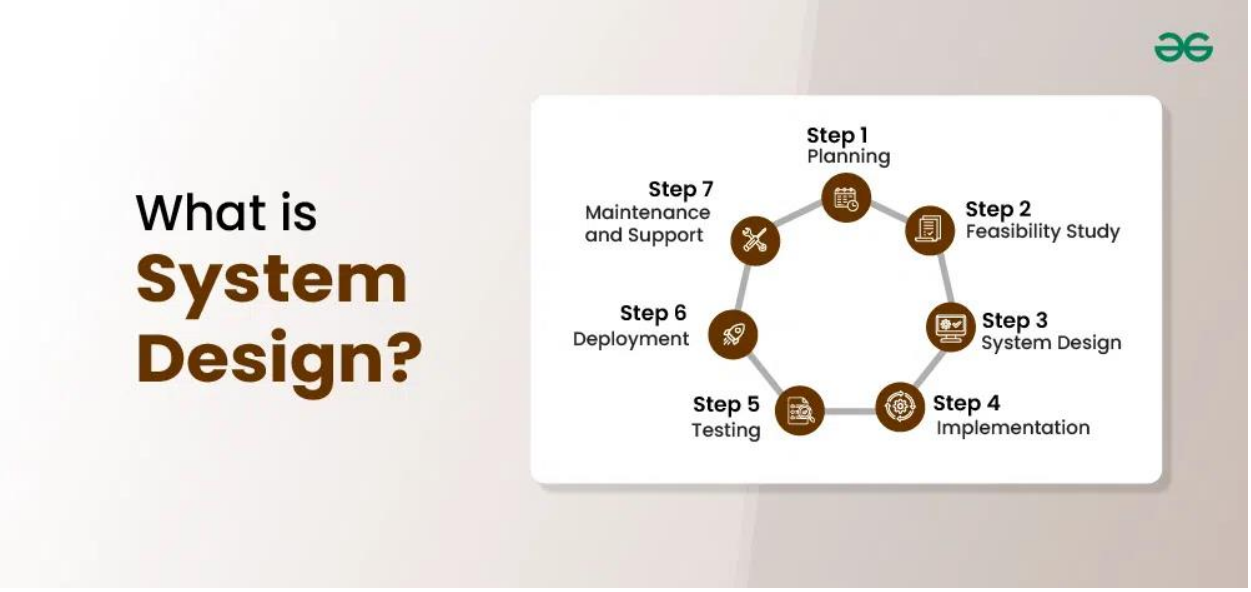
Week-3

System Designing:

Systems Design is the process of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements. It involves translating user requirements into a detailed blueprint that guides the implementation phase.



Design Patterns: Design patterns are standardized solutions to common problems that arise in software design. They provide a template or blueprint that developers can use to address specific challenges in a flexible and efficient manner. Design patterns help promote best practices,enhance code reusability, and improve communication among developers.

There are three main categories of design patterns:

− Creational Patterns: These patterns deal with object creation mechanisms, aiming to create

objects in a manner suitable to the situation. Examples include Singleton, Factory Method,

and Abstract Factory.
− Structural Patterns: These patterns focus on how classes and objects are composed to form
larger structures. Examples include Adapter, Composite, and Decorator.
− Behavioral Patterns: These patterns define how objects interact and communicate with one
another. Examples include Observer, Strategy, and Command.
By using design patterns, developers can improve the scalability, maintainability, and clarity of their
code, ultimately leading to more robust software solutions.

Types:

1. High-Level Design (HLD)
2. Low-Level Design (LLD)
3. Distributed System Design
4. Object-Oriented Design (OOD)
5. Microservices Design
6. Database System Design
7. Real-Time System Design
8. Cloud-Based System Design
9. Network System Design
10. Embedded System Design
11. Service-Oriented Architecture (SOA) Design
12. Event-Driven System Design
13. Enterprise System Design

The key components of system design:

1. Scalability: Ability to handle increased load by adding resources.
2. Reliability: Ensuring the system performs consistently under defined conditions.
3. Availability: Ensuring the system is operational and accessible when needed.
4. Performance: Measures response time, throughput, and resource usage.
5. Latency: Time taken for data to travel from source to destination.
6. Throughput: Amount of data processed over a given time period.
7. Consistency: Ensuring data remains the same across different nodes.
8. Partition Tolerance: System continues to operate even when network partitions occur.
9. Fault Tolerance: Ability to continue functioning when some components fail.
10. Load Balancing: Distributing workloads across multiple resources.
11. Caching: Storing frequently accessed data for faster retrieval.
12. Database Design: Choosing the right database (SQL/NoSQL) and schema design.
13. Sharding: Partitioning data across multiple databases or servers.
14. Concurrency: Managing multiple operations simultaneously.
15. Message Queuing: Decoupling processes through asynchronous communication.

16. Security: Protecting system from unauthorized access and attacks.
17. Monitoring and Logging: Tracking system health and performance over time.
18. Data Replication: Copying data across systems for redundancy and reliability.
19. API Design: Creating interfaces for communication between system components.
20. Eventual Consistency: Allowing temporary inconsistencies with the guarantee that data will converge over time.

System Designing Concepts :

1. Load Balancers

Definition

A load balancer is a device or software that distributes incoming network traffic across multiple servers.

Importance in System Design

Load balancers enhance application availability and reliability by ensuring that no single server is overwhelmed with traffic. They are essential in designing scalable systems that can handle varying loads.

Key Considerations

- Types of Load Balancers: Hardware vs. Software
- Load Balancing Algorithms: Round Robin, Least Connections, IP Hashing
- Benefits: Fault tolerance, improved performance

2. Caching

Definition
Caching involves storing frequently accessed data in a temporary storage area to reduce retrieval time.
Importance in System Design
Caching improves performance and reduces latency, making it critical for high-traffic applications. Effective caching strategies are vital for optimizing resource usage.
Key Considerations

- Types of Caches: In-memory vs. Disk caching
- Cache Invalidation Strategies: Time-based, event-based
- Cache Eviction Policies: LRU, LFU

3. Sharding (Data Partitioning)
Definition
Sharding is the practice of dividing a database into smaller, more manageable pieces called shards.
Importance in System Design
Sharding enhances performance and scalability by distributing data across multiple servers. It is crucial for handling large datasets and high throughput.
Key Considerations

- Horizontal vs. Vertical Sharding
- Benefits: Improved parallelism, reduced load on individual databases

4. Indexes (Indexing)
Definition
Indexes are data structures that improve the speed of data retrieval operations in databases.
Importance in System Design
Effective indexing strategies are essential for optimizing query performance, directly impacting application responsiveness.

Key Considerations

- Types of Indexes: B-tree, Hash, Full-text
- Trade-offs: Faster reads vs. slower writes

5. Proxy Server

Definition

A proxy server acts as an intermediary that forwards requests from clients to other servers.

Importance in System Design

Proxy servers enhance security, load balancing, and caching, making them a valuable component in designing robust systems.

Key Considerations

- Types of Proxies: Forward vs. Reverse proxies
- Use Cases: Security, anonymity, load distribution

6. Messaging Queue

Definition

A messaging queue facilitates asynchronous communication between services.

Importance in System Design

Messaging queues decouple service dependencies, enabling systems to handle high loads and improve resilience.

Key Considerations

- Popular Tools: RabbitMQ, Apache Kafka
- Benefits: Buffers messages, reduces coupling

7. Choosing Database (SQL vs. NoSQL)

Definition

The choice between SQL (relational) and NoSQL (non-relational) databases significantly impacts how data is structured and accessed.

Importance in System Design

Understanding the strengths and weaknesses of each database type is crucial for designing data-driven applications.

Key Considerations

- SQL: Structured data, ACID properties
- NoSQL: Flexibility, scalability

8. Monolithic vs. Microservices Architecture

Definition

Monolithic architecture refers to a single unified codebase, while microservices architecture breaks the application into smaller, independently deployable services.

Importance in System Design

Choosing the right architectural style affects scalability, maintainability, and complexity, making it a key decision in system design.

Key Considerations

- Trade-offs: Simplicity vs. complexity, flexibility vs. performance

9. REST APIs

Definition

REST (Representational State Transfer) APIs are architectural styles for designing networked applications.

Importance in System Design

RESTful APIs enable seamless communication between client and server, a core aspect of modern web applications.

Key Considerations

- Principles: Stateless communication, resource-based URLs
- Benefits: Scalability, integration ease

10. Hashing

Definition

Hashing is the process of converting input data into a fixed-size string (hash) using a hash function.

Importance in System Design

Hashing is essential for data integrity checks and efficient data retrieval, particularly in databases.

Key Considerations

- Common Algorithms: MD5, SHA-256
- Applications: Hash tables, data integrity verification

11. Consistent Hashing

Definition

Consistent hashing is a technique that minimizes reorganization when nodes are added or removed in a distributed system.

Importance in System Design

It enhances load distribution and reduces data movement, which is critical for scalable systems.

Key Considerations

- Use Cases: Distributed caching, sharding

12. Kafka

Definition

Kafka is a distributed event streaming platform for high-throughput, low-latency data pipelines.

Importance in System Design

Kafka enables real-time analytics and data integration, making it essential for building responsive applications.

Key Considerations

- Components: Producers, Consumers, Brokers
- Use Cases: Log aggregation, real-time data processing

13. LRU Cache

Definition

An LRU (Least Recently Used) cache eviction policy removes the least recently accessed items first.

Importance in System Design

Implementing LRU caching optimizes memory usage and improves data retrieval speeds in systems.

Key Considerations

- Implementation Strategies: Doubly linked list, HashMap

14. Apache Hadoop

Definition

Hadoop is an open-source framework for distributed storage and processing of large datasets.

Importance in System Design

Understanding Hadoop's architecture is vital for designing big data applications and processing workflows.

Key Considerations

- Components: HDFS, MapReduce
- Use Cases: Big data analytics, data lakes

15. HDFS (Hadoop Distributed File System)

Definition

HDFS is a distributed file system designed for storing large files across multiple nodes.

Importance in System Design

HDFS is critical for managing data in big data systems and ensuring fault tolerance.

Key Considerations

- Features: High throughput, fault tolerance
- Use Cases: Storing large datasets

16. HBase

Definition

HBase is a distributed NoSQL database built on HDFS for real-time read/write access to large datasets.

Importance in System Design

HBase is essential for applications requiring fast data access and high scalability.

Key Considerations

- Data Model: Column-oriented storage
- Use Cases: Real-time analytics

17. Zookeeper

Definition

Zookeeper is a centralized service for managing configuration information and providing distributed synchronization.

Importance in System Design

Zookeeper simplifies coordination between distributed services, enhancing system reliability.

Key Considerations

- Features: High availability, fault tolerance
- Use Cases: Configuration management, leader election

18. Solr

Definition

Solr is an open-source search platform built on Apache Lucene for powerful full-text search capabilities.

Importance in System Design

Integrating Solr into applications enables complex search functionalities essential for user experience.

Key Considerations

- Features: Faceted search, distributed indexing
- Use Cases: Search engines, big data applications

19. Cassandra

Definition

Cassandra is a distributed NoSQL database designed for high availability and scalability without a single point of failure.

Importance in System Design

Cassandra is crucial for applications that require handling large amounts of structured data.

Key Considerations

- Data Model: Column-family
- Use Cases: Real-time data applications

21. URL Shortener Design

Definition

A URL shortener is a service that converts long URLs into shorter, more manageable links.

Importance in System Design

Designing a URL shortener involves practical applications of hashing, database design, and user interaction.

Key Considerations

- Collision Handling: Ensuring unique short URLs
- Database Design: Mapping original URLs to short URLs

22. Pastebin-like Service Design

Definition

A Pastebin-like service allows users to store and share text snippets or documents.

Importance in System Design

This service design illustrates data storage, retrieval, and user management principles in application development.

Key Considerations

- Data Storage: Managing user inputs and pastes
- Privacy Options: Expiration of pastes and access controls

23. Design Dropbox/OneDrive-like Services

Definition

These services provide cloud storage solutions for file sharing and synchronization.

Importance in System Design

Designing such services requires a deep understanding of file management, security, and user experience.

Key Considerations

- Synchronization Algorithms: Ensuring consistency across devices
- User Authentication: Managing access and security

24. HTTP vs. HTTPS

Definition

HTTP (Hypertext Transfer Protocol) and HTTPS (HTTP Secure) are protocols for transferring data over the web.

Importance in System Design

Understanding the differences between these protocols is essential for designing secure web applications.

Key Considerations

- Security Features: Encryption and integrity checks in HTTPS
- Use Cases: E-commerce, sensitive data transfer

25. CAP Theorem

Definition

The CAP theorem states that in a distributed system, you can only achieve two out of three guarantees: Consistency, Availability, and Partition Tolerance.

Importance in System Design

Understanding the CAP theorem is vital for making architectural decisions in distributed systems.

Key Considerations

- Trade-offs: Deciding which two guarantees to prioritize
- Use Cases: Designing scalable and resilient systems