System design is the process of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements. It involves breaking down complex systems into manageable parts, ensuring that all components work together efficiently.

Key aspects of system design include:

1. **Requirements Analysis**: Understanding what the system needs to accomplish, including functional and non-functional requirements.
2. **Architecture**: Defining the overall structure, including how different components interact, the technologies to be used, and the deployment environment.
3. **Component Design**: Specifying the individual parts of the system, their responsibilities, and how they communicate.
4. **Data Design**: Organizing how data is stored, accessed, and manipulated, often involving database design.
5. **Scalability and Performance**: Ensuring the system can handle growth and performs efficiently under load.
6. **Security**: Incorporating measures to protect the system from unauthorized access and vulnerabilities.
7. **Testing and Validation**: Planning how to verify that the system meets its requirements and functions as intended.

A large-scale distributed system is a system that consists of multiple interconnected components spread across various locations, working together to achieve a common goal. These systems can handle a significant amount of data and user requests while providing resilience, scalability, and fault tolerance.

Design patterns are general reusable solutions to common problems that occur in software design. They represent best practices and can be adapted to fit specific situations. Understanding design patterns can significantly improve code organization, readability, and maintainability.

Live Streaming System Design

Key Components:

1. **Video Capture**: Cameras or devices to capture video content.
2. **Encoding**: Compressing and converting video to a suitable format (e.g., H.264).

3. **Streaming Server**: Distributes the encoded video to users (e.g., using protocols like RTMP, HLS).
4. **Content Delivery Network (CDN)**: Ensures low-latency delivery by caching content across multiple locations.
5. **Player**: Web/mobile app component that receives and displays the video stream.

Fault Tolerance

Definition: Fault tolerance is the ability of a system to continue operating properly in the event of the failure of some of its components.

Key Concepts:

1. **Redundancy**: Implementing duplicate components (e.g., servers, databases) to take over in case of failure.
2. **Error Detection**: Mechanisms to identify faults, such as checksums and heartbeats.
3. **Graceful Degradation**: Maintaining partial functionality even when some components fail.
4. **Replication**: Keeping copies of data across multiple locations to prevent data loss.
5. **Failover**: Automatically switching to a standby system or component when a failure occurs.

Benefits:

- Increases system reliability and availability.
- Minimizes downtime and service disruption.
- Enhances user confidence in system performance.

Extensibility

Definition: Extensibility is the design characteristic that allows a system to accommodate future growth or changes without major alterations to its existing structure.

Key Aspects:

- **Modular Architecture**: Components can be added or replaced easily.
- **Plugins**: Support for third-party extensions or features.

- **APIs**: Well-defined interfaces to integrate new functionalities.

Testing

Importance: Testing ensures that the system meets its requirements and functions as expected. It helps identify and fix bugs before deployment.

Types of Testing:

- **Unit Testing**: Tests individual components for correctness.
- **Integration Testing**: Ensures that combined components work together.
- **End-to-End Testing**: Tests the complete application workflow.
- **Load Testing**: Assesses system performance under heavy load.

Core Requirement - Streaming Video

Key Requirements:

- **Low Latency**: Quick delivery for real-time interaction.
- **High Availability**: Continuous access during peak loads.
- **Adaptive Bitrate Streaming**: Adjusts video quality based on network conditions.
- **Security**: Protects content from unauthorized access.

Diagramming Approaches

1. **Architecture Diagram**: Visualizes the overall system architecture, showing components like encoding, streaming servers, CDNs, and clients.
2. **Data Flow Diagram**: Illustrates how data moves through the system, from video capture to playback.
3. **Sequence Diagram**: Depicts the interactions between components during video streaming, highlighting processes like encoding, distribution, and playback.

API Design

Definition: Creating a set of rules and protocols for how software components communicate. Focus on RESTful or GraphQL design for efficient data retrieval and manipulation.

Database Design

Definition: Structuring a database to optimize performance, scalability, and data integrity. Key considerations include normalization, indexing, and choosing between SQL vs. NoSQL databases.

Network Protocols

Overview: Set of rules governing data transmission over networks. Common protocols include TCP/IP for reliability and HTTP/HTTPS for web communications.

Choosing a Datastore

Considerations:

- **Data Structure**: JSON, relational data, or time-series.
- **Scalability**: Ability to handle growth in data and user load.
- **Performance**: Read/write speeds and query efficiency.

Uploading Raw Video Footage

Process:

- Use APIs to handle video uploads.
- Implement chunked uploads for large files.
- Ensure file validation and storage in a reliable format.

MapReduce for Video Transformation

Purpose: Distributed processing framework to handle video transformation tasks, such as transcoding or editing, across multiple nodes for efficiency and scalability.

WebRTC vs. MPEG DASH vs. HLS

- **WebRTC**: Peer-to-peer protocol for real-time communication; low latency.
- **MPEG DASH**: Adaptive streaming standard that delivers content based on user bandwidth.
- **HLS (HTTP Live Streaming)**: Apple's adaptive streaming protocol, widely supported, but with slightly higher latency compared to WebRTC.

Content Delivery Networks (CDN)

Overview: CDNs are systems of distributed servers that deliver content to users based on their geographic location. They enhance performance and reduce latency by caching content closer to users.

High-Level Summary

Key Points:

- **Purpose**: Improve load times and provide redundancy.
- **Components**: Edge servers, origin servers, and a control plane for managing requests.
- **Benefits**: Increased availability, scalability, and reliability for streaming services.

Introduction to Low-Level Design

Focus: Translating high-level architecture into detailed component designs. This involves specifying data structures, algorithms, and interactions between components to ensure efficient implementation.

Video Player Design

Key Features:

- **UI/UX**: Intuitive interface for playback controls, settings, and playlists.
- **Functionality**: Supports adaptive bitrate streaming, captions, and DRM for content protection.
- **Compatibility**: Works across various devices and browsers, ensuring seamless playback.

Engineering Requirements

Considerations:

- **Performance**: Minimal buffering and quick load times.
- **Scalability**: Ability to handle high traffic loads during peak times.
- **Reliability**: Fault tolerance to maintain uptime and user experience.
- **Security**: Protects against unauthorized access and ensures data integrity.

Use Case UML Diagram

- **Description**: Visualizes the system's functionality from a user’s perspective.

- **Elements**:
    - **Actors**: Users or other systems interacting with the system.
    - **Use Cases**: Specific functionalities or processes the system offers.
    - **Relationships**: Lines connecting actors to use cases, indicating interactions.

Class UML Diagram

- **Description**: Depicts the static structure of the system, highlighting classes and their relationships.
- **Elements**:
    - **Classes**: Defined by attributes and methods.
    - **Relationships**: Includes associations, generalizations (inheritance), and aggregations/compositions.
- **Purpose**: Helps in understanding the data model and how different entities relate.

Sequence UML Diagram

- **Description**: Illustrates how objects interact over time for a specific use case.
- **Elements**:
    - **Objects**: Represented as lifelines.
    - **Messages**: Arrows indicating communication between objects.
- **Purpose**: Focuses on the order of operations and the flow of control.

Coding the Server

- **Overview**: The implementation phase where the server-side logic is developed.
- **Considerations**:
    - **Technology Stack**: Choice of programming language and frameworks (e.g., Node.js, Django).
    - **Architecture**: Designing RESTful APIs or GraphQL endpoints.
    - **Database Interaction**: Handling data storage and retrieval.

Resources for System Design

- **Types of Resources**:

- **Books**: Recommended reads on system architecture and design patterns.
- **Online Courses**: Platforms like Coursera or Udacity offering system design courses.
- **Documentation**: Framework and technology-specific documentation for best practices.
- **Communities**: Forums and discussion groups (e.g., Stack Overflow, Reddit) for peer advice and shared experiences.

Load balancers are crucial components in network architecture that help distribute incoming traffic across multiple servers to ensure reliability, efficiency, and availability. Here's a detailed overview:

What is a Load Balancer?

A load balancer acts as a traffic manager, sitting between client devices and server resources. It receives incoming requests and routes them to the appropriate server based on various algorithms.

Types of Load Balancers

1. **Hardware Load Balancers**:
   - Physical devices dedicated to distributing network or application traffic.
   - Often offer high performance and reliability.
2. **Software Load Balancers**:
   - Applications that run on standard hardware or virtual machines.
   - More flexible and cost-effective (e.g., NGINX, HAProxy).
3. **Cloud Load Balancers**:
   - Offered by cloud service providers (e.g., AWS Elastic Load Balancing, Google Cloud Load Balancing).
   - Automatically scales and manages traffic based on demand.

Load Balancing Algorithms

1. **Round Robin**: Distributes requests evenly across servers in a rotating manner.
2. **Least Connections**: Routes traffic to the server with the fewest active connections.

3. **IP Hash**: Uses the client's IP address to determine which server should handle the request, ensuring that the same client is consistently directed to the same server.
4. **Weighted Load Balancing**: Assigns different weights to servers based on their capacity, routing more traffic to more powerful servers.

Benefits of Load Balancers

- **Improved Availability**: If one server fails, the load balancer redirects traffic to other operational servers.
- **Scalability**: Easily add or remove servers based on traffic demands without downtime.
- **Enhanced Performance**: Distributes workloads to optimize resource utilization and response times.
- **Health Monitoring**: Continuously checks the health of servers and ensures that traffic is only sent to healthy instances.

Use Cases

- **Web Applications**: Ensures consistent user experience by distributing user requests across multiple application servers.
- **Microservices Architecture**: Balances requests among various microservices, improving performance and reliability.
- **API Management**: Helps manage traffic to backend services, ensuring efficient resource use.

Caching is a technique used to temporarily store data in a way that allows for faster access. It helps improve application performance, reduce latency, and lower the load on underlying data sources. Here's an overview of caching:

What is Caching?

Caching involves storing copies of frequently accessed data in a temporary storage layer, known as a cache. When a request for data is made, the application first checks the cache before querying the primary data source.

Types of Caches

1. **Memory Cache**:
    - Stores data in RAM for extremely fast access.

- Examples: Redis, Memcached.

2. **Disk Cache**:
    - Stores data on disk to persist beyond application restarts, but with slower access than memory.
    - Example: Local disk caching, browser caching.
3. **CDN Cache** (Content Delivery Network):
    - Caches static content (images, scripts) at various geographical locations to speed up delivery to users.
4. **Database Cache**:
    - Caches results of database queries to reduce load on the database and speed up responses.

Cache Strategies

1. **Cache-aside (Lazy Loading)**:
    - The application code manages the cache. It checks the cache first; if the data is not present, it loads it from the database and then stores it in the cache.
2. **Write-through**:
    - Every time data is written to the database, it is also written to the cache simultaneously.
3. **Write-back**:
    - Writes are made only to the cache first, and the database is updated later, improving write performance but introducing potential consistency issues.
4. **Time-based Expiration**:
    - Cached data is invalidated after a set period, ensuring that stale data isn't served.
5. **Eviction Policies**:
    - Determines which data to remove from the cache when it reaches capacity. Common policies include:

    - **LRU (Least Recently Used)**: Removes the least recently accessed items.
    - **FIFO (First In, First Out)**: Removes the oldest items.
    - **LFU (Least Frequently Used)**: Removes items that are least frequently accessed.

Benefits of Caching

- **Improved Performance**: Significantly reduces data retrieval times.
- **Reduced Latency**: Quick access to data decreases the time users wait for responses.
- **Decreased Load on Databases**: Less frequent database queries reduce load and improve scalability.
- **Cost Efficiency**: Less processing and bandwidth usage can lead to lower operational costs.

Use Cases

- **Web Applications**: Caching user sessions, HTML pages, and API responses to enhance performance.
- **E-commerce Platforms**: Storing product data and user preferences to speed up load times.
- **Data Processing Pipelines**: Caching intermediate results in ETL processes to optimize processing times.

Sharding is a database architecture pattern used to improve performance and scalability by distributing data across multiple servers or instances. Here's an overview of what sharding involves:

What is Sharding?

Sharding involves partitioning a database into smaller, more manageable pieces called "shards." Each shard is a separate database that holds a subset of the overall data. This technique allows for horizontal scaling, meaning you can add more servers to handle increased load rather than upgrading a single server.

How Sharding Works

1. **Data Partitioning**: Data is divided based on a sharding key, which could be a user ID, geographical region, or any other logical segment. Each shard holds a portion of the data based on this key.
2. **Routing Requests**: When an application needs to access data, it determines which shard to query based on the sharding key. This can be done using a routing layer or by including shard logic in the application.
3. **Independent Shards**: Each shard operates independently, allowing for parallel processing and improved performance.

Sharding Strategies

1. **Horizontal Sharding**:
    - Distributes rows of a database table across multiple shards. For example, user accounts might be split by user ID ranges.
2. **Vertical Sharding**:
    - Distributes different tables or columns across different shards. For example, user information might be on one shard while user transactions are on another.
3. **Directory-based Sharding**:
    - Uses a lookup table or directory to map each data item to its corresponding shard. This allows more flexibility but adds a layer of complexity.

Benefits of Sharding

- **Scalability**: Easily scale out by adding more servers as data grows.
- **Performance**: Distributes the load, leading to faster query responses and reduced latency.
- **Fault Isolation**: A failure in one shard doesn't affect others, enhancing system reliability.
- **Improved Resource Utilization**: Each shard can be hosted on different hardware configurations based on its needs.

Challenges of Sharding

- **Complexity**: Managing multiple shards increases the complexity of the application and the database architecture.

- **Data Consistency**: Ensuring consistent data across shards can be challenging, especially for transactions that span multiple shards.
- **Rebalancing**: If data grows unevenly across shards, rebalancing (redistributing data) may be required, which can be a complex operation.

Use Cases

- **Large-scale Applications**: Websites with millions of users, like social media platforms, often use sharding to manage user data.
- **E-commerce Platforms**: To handle large volumes of transactions and customer data efficiently.
- **Big Data Systems**: Applications that require processing large datasets, where sharding helps in distributed data processing.

Indexes are a crucial component of database management systems that enhance the speed of data retrieval operations. Here's a comprehensive overview of what indexes are, how they work, and their benefits:

What is an Index?

An index is a data structure that improves the speed of data retrieval on a database table at the cost of additional space and slower write operations. It works similarly to an index in a book, allowing the database to find data without scanning the entire table.

How Indexes Work

- **Structure**: Most indexes are implemented as B-trees or hash tables. They store key-value pairs where the key is the indexed column and the value points to the actual row in the table.
- **Data Lookup**: When a query is executed, the database can quickly locate the indexed values, leading to faster access times.
- **Maintenance**: Indexes must be updated whenever data is modified (inserted, updated, or deleted), which can affect performance during write operations.

Types of Indexes

1. **Single-column Index**: An index on a single column of a table. It improves performance for queries filtering or sorting on that column.
2. **Composite Index**: An index on multiple columns. It can be beneficial for queries that involve filtering or sorting by more than one column.

3. **Unique Index**: Ensures that the indexed values are unique across the table. This is often used to enforce primary key constraints.
4. **Full-text Index**: Optimizes searches on text columns, allowing for efficient searching of textual data.
5. **Spatial Index**: Used for indexing spatial data, like geographical locations, to optimize queries involving spatial relationships.
6. **Bitmap Index**: Uses bit arrays to represent distinct values in columns, efficient for columns with a limited number of unique values.

Benefits of Indexes

- **Faster Query Performance**: Significantly speeds up SELECT queries by reducing the number of data pages the database engine needs to scan.
- **Efficient Sorting**: Indexes can optimize ORDER BY clauses, improving the performance of sorted queries.
- **Improved JOIN Operations**: Facilitates faster lookups during joins between tables.

Drawbacks of Indexes

- **Increased Storage**: Indexes require additional disk space, which can be substantial for large tables.
- **Slower Writes**: Insertions, updates, and deletions may become slower because indexes need to be updated along with the data.
- **Maintenance Overhead**: Indexes require ongoing maintenance, and poorly chosen indexes can lead to performance degradation.

When to Use Indexes

- Use indexes on columns that are frequently queried or used in WHERE clauses.
- Index columns used for JOINs and ORDER BY clauses.
- Consider composite indexes when multiple columns are often queried together.

A proxy server acts as an intermediary between clients and other servers, facilitating various network requests. Here's an overview of proxy servers in system design:

What is a Proxy Server?

A proxy server is a server that sits between client devices and the internet, handling requests from clients seeking resources from other servers. It forwards requests and responses, often modifying them in the process.

Types of Proxy Servers

1. **Forward Proxy**:
    - **Description**: Acts on behalf of clients, routing requests to various servers.
    - **Use Cases**: Accessing restricted content, enhancing privacy, or bypassing geo-blocks.
2. **Reverse Proxy**:
    - **Description**: Acts on behalf of servers, receiving client requests and routing them to the appropriate server.
    - **Use Cases**: Load balancing, SSL termination, and caching.
3. **Transparent Proxy**:
    - **Description**: Intercepts and redirects requests without modifying them. Clients may not be aware of its presence.
    - **Use Cases**: Monitoring traffic or caching content without client configuration.
4. **Anonymous Proxy**:
    - **Description**: Hides the client's IP address from the target server.
    - **Use Cases**: Enhancing privacy and security during web browsing.
5. **High Anonymity Proxy**:
    - **Description**: Provides maximum privacy by not revealing that it is a proxy.
    - **Use Cases**: Sensitive browsing where anonymity is paramount.

Benefits of Using a Proxy Server

- **Improved Security**: Can mask client IP addresses, providing anonymity and reducing exposure to attacks.
- **Load Balancing**: Distributes traffic among multiple servers to prevent overload and improve performance.

- **Caching**: Stores frequently requested content, speeding up access for clients and reducing load on backend servers.
- **Content Filtering**: Can block access to certain websites or content based on policies, useful in corporate or educational environments.
- **SSL Termination**: Can handle SSL encryption, offloading this processing from application servers.

Challenges and Considerations

- **Single Point of Failure**: If the proxy server goes down, all clients relying on it may lose access.
- **Latency**: Introducing a proxy can add additional latency to requests, although caching can mitigate this.
- **Complexity**: Requires careful configuration and management to ensure it meets security and performance needs.
- **Security Risks**: Misconfigured proxies can expose sensitive data or create vulnerabilities.

Use Cases in System Design

- **Web Applications**: Using a reverse proxy to manage traffic, provide SSL termination, and cache responses.
- **Microservices Architecture**: Implementing a service mesh with proxy functionality to manage inter-service communication.
- **Corporate Networks**: Deploying forward proxies for secure internet access, content filtering, and monitoring employee web usage.

Message queues are vital in system design, facilitating communication between different components of an application by allowing messages to be sent asynchronously. Apache Kafka is a popular distributed messaging system known for its scalability and fault tolerance. Here's an overview of Kafka and its role in system design:

What is Kafka?

Kafka is a distributed event streaming platform that is designed to handle real-time data feeds. It allows producers to send messages to topics, which are then consumed by consumers. Kafka is highly scalable, fault-tolerant, and optimized for throughput.

Key Concepts in Kafka

1. **Producers**: Applications or services that publish messages to Kafka topics.
2. **Consumers**: Applications or services that subscribe to topics to read messages.
3. **Topics**: Categories or feeds to which messages are published. Each topic can have multiple partitions for parallel processing.
4. **Partitions**: A topic is divided into partitions, allowing Kafka to scale horizontally. Each partition is an ordered, immutable sequence of records.
5. **Brokers**: Kafka servers that store messages and serve clients. A Kafka cluster can consist of multiple brokers for redundancy and load distribution.
6. **Consumer Groups**: A group of consumers that work together to consume messages from a topic. Each message in a partition is delivered to one consumer in the group, enabling parallel processing.

Benefits of Using Kafka

- **High Throughput**: Kafka is designed for high volume data processing, making it suitable for real-time analytics and large-scale data ingestion.
- **Scalability**: Easily scale by adding more brokers and partitions to handle increased loads without downtime.
- **Fault Tolerance**: Data is replicated across multiple brokers, ensuring durability and availability even in case of server failures.
- **Decoupling of Services**: Producers and consumers can evolve independently, leading to a more flexible architecture.
- **Durability**: Messages can be configured to persist on disk for a specified retention period, allowing for reprocessing if needed.

Use Cases

- **Real-time Data Processing**: Stream processing applications that analyze data in real-time (e.g., fraud detection, analytics dashboards).
- **Log Aggregation**: Collecting logs from multiple services and making them available for monitoring and analysis.
- **Event Sourcing**: Storing state changes as a sequence of events, allowing for reconstructing application state.

- **Microservices Communication**: Enabling asynchronous communication between microservices, enhancing resilience and scalability.

Challenges

- **Complexity**: Setting up and managing a Kafka cluster can be complex, requiring careful configuration and monitoring.
- **Message Ordering**: While Kafka maintains order within a partition, achieving global ordering across partitions can be challenging.
- **Data Duplication**: Ensuring exactly-once delivery semantics can be complex and may require additional handling.

SQL and NoSQL databases serve different purposes and have distinct characteristics. Here's a comparison to help you understand their differences, advantages, and typical use cases.

SQL Databases

Characteristics:

- **Structured Data**: Use a fixed schema, meaning data must adhere to a defined structure.
- **ACID Compliance**: Provide strong consistency with ACID (Atomicity, Consistency, Isolation, Durability) properties, making them reliable for transactions.
- **Relational Model**: Organize data into tables with predefined relationships (foreign keys).

Examples:

- MySQL
- PostgreSQL
- Microsoft SQL Server
- Oracle Database

Advantages:

- **Complex Queries**: Support advanced querying capabilities with SQL, allowing for complex joins and aggregations.
- **Data Integrity**: Enforce data integrity and relationships through constraints.

- **Mature Ecosystem**: Established tools and support for management, reporting, and analytics.

Use Cases:

- Applications requiring complex transactions (e.g., banking, e-commerce).
- Systems where data integrity and relationships are critical (e.g., CRM systems).

NoSQL Databases

Characteristics:

- **Flexible Schema**: Allow for unstructured or semi-structured data with dynamic schemas.
- **Eventual Consistency**: Many NoSQL databases favor eventual consistency, which can lead to faster writes but may not guarantee immediate consistency.
- **Diverse Models**: Include various data models, such as document, key-value, column-family, and graph.

Examples:

- MongoDB (Document)
- Cassandra (Column-family)
- Redis (Key-value)
- Neo4j (Graph)

Advantages:

- **Scalability**: Designed for horizontal scaling, making it easier to distribute data across multiple servers.
- **High Performance**: Generally optimized for read/write operations, particularly in high-volume environments.
- **Varied Data Types**: Can handle a wide variety of data formats, making them suitable for big data applications and rapidly evolving data structures.

Use Cases:

- Real-time analytics and big data applications.

- Content management systems and applications with varied data types (e.g., social media).
- Internet of Things (IoT) applications requiring fast writes and flexible schemas.

Summary

- **SQL** databases are best suited for applications needing structured data, complex queries, and strong consistency.
- **NoSQL** databases excel in scenarios requiring flexibility, scalability, and the ability to handle varied data types.

Monolithic and microservices architectures are two different approaches to building and organizing applications. Here's a detailed comparison of both architectures:

Monolithic Architecture

Characteristics:

- **Single Codebase**: The entire application is built as a single unit, where all components are tightly integrated and deployed together.
- **Unified Deployment**: Any change, even minor, requires redeploying the entire application.
- **Shared Resources**: Components share the same database and resources, which can simplify communication between them.

Advantages:

- **Simplicity**: Easier to develop, test, and deploy, especially for smaller applications or startups.
- **Performance**: Communication between components happens in-memory, resulting in lower latency.
- **Ease of Management**: Fewer moving parts mean less operational overhead.

Disadvantages:

- **Scalability Challenges**: Scaling requires replicating the entire application rather than individual components.

- **Tight Coupling**: Changes in one part of the application can impact others, making maintenance difficult.
- **Deployment Risks**: A failure in one component can lead to the entire application being down.

Use Cases:

- Small to medium-sized applications.
- Projects with well-defined scopes that are unlikely to change drastically over time.

Microservices Architecture

Characteristics:

- **Decentralized Services**: The application is composed of small, independent services, each responsible for a specific business capability.
- **Independent Deployment**: Services can be developed, deployed, and scaled independently.
- **Diverse Technology Stack**: Different services can use different technologies or programming languages suited to their specific needs.

Advantages:

- **Scalability**: Individual services can be scaled independently based on demand.
- **Flexibility**: Teams can work on different services simultaneously, enabling faster development cycles.
- **Resilience**: A failure in one service doesn't necessarily bring down the entire application.

Disadvantages:

- **Complexity**: Increased operational overhead and complexity in managing multiple services and their intercommunications.
- **Network Latency**: Communication between services happens over the network, which can introduce latency.
- **Data Management**: Requires strategies for data consistency and management across distributed services.

Use Cases:

- Large, complex applications that need to evolve rapidly.
- Organizations that adopt DevOps practices and need to scale teams.

Summary

- **Monolithic Architecture**: Best for simpler applications with limited scope. It offers ease of development and management but can struggle with scalability and flexibility as the application grows.
- **Microservices Architecture**: Ideal for complex, large-scale applications requiring agility, scalability, and resilience. It allows teams to innovate and iterate quickly but introduces complexity in deployment and management.

A REST API (Representational State Transfer Application Programming Interface) is a widely used architectural style for building web services that allow different software applications to communicate over the internet. Here's a comprehensive overview:

What is REST?

REST is an architectural principle that uses standard HTTP methods and principles to facilitate communication between clients and servers. It is stateless and relies on the concept of resources, which can be identified and manipulated using URIs (Uniform Resource Identifiers).

Key Principles of REST

1. **Statelessness**: Each request from a client to the server must contain all the information needed to understand and process the request. The server does not store client context between requests.
2. **Client-Server Architecture**: The client and server are separate entities that interact with each other through requests and responses. This separation allows for independent development and scaling.
3. **Resource-Based**: Everything is considered a resource, which can be accessed and manipulated via URIs. Resources can be represented in various formats, such as JSON, XML, or HTML.
4. **Use of Standard HTTP Methods**:
    - **GET**: Retrieve a resource.

    - **POST**: Create a new resource.
    - **PUT**: Update an existing resource.
    - **DELETE**: Remove a resource.
5. **Stateless Communication**: Each request is independent, and the server does not retain any session information.
6. **Cacheability**: Responses must indicate whether they can be cached or not, improving performance by reducing the number of requests to the server.

Components of a REST API

1. **Endpoint**: A specific URL where a resource can be accessed (e.g., https://api.example.com/users).
2. **HTTP Methods**: Actions that can be performed on resources (GET, POST, PUT, DELETE).
3. **Request and Response**: Clients send requests to the API endpoints, and the server responds with data, often in JSON or XML format.
4. **Status Codes**: HTTP status codes communicate the result of a request (e.g., 200 for success, 404 for not found, 500 for server error).

Benefits of REST APIs

- **Simplicity**: REST APIs use standard HTTP protocols, making them easy to understand and implement.
- **Scalability**: Statelessness allows for easier scaling, as servers do not need to store session information.
- **Flexibility**: REST APIs can handle multiple formats for data exchange (e.g., JSON, XML), allowing clients to choose their preferred format.
- **Interoperability**: RESTful services can be consumed by any client that understands HTTP, making it easier to integrate with various platforms.

Use Cases

- **Web Services**: Connecting web applications to backend services or databases.
- **Mobile Applications**: Allowing mobile apps to communicate with server-side resources.

- **Microservices**: Enabling communication between microservices in a distributed architecture.

An LRU (Least Recently Used) cache is a type of data structure that manages a fixed-size cache by removing the least recently accessed items when it becomes full. This caching strategy is useful for optimizing the performance of applications by keeping frequently accessed data readily available.

Key Concepts of LRU Cache

1. **Cache Size**: The maximum number of items the cache can hold. Once this limit is reached, the cache must evict the least recently used item to make space for new data.
2. **Data Structure**:
    - **Hash Map**: Used to store key-value pairs for O(1) average time complexity for lookups.
    - **Doubly Linked List**: Maintains the order of access, allowing for quick removal of the least recently used items. The most recently used items are at one end, and the least recently used items are at the other.
3. **Operations**:
    - **Get(key)**: Retrieve the value associated with the key if it exists in the cache. If found, the item is marked as recently used (moved to the front of the linked list).
    - **Put(key, value)**: Insert or update the value for the key. If the key already exists, update it and mark it as recently used. If it doesn't exist and the cache is full, evict the least recently used item before adding the new item.

Advantages of LRU Cache

- **Efficiency**: Provides O(1) time complexity for both get and put operations, making it very efficient.
- **Optimal Eviction Strategy**: By evicting the least recently used items, it keeps frequently accessed data in the cache, which can significantly enhance performance in read-heavy applications.

- **Implementation Example (Python)**
- Here's a simple implementation of an LRU Cache in Python:

```
class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.prev = None
        self.next = None

class LRUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.cache = {}  # Map of key to node
        self.head = Node(0, 0)  # Dummy head
        self.tail = Node(0, 0)  # Dummy tail
        self.head.next = self.tail
        self.tail.prev = self.head

    def _remove(self, node: Node):
        prev, nxt = node.prev, node.next
        prev.next, nxt.prev = nxt, prev

    def _add_to_front(self, node: Node):
        node.prev = self.head
        node.next = self.head.next
        self.head.next.prev = node
```

```
        self.head.next = node

    def get(self, key: int) -> int:
        if key in self.cache:
            node = self.cache[key]
            self._remove(node)
            self._add_to_front(node)
            return node.value
        return -1

    def put(self, key: int, value: int) -> None:
        if key in self.cache:
            self._remove(self.cache[key])
        elif len(self.cache) >= self.capacity:
            lru_node = self.tail.prev
            self._remove(lru_node)
            del self.cache[lru_node.key]
        new_node = Node(key, value)
        self._add_to_front(new_node)
        self.cache[key] = new_node
```

Apache Hadoop

Apache Hadoop is an open-source framework for distributed storage and processing of large datasets using a cluster of commodity hardware. It is designed to scale from a single server to thousands of machines, each offering local computation and storage.

Key Components:

1. **Hadoop Common**: The common utilities and libraries that support the other Hadoop modules.
2. **Hadoop Distributed File System (HDFS)**: A distributed file system that stores data across multiple machines.
3. **Hadoop YARN (Yet Another Resource Negotiator)**: A resource management layer that schedules and manages computing resources in the cluster.
4. **Hadoop MapReduce**: A programming model for processing large datasets in parallel across the cluster.

HDFS (Hadoop Distributed File System)

HDFS is the storage component of Hadoop. It is designed to handle large files and provides high-throughput access to application data.

Characteristics:

- **Distributed Storage**: Splits large files into smaller blocks (default 128 MB or 256 MB) and distributes them across the cluster.
- **Fault Tolerance**: Replicates each block (default 3 copies) across different nodes to ensure data availability even in case of hardware failure.
- **Scalability**: Can scale horizontally by adding more nodes to the cluster.
- **Streaming Data Access**: Optimized for high-throughput access to large datasets rather than low-latency access.

HBase

Apache HBase is a distributed, scalable, NoSQL database built on top of HDFS. It is designed to provide real-time read and write access to large datasets.

Characteristics:

- **Column-oriented**: Stores data in columns rather than rows, which makes it efficient for sparse data.
- **Scalable**: Can handle large amounts of data by scaling horizontally.
- **Random Access**: Provides quick random access to data through a key-based lookup.
- **Strong Consistency**: Supports strong consistency for read and write operations.

Use Cases:

- Real-time analytics and applications that require random access to large datasets (e.g., social media data, user activity logs).

ZooKeeper

Apache ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and coordinating distributed applications.

Characteristics:

- **Coordination Service**: Helps manage distributed systems by keeping track of configuration data, group memberships, and synchronization.
- **High Availability**: Ensures that services remain available even if some nodes fail.
- **Simple API**: Provides a straightforward API for clients to interact with.

Use Cases:

- Managing distributed configurations for Hadoop and HBase.
- Coordinating distributed applications and services (e.g., leader election, group management).

Apache Solr

Apache Solr is an open-source search platform built on Apache Lucene. It is designed for searching and indexing large volumes of text data and is widely used for building search applications.

Key Features:

- **Full-Text Search**: Supports powerful full-text search capabilities, including advanced queries and faceting.
- **Scalability**: Designed to handle large volumes of data and can scale horizontally.
- **Faceting**: Allows for aggregating and categorizing search results, making it easier for users to navigate.
- **Real-Time Indexing**: Supports real-time indexing, allowing for immediate searchability of newly added documents.
- **Distributed Search**: Can perform distributed searching across multiple nodes.

Use Cases:

- Enterprise search applications.
- E-commerce product search.
- Data analytics and reporting.

Apache Cassandra

Apache Cassandra is a distributed NoSQL database designed for handling large amounts of structured data across many commodity servers. It provides high availability with no single point of failure.

Key Features:

- **Scalability**: Supports horizontal scaling, allowing you to add more nodes without downtime.
- **Fault Tolerance**: Data is replicated across multiple nodes, ensuring availability even if some nodes fail.
- **Flexible Data Model**: Uses a wide-column store model, allowing for dynamic and flexible schema design.
- **Tunable Consistency**: Provides options for consistency levels, allowing you to balance between consistency and availability.

Use Cases:

- Applications requiring high write and read throughput (e.g., social media, IoT data).
- Event logging and real-time analytics.

HTTP vs. HTTPS

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

Key Differences:

1. **Security**:
    - **HTTP**: Data is sent in plaintext, making it vulnerable to interception and attacks.

    - **HTTPS**: Encrypts data using SSL/TLS, ensuring secure transmission between client and server.
2. **Port**:
    - **HTTP**: Typically uses port 80.
    - **HTTPS**: Typically uses port 443.
3. **SEO Impact**:
    - HTTPS is favored by search engines, potentially improving search rankings.
4. **Trust**:
    - Browsers indicate secure connections with HTTPS by showing a padlock icon, enhancing user trust.

CAP Theorem

The CAP theorem, formulated by Eric Brewer, states that in a distributed data store, it is impossible to simultaneously guarantee all three of the following properties:

1. **Consistency (C)**: Every read receives the most recent write for a given piece of data. All nodes see the same data at the same time.
2. **Availability (A)**: Every request (read or write) receives a response, regardless of whether the data is the most recent.
3. **Partition Tolerance (P)**: The system continues to operate despite network partitions or failures between nodes.

Implications:

- You can only choose two out of the three properties to prioritize, which leads to different types of databases:
    - **CP (Consistency and Partition Tolerance)**: Systems that prioritize consistency over availability (e.g., HBase).
    - **AP (Availability and Partition Tolerance)**: Systems that prioritize availability over consistency (e.g., Cassandra).
    - **CA (Consistency and Availability)**: Not achievable in a distributed system due to the possibility of network partitions.