Section 1 (Functionality)

1. Technology:

To build the chat application I am going to use the following thechnologies.

- **Programming Language:** Java
- **IDE:** Android Studio
- **UI Design:** XML.
- **Networking:** HTTP/HTTPS protocol for server client communication
- **Real Time Communication:** WebSocket for real time communication.
- **Backend Server:** Firebase
- **Database:** Firebase Realtime Database

2. Description:

- **Project Overview:**

  The goal of this project is to develop a robust and user-friendly Android chat application that allows users to connect with each other in real-time. The application will provide a seamless messaging experience with features such as text messaging, multimedia sharing, and real-time notifications. The project will involve both frontend and backend development to ensure a comprehensive and scalable solution.

- **Key Features:**
  - User registration and Authentication
  - User Profile
  - Real-Time Messaging
  - Multimedia Sharing
  - Contact Management
  - Group Chat

3. Logic:

1. User Authentication:

- Objective: Allow users to register, log in, and authenticate securely.
- Logic:
  - Create user registration and login screens.
  - Implement authentication using email/password or third-party services (Google Sign-In, Facebook Login).

    - Use Firebase Authentication or a custom backend for user authentication.

2. User Interface (UI):

- Objective: Design an intuitive and user-friendly interface for messaging.
- Logic:
    - Create main screens for chat, contacts, and user profiles using XML layouts.
    - Use RecyclerView for displaying lists of messages and contacts.
    - Implement a chat interface with message bubbles, sender information, and a text input field.

3. Real-Time Communication:

- Objective: Enable real-time messaging between users.
- Logic:
    - Utilize WebSockets or a real-time database (Firebase Realtime Database, Firestore) for instant message updates.
    - Implement sending and receiving messages in real-time.
    - Update the UI dynamically when new messages are received.

4. Message Handling:

- Objective: Develop logic to handle different types of messages.
- Logic:
    - Define a message structure with sender, receiver, timestamp, and content.
    - Handle different content types (text, images, videos).
    - Implement logic for displaying multimedia content within the chat interface.

5. Contact Management:

- Objective: Allow users to manage their contacts.
- Logic:
    - Implement functionality to add and remove contacts.
    - Display the online/offline status of contacts.
    - Organize contacts into a list or provide a search feature.

6. Group Chat:

- Objective: Enable users to create and participate in group conversations.
- Logic:

- Implement group creation, adding/removing members, and group chat interfaces.
- Handle notifications for group events.

7. User Profile:

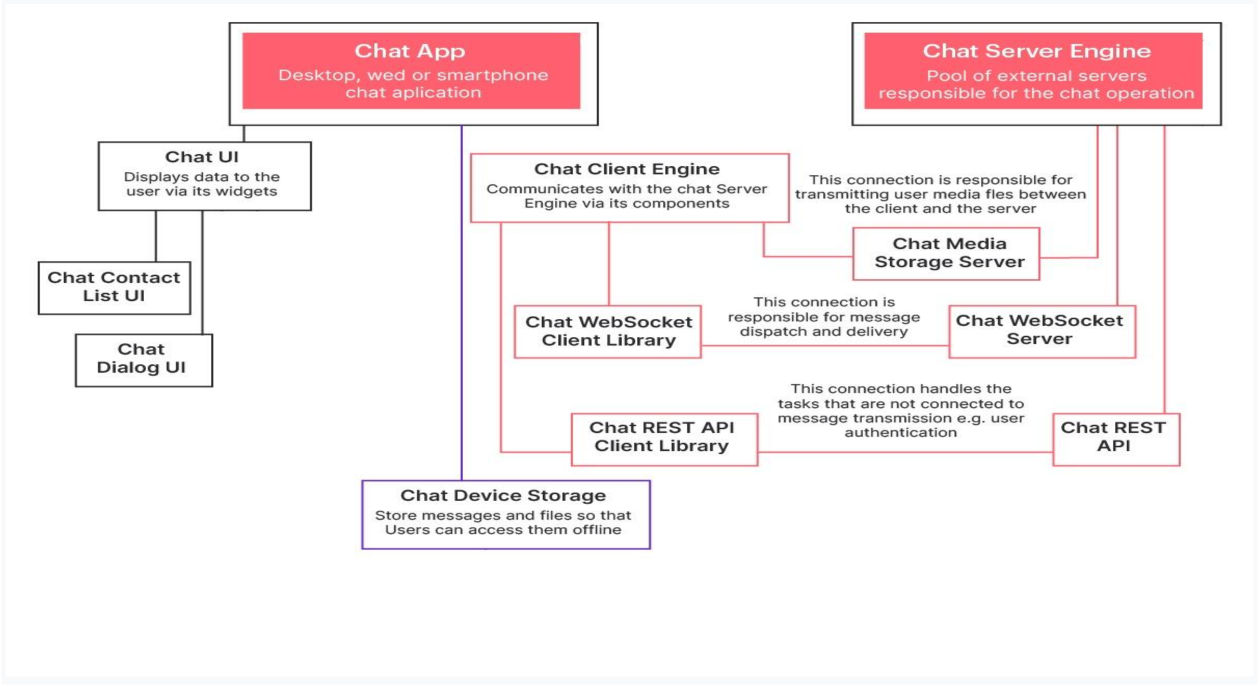
- Objective: Allow users to set up and manage their profiles.
- Logic:
  - Implement features such as setting a profile picture, updating status, and adding a bio.
  - Display user profiles within the app.

8. Database Interaction:

- Objective: Store and retrieve user data, chat history, and other relevant information.
- Logic:
  - Use a database (SQLite, Firebase Realtime Database, etc.) to store data.
  - Implement CRUD operations for managing user and message data.

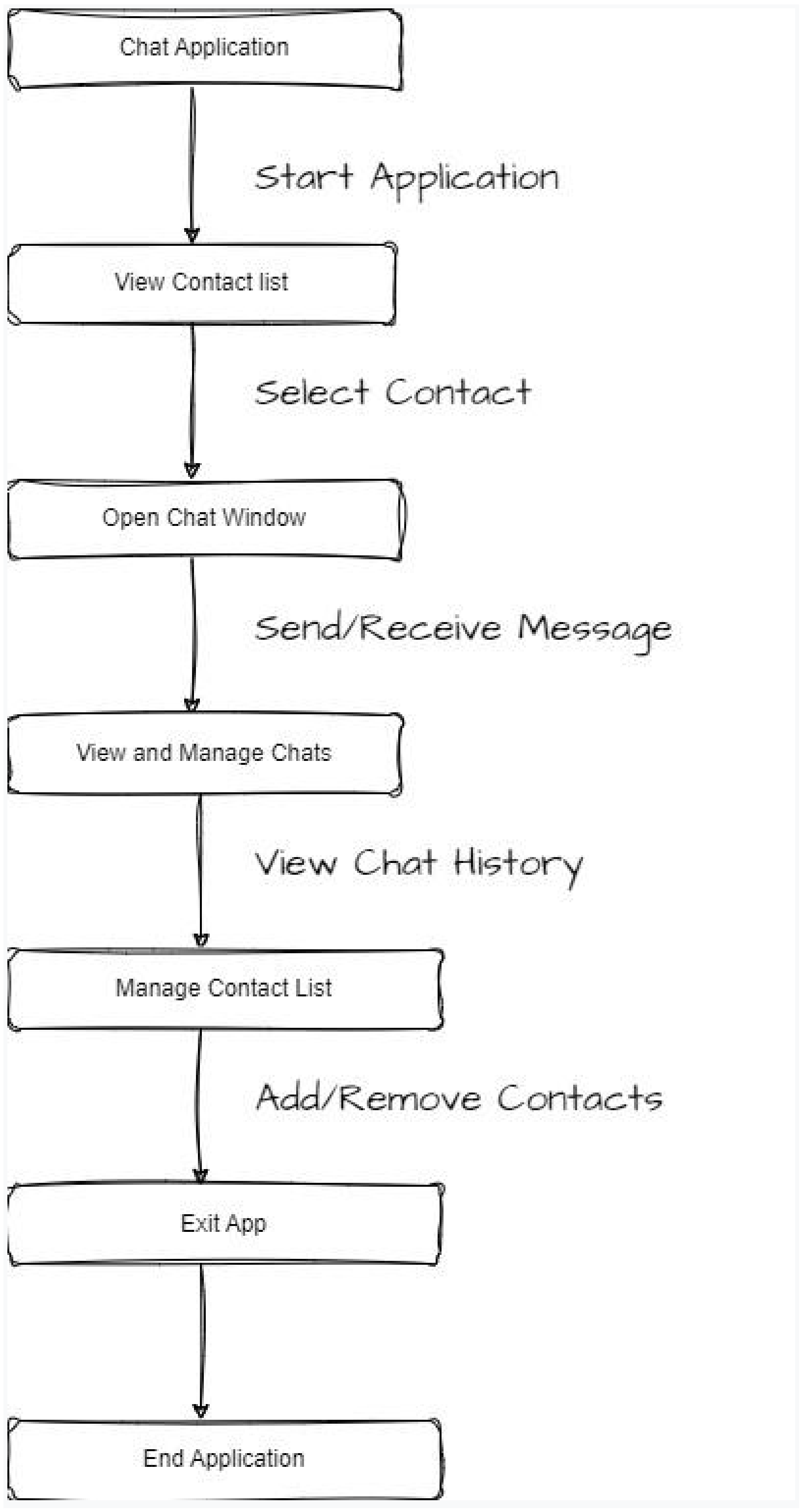
Section 2 (UML Diagram)

❖ Architecture Diagram

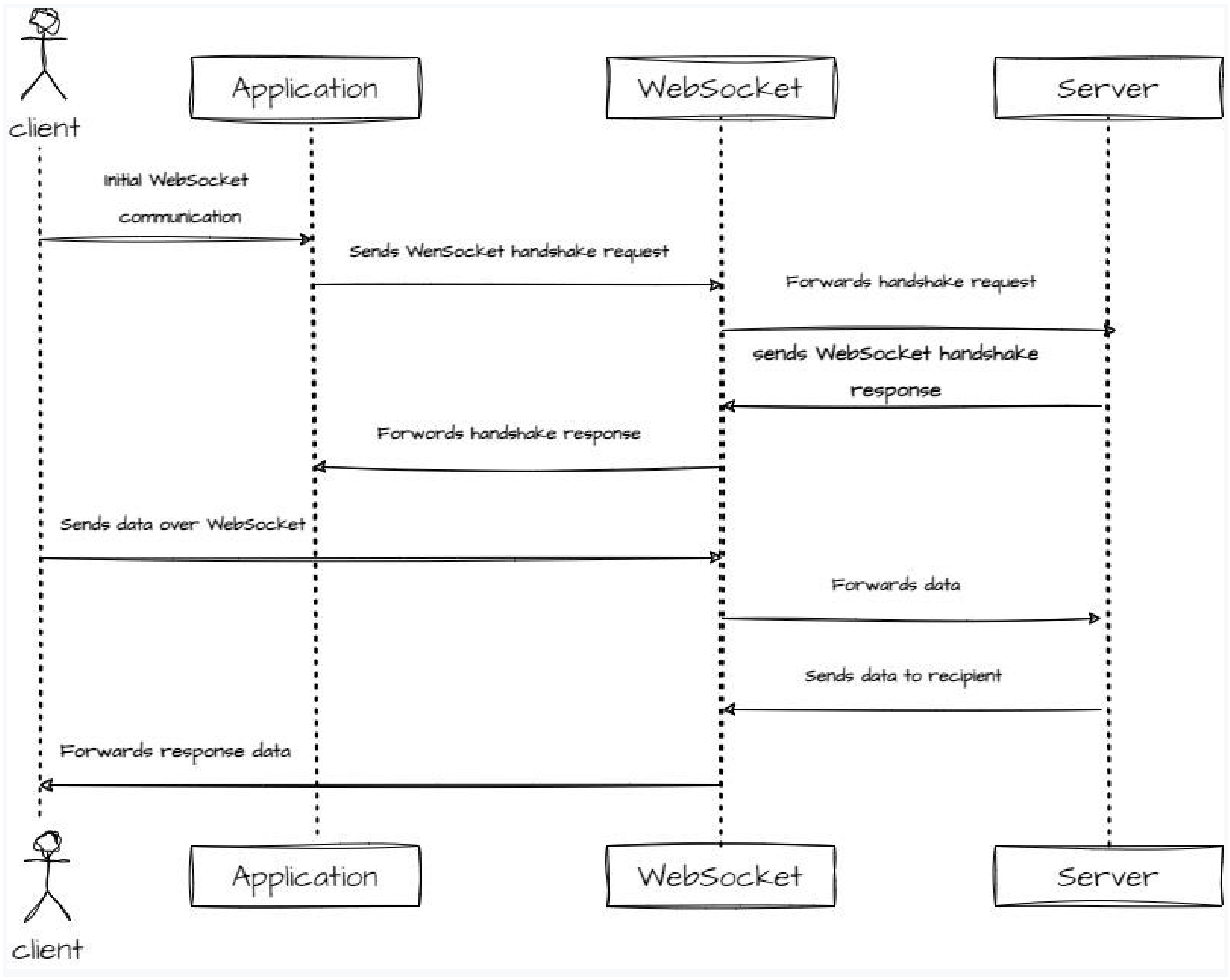


Chat App
Desktop, wed or smartphone chat aplication
Chat Server Engine
Pool of external servers responsible for the chat operation
Chat UI
Displays data to the user via its widgets
Chat Client Engine
Communicates with the chat Server Engine via its components
This connection is responsible for transmitting user media fles between the client and the server
Chat Media Storage Server
Chat Contact List UI
Chat Dialog UI
Chat WebSocket Client Library
This connection is responsible for message dispatch and delivery
Chat WebSocket Server
This connection handles the tasks that are not connected to message transmission e.g. user authentication
Chat REST API Client Library
Chat REST API
Chat Device Storage
Store messages and files so that Users can access them offline

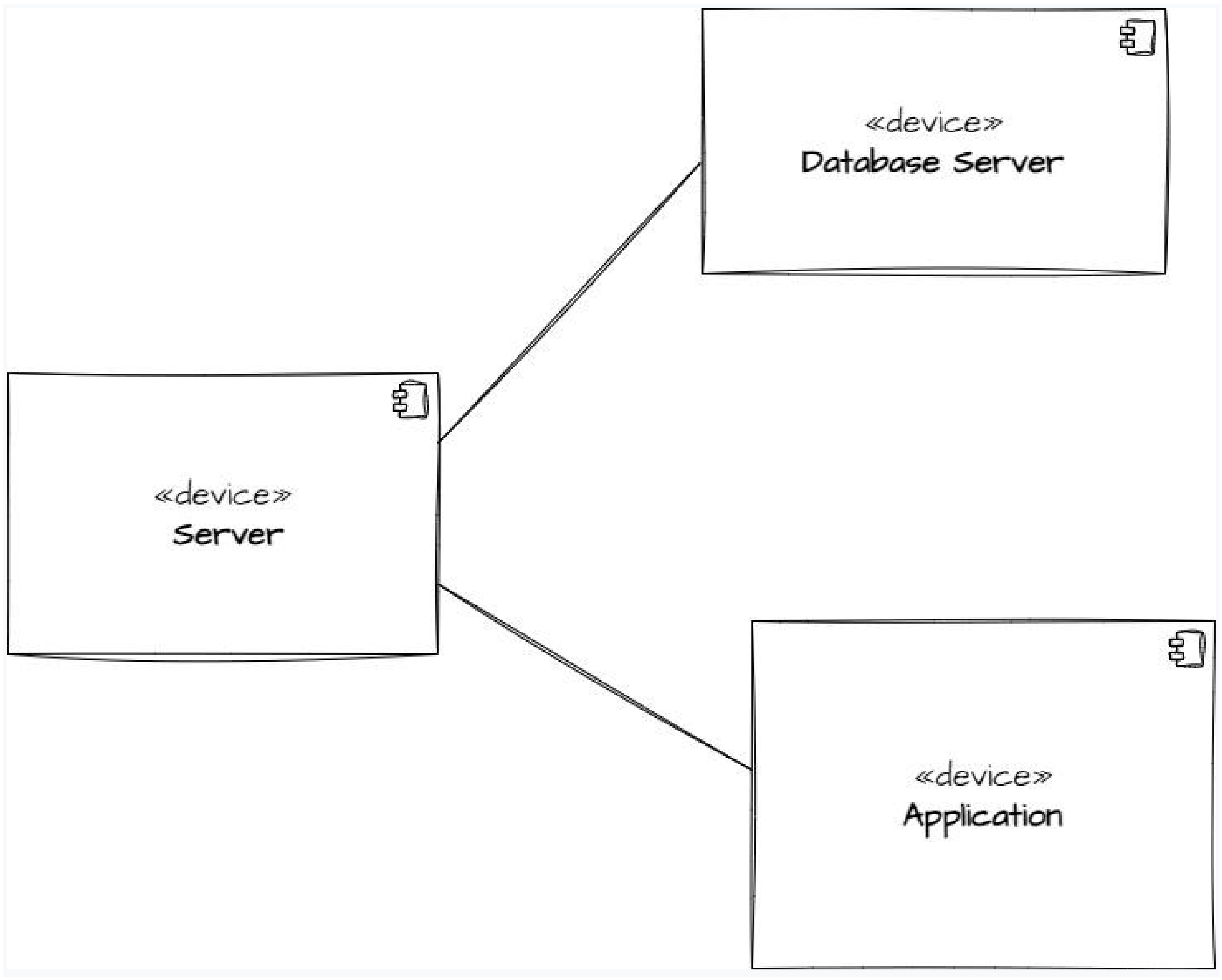
❖ Activity Diagram



❖ Communication Diagram


client
Application
WebSocket
Server
Initial WebSocket communication
Sends WenSocket handshake request
Forwards handshake request
sends WebSocket handshake response
Forwords handshake response
Sends data over WebSocket
Forwards data
Sends data to recipient
Forwards response data
client
Application
WebSocket
Server

❖ Deployment Diagram



Section 3

❖ Explain EJB, JMS, MDB in detail

EJB, JMS, and MDB are Java EE (Enterprise Edition) technologies that play crucial roles in building distributed and enterprise-level Java applications. Here's a detailed explanation of each:

1. EJB (Enterprise JavaBeans):

Definition:
Enterprise JavaBeans (EJB) is a server-side component architecture for building scalable, distributed, and transactional enterprise-level applications in Java.

Key Features:

- Components: EJBs are server-side components that encapsulate business logic and can be deployed on a Java EE-compliant application server.
- Distributed Computing: EJBs enable the development of distributed applications by allowing components to be distributed across multiple tiers.
- Transaction Management: EJB provides built-in support for managing transactions, ensuring the consistency and reliability of business operations.
- Security: EJB includes a security model for securing enterprise applications, including authentication and authorization mechanisms.
- Concurrency Control: EJB supports concurrent access to components, and it provides mechanisms for managing concurrent transactions and preventing conflicts.

Types of EJBs:

1. **Session Beans:** Used to represent business logic in a transactional and non-persistent manner.
2. **Entity Beans (deprecated):** Used to represent persistent data in a relational database. Note that Entity Beans have been deprecated in later versions of Java EE in favor of other persistence mechanisms like JPA (Java Persistence API).
3. **Message-Driven Beans (MDB):** Used for asynchronous processing of messages.

2. JMS (Java Message Service):

Definition:

Java Message Service (JMS) is a Java API that allows applications to create, send, receive, and read messages asynchronously.

Key Features:

- Messaging Paradigms: JMS supports both Point-to-Point (P2P) and Publish/Subscribe (Pub/Sub) messaging paradigms.
- Message Types: It allows the exchange of messages between components in a distributed environment and supports various message types, such as TextMessage, MapMessage, ObjectMessage, etc.
- Asynchronous Communication: JMS enables asynchronous communication between components, providing loose coupling and improving system scalability and responsiveness.
- Reliability: JMS ensures the reliable delivery of messages through features like acknowledgment and transactional processing.
- Message Listeners: JMS allows the creation of MessageListeners to process messages as they arrive.

3. MDB (Message-Driven Bean):

Definition:

A Message-Driven Bean (MDB) is a type of EJB specifically designed to process JMS messages asynchronously.

Key Features:

- Asynchronous Processing: MDBs are used to perform asynchronous processing of JMS messages. They are triggered by the arrival of messages in a JMS destination (queue or topic).
- EJB Component: Despite being specifically related to JMS, MDB is still an EJB component, benefiting from features such as transaction management and security.
- Activation Configuration: MDBs are configured using activation specifications, which define the connection details to the JMS provider and other properties.

Lifecycle of an MDB:

1. Creation: An MDB is created when a message arrives at the associated JMS destination.
2. Processing: The onMessage() method of the MDB is called to process the received message.
3. Destruction: The container removes the MDB after it has processed the message, and it can be garbage collected.

❖ Explain replication

Distributed systems allow sharing of information and communications. Data Replication in Distributed Systems enhances fault tolerance, reliability, and accessibility by maintaining consistency amongst redundant resources such as software and hardware components.

Data Replication is the process of generating numerous copies of data. You then store these copies also called replicas in various locations for backup, fault tolerance, and improved overall network accessibility. The data replicates can be stored on on-site and off-site servers, as well as cloud-based hosts, or all within the same system.

Data Replication in Distributed Systems refers to the distribution of data from a source server to other servers while keeping the data updated and synced with the source so that users can access data relevant to their activities without interfering with the work of others.

❖ Explain Distributed File Systems

A Distributed File System (DFS) is a file system that allows multiple computers to share files and storage resources across a network. The goal of a distributed file system is to provide a unified, transparent view of distributed storage, making it appear to users as if they are accessing a centralized file system. This enables efficient data sharing, collaboration, and scalability in large-scale computing environments. Here are key concepts and features of distributed file systems.

Key Concepts:

- ➢ **Distributed Storage:** Data is stored across multiple nodes or servers in the network, and the file system provides a mechanism to manage and access this distributed storage.

- **Transparency:** Users and applications interact with the distributed file system without needing to be aware of the underlying complexities of the distributed architecture. This includes location transparency, where the physical location of data is hidden from users.
- **Scalability:** Distributed file systems are designed to scale horizontally by adding more nodes to the network. This allows them to handle increasing amounts of data and user requests.
- **Fault Tolerance:** Distributed file systems often incorporate mechanisms for fault tolerance to ensure that the system continues to operate even in the presence of node failures or network issues.
- **Consistency:** Ensuring data consistency across distributed nodes is a crucial aspect of distributed file systems. Changes made to files or directories by one user or node should be reflected consistently across the entire system.
- **Concurrency Control:** Managing concurrent access to files is essential to prevent conflicts and data corruption. Distributed file systems employ locking mechanisms or other concurrency control techniques.
- **Caching:** Caching strategies are employed to improve performance by storing frequently accessed data closer to the user or application. This helps reduce the need for repeated access to remote nodes.

❖ Security

Security in distributed applications is a critical aspect that involves safeguarding data, resources, and communication across multiple nodes or components within a network. Ensuring the confidentiality, integrity, and availability of information is paramount in distributed systems. Here are key considerations and practices for implementing security in distributed applications:

- Authentication
- Authorization
- Encryption
- Verification
- Secure Communication Protocols
- Secure APIs
- Intrusion Detection and Prevention
- Secure Configuration Management
- Distributed Firewall and Network Security
- Incident Response Plan

❖ Container

A container is a lightweight, portable, and self-sufficient software unit that encapsulates an application and its dependencies. Containers provide a consistent and isolated environment for running applications across different computing environments, from development to testing and production. They are part of containerization technology, which enables the packaging, distribution, and deployment of software in a consistent and reproducible manner. Here are key concepts and components related to containers:

Key Concepts:

1. Containerization: The process of encapsulating an application, its dependencies, and runtime environment into a container.

2. Container Image: A lightweight, standalone, and executable package that includes everything needed to run a piece of software, including the code, runtime, libraries, and system tools.

3. Docker: Docker is one of the most widely used containerization platforms. It provides tools and a platform for developing, shipping, and running applications in containers.

4. Container Orchestrators: Tools that automate the deployment, scaling, and management of containerized applications. Examples include Kubernetes, Docker Swarm, and Apache Mesos.

5. Microservices: Architectural style where applications are composed of small, independent, and loosely coupled services. Containers are often used to deploy and manage microservices.

6. Container Registry: A repository for storing and managing container images. Docker Hub is a popular public container registry, and organizations often use private container registries for their images.

❖ Explain what is Docker and kubernetes

Docker:

Docker is an open-source platform that enables developers to automate the deployment of applications inside lightweight, portable containers. Containers are self-sufficient units that encapsulate an application and its dependencies, ensuring consistency across different environments.

Key Concepts:

1. Containerization: Docker uses containerization to package applications and their dependencies into a standardized unit called a container. Containers are isolated and share the host operating system's kernel, providing lightweight and efficient deployment.

2. Docker Image: An image is a lightweight, standalone, and executable package that includes everything needed to run an application, such as the code, runtime, libraries, and system tools. Images are used to create containers.

3. Dockerfile: A Dockerfile is a script that contains instructions for building a Docker image. It specifies the base image, sets up the environment, and defines the steps to create the final image.

4. Docker Engine: The Docker Engine is the software responsible for running and managing containers on a host system. It includes a daemon process, REST API, and a command-line interface.

5. Docker Hub: Docker Hub is a cloud-based registry that hosts and allows users to share Docker images. It provides a repository for storing and managing container images.

Kubernetes:

Kubernetes, often abbreviated as K8s, is an open-source container orchestration platform for automating the deployment, scaling, and management of containerized applications. It provides a robust and extensible framework for managing containerized workloads and services.

Key Concepts:

1. Container Orchestration: Kubernetes automates the deployment and management of containers, handling tasks such as scaling, load balancing, and rolling updates. It abstracts the underlying infrastructure, making it easier to manage distributed systems.

2. Pods: The smallest deployable units in Kubernetes are pods. A pod represents a group of one or more containers that share the same network namespace and storage. Containers within a pod can communicate with each other using localhost.

3. Replication Controllers and Deployments: Replication Controllers (deprecated in favor of Deployments) manage the desired number of pod replicas, ensuring that the specified number of instances is running at all times. Deployments provide more advanced features for managing rolling updates and rollbacks.

4. Services: Kubernetes Services enable communication between different pods in a consistent and abstract manner. Services provide a stable IP address and DNS name for accessing pods, even as they scale or move across nodes.

5. Nodes: Nodes are the individual machines (physical or virtual) that make up a Kubernetes cluster. Nodes run the containers and host the necessary components for communication with the master.

6. Master and Control Plane: The master is responsible for managing and controlling the cluster. It includes components such as the API server, controller manager, scheduler, and etc (a distributed key-value store for configuration data).