Week 3 Assignment

1. **What is Docker?**

Docker is an open-source platform that allows developers to build, package, and deploy applications inside **containers**. Containers are lightweight, portable, and self-sufficient environments that include everything needed to run an application (code, libraries, dependencies).

1. **Why is Docker Used?**
2. **Portability:** Dockerized applications run consistently across different environments (local, testing, production, cloud).
3. **Efficiency:** Uses fewer resources compared to traditional Virtual Machines (VMs) since containers share the host OS kernel.
4. **Scalability:** Easily scales applications by running multiple container instances.
5. **Faster Deployment:** Containers start quickly, improving development and deployment speed.
6. **Microservices Support:** Ideal for breaking applications into smaller, manageable services.
7. **Security & Isolation:** Each container runs independently, reducing conflicts between applications.
8. **CI/CD & DevOps Integration:** Works seamlessly with automation tools like Jenkins, Kubernetes, and Docker Compose for smooth development workflows.
9. **Docker Images and Containers ?**
10. Docker Image → A blueprint for creating containers, containing code, dependencies, and configurations.
    * Example: docker pull nginx (Downloads an image).
11. Docker Container → A running instance of an image, providing an isolated environment.
    * Example: docker run -d -p 8080:80 nginx (Runs a container).
12. Key Differences:
    * Image is read-only; Container is a running process.
    * Image is stored in a registry; Container runs on a host.

Docker containers allow efficient, scalable, and portable application deployment**.**

1. **Dockerfile and how to create it**

A **Dockerfile** is a script containing a set of instructions to automate the process of creating a Docker image. It defines everything required to set up an application inside a container.Steps to Create a Dockerfile:

It consists of various instructions like

FROM , RUN , COPY , WORKDIR ,CMD

Mkdir myapp

Cd myapp

Touch Dockerfile

Now write a simple Dockerfile

# Use Python base image

FROM python:3.9

# Set working directory

WORKDIR /app

# Copy files

COPY . .

# Install dependencies

RUN pip install -r requirements.txt

# Expose port

EXPOSE 5000

# Run the app

CMD ["python", "app.py"]

1. **Docker Networking Basics**

**1. Types of Docker Networks**

1. **Bridge (Default) – Isolates containers but allows them to communicate.**

**docker network create my\_bridge**

1. **Host – Removes isolation and makes the container use the host’s network.**
2. **Overlay – Used in multi-host Swarm setups.**
3. **Macvlan – Assigns a unique MAC address to each container.**
4. **None – Disables networking for the container.**

**2. Basic Commands**

* **List networks: docker network ls**
* **Inspect network: docker network inspect my\_bridge**
* **Connect container to network: docker network connect my\_bridge my\_container**

1. **Docker Volumes**

Docker volumes are used for persistent data storage, ensuring data is not lost when containers stop or restart.

1. Types of Storage in Docker

1. Volumes – Managed by Docker, stored in /var/lib/docker/volumes/.
2. Bind Mounts – Directly links a host directory to a container.
3. Tmpfs Mounts – Stored in RAM, used for temporary data.

2. Basic Commands

* #Create a volume:
* docker volume create my\_data
* #List volumes:
* docker volume ls
* #Use a volume in a container:
* docker run -v my\_data:/app/data myapp
* #Inspect volume:
* docker volume inspect my\_data
* #Remove volume:
* docker volume rm my\_data

Docker volumes help in data persistence, sharing, and backup for containers.

1. **Docker Compose**

Docker Compose is a tool for managing multi-container applications using a simple YAML configuration file. It allow to define and run multiple services (containers) together.

Key Features:

* Manages multiple containers with a single command.
* Defines dependencies between services (e.g., a database starts before an app).
* Simplifies networking and volume management.
* Uses a single YAML file for configuration.

Common Usage:

* Start all services together.
* Stop and remove all services easily.
* Check the status of running containers.

Docker Compose makes containerized application deployment easier and more efficient.

1. **Docker Registry**

A Docker Registry is a storage system for Docker images, allowing users to store, share, and manage container images.

Types of Docker Registries:

1. Docker Hub – The default public registry for storing and sharing images.
2. Private Registry – A self-hosted registry for secure image storage.
3. Cloud Registries – Services like AWS Elastic Container Registry (ECR), Google Container Registry (GCR), and Azure Container Registry (ACR).

Why Use a Docker Registry?

* Centralized storage for container images.
* Easy sharing and deployment of images across teams.
* Version control for different application builds.
* Security by restricting image access in private registries.

Docker registries help streamline image management and deployment in containerized environments.

1. **Docker Swarm**

Docker Swarm

Docker Swarm is a container orchestration tool that allows managing and scaling multiple Docker containers across multiple hosts. It turns a group of Docker hosts into a single virtual cluster for easier deployment and management.

Key Features of Docker Swarm:

1. Clustering: Groups multiple Docker nodes into a single system.
2. Load Balancing: Distributes traffic across multiple containers for better performance.
3. Service Scaling: Allows increasing or decreasing the number of container instances easily.
4. Fault Tolerance: Automatically restarts failed containers to maintain uptime.
5. Secure Communication: Uses TLS encryption for secure interaction between nodes.

Docker Swarm simplifies container orchestration for high availability, scalability, and efficient resource management.

1. **Docker Commands**

Docker Commands

Docker provides various commands to manage containers, images, networks, and volumes efficiently.

1. Basic Commands

* Check Docker version → docker --version
* View running containers → docker ps
* View all containers (including stopped ones) → docker ps -a

2. Image Management

* Pull an image → docker pull image\_name
* List downloaded images → docker images
* Remove an image → docker rmi image\_name

3. Container Management

* Create and run a container → docker run image\_name
* Start/Stop a container → docker start/stop container\_id
* Remove a container → docker rm container\_id

4. Docker Networking

* List networks → docker network ls
* Create a network → docker network create network\_name

5. Docker Volume Management

* List volumes → docker volume ls
* Create a volume → docker volume create volume\_name

These commands help efficiently manage Docker environments.

1. **Docker vs Virtual Machines (VMs)**

Docker and Virtual Machines both enable application isolation, but they work differently in terms of efficiency, resource usage, and scalability.

1. Key Differences:

|  |  |  |
| --- | --- | --- |
| Feature | Docker (Containers) | Virtual Machines (VMs) |
| Architecture | Shares the host OS kernel | Each VM has its own OS |
| Startup Time | Fast (seconds) | Slow (minutes) |
| Resource Usage | Lightweight, consumes less memory | Heavy, requires more CPU and RAM |
| Isolation | Process-level isolation | Full OS-level isolation |
| Portability | Highly portable across different platforms | Less portable, depends on hypervisor |
| Use Case | Microservices, cloud apps, CI/CD | Running multiple OS instances, legacy apps |

2. When to Use:

* Use Docker when you need fast, scalable, and lightweight containerized applications.
* Use Virtual Machines when you need full OS isolation or to run different operating systems.

Docker is more efficient for modern application deployment, while VMs are better for running diverse OS environments.

1. **What is Kubernetes and why is it used?**

What is Kubernetes?

Kubernetes (K8s) is an open-source container orchestration platform that automates the **deployment, scaling, and management** of containerized applications. It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF).

Why is Kubernetes Used?

1. Automated Deployment & Scaling – Automatically deploys and scales containers based on demand.
2. Load Balancing – Distributes traffic efficiently across containers to prevent failures.
3. Self-Healing – Detects and restarts failed containers automatically.
4. Service Discovery – Manages internal communication between containers without needing IP configurations.
5. Multi-Cloud Compatibility – Runs seamlessly on on-premises, public, and hybrid cloud environments.
6. Efficient Resource Utilization – Optimizes computing power by distributing workloads across nodes.
7. Rolling Updates & Rollbacks – Deploys new versions of applications with zero downtime and allows reverting if needed.

Kubernetes is essential for managing large-scale containerized applications efficiently and ensuring high availability.

1. **Kubernetes Pods and Deployments**

Kubernetes Pods and Deployments

A Pod is the smallest unit in Kubernetes, containing one or more containers that share storage and networking. It ensures applications run in isolated environments but is temporary, meaning Kubernetes may replace it if it fails.

A Deployment manages multiple Pods, ensuring high availability, scalability, and automatic updates. It allows rolling updates without downtime and automatically replaces failed Pods.

Pods run applications, while Deployments ensure they remain available and up to date.

1. **Kubernetes Services**

A Service in Kubernetes ensures stable communication between different components of an application, even when Pods are dynamically created or deleted.

Types of Services:

1. ClusterIP (Default) – Internal communication only, not accessible outside the cluster.
2. NodePort – Exposes the service on a specific port of each node, allowing external access.
3. LoadBalancer – Uses a cloud provider’s load balancer to expose the service externally.
4. ExternalName – Redirects traffic to an external domain instead of a Kubernetes resource.
5. **Why Use Kubernetes Services?**

* Provide a fixed endpoint for dynamically changing Pods.
* Enable load balancing to distribute traffic efficiently.
* Support service discovery, allowing components to find each other.

1. **ConfigMaps & Secrets in Kubernetes**

Both ConfigMaps and Secrets help store configuration data separately from application code, ensuring flexibility and security in Kubernetes.

1. ConfigMaps (For Non-Sensitive Data)

* Stores plain text configuration (e.g., environment variables, command-line arguments).
* Used for configuring applications dynamically without modifying container images.
* Example Use Cases: Database URLs, application settings, feature flags.

2. Secrets (For Sensitive Data)

* Stores confidential information (e.g., passwords, API keys, TLS certificates).
* Data is Base64 encoded but should be encrypted for better security.
* Helps avoid storing sensitive info in Pod definitions or ConfigMaps.

Key Differences:

|  |  |  |
| --- | --- | --- |
| Feature | ConfigMap | Secret |
| Data Type | Plain text (non-sensitive) | Encoded (for sensitive info) |
| Use Case | App settings, URLs | Passwords, API keys, certificates |
| Security | Stored as text | More secure, recommended for credentials |
| Encoding | No encoding | Base64 encoded (not encrypted) |

Why Use Them?

* Decouple configuration from code for easy updates.
* Improve security by keeping secrets separate.
* Enable dynamic changes without rebuilding containers.

1. **Namespaces in Kubernetes**

A **Namespace** in Kubernetes is a way to **logically separate** resources within a cluster. It helps manage multiple applications, teams, or environments efficiently.

**Why Use Namespaces?**

1. **Resource Isolation** – Different teams or projects can have their own space.
2. **Access Control** – Restrict access using Role-Based Access Control (RBAC).
3. **Resource Management** – Set CPU and memory limits per namespace.

**Default Namespaces in Kubernetes:**

* **default** – Used when no namespace is specified.
* **kube-system** – Contains system-related components (e.g., API server, scheduler).
* **kube-public** – Publicly accessible resources.
* **kube-node-lease** – Stores node heartbeat data for availability tracking.

**When to Use Namespaces?**

* Large organizations managing multiple teams.
* Running **staging, testing, and production** environments in the same cluster.
* Enforcing **resource quotas** for different applications.

1. **Kubernetes Volumes**

Kubernetes Volumes provide **persistent storage** for containers, ensuring data is not lost when a Pod restarts or moves. Unlike ephemeral storage in containers, volumes help retain important files and logs.

**Types of Kubernetes Volumes:**

1. **emptyDir** – Temporary storage that lasts as long as the Pod runs. Gets deleted if the Pod stops.
2. **hostPath** – Uses a folder from the worker node’s filesystem. Useful for debugging but not ideal for production.
3. **Persistent Volume (PV) & Persistent Volume Claim (PVC)** – Provides long-term storage managed by Kubernetes. Supports cloud storage solutions.
4. **ConfigMap & Secret Volumes** – Used for injecting configurations or sensitive data into a Pod.
5. **NFS (Network File System)** – Allows multiple Pods to access shared storage across nodes.
6. **CSI (Container Storage Interface)** – A flexible method to integrate with external storage providers like AWS EBS, Google Persistent Disk, etc.

**Why Use Kubernetes Volumes?**

* **Data Persistence** – Ensures storage survives Pod restarts.
* **Shared Storage** – Allows multiple containers in a Pod to share data.
* **Scalability** – Supports cloud-based and distributed storage solutions.

1. **Autoscaling in Kubernetes**

Autoscaling in Kubernetes automatically adjusts the number of **Pods, Nodes, or Resources** based on workload demand. This helps optimize performance and resource usage.

**Types of Autoscaling:**

1. **Horizontal Pod Autoscaler (HPA)**
   * **Scales the number of Pods** up or down based on CPU, memory, or custom metrics.
   * Example: If CPU usage exceeds 70%, HPA adds more Pods.
2. **Vertical Pod Autoscaler (VPA)**
   * **Adjusts CPU and memory** requests/limits for existing Pods.
   * Example: If a Pod needs more memory, VPA increases its allocation instead of adding new Pods.
3. **Cluster Autoscaler (CA)**
   * **Adds or removes worker nodes** based on resource needs.
   * Example: If all nodes are full, CA provisions a new node; if underutilized, it removes nodes.

**Why Use Autoscaling?**

* **Optimizes performance** by handling traffic spikes.
* **Reduces costs** by scaling down when demand is low.
* **Improves availability** by automatically managing resources.

1. **Role-Based Access Control (RBAC) in Kubernetes**

RBAC in Kubernetes **controls access** to resources based on a user’s role. It helps enforce **security** by restricting actions like creating, modifying, or deleting resources.

**Key RBAC Components:**

1. **Role & ClusterRole**
   * **Role**: Defines permissions within a single **namespace**.
   * **ClusterRole**: Defines permissions **cluster-wide** (across all namespaces).
2. **RoleBinding & ClusterRoleBinding**
   * **RoleBinding**: Assigns a **Role** to a user or service account **within a namespace**.
   * **ClusterRoleBinding**: Assigns a **ClusterRole** to a user **across the cluster**.

**Why Use RBAC?**

* **Security** – Restricts unauthorized access to Kubernetes resources.
* **Granular Control** – Assigns permissions at namespace or cluster level.
* **Compliance** – Helps meet security policies by enforcing access rules.

1. **Helm Charts – Kubernetes Package Management**

**Helm** is a package manager for Kubernetes that simplifies the deployment of applications by using **Helm Charts** (pre-configured templates).

**Why Use Helm?**

* **Simplifies Deployment** – Automates application installation and updates.
* **Reusable & Versioned** – Helm Charts allow version control and reuse.
* **Customizable** – Supports configuration changes via **values.yaml**.

**Key Helm Components:**

1. **Chart** – A packaged application containing Kubernetes manifests.
2. **Values.yaml** – Defines customizable settings for the chart.
3. **Release** – A deployed instance of a Helm Chart.
4. **Repositories** – Storage locations for Helm Charts (e.g., ArtifactHub, Bitnami).

**Common Use Cases:**

* Deploying databases (MySQL, PostgreSQL) easily.
* Managing microservices in Kubernetes.
* Automating application rollouts and rollbacks.

1. **Basic Kubernetes Commands**

**1. Cluster & Node Management**

* kubectl cluster-info → Displays cluster details.
* kubectl get nodes → Lists all nodes in the cluster.
* kubectl describe node <node-name> → Shows details of a specific node.

**2. Pod Management**

* kubectl get pods → Lists all running Pods.
* kubectl describe pod <pod-name> → Shows details of a specific Pod.
* kubectl delete pod <pod-name> → Deletes a Pod.

**3. Deployment & Service Management**

* kubectl get deployments → Lists all Deployments.
* kubectl get services → Lists all Services.
* kubectl delete service <service-name> → Deletes a Service.

**4. Namespace Management**

* kubectl get namespaces → Lists all namespaces.
* kubectl create namespace <namespace-name> → Creates a new namespace.
* kubectl delete namespace <namespace-name> → Deletes a namespace.

**5. Logs & Debugging**

* kubectl logs <pod-name> → Retrieves logs from a Pod.
* kubectl exec -it <pod-name> -- /bin/sh → Accesses a running Pod’s shell.