**Microservices**

**Que: What is Microservices ?**

**Ans:** Microservices, also known as Microservices Architecture, is basically an SDLC approach in which large applications are built as a collection of small functional modules.

It is a software architectural style where an application is structured as a collection of small, loosely coupled, independently deployable services. Each microservice focuses on a specific business domain or functionality and operates as a separate process or service within a larger system.

Microservices architecture contrasts with monolithic architecture, where an application is built as a single, tightly integrated units.

* Maintainable and testable.
* Loosely Coupled.
* Independently deployable.
* Designed or organized around business capabilities.
* Managed by a small team.

**Que: Mention the main features of Microservices.**

**Que: Benefits of using Microservices.**

**Ans**: Key characteristics of microservices include:

**1. Decomposition:**

Microservices decompose a complex application into smaller, manageable services that can be developed, deployed, and scaled independently. Each microservice typically handles a specific business capability, such as user authentication, payment processing, or data management.

**2. Loose Coupling:**

Microservices are loosely coupled, meaning they have minimal dependencies on other services. They communicate through well-defined APIs (Application Programming Interfaces), often using lightweight protocols like HTTP or messaging systems.

**3. Independence:**

Microservices can be developed, tested, deployed, and updated independently of each other.

**4. Scalability:**

Microservices architecture enables horizontal scaling by deploying multiple instances of individual services based on demand.

**5. Technology Diversity:**

Microservices allow for technology diversity, meaning each service can use different programming languages, frameworks, databases, and tools based on its specific requirements. This flexibility enables teams to choose the best technologies for each microservice.

**6. Resilience:**

Microservices promote resilience by isolating failures to specific services. If one microservice experiences a failure or performance issue, it does not necessarily impact the entire system, and other services can continue to function independently.

**7. DevOps and Automation:**

Microservices align well with DevOps practices, automation, and continuous delivery. Teams can use automated pipelines for testing, deployment, monitoring, and scaling of microservices, facilitating rapid and reliable delivery of software updates.

**8. Containerization and Orchestration:**

Microservices are often deployed using containerization technologies like Docker, and orchestrated using platforms like Kubernetes. Containerization provides consistency in development and deployment environments, while orchestration simplifies management, scaling, and fault tolerance of microservices in production.

**Que: What are the main components of Microservices?**

**Ans:** The main components of a microservices architecture include:

**1. Microservices:**

Microservices are the core building blocks of the architecture.

**2. API Gateway:**

An API Gateway is a centralized entry point that manages and routes external client requests to the appropriate microservices. It often handles tasks like authentication, authorization, rate limiting, and request routing. The API Gateway can also aggregate responses from multiple microservices into a single response for the client.

**3. Service Discovery:**

Service Discovery is a mechanism that enables microservices to dynamically discover and communicate with each other without hardcoded dependencies. It allows services to register themselves with a service registry and look up the locations of other services at runtime.

**4. Service Registry:**

A Service Registry is a database or registry that maintains a list of available microservices and their network locations (e.g., IP addresses, ports). It helps in dynamic service discovery and allows services to be resilient to changes in their deployment environment.

**5. Load Balancer:**

Load Balancers distribute incoming client requests across multiple instances of microservices to achieve scalability, improve performance, and ensure high availability. Load balancing can be performed at various levels, such as network-level load balancing or application-level load balancing.

**6. Configuration Management:**

Configuration Management involves managing configuration settings and parameters for microservices in a centralized and scalable manner. It includes tools and practices for storing, versioning, and deploying configuration settings to microservices.

**7. Distributed Logging and Monitoring:**

Distributed Logging and Monitoring tools help in monitoring the health, performance, and behavior of microservices across a distributed system. They collect metrics, logs, and traces from microservices and provide insights into system performance, resource utilization, and error tracking.

**8. Containerization and Orchestration:**

Containerization technologies like Docker allow microservices to be packaged into lightweight, portable containers that encapsulate their dependencies and runtime environment. Orchestration platforms like Kubernetes provide tools for deploying, managing, scaling, and monitoring containerized microservices in production.

**9. Event Bus or Message Broker:**

An Event Bus or Message Broker facilitates asynchronous communication and event-driven architectures between microservices. It enables services to publish and subscribe to events, decoupling components and promoting scalability, loose coupling, and flexibility.

**10. Database per Service:**

In a microservices architecture, each microservice typically has its database (Database per Service pattern). This pattern promotes service autonomy, isolation, and scalability but requires careful consideration of data consistency, synchronization, and transaction management across services.

**Que: What are drawbacks of Microservices?**

**Ans**: Here are some common drawbacks of microservices:

**1. Complexity of Distributed Systems:**

Microservices introduce complexity in managing distributed systems. As the number of microservices increases, so does the complexity of managing communication between services, ensuring data consistency, handling inter-service dependencies, and monitoring the overall system.

**2. Service Communication Overhead:**

Microservices communicate with each other over the network, which introduces latency and potential network failures. Service communication overhead includes serialization/deserialization of data, network latency, and managing communication protocols.

**3. Data Consistency and Transaction Management:**

Ensuring data consistency and managing transactions across multiple microservices can be challenging. Distributed transactions are complex.

**4. Testing and Quality Assurance:**

Testing microservices involves testing individual services in isolation as well as testing interactions between services.

**5. Dependency Management:**

Microservices have dependencies on external services, libraries, and APIs. Managing dependencies, versioning, backward compatibility, and API changes across multiple services can be challenging.

**6. Cost of Infrastructure and Scalability:**

Microservices architecture may require additional infrastructure resources for managing containerization, orchestration, service discovery, and monitoring tools.

**7. Organizational Changes, Overall Cultural Shifts and Team Collaboration:**

Adopting microservices often requires organizational changes, such as forming cross-functional teams, adopting DevOps practices, and promoting a culture of autonomy and accountability.

**Que: Mention some common tools used in Microservice Architecture.**

**Ans**: Microservices architecture involves a range of tools and technologies to support development, deployment, monitoring, and management of microservices-based systems. Here are some common tools used in various stages of the microservices lifecycle:

**1. Service Development and Integration:**

**Spring Boot:** A popular Java-based framework for building microservices. It simplifies development by providing pre-configured components for dependency injection, web services, and data access.

**Express.js**: A lightweight Node.js framework for building web applications and APIs, commonly used in microservices development with JavaScript.

**gRPC:** A high-performance RPC (Remote Procedure Call) framework that supports communication between microservices using protocol buffers and HTTP/2.

**Swagger/OpenAPI:** Tools for designing, documenting, and testing APIs, promoting API-first development in microservices architectures.

**2. Containerization and Orchestration:**

**Docker:** A containerization platform that allows packaging microservices into lightweight, portable containers along with their dependencies.

**Kubernetes**: An open-source container orchestration platform for automating deployment, scaling, and management of containerized applications, including microservices.

**Docker Compose:** A tool for defining and running multi-container Docker applications, useful for local development and testing of microservices.

**3.API Gateway and Service Mesh:**

**NGINX:** A high-performance web server and reverse proxy that can also act as an API gateway for routing and managing API requests.

**Kong:** An open-source API gateway that provides features like authentication, rate limiting, logging, and monitoring for microservices.

**Envoy Proxy:** A cloud-native proxy designed for modern microservices architectures, often used as part of a service mesh for handling service-to-service communication, load balancing, and observability.

**4. Service Discovery and Load Balancing:**

**Consul**: A service discovery and configuration tool that helps microservices locate and communicate with each other dynamically.

**Zookeeper**: A centralized service for distributed synchronization, coordination, and discovery, commonly used in microservices environments.

**Netflix Eureka**: A service registry and discovery server from Netflix, designed for cloud-based microservices architectures.

**5. Monitoring and Logging:**

**Prometheus:** An open-source monitoring and alerting toolkit for collecting metrics, monitoring microservices performance, and generating alerts.

**Grafana:** A visualization and analytics platform that works with Prometheus and other data sources to create customizable dashboards for monitoring microservices.

**ELK Stack (Elasticsearch, Logstash, Kibana**): A combination of tools for centralized logging, log processing, and log visualization, commonly used for managing logs in microservices environments.

**6. Continuous Integration and Deployment (CI/CD):**

**Jenkins:** An open-source automation server for implementing continuous integration and continuous delivery pipelines for microservices deployment.

**GitLab CI/CD:** A built-in CI/CD tool in GitLab for automating software delivery pipelines, including building, testing, and deploying microservices.

**CircleCI:** A cloud-based CI/CD platform for automating software development workflows, including integration with Docker and Kubernetes for microservices deployment.

**7. Configuration Management:**

**HashiCorp Consul**: In addition to service discovery, Consul also provides features for centralized configuration management and dynamic updates for microservices.

**Spring Cloud Config:** A configuration management tool for managing application configurations across multiple microservices in a distributed environment.

**8. Security and Identity Management:**

**Keycloak:** An open-source identity and access management solution for securing microservices with features like authentication, authorization, and single sign-on (SSO).

**OAuth 2.0/OpenID Connect:** Protocols and frameworks for securing APIs and microservices by implementing token-based authentication and authorization mechanisms.

**Que: Explain the working of Microservice Architecture?**

**Ans:** Each microservice is designed to perform a specific business function and communicates with other microservices through well-defined APIs. The working of microservices architecture involves several key aspects:

**1. Client Sends Request:**

The flow begins when a client (such as a web browser or mobile app) sends a request to the microservices-based application. This request could be an HTTP request for a specific functionality, such as retrieving user information or placing an order.

**2. API Gateway Receives Request:**

The request first hits the API Gateway, which serves as the entry point for incoming client requests. The API Gateway may handle tasks like authentication, rate limiting, request validation, and routing the request to the appropriate microservice.

* **Service Discovery:**

If the API Gateway needs to route the request to a specific microservice, it may use service discovery mechanisms to locate the appropriate instance of the microservice. Service discovery helps in dynamically finding the network location (IP address, port) of the microservice.

* **Load Balancing:**

In a clustered or scaled-out environment, a load balancer may distribute the incoming requests across multiple instances of the same microservice to achieve load balancing and improve performance.

**3. Microservice Handles Request:**

The request is then forwarded to the respective microservice responsible for handling that particular functionality or business capability. Each microservice is designed to perform a specific task or operation.

**Inter-Service Communication:**

If the operation requires data or functionality from other microservices, the handling microservice communicates with those services using synchronous (e.g., RESTful APIs) or asynchronous (e.g., message queues) communication patterns.

**4. Data Retrieval and Processing:**

The microservice retrieves data from databases or external APIs, performs business logic, and processes the request. This may involve querying databases, performing calculations, validating input, and applying business rules.

**5. Response Generation:**

Once the processing is complete, the microservice generates a response. The response includes the necessary data or information requested by the client, formatted according to the API contract or specification.

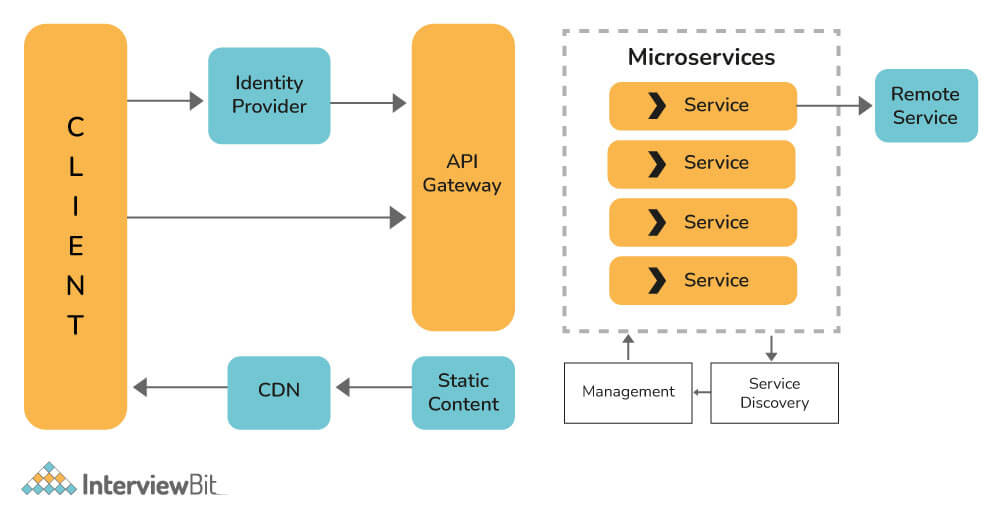
**6. API Gateway Returns Response:**

If necessary, the API Gateway may receive the response from the microservice and perform additional processing, such as aggregating data from multiple microservices, transforming the response format, or applying security policies.

**7. Client Receives Response:**

Finally, the API Gateway sends the response back to the client that initiated the request. The client processes the response and may display the information to the user or perform further actions based on the received data.

***Client -> API Gateway -> Service Discovery -> Load Balancer -> Microservice -> Inter-Service Communication -> Data Retrieval and Processing -> Response Generation -> API Gateway -> Client***



**Que: What is difference between Monolithic, SOA and Microservices?**

**Ans:** Monolithic architecture, Service-Oriented Architecture (SOA), and Microservices architecture are three distinct software design approaches, each with its characteristics and differences:

**Monolithic Architecture:**

* In a monolithic architecture, the entire application is built as a single, tightly integrated unit.
* All components (UI, business logic, data access, etc.) are tightly coupled and packaged together.
* Scaling is done by replicating the entire application, which can lead to resource inefficiency.
* Deployment and updates require deploying the entire application, causing downtime and risk during updates.
* Development teams often work on the same codebase, leading to coordination challenges and slower development cycles.
* Testing and debugging can be complex due to the interconnected nature of components.

**Service-Oriented Architecture (SOA):**

* SOA is an architectural approach where an application is divided into loosely coupled services that communicate via standard protocols.
* Services in SOA are typically larger in granularity compared to microservices and can encapsulate multiple functionalities.
* Communication between services often occurs through a centralized Enterprise Service Bus (ESB) or messaging systems.
* SOA promotes reusability, flexibility, and interoperability by exposing services as reusable components that can be shared across applications.
* However, SOA can still suffer from some of the downsides of monolithic architectures, such as limited scalability and challenges in managing dependencies.

**Microservices Architecture:**

* Microservices architecture breaks down an application into small, independent services that focus on specific business capabilities.
* Each microservice is self-contained, with its database (in some cases), business logic, and API.
* Microservices communicate via lightweight protocols like HTTP/REST, gRPC, or message queues.
* Services in a microservices architecture are highly decoupled, allowing teams to develop, deploy, and scale them independently.
* Scaling is done at the service level, enabling more efficient resource utilization and better responsiveness to changes in demand.
* Microservices promote agility, scalability, resilience, and technology diversity but require careful design, orchestration, and management of service interactions.

**Que: What do you meant by Cohesion and Coupling?**

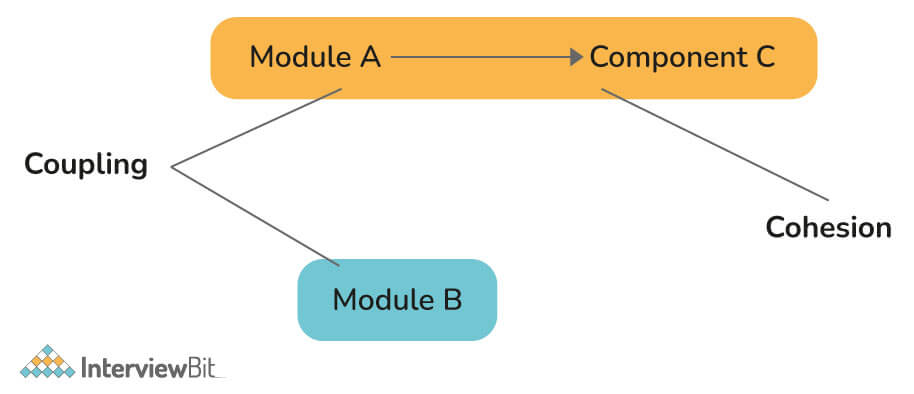
**Ans**: In software engineering, cohesion and coupling are two fundamental concepts that describe the design quality and relationships between components/modules in a system. Let's understand each term:

**Cohesion**:

* Cohesion refers to the degree of relatedness and focus within a single module or component. It measures how well the elements within a module are related to each other and how closely they work together to achieve a common purpose or functionality.

**Coupling**:

* Coupling refers to the degree of interdependence or connectivity between modules or components within a system. It measures how closely modules are connected or reliant on each other.
* It is defined as a relationship between software modules A and B, and how much one module depends or interacts with another one.
* Couplings fall into three major categories. Modules can be highly coupled (highly dependent), loosely coupled, and uncoupled from each other. The best kind of coupling is loose coupling, which is achieved through interfaces.

****

**Que: Explain Domain Driven Design?**

**Ans:** Domain-Driven Design (DDD) is an approach to software development that focuses on understanding and modeling the core domain of a business or problem space. It aims to align the software design closely with the domain's concepts, rules, and logic, leading to more effective and maintainable systems. DDD emphasizes collaboration between domain experts, developers, and stakeholders to build software that accurately reflects the complexities of the domain.

**Que: What do you meant by Bounded Context?**

**Ans:** A Bounded Context is a central pattern in DDD (Domain-Driven Design), which deals with collaboration across large models and teams. DDD breaks large models down into multiple contexts to make them more manageable. Additionally, it explains their relationship explicitly. The concept promotes an object-oriented approach to developing services bound to a data model and is also responsible for ensuring the integrity and mutability of said data model.

**Que: Explain how independent microservices communicate with each other.**

**Ans:** Independent microservices communicate with each other through well-defined communication protocols and patterns. Here are some common methods of communication between microservices:

**1. HTTP/RESTful APIs:**

* One of the most common ways for microservices to communicate is through HTTP-based RESTful APIs (Representational State Transfer).
* Each microservice exposes a set of RESTful endpoints that other microservices can call to request or send data.
* RESTful APIs use standard HTTP methods like GET, POST, PUT, DELETE, etc., to perform CRUD (Create, Read, Update, Delete) operations on resources.

**2. Messaging Queues/Systems:**

* Microservices can communicate asynchronously through messaging queues or message brokers such as Apache Kafka, RabbitMQ, or Amazon SQS (Simple Queue Service).
* In this pattern, a microservice publishes messages to a queue, and other microservices consume messages from the queue as needed.
* Messaging systems support reliable communication, decouple services, and enable event-driven architectures.

**3. gRPC (Remote Procedure Call):**

* gRPC is a high-performance RPC (Remote Procedure Call) framework developed by Google.
* It allows microservices to define service contracts using Protocol Buffers (protobufs) and communicate over HTTP/2, providing efficient and low-latency communication.
* gRPC supports synchronous communication with strong typing and automatic code generation for client-server communication.

**4. GraphQL:**

* GraphQL is a query language for APIs that allows clients to request specific data from a server.
* Microservices can expose GraphQL endpoints, and clients can send GraphQL queries to retrieve only the data they need, reducing over-fetching and under-fetching of data.
* GraphQL provides a flexible and efficient way to fetch data from multiple microservices in a single request.

**5. Event-Driven Architecture (EDA):**

* In an event-driven architecture, microservices communicate through events and event streams.
* A microservice can publish events (e.g., user-created, order-placed) to an event bus or broker, and other microservices can subscribe to these events and react accordingly.
* EDA promotes loose coupling, scalability, and real-time processing of events.

**6. Service Mesh:**

* Service mesh technologies like Istio, Linkerd, and Envoy provide a way to manage and control communication between microservices.
* Service meshes offer features such as service discovery, load balancing, circuit breaking, retries, and observability, enhancing communication reliability and resilience.

**Que: What do you mean by client certificates?**

**Ans:** Client certificates, also known as mutual TLS (Transport Layer Security) certificates or two-way SSL certificates, are a type of digital certificate used in secure communication between clients (such as web browsers, applications, or devices) and servers over HTTPS or other secure protocols. Unlike traditional SSL/TLS certificates that authenticate the server to the client, client certificates authenticate the client to the server, establishing a two-way trust relationship.

**Authentication of Clients:**

* Client certificates are used for authenticating clients (users or devices) to servers during SSL/TLS handshake. This authentication ensures that only authorized clients can access protected resources or services on the server.

**Components of Client Certificates:**

* Public Key: Like server certificates, client certificates contain a public key that corresponds to a private key kept securely by the client.
* Client's Identity Information: Client certificates also contain information about the client's identity, such as their name, organization, and a unique identifier (usually included in the Subject field of the certificate).

**SSL/TLS Handshake Process with Client Certificates:**

* When a client with a client certificate initiates a connection to a server:
* The server requests a client certificate during the SSL/TLS handshake.
* The client presents its client certificate, including its public key and identity information.
* The server verifies the client certificate by validating its digital signature using the corresponding CA (Certificate Authority) public key.
* If the client certificate is valid and trusted, the server allows the client to access protected resources or services.

**Que: What do you mean by Distributed Transaction?**

**Ans**: A distributed transaction refers to a transaction that involves multiple resources or systems distributed across different nodes or locations in a network. In other words, it is a transaction that spans multiple databases, services, or applications, where each participant may be located on separate machines or even in different geographical locations.

**System Design**

**Que: What is System design?**

**Ans:** System design is the process of designing the architecture, components, modules, interfaces, and data structures of a software system or application to meet specified requirements, achieve desired functionalities, and address various quality attributes such as performance, scalability, reliability, maintainability, and security. It involves making high-level decisions about how the system will be structured, how its components will interact, and how it will handle different use cases and scenarios.

Here are key aspects and concepts related to system design:

**1. Requirements Analysis:**

* System design starts with understanding and analyzing the requirements of the system, including functional requirements (what the system should do) and non-functional requirements (qualities the system should have).
* Requirements gathering involves discussions with stakeholders, domain experts, and end-users to identify needs, use cases, constraints, and expectations.

**2. Architecture Design:**

* Architecture design involves defining the overall structure of the system, including its components, layers, and interactions.
* Common architectural styles and patterns used in system design include client-server architecture, microservices architecture, monolithic architecture, layered architecture, event-driven architecture, and service-oriented architecture (SOA).

**3. Component Design:**

* Component design focuses on designing individual components or modules within the system, such as user interface components, business logic components, data access components, and external integrations.
* It includes defining interfaces, APIs, data models, algorithms, and internal workings of each component.

**4. Database Design:**

* Database design involves designing the structure, schema, relationships, and queries of the system's database(s).
* It includes defining tables, indexes, constraints, normalization, denormalization, data integrity rules, and access control policies.

**5. Interface Design:**

* Interface design deals with designing user interfaces (UIs), application programming interfaces (APIs), and communication interfaces between system components.
* It focuses on usability, accessibility, responsiveness, data presentation, user interaction flows, and API contract design.

**6. Scalability and Performance Design:**

* Scalability and performance design address the system's ability to handle increasing loads, concurrent users, and data volume while maintaining acceptable performance levels.
* Techniques such as load balancing, caching, sharding, asynchronous processing, and distributed architectures are used to improve scalability and performance.

**7. Reliability and Fault Tolerance:**

* Reliability and fault tolerance design aims to ensure that the system operates correctly under normal conditions and gracefully handles failures, errors, and unexpected events.
* Strategies include redundancy, failover mechanisms, error handling, logging, monitoring, and recovery procedures.

**8. Security Design:**

* Security design focuses on designing security measures and controls to protect the system against threats, vulnerabilities, unauthorized access, data breaches, and malicious attacks.
* It includes authentication, authorization, encryption, secure communication protocols, access control, and compliance with security standards and best practices.

**9. Maintainability and Extensibility:**

* Maintainability and extensibility design addresses the ease of maintaining, updating, and extending the system over time.
* Design principles such as modularity, encapsulation, separation of concerns, code reusability, versioning, and documentation contribute to maintainable and extensible systems.

**10. Testing and Quality Assurance:**

* System design also involves designing testing strategies, test cases, and quality assurance processes to verify and validate the system's functionality, performance, security, and reliability.
* Testing includes unit testing, integration testing, system testing, performance testing, security testing, and user acceptance testing.

**Que: What is CAP theorem?**

**Ans:** The CAP theorem, also known as Brewer's theorem, is a fundamental concept in distributed computing that describes the trade-offs between three desirable properties in a distributed system: Consistency, Availability, and Partition Tolerance. The CAP theorem states that in a distributed system, it is impossible to simultaneously guarantee all three properties under certain conditions.

Here's a breakdown of the CAP theorem and its implications:

**Consistency (C):**

* Consistency refers to all nodes in a distributed system having the same data at the same time. In other words, whenever a piece of data is updated or written to the system, all subsequent reads should return the updated data or a consistent view of the data.
* Strong consistency ensures that there are no inconsistencies or conflicts between replicas or nodes, providing a linearizable and coherent view of the data across the system.

**Availability (A):**

* Availability refers to the ability of a distributed system to respond to read and write requests from clients at any given time, even in the presence of failures or network issues.
* Highly available systems remain operational and accessible to users despite individual node failures or network partitions, ensuring uninterrupted service and responsiveness.

**Partition Tolerance (P):**

* Partition tolerance deals with the system's ability to function and maintain consistency even when network partitions occur, causing communication failures between nodes or subsets of nodes.
* Network partitions can lead to nodes being unable to communicate with each other, creating disjoint sets of nodes that cannot exchange data directly.

**Que: How is Horizontal scaling different from Vertical scaling?**

**Ans:** Horizontal scaling and vertical scaling are two approaches used to increase the capacity and performance of a system or application, but they differ in their strategies and implementations.

**Horizontal Scaling:**

* Horizontal scaling, also known as scaling out, involves adding more machines or instances to a system to distribute the workload across multiple resources.
* In horizontal scaling, each new instance runs a copy of the application or service and shares the overall workload with existing instances.
* Horizontal scaling typically requires load balancing mechanisms to evenly distribute incoming requests or tasks among the instances.

**Vertical Scaling:**

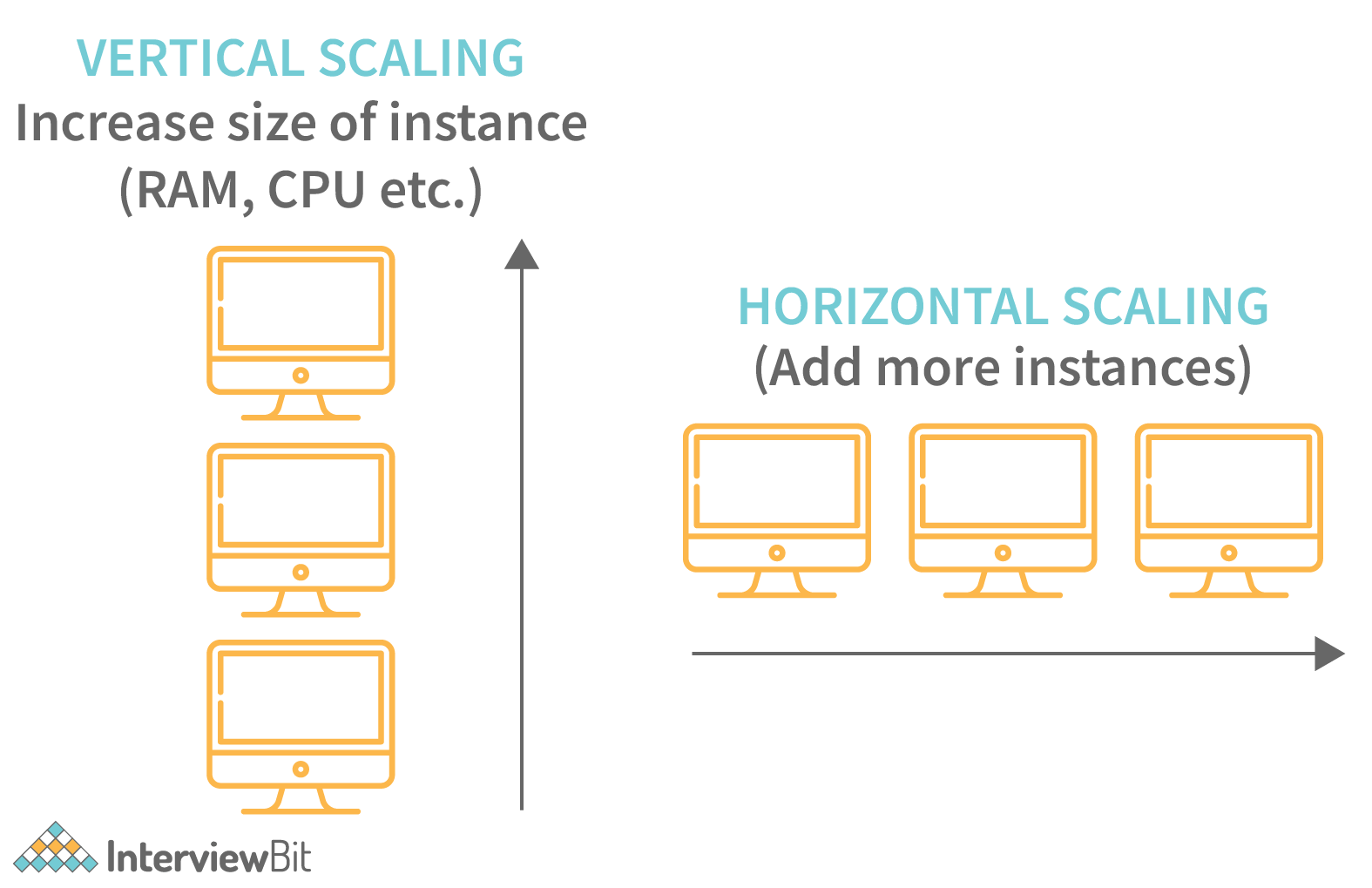
* Vertical scaling, also known as scaling up, involves increasing the resources (such as CPU, memory, storage) of existing machines or instances to handle higher workloads.
* In vertical scaling, a single instance is upgraded with more powerful hardware, such as adding more CPU cores, increasing RAM, or expanding storage capacity.
* Vertical scaling aims to improve the performance and capacity of a single instance by increasing its processing power and resources.

**Horizontal Scaling (Scaling Out):**

* Imagine you're baking cookies for a party, and you need to bake a lot of cookies quickly. Instead of using one oven, you decide to use multiple ovens.
* Each oven can bake a batch of cookies independently, so you can bake more cookies in the same amount of time by using multiple ovens simultaneously.
* In this scenario, adding more ovens (instances) is like horizontal scaling. It's about increasing the number of resources (ovens) to handle a larger workload (baking more cookies).

**Vertical Scaling (Scaling Up):**

* Now, let's say you're using a single oven to bake cookies, but you realize that it's taking too long to bake all the cookies because the oven is too small.
* To solve this, you decide to replace the small oven with a bigger and more powerful oven that can bake more cookies at once.
* In this case, upgrading to a bigger oven (increasing the resources of the existing oven) is like vertical scaling. It's about making the existing resource (oven) more capable to handle a larger workload (baking more cookies at once).



**Que: What do you understand by Latency, throughput, and availability of a system?**

**Ans:**

**Latency:**

* Latency refers to the time it takes for a system to respond to a request or perform a task. It's the delay between initiating an action and receiving a response.
* Imagine you're ordering food online. Latency is like the time it takes for the restaurant to receive your order, prepare the food, and deliver it to your doorstep. The shorter the latency, the faster you get your food.

**Throughput:**

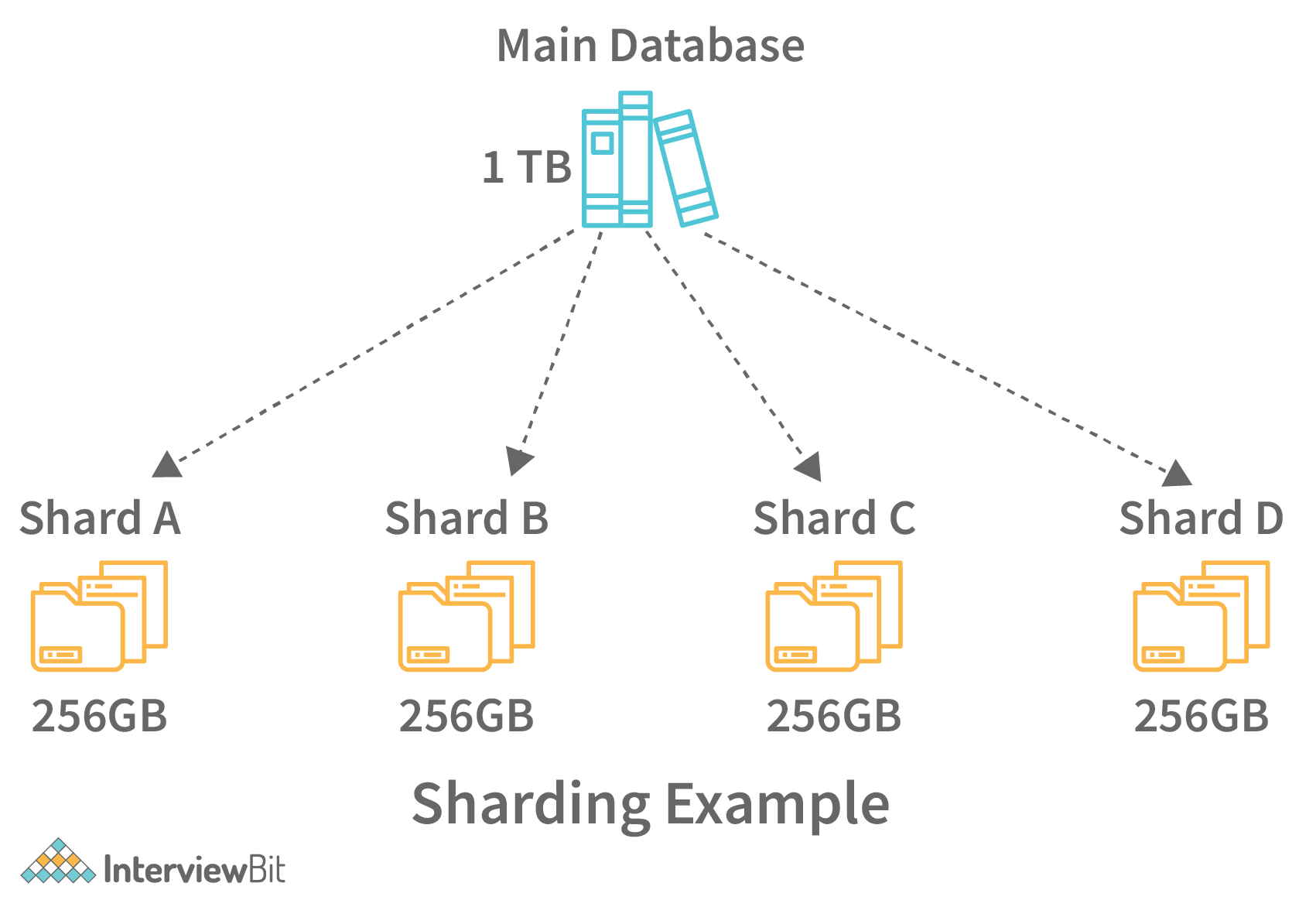
* Throughput measures the rate at which a system can process or handle a certain number of requests or tasks within a given time period.
* This is the amount of data successfully transmitted through a system in a given amount of time. It is measured in bits per second.
* Using the food delivery example again, throughput is like how many orders the restaurant can fulfill in an hour. A higher throughput means the restaurant can handle more orders efficiently.

**Availability**:

* Availability refers to the percentage of time that a system or service is operational and accessible to users.
* This determines the amount of time a system is available to respond to requests. It is calculated: System Uptime / (System Uptime+Downtime).
* Let's stick with the food delivery analogy. If the restaurant is available 24/7 and can deliver food whenever you want, it has high availability. But if the restaurant is closed or experiences downtime frequently, its availability is lower.

**Que: What is Sharding?**

**Ans**: Sharding is a process of splitting the large logical dataset into multiple databases. It also refers to horizontal partitioning of data as it will be stored on multiple machines. By doing so, a sharded database becomes capable of handling more requests than a single large machine. Consider an example - in the following image, assume that we have around 1TB of data present in the database, when we perform sharding, we divide the large 1TB data into smaller chunks of 256GB into partitions called shards.



**Que: How is sharding different from partitioning?**

**Ans:** Let's clarify the differences between sharding and partitioning:

**Sharding:**

* Sharding is a specific type of horizontal partitioning that involves splitting a database or dataset into multiple shards or partitions, each managed independently.
* Each shard contains a subset of the data, and the entire dataset is distributed across multiple servers or nodes in a distributed system.

**Partitioning:**

* Partitioning, also known as data partitioning or vertical partitioning, involves dividing a database or dataset into smaller logical units called partitions based on specific criteria.
* Unlike sharding, which distributes data horizontally across shards, partitioning focuses on organizing data vertically into subsets or categories within the same database or storage system.

In summary, sharding is a distributed data management technique that partitions data across multiple nodes, while partitioning is a data organization technique within a single database or storage system. Sharding is often used in distributed systems to achieve scalability and fault tolerance by spreading the workload, whereas partitioning is used for data organization, optimization, and management within a centralized system.

**Que: What are some of the design issues in distributed systems?**

**Ans:** Designing distributed systems involves addressing various challenges and design issues to ensure reliability, scalability, performance, and fault tolerance. Some of the key design issues in distributed systems include:

* **Consistency and Replication**: Balancing data consistency with scalability and availability.
* **Communication and Messaging**: Efficient communication protocols and message handling.
* **Fault Tolerance and Resilience**: Designing for system reliability and recovery from failures.
* **Scalability and Load Balancing:** Handling increasing workloads and distributing load evenly.
* **Consensus and Coordination:** Achieving agreement among distributed nodes and processes.
* **Data Partitioning and Sharding:** Organizing data across multiple nodes or shards.
* **Security and Privacy:** Implementing security measures and protecting sensitive data.
* **Monitoring and Management:** Tools for monitoring, managing, and troubleshooting distributed systems.

**Design Pattern**

**Que: What are Design Patterns?**

**Ans**: Design patterns are reusable solutions to common software design problems that occur repeatedly in software development. They provide proven solutions and best practices for designing and structuring code to achieve specific goals, such as improving code quality, maintainability, scalability, and flexibility.

Some of the advantages of using design patterns in Java are:

* They are reusable and can be used in multiple projects.
* They provide template solutions for defining system architecture.
* They provide transparency to software design.
* They are well-tested and proven means of developing robust solutions effortlessly.

**Que: What are types of Design Pattern ?**

**Ans**: Here are some common categories of design patterns:

**1. Creational Patterns:**

* These patterns deal with object creation mechanisms, providing flexibility in creating objects and hiding the instantiation logic.
* These patterns provide freedom of choice between creating objects by hiding the logic.
* The objects constructed are decoupled from the implemented system.
* Examples: Singleton, Factory Method, Abstract Factory, Builder, Prototype.

**2. Structural Patterns:**

* Structural patterns focus on organizing classes and objects to form larger structures while keeping them flexible and efficient.
* These patterns help in defining how the structures of classes and objects should be like for defining the composition between classes, interfaces and objects.
* Examples: Adapter, Bridge, Composite, Decorator, Facade, Proxy.

**3. Behavioral Patterns:**

* Behavioral patterns are concerned with how objects interact and communicate with each other, defining algorithms and communication patterns.
* Examples: Observer, Strategy, Command, Template Method, Chain of Responsibility, Iterator, State, Visitor.

**Que: What are all Creational Design Pattern?**

**Ans:** Creational design patterns in software engineering are focused on object creation mechanisms, providing ways to create objects while hiding the creation logic, managing object lifecycles, and enhancing flexibility in object creation. Let's explore each creational design pattern along with an explanation and example:

**1.Singleton Pattern:**

* Purpose: Ensures that a class has only one instance and provides a global point of access to that instance.
* Example: Logging systems, Configuration settings, Database connection pools.
* Implementation: Ensure the constructor of the class is private, provide a static method to access the single instance, and use lazy initialization or eager initialization based on the requirements.

**2. Factory Method Pattern:(fun interface wala hai )**

* Purpose: Defines an interface for creating objects but lets subclasses decide which class to instantiate.
* Example: Abstracting object creation in a class hierarchy, like different types of documents (PDF, Word, etc.) in a document processing application.
* Implementation: Define an interface (or abstract class) for creating objects, and let subclasses implement the factory method to create specific instances of objects.

**3. Abstract Factory Pattern:**

* Purpose: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
* Example: GUI toolkits creating different types of UI components (buttons, menus) for different platforms (Windows, macOS).
* Implementation: Define abstract factories for each family of related objects, and let concrete factories create instances of objects belonging to the respective families.

**4. Prototype Pattern:**

* Purpose: Creates new objects by copying an existing object, known as the prototype, instead of creating instances from scratch.
* Example: Cloning objects, like creating multiple copies of a predefined document template with slight variations.
* Implementation: Create a prototype interface with a method to clone objects, and let concrete prototypes implement the cloning method to create copies of themselves.

**5. Builder Pattern:**

* Purpose: Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
* Example: Building complex objects step by step, such as creating a customizable meal with different components (burger, fries, drink).
* Implementation: Define a builder interface with methods to build different parts of an object, and use a concrete builder to construct the object step by step, providing flexibility in constructing varied object representations.

**Que: What are Singleton Design Pattern?**

**Ans:** The Singleton design pattern is a creational pattern that ensures a class has only one instance and provides a global point of access to that instance. This pattern is used when you want to restrict the instantiation of a class to a single object, which can be useful for scenarios such as managing shared resources, maintaining global state, or controlling access to a resource.

**1. Key Characteristics of Singleton Pattern:**

* **Private Constructor:** The Singleton class has a private constructor to prevent direct instantiation from outside the class.
* **Static Instance:** The Singleton class provides a static method to access the single instance of the class.
* **Lazy Initialization or Eager Initialization**: The Singleton instance can be lazily initialized (created when first accessed) or eagerly initialized (created at class loading time).

**Implementation Steps for Singleton Pattern:**

* **Private Constructor:** Ensure that the constructor of the Singleton class is private to prevent external instantiation.
* **Static Instance**: Provide a static method that returns the single instance of the class. This method should check if the instance exists and create it if necessary.
* **Lazy Initialization or Eager Initialization:** Decide whether to lazily initialize the instance (create it when first accessed) or eagerly initialize it (create it at class loading time).

**Advantages of Singleton Pattern:**

* **Controlled Access:** Ensures that only one instance of a class is created and provides a global point of access to that instance.
* **Resource Sharing:** Useful for managing shared resources, such as database connections or configuration settings.
* **Lazy Initialization**: Allows for lazy initialization of the instance, reducing memory consumption until needed.

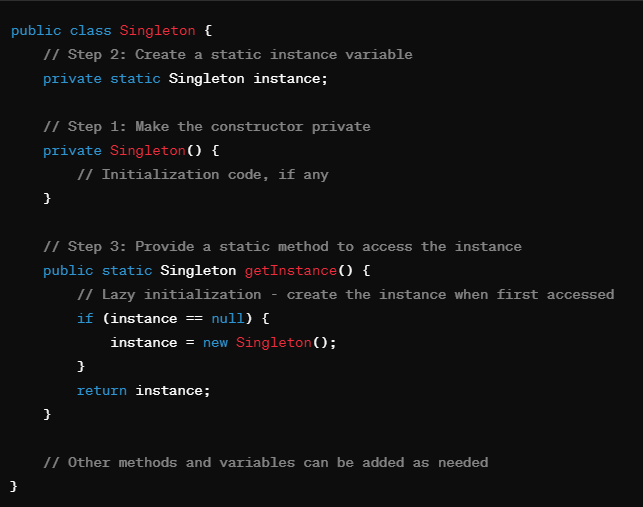
**Disadvantages of Singleton Pattern:**

* Global State: May introduce global state, making it harder to manage dependencies and testability.
* Concurrency Issues: Lazy initialization in a multi-threaded environment may lead to race conditions, requiring synchronization or other techniques to ensure thread safety.

**Que: How will you create a Singleton Design Pattern?**

**Ans:** To create a Singleton design pattern in Java, you need to follow a few key steps:

1. Make the constructor private to prevent external instantiation.
2. Create a static instance of the class.
3. Provide a static method to get the single instance.
4. Optionally, implement lazy initialization or eager initialization based on your requirements.



In this example:

* The constructor of the Singleton class is made private to prevent external instantiation.
* The getInstance() method provides access to the single instance of the class and creates it lazily (on-demand) if it does not exist.
* The instance variable is declared as static so that it's shared across all instances of the class.

**Que: How to make a Singleton Class Thread-Safe?**

**Ans:** To make a Singleton class thread-safe, ensuring that it works correctly in a multi-threaded environment where multiple threads may access and potentially modify the singleton instance concurrently, you can employ various techniques. Here are a few commonly used methods:

**1. Synchronized getInstance() Method:**

Synchronize the getInstance() method to ensure that only one thread can enter the critical section at a time, preventing concurrent creation of multiple instances.

**public class Singleton {**

**private static Singleton instance;**

**private Singleton() {**

**// Initialization code, if any**

**}**

**public synchronized static Singleton getInstance() {**

**if (instance == null) {**

**instance = new Singleton();**

**}**

**return instance;**

**}**

**}**

**2. Eager Initialization with Static Final Instance:**

In this approach, the singleton instance is eagerly initialized at class loading time using a static final variable. Since static final variables are initialized only once, this ensures thread safety.

class loading ke time hi ban jayega toh thread usko disturb hi ni kar payega.

***public class Singleton {***

***private static final Singleton instance = new Singleton();***

***private Singleton() {***

***// Initialization code, if any***

***}***

***public static Singleton getInstance() {***

***return instance;***

***}***

***}***

**3. Double-Checked Locking (DCL):**

Use double-checked locking to reduce the overhead of synchronization after the first initialization. Check if the instance is null, and then synchronize only if needed to create the instance.

***public class Singleton {***

***private static volatile Singleton instance;***

***private Singleton() {***

***// Initialization code, if any***

***}***

***public static Singleton getInstance() {***

***if (instance == null) {***

***synchronized (Singleton.class) {***

***if (instance == null) {***

***instance = new Singleton();***

***}***

***}***

***}***

***return instance;***

***}***

***}***

**Que: What happens when more threads trying to get the singleton object and how to prevent from that?**

**Ans:** When multiple threads try to access the Singleton object concurrently without proper synchronization or thread safety mechanisms, it can lead to various issues such as race conditions, multiple instance creation, and inconsistent state. To prevent these problems and ensure that only one instance of the Singleton class is created in a multi-threaded environment.

Rest answer is in the below question.

**Que: What would happen if we do not have a synchronized method for returning Singleton instance in a multi-threaded environment?**

**Ans:** If you do not use synchronization in a multi-threaded environment for returning a Singleton instance, it can lead to potential issues and inconsistencies, especially during concurrent access to the getInstance() method. Here are some of the consequences:

**1. Race Conditions:**

* Multiple threads can concurrently check if the instance variable is null and create multiple instances of the Singleton class, violating the Singleton pattern.
* This race condition occurs when one thread checks instance == null, another thread creates an instance before the first thread enters the synchronized block, and then the first thread also creates a new instance.

**2. Multiple Instances:**

* Without synchronization, multiple threads can create separate instances of the Singleton class simultaneously, which defeats the purpose of having a single global instance.

**3. Inconsistent State:**

* If multiple instances of the Singleton class are created due to lack of synchronization, it can lead to inconsistent state and behavior in the application.
* Each thread may operate on its own instance, leading to unexpected results and errors.

**Que: What is Lazy initialization and Eager initialization?**

**Ans:** Lazy initialization and eager initialization are two approaches for initializing objects in programming, especially in the context of design patterns like Singleton.

**Lazy Initialization:**

* Lazy initialization means that an object is created only when it is first accessed or required, rather than at the time of class loading or instantiation. This approach defers the creation of the object until it is actually needed.
* Lazy initialization is often used to optimize memory usage and improve performance by delaying the creation of objects until they are necessary.
* In the context of the Singleton design pattern, lazy initialization means that the Singleton instance is created when the getInstance() method is first called, rather than at the time of class loading or initialization.

***public class Singleton {***

***private static Singleton instance;***

***private Singleton() {***

***// Private constructor to prevent external instantiation***

***}***

***public static synchronized Singleton getInstance() {***

***if (instance == null) {***

***instance = new Singleton(); // Lazy initialization***

***}***

***return instance;***

***}***

***}***

*In this example, the Singleton instance is lazily initialized inside the getInstance() method, i.e., it is created when getInstance() is first called.*

**Eager Initialization:**

* Eager initialization means that an object is created at the time of class loading or instantiation, regardless of whether it is immediately needed or not. This approach ensures that the object is available and ready for use as soon as possible.
* Eager initialization is straightforward and guarantees that the object is always available, but it may lead to increased memory usage and slower startup times, especially if the object is not always needed.
* In the context of the Singleton design pattern, eager initialization means that the Singleton instance is created when the class is loaded or initialized, ensuring that it is available for use immediately.

***public class Singleton {***

***private static final Singleton instance = new Singleton();***

***private Singleton() {***

***// Private constructor to prevent external instantiation***

***}***

***public static Singleton getInstance() {***

***return instance; // Eager initialization***

***}***

***}***

*In this example, the Singleton instance is eagerly initialized as a static final variable when the class is loaded, ensuring that it is available immediately for use.*

*static final kar diya hai, static varible belong to class itself issi lie class load hote hi ban jayega.*

**Que: What are Factory Design Pattern?**

**Ans**: The Factory Design Pattern is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. It is used when you want to centralize and decouple the process of object creation from the client code that uses those objects.

**Key Components of Factory Design Pattern:**

**1. Factory Interface or Abstract Class:**

Defines an interface or abstract class for creating objects. This interface usually contains a factory method that returns objects of a specific type.

**2. Concrete Factories:**

Concrete subclasses or implementations of the factory interface/abstract class. Each concrete factory is responsible for creating objects of a specific type or family of related types.

**3. Product Interface or Abstract Class:**

Defines an interface or abstract class for the products (objects) created by the factory. This interface typically declares common methods that all products must implement.

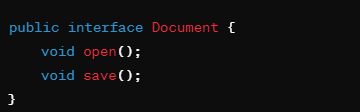
**4. Concrete Products:**

Concrete subclasses or implementations of the product interface/abstract class. These are the actual objects that are created by the factory.

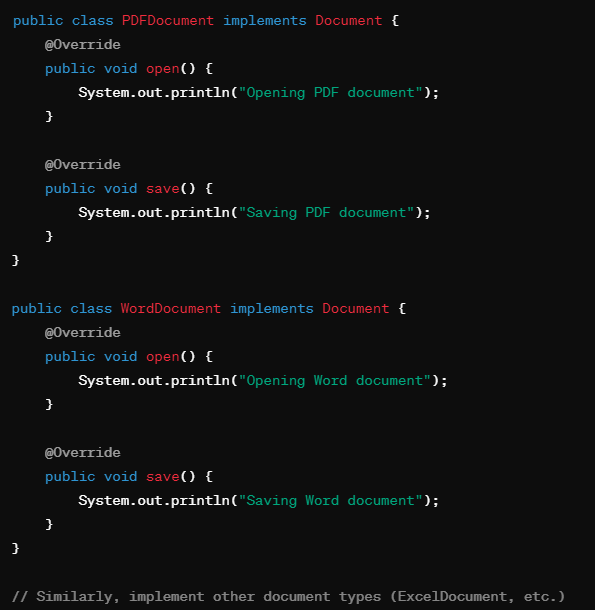
**Example Scenario:**

Consider a scenario where you are developing a document processing application that can handle different types of documents, such as PDF documents, Word documents, and Excel documents. The Factory Design Pattern can be used to create instances of these document types without exposing the creation logic to the client code.

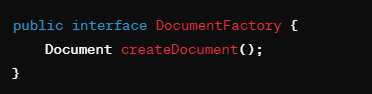
**1. Define Document Interface (Product):**

****

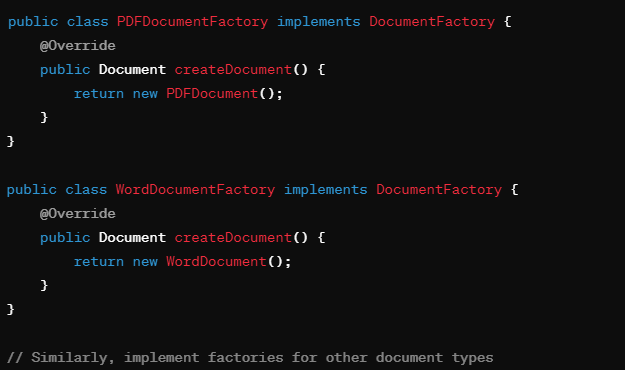
**2. Implement Concrete Document Classes:**

****

**3. Define DocumentFactory Interface or Abstract Class:**

****

**4. Implement Concrete DocumentFactory Classes:**

****

**Que: What are the SOLID Principles?**

**Ans:** The SOLID principles are a set of design principles in object-oriented programming (OOP) intended to make software designs more understandable, flexible, and maintainable. The term "SOLID" is an acronym, where each letter represents one of these principles:

**Single Responsibility Principle (SRP):**

* A class should have only one reason to change, meaning it should have a single responsibility or purpose. This principle promotes cohesion by ensuring that each class is focused on a specific functionality or aspect of the system.
* Example: A class that manages database connections should not also be responsible for handling user authentication.

**Open/Closed Principle (OCP):**

* Software entities (classes, modules, functions) should be open for extension but closed for modification. This means that you should be able to add new functionality to a system without altering existing code.
* Example: Using inheritance or interfaces to extend the behavior of a class without modifying its source code.

**Liskov Substitution Principle (LSP):**

* Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program. In other words, subclasses should be substitutable for their base classes.
* Example: If you have a Bird superclass and a Duck subclass, you should be able to use a Duck object wherever a Bird object is expected.

**Interface Segregation Principle (ISP):**

* Clients should not be forced to depend on interfaces they do not use. Instead, interfaces should be fine-grained and specific to the needs of the clients to avoid unnecessary dependencies.
* Example: Breaking a large interface into smaller, more focused interfaces, so clients can implement only the methods they need.

**Dependency Inversion Principle (DIP):**

* High-level modules/classes should not depend on low-level modules/classes directly. Instead, both should depend on abstractions (e.g., interfaces or abstract classes). Additionally, abstractions should not depend on details, but details should depend on abstractions.
* Example: Using dependency injection to pass dependencies into classes rather than creating dependencies within the class.