Docker Mock Interview Questions and answers - Pankaj Dhapade

Absolutely, Pankaj. Below are detailed, professional-style answers for the first 15 Docker interview questions, tailored to reflect the knowledge and confidence of a DevOps engineer with 3 years of experience working in real-world setups.

---

1. What is Docker?

Docker is an open-source platform designed to automate the deployment, scaling, and management of applications using containerization. Containers are lightweight, portable, and consistent environments that bundle an application with all its dependencies, libraries, and configurations. Docker allows developers and DevOps engineers to build once and run anywhere—be it on a developer's laptop, a staging server, or production.

Real-world note:
In production, Docker enables teams to ship microservices independently, supports immutable infrastructure, and integrates tightly with CI/CD pipelines and orchestrators like Kubernetes.

---

2. What is a Docker Container?

A Docker container is a runtime instance of a Docker image. It includes the application code along with its dependencies, configuration files, and runtime libraries, all isolated from the host system.

Key characteristics:

- Lightweight (shares the host OS kernel)
- Isolated (via namespaces and cgroups)
- Portable and reproducible

Production tip:
Containers are ideal for microservices architectures because they start fast, are easily replaceable, and support horizontal scaling.

---

3. What are Docker Images?

A Docker image is a read-only template that contains the application code, dependencies, libraries, and the file system required to run a container. It acts as a blueprint for containers.

Lifecycle in production:

- Developers build images using a `Dockerfile`.

- These are versioned and stored in a registry (like Docker Hub or AWS ECR).
- CI/CD pipelines fetch and deploy these images to environments.

---

4. What is Docker Hub?

Docker Hub is a cloud-based container registry provided by Docker Inc., where developers can find, share, and manage Docker images. It offers:

- Public and private repositories
- Automated build triggers
- Image version tagging

In teams:
Private registries are used in enterprises to secure internal images, while public ones like `nginx`, `redis`, and `node` are widely pulled for base image layers.

---

5. Explain Docker Architecture.

Docker has a client-server architecture consisting of:

- **Docker Client:** CLI tool (`docker`) that users interact with.
- **Docker Daemon (`dockerd`):** Background service that manages containers and images.
- **Docker Images:** Stored in a registry, pulled and run by the daemon.
- **Docker Registries:** Like Docker Hub or private repositories.

Interaction Flow:

```
docker client  → REST API  → Docker daemon → Image → Container
```

Real-world insight:
On a production server, `dockerd` often runs as a service controlled by `systemd`, with strict firewall and resource controls.

---

6. What is a Dockerfile?

A Dockerfile is a script with a series of instructions used to build a Docker image. It defines:

- Base image
- App dependencies
- Configuration
- Entry points

Example:

```
FROM python:3.9
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
CMD ["python", "app.py"]
```

Tip:
Use multi-stage builds to keep images lean, and avoid installing unnecessary packages in the final stage.

---

7. Tell us something about Docker Compose.

Docker Compose is a tool for defining and running multi-container Docker applications using a YAML file. It allows you to:

- Define services, networks, and volumes
- Use `docker-compose up` to start the entire stack

Use case:
In development, Compose is used to spin up microservices like `web`, `api`, and `db` together with isolated networks.

---

8. What is Docker Swarm?

Docker Swarm is Docker's native clustering and orchestration tool. It turns a group of Docker hosts into a single virtual Docker host (a swarm).

Key features:

- High availability
- Load balancing
- Scaling services
- Rolling updates

Note:
While Swarm is simpler than Kubernetes, it's less feature-rich and is often used in smaller-scale setups.

---

9. What is a Docker Namespace?

Namespaces provide container isolation in Docker. They ensure that processes, network interfaces, file systems, etc., inside a container are isolated from the host and other containers.

Types:

- `pid`, `net`, `ipc`, `mnt`, `uts`, `user`

Production relevance:
This isolation ensures that containers do not interfere with one another, enabling multi-tenant deployments securely.

---

10. What is the Lifecycle of a Docker Container?

1. **Create** – Container is created using an image.
2. **Start** – Container is started and runs the defined command.
3. **Running** – Active state.
4. **Paused** – Temporarily halted.
5. **Stopped/Exited** – Container has finished execution or been stopped.
6. **Removed** – Deleted from the system.

Commands:

```
docker create / start / stop / pause / unpause / rm
```

---

11. What is Docker Machine?

Docker Machine was a tool to provision Docker hosts (VMs) on local or cloud environments. It created virtual machines with Docker pre-installed.

Status:
Deprecated in most real-world scenarios in favor of Docker Desktop or infrastructure as code tools like Terraform and cloud-native provisioning via ECS/EKS.

---

12. How to Check for Docker Client and Docker Server Version?

```
docker version
```

Sample Output:

```
Client: Docker Engine - Community
 Version:           24.0.6
...
Server: Docker Engine - Community
 Engine:
  Version:          24.0.6
```

Useful in interviews and troubleshooting version mismatches between client and daemon.

---

13. How Do You Get the Number of Containers Running, Paused, and Stopped?

```
docker info
```

Relevant Output:

```
Containers: 10
 Running: 3
 Paused: 1
 Stopped: 6
```

Also, you can filter with:

```
docker ps -a --filter "status=running"
```

---

14. If You Vaguely Remember a Command and Want Help, How Do You Get It?

Use Docker's built-in help system:

```
docker --help
docker container --help
docker run --help
```

Pro tip:
Use `man docker` for the manual page if available, or check `docker docs` online.

---

15. How to Login into Docker Repository?

```
docker login
```

Steps:

- Prompts for Docker Hub or private registry credentials
- Stores them locally in `~/.docker/config.json`

Important:
Use secrets management for credentials in CI/CD pipelines rather than storing them in plaintext.

Would you like me to continue with questions **16 to 30** next? I'll maintain the same clarity, format, and real-world insight to help you sound confident and production-ready.

Perfect, Pankaj. Here are the next set of **Docker interview questions (16 to 30)**, with detailed, professional-style answers that reflect real production knowledge and the tone of a DevOps engineer with 3 years of experience.

---

16. How do you create a Docker container from an image?

To create and start a container from a Docker image:

```
docker run -d --name my_container_name image_name
```

Example:

```
docker run -d --name nginx_web -p 80:80 nginx
```

Flags used:

- `-d`: Run container in detached mode
- `--name`: Assign a custom name
- `-p`: Map host port to container port

Production tip:
Always use `--restart unless-stopped` in production to ensure the container starts after reboot.

---

17. Once you've worked with an image, how do you push it to Docker Hub?

Steps:

1. **Tag the image:**

```
docker tag myapp:latest mydockerhubusername/myapp:1.0
```

2. **Login to Docker Hub:**

```
docker login
```

3. **Push the image:**

```
docker push mydockerhubusername/myapp:1.0
```

Note:
Use proper semantic versioning for image tags and avoid using `latest` in production CI/CD pipelines to ensure immutability.

---

18. How to build a Dockerfile?

Navigate to the directory containing your Dockerfile and run:

```
docker build -t myapp:1.0 .
```

- `-t`: Tags the image with a name and version
- `.`: Refers to the current directory (Dockerfile and app code)

Pro tip:
Add `.dockerignore` to exclude files like `.git`, `node_modules`, and temporary build files to reduce

image size.

---

19. Do you know why `docker system prune` is used? What does it do?

`docker system prune` is used to clean up unused Docker resources.

```
docker system prune -a
```

Removes:

- Stopped containers
- Unused images
- Dangling volumes
- Unused networks

Warning:
Use with caution in production as it may delete images not associated with running containers.

---

20. Will you lose your data when a Docker container exits?

Yes, data stored inside a container is ephemeral and lost when the container stops or is removed—unless volumes are used.

Solution:
Use **Docker volumes** to persist data:

```
docker run -v myvolume:/app/data myimage
```

Real-world usage:
Databases, logs, or any stateful services should write to mounted volumes for durability.

---

21. Where all do you think Docker is being used?

Docker is widely used across:

- **Development environments** – Ensuring consistency across teams
- **CI/CD pipelines** – Packaging and testing applications
- **Microservices architectures** – Isolated services, container-per-service
- **Hybrid cloud deployments** – Portability across AWS, Azure, GCP
- **Testing/QA** – Isolated test environments with versioned containers
- **Production environments** – Deployed directly or via orchestrators like Kubernetes

---

22. How is Docker different from other containerization methods?

Compared to traditional methods (like LXC or chroot):

- **Simplified tooling and developer UX**
- **Rich ecosystem (Docker Hub, Compose, Swarm)**
- **Better community support and integrations**
- **Cross-platform support (Linux, Windows, macOS)**

Real-world impact:
Docker standardized the container format (OCI), making it easier to integrate into modern CI/CD and orchestration systems.

---

23. Can I use JSON instead of YAML for my Compose file in Docker?

No, Docker Compose officially supports **YAML** (`.yml`) format only.

Reason:

- YAML offers better readability for complex configurations.
- While JSON is technically a subset of YAML, Compose parsers are optimized for `.yml`.

Tip:
Validate YAML files using `yamllint` or CI jobs to prevent indentation errors.

---

24. How have you used Docker in your previous position?

In my previous role, I used Docker extensively to containerize microservices written in Spring Boot and Node.js. Each microservice was defined with a separate Dockerfile and built using multi-stage builds to optimize image sizes.

I used Docker Compose for local development to orchestrate dependent services like Redis, PostgreSQL, and RabbitMQ. In CI/CD, Docker images were built in Jenkins pipelines, tested using container-based integration tests, and then pushed to a private registry (Harbor). The final deployment was handled through Kubernetes using Helm charts.

---

25. How far do Docker containers scale? Are there any requirements for the same?

Docker containers themselves are lightweight and can scale horizontally based on infrastructure capacity and orchestration logic.

To scale effectively:

- Use orchestration tools like **Kubernetes**, **Docker Swarm**, or **ECS**.
- Design stateless containers

- Use external services for config and secrets (e.g., AWS SSM or Vault)
- Monitor CPU/memory usage to auto-scale resources

Real-world scalability example:
Containers for web services can scale to hundreds of instances behind a load balancer, provided the infra supports it.

---

26. What platforms does Docker run on?

Docker runs on:

- **Linux distributions** (Ubuntu, CentOS, Debian, Amazon Linux)
- **macOS** (via Docker Desktop with a Linux VM)
- **Windows 10/11 Pro & Enterprise** (via WSL2 or Hyper-V backend)
- **Cloud platforms** (AWS, GCP, Azure via ECS, EKS, GKE, AKS)

Pro insight:
For CI pipelines, Linux-based containers are preferred due to lower overhead and faster boot time.

---

27. Is there a way to identify the status of a Docker container?

Yes, use the following commands:

```
docker ps               # Lists running containers
docker ps -a            # Lists all containers (running and stopped)
docker inspect <ID>     # Detailed info including status
docker container ls     # Alias for `docker ps`
```

You can also use:

```
docker container ls --filter "status=exited"
```

---

28. Can you remove a paused container from Docker?

No, a **paused** container must first be **unpaused or stopped** before it can be removed.

Commands:

```
docker unpause <container_id>
docker stop <container_id>
docker rm <container_id>
```

Reason:
Paused containers are in a frozen state (via `cgroups`), so they can't be safely deleted.

---

29. Suppose you have an application that has many dependent services. Will Docker Compose wait for the current container to be ready before running the next service?

No, Docker Compose does **not** wait for a container to be "ready"—it only ensures that containers **start** in the order defined (if `depends_on` is used), but not that the service inside is healthy.

Solution:

- Use **health checks** in `docker-compose.yml`
- Use tools like `wait-for-it.sh` or `dockerize` to block service start until dependencies are ready

```
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
```

---

30. How will you monitor Docker in production?

Monitoring strategies:

- Use `docker stats` for live container resource usage
- Aggregate logs via **ELK stack**, **Fluentd**, or **AWS CloudWatch**
- Use Prometheus + Grafana with **cAdvisor** or Docker Engine metrics
- Enable Docker daemon metrics (`--metrics-addr`)
- Use third-party tools like Datadog, Dynatrace, or New Relic

Production tip:
Don't rely on `docker stats` alone—set up proper monitoring pipelines and alerting thresholds for CPU/memory/network usage per container.

Let me know if you'd like to proceed with **questions 31–45** next. I'll continue in the same structured, professional format with production insights and confidence-ready phrasing.

Great! Here's the next set of **Docker interview questions (31 to 45)** with **well-structured, real-world answers** that reflect the level expected from a DevOps engineer with 3 years of hands-on experience in production environments.

---

31. Is it a good practice to run Docker Compose in production?

Not recommended in production environments.

While Docker Compose is excellent for **local development and testing**, it's not designed for:

- High availability
- Auto-scaling

- Health-based restarts
- Robust networking and service discovery

Instead, use:

- **Kubernetes**, **Docker Swarm**, or **AWS ECS** for production orchestration.

If Compose *must* be used (e.g., small internal apps):

- Use `docker-compose -f docker-compose.prod.yml up -d`
- Combine with reverse proxies, health checks, monitoring, and volumes.

---

32. What changes are expected in your Docker Compose file while moving it to production?

Here's how production `docker-compose` files differ from development:

Aspect	Dev	Prod
Image	`build: .`	`image: myrepo/app:1.0.0`
Volumes	Bind mounts	Named volumes or cloud block storage
Environment	`.env` file	Secrets manager or environment variables
Logging	Default (stdout)	Centralized logging drivers (e.g., `fluentd`, `gelf`)
Networks	Bridge (default)	Custom overlay or host networking
Scale	1 replica	Multiple replicas with orchestration
Restart policy	Not defined	`restart: always` or `unless-stopped`

Best Practice:
Split Compose files into base and override (e.g., `docker-compose.yml` and `docker-compose.override.yml`).

---

33. Explain the difference between Dockerfile and docker-compose.yml

Dockerfile	docker-compose.yml
Blueprint to build a **Docker image**	Blueprint to run **multiple containers** as a stack
Focuses on a single app/service	Defines how multiple containers interact (networks, volumes)
Executed with `docker build`	Executed with `docker-compose up`
Contains instructions like `FROM`, `RUN`, `CMD`	Contains services, volumes, networks, dependencies

Example:

- `Dockerfile` builds the Node.js app image
- `docker-compose.yml` runs Node.js + Redis + MongoDB services together

---

34. Describe the role of Docker Hub in containerization.

Docker Hub is Docker's default public registry that:

- Stores and distributes Docker images
- Hosts **official images** (e.g., `nginx`, `mysql`, `ubuntu`)
- Acts as a **collaboration hub** with teams and organizations
- Allows **versioned tagging** of images
- Can integrate with CI/CD to pull/push images

Real-world use:
Your Jenkins/GitHub Actions pipeline can push a newly built image to Docker Hub, from where Kubernetes or another environment can pull it.

---

35. How do you design a scalable Docker architecture?

Key design considerations:

- **Use orchestration:** Kubernetes, ECS, or Swarm
- **Design stateless containers** – offload state to external databases
- **Horizontal scaling:** via ReplicaSets or service scaling
- **Service discovery and networking:** Leverage internal DNS, load balancers
- **Persistent storage:** Use volumes or cloud block storage (EBS, Azure Disks)
- **CI/CD integration:** Automate image build, push, and deployment

Diagram example:

- Load Balancer (ALB)
  → Web containers (Nginx)
  → API containers (Spring Boot)
  → External RDS
  → S3 for static assets

---

36. Explain Docker's layered filesystem and its benefits.

Docker uses a **union filesystem** (e.g., OverlayFS) that layers image changes.

How it works:

- Base image → Intermediate layers (RUN, COPY, etc.) → Final image
- Only the **top writable layer** is modified when the container runs

Benefits:

- **Reusability:** Shared base images across containers
- **Efficiency:** Only changed layers need to be rebuilt or pushed
- **Faster CI/CD:** Cache intermediate layers
- **Smaller images:** Reduce redundancy

Best Practice:
Reorder Dockerfile instructions to **maximize cache usage**.

---

37. How do you optimize Docker image sizes?

✅ **Techniques:**

- Use **alpine-based** images (e.g., `python:3.10-alpine`)
- Use **multi-stage builds**:

```
FROM golang:alpine AS builder
...
FROM alpine
COPY --from=builder /app /app
```

- Minimize layers: Combine `RUN` commands with `&&`
- Use `.dockerignore`
- Remove unnecessary packages, temp files, cache (`rm -rf /var/lib/apt/lists/*`)

CI/CD note:
Always scan and validate image size before pushing to the registry.

---

38. Explain Docker's security features (e.g., SELinux, AppArmor).

Docker integrates with **Linux Security Modules (LSMs)** for container isolation:

- **SELinux** (Red Hat-based): Enforces mandatory access controls
- **AppArmor** (Ubuntu/Debian): Applies security profiles to containers
- **Seccomp**: Limits system calls (default seccomp profile blocks 44+ dangerous syscalls)
- **Capabilities**: Drops root-level permissions not needed (e.g., `NET_ADMIN`)
- **User namespaces**: Maps container UID to unprivileged host UID
- **Read-only filesystems** for immutable containers

---

39. How do you secure Docker containers from unauthorized access?

Security best practices:

- Run containers as **non-root** users
- Use **minimal base images**
- Apply **least privilege** (drop capabilities)
- Scan images for CVEs (e.g., Trivy, Clair)
- Use **network segmentation** (bridge/overlay networks)
- Configure **firewall rules** or **iptables**
- Regularly patch base images
- Use signed images with Docker Content Trust

---

40. Describe Docker's network security options.

Available options:

- **Bridge network:** Default, isolated, NATed traffic
- **Host network:** Shares host's network stack (less isolation)
- **Overlay network:** Cross-host networking (Swarm/Kubernetes)
- **Macvlan:** Assigns real MAC/IP from the host
- **Custom networks:** Better isolation, DNS resolution, aliases

Security enhancements:

- Control inter-container communication
- Enforce network-level policies (e.g., Cilium, Calico in K8s)
- Avoid exposing ports unnecessarily (`EXPOSE` vs `-p`)

---

41. How do you manage Docker image vulnerabilities?

✅ **Steps to manage image vulnerabilities:**

- Use vulnerability scanners:
  - **Trivy**
  - **Clair**
  - **Anchore Engine**
- Automate scans in CI/CD
- Track base image versions (don’t use `latest`)
- Apply CVE patches promptly
- Prefer official, minimal, or hardened images

- Set up alerts for high/critical CVEs
- Maintain an internal trusted image registry (e.g., Harbor)

---

42. Explain Docker Content Trust.

Docker Content Trust (DCT) enables **image signature verification** using **Notary**.

What it does:

- Ensures only **signed images** are pulled/pushed
- Prevents use of tampered or unverified images

How to enable:

```
export DOCKER_CONTENT_TRUST=1
```

Limitations:

- Not all registries support signed images
- Needs integration with secure signing keys

---

43. Explain Docker's networking modes (e.g., bridge, host).

Mode	Description
Bridge	Default, NAT-based, isolated from host
Host	Shares host's network stack, faster but less isolated
None	No network interface assigned
Overlay	Enables multi-host networking (Docker Swarm, K8s)
Macvlan	Assigns unique MAC/IP address (used in legacy systems)

---

44. How do you configure Docker networking for multiple containers?

Use Docker user-defined bridge networks:

```
docker network create my_network
docker run -d --name app --network my_network app_image
docker run -d --name db --network my_network db_image
```

Benefits:

- Containers can communicate using **container names** as hostnames
- Avoids hardcoded IPs
- Cleaner DNS resolution

Best Practice:
Use `docker-compose.yml` to define networks and assign aliases.

---

45. Describe Docker's DNS resolution mechanisms.

Docker provides **internal DNS resolution** for containers within the same network.

- Docker automatically sets up a DNS server inside containers
- Containers can resolve each other via **container name** or aliases
- Works well with user-defined bridge and overlay networks

Example:
In `docker-compose.yml`:

```
services:
  web:
    networks:
      - backend
  db:
    networks:
      - backend
networks:
  backend:
```

`web` can resolve `db` as hostname.

Let me know when you'd like to continue with **questions 46–60**, where we'll cover **port exposure, load balancing, debugging, logging, performance optimization, and Docker security in depth**.

Excellent. Here's the next batch: **Docker interview questions 46–60** with **clean, structured, and real-world answers**, aligned with what a confident DevOps engineer with 3 years of experience should convey during interviews.

---

46. How do you expose container ports to the host?

You can expose container ports using:

Method 1: Docker run

```
docker run -d -p 8080:80 nginx
```

> This maps port `80` inside the container to port `8080` on the host.

Method 2: Docker Compose

```yaml
services:
  web:
    image: nginx
    ports:
      - "8080:80"
```

✅ **Best Practices:**

- Only expose required ports.
- Avoid exposing DB ports externally.
- Use firewalls or security groups to limit access.

---

47. Explain Docker's load balancing options

Docker's native load balancing is **limited**. Depending on your stack:

In Docker Swarm:

- Uses **internal routing mesh**.
- All nodes in a service can accept requests; Swarm routes them to healthy containers.

In Kubernetes (recommended for production):

- Uses **Services (ClusterIP, NodePort, LoadBalancer)** and **Ingress controllers**.
- Integrate with **NGINX**, **HAProxy**, or **Envoy** for Layer 7 routing.

In simple Docker setups:

- Use an external reverse proxy (e.g., Nginx, Traefik) to load balance between multiple containers.

---

48. Explain Docker's service discovery mechanisms

Service discovery allows containers to **find and communicate** with each other by name.

Methods:

- **User-defined bridge networks**: Containers can resolve others using their names.
- **Docker Compose**: Automatically handles DNS between services.
- **Swarm mode**: Uses **internal DNS** and **service names**.
- **Third-party tools**: Consul, etcd, or Kubernetes' built-in DNS.

> **In production**, service discovery is typically handled by **Kubernetes CoreDNS** or **Istio**.

---

49. How do you deploy a Docker application using Kubernetes?

Steps:

1. **Write Dockerfile** → Build and push image to registry

2. **Create Kubernetes YAMLs:**
    - `Deployment` for app
    - `Service` for networking
    - `Ingress` (optional) for HTTP routing

```
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
```

3. **Verify:**

```
kubectl get pods
kubectl get svc
```

4. Use `kubectl port-forward` or Ingress to access the app.

✅ **Pro Tip:** Use Helm charts or Kustomize for reusable, templated deployments.

---

50. Describe Docker's rolling update strategy

In **Docker Swarm**, rolling updates are managed via the `docker service update` command.

```
docker service update --image app:v2 --update-parallelism 2 --update-delay 10s
my_service
```

It:

- Updates containers **in batches**
- Ensures **minimum downtime**
- Rolls back if health checks fail

In **Kubernetes**, rolling updates are handled via `Deployment` objects automatically.

---

51. Explain Docker's self-healing capabilities

Docker by itself has **basic self-healing** through restart policies:

```
docker run --restart unless-stopped my_container
```

This helps:

- Restart containers on failure
- Auto-restart on host reboot

For true self-healing in production:

- Use **Docker Swarm** (`--replicas`, reschedules failed containers)

- Use **Kubernetes**, which has health checks and automatic pod rescheduling

---

52. How do you debug Docker container issues

✅ **Step-by-step debugging:**

1. **Check logs:**

   ```
   docker logs <container_id>
   ```

2. **Inspect container:**

   ```
   docker inspect <container_id>
   ```

3. **Enter the container:**

   ```
   docker exec -it <container_id> /bin/bash
   ```

4. **Check health status and events:**

   ```
   docker ps --format '{{.Names}} {{.Status}}'
   docker events
   ```

5. **Network & volume checks:**
   - Use `docker network inspect`, `docker volume inspect`

👉 **Common issues:** DNS failure, missing ENV vars, permission issues, incorrect CMD/ENTRYPOINT.

---

53. Explain Docker's logging mechanisms

Docker provides **multiple logging drivers**:

- **default:** `json-file` (stored in `/var/lib/docker/containers/<id>/`)
- **others:** `syslog`, `journald`, `fluentd`, `gelf`, `awslogs`, `splunk`, etc.

How to set logging driver:

```
docker run --log-driver=syslog my_container
```

Production Practice:

Use centralized logging via:

- ELK stack (Elasticsearch, Logstash, Kibana)
- Fluentd + CloudWatch or Loki + Grafana
- Avoid `json-file` for long-term logs

---

54. Describe Docker's monitoring tools (e.g., Docker Stats)

✅ **Docker built-in:**

```
docker stats
```

Shows real-time CPU, memory, I/O per container.

✅ **Production-grade monitoring tools:**

- **cAdvisor**: Container resource metrics
- **Prometheus + Grafana**: Time-series monitoring
- **Datadog, New Relic, Sysdig**: Full-stack monitoring

Use exporters (like node-exporter, cadvisor) for custom metrics in Kubernetes.

---

55. How do you troubleshoot Docker network connectivity?

1. **Ping between containers:**

   ```
   docker exec -it app ping db
   ```

2. **Inspect network:**

   ```
   docker network inspect my_bridge
   ```

3. **Verify ports are exposed:**

   ```
   docker ps
   ```

4. **Check DNS:**
   - Use `/etc/resolv.conf` in container
   - Confirm container can resolve names

5. **Try curl/telnet inside container:**

   ```
   docker exec -it app curl http://db:3306
   ```

> Use `--network` flag to attach containers to custom networks.

---

56. Explain Docker's error handling mechanisms

Docker doesn't have native exception handling but provides:

- **Exit codes** (check with `docker inspect`)
- **Restart policies**
- **Health checks** (for orchestrators to act on failures)

In CI/CD pipelines:

- Handle failures using `set -e` in shell scripts
- Integrate with tools (e.g., Jenkins/GitHub Actions) to catch `docker build` or `run` errors

---

57. Explain Docker's best practices for image creation

✅ **Best Practices:**

- Use **official base images** (or minimal images like Alpine)
- Minimize layers (combine `RUN` commands)
- Leverage `.dockerignore`
- Avoid installing unnecessary packages
- Use **multi-stage builds**
- Specify `ENTRYPOINT` and `CMD` properly
- Tag images with version (`v1.2.3`) instead of `latest`

---

58. Describe Docker's guidelines for container management

- Use **non-root users**
- Mount **volumes** for persistence
- Set **resource limits** (`--memory`, `--cpus`)
- Clean up unused containers/images (`docker system prune`)
- Use **labels** for metadata and organization
- Avoid long-running containers doing too many things → follow **one process per container**

---

59. How do you optimize Docker container performance?

✅ **Tips:**

- Use lightweight base images
- Reduce image size to decrease container startup time
- Use appropriate resource limits (CPU/memory)
- Precompile assets (Java, Node.js) in multi-stage builds
- Minimize log volume (avoid `print` spam)
- Pin dependencies and OS packages

Also, monitor using `docker stats`, Prometheus, and analyze bottlenecks.

---

60. Explain Docker's recommendations for security

- Avoid running containers as **root**
- Use **read-only root filesystem**

- Use **trusted base images**
- Scan images (Trivy, Clair)
- Apply **resource limits**
- Drop unneeded Linux capabilities (`--cap-drop`)
- Enable Docker Content Trust
- Store secrets securely (not in ENV or Dockerfile)
- Restrict host access via seccomp, AppArmor/SELinux

Let me know when you're ready for the **final batch (61–81)** — we'll cover **scalability, daemon.json, resource management, network drivers, Swarm vs Kubernetes, and best practices for container security, discovery, and performance**.

Shall we continue?

Great! Let's complete the **final batch (Docker Questions 61–81)** — these focus on **scalable architecture, advanced networking, security, service discovery, deployment with Kubernetes, and real-world best practices**. Answers are tailored for a **3-year DevOps role** with production context.

---

61. Describe Docker's advice for scalable architecture

✅ **Scalability in Docker depends on:**

- **Using orchestrators** like Kubernetes or Docker Swarm
- Designing **stateless containers**
- Externalizing state (e.g., using AWS RDS, S3)
- Using **Docker Compose** for local simulation, not production
- Implementing **load balancing** via Ingress, Nginx, or cloud LBs
- Using **auto-scaling** features in orchestrators (e.g., HPA in K8s)

> In production, **Kubernetes + CI/CD pipelines + GitOps** is the preferred scalable setup.

---

62. How do you configure Docker to use a custom registry?

To use a private/custom registry:

Login:

```
docker login myregistry.example.com
```

Tag and push:

```
docker tag myapp myregistry.example.com/myapp:v1
docker push myregistry.example.com/myapp:v1
```

Configure Docker daemon (optional):

Edit `/etc/docker/daemon.json`:

```
{
  "insecure-registries": ["myregistry.example.com"]
}
```

> For secure registries, ensure **TLS certificates** are configured correctly.

---

63. Explain Docker's configuration file (`daemon.json`)

Located at:

- 📍 **Linux:** `/etc/docker/daemon.json`
- 📍 **Windows:** `%programdata%\docker\config\daemon.json`

Common fields:

```
{
  "log-driver": "json-file",
  "insecure-registries": ["myregistry.local"],
  "registry-mirrors": ["https://mirror.gcr.io"],
  "default-address-pools": [
    {
      "base": "10.10.0.0/16",
      "size": 24
    }
  ]
}
```

🧠 **Tip:** Restart Docker after changes:

```
sudo systemctl restart docker
```

---

64. How do you manage Docker container logs?

Check logs:

```
docker logs <container_id>
```

Best practices:

- Use logging drivers (e.g., `fluentd`, `syslog`, `awslogs`)
- Mount log directories using `volumes`
- Ship logs to ELK, Loki, or CloudWatch for centralized logging

In Kubernetes: Use **sidecar log collectors** (e.g., Fluent Bit or Filebeat).

---

65. Describe Docker's resource management capabilities

Docker supports:

- **Memory limits** (`--memory`)
- **CPU limits** (`--cpus`, `--cpuset-cpus`)
- **Block IO weight** (`--blkio-weight`)
- **PIDs limit** (`--pids-limit`)

> These are enforced using **Linux cgroups**.

Helps prevent noisy neighbor issues and ensures **fair resource sharing** in production.

---

66. How do you configure Docker's CPU and memory limits?

```
docker run --memory="512m" --cpus="1.5" my_app
```

Docker Compose example:

```yaml
services:
  app:
    image: my_app
    deploy:
      resources:
        limits:
          memory: 512M
          cpus: '0.5'
```

> Kubernetes provides more advanced resource requests/limits control via YAML.

---

67. Explain Docker's networking model

Docker has multiple networking types:

1. **Bridge** (default): NATed access to containers on the same host
2. **Host**: Shares host's network stack
3. **None**: No networking (isolated)
4. **Overlay** (Swarm/Kubernetes): Multi-host networking
5. **Macvlan**: Assigns a MAC address to containers

Each container has a virtual eth interface (`eth0`) and connects via veth pairs.

---

68. How do you create a Docker network?

```
docker network create --driver bridge my_network
```

To use it:

```
docker run -d --network my_network my_app
```

> Helps containers **communicate via DNS names** and avoid exposing all ports.

---

69. Describe Docker's network drivers

Driver	Purpose
`bridge`	Default, isolated containers on same host
`host`	Shares host network, no isolation
`overlay`	Multi-host networking (Swarm, Kubernetes)
`macvlan`	Assigns MAC address, useful for legacy networks
`none`	No network access

> Use `overlay` or CNI plugins like **Calico/Flannel** in K8s for advanced networking.

---

70. How do you configure Docker's DNS resolution?

Docker uses its internal **DNS server (127.0.0.11)** when using user-defined networks.

You can override DNS:

```
docker run --dns 8.8.8.8 my_container
```

> Containers can resolve other services by name when on the same network.

In Kubernetes: Use **CoreDNS**.

---

71. Explain Docker's load balancing options

✅ **In Docker Swarm:**

- Uses internal **routing mesh** to distribute requests

✅ **In Kubernetes:**

- Uses **Services** (ClusterIP, NodePort, LoadBalancer)
- Advanced routing with **Ingress Controllers (NGINX, Traefik)**

✅ **Externally:**

- Use **HAProxy, NGINX, or cloud LBs**

> Docker alone has limited LB features. Real-world setups use orchestration + external tools.

---

72. Explain Docker's security features

- **Namespaces** for isolation
- **Control groups (cgroups)** for resource limits
- **Seccomp** for system call filtering
- **AppArmor/SELinux** for access control
- **Docker Content Trust (DCT)** for image signing
- **Rootless containers** support
- **User namespaces** to map root inside container to non-root outside

---

73. How do you secure Docker containers?

✅ **Best Practices:**

- Run as **non-root user**
- Use **minimal base images** (Alpine, distroless)
- Set `--read-only` file system
- Drop Linux capabilities with `--cap-drop`
- Scan images (Trivy, Clair)
- Use **secrets management** (not ENV vars)
- Enable **DCT** and use signed images

---

74. Describe Docker's network security options

- Use **firewall rules** (iptables, security groups)
- Avoid `--network=host` unless necessary
- Use **isolated user-defined networks**
- Configure **TLS for Docker API**
- Use **macvlan** for segmented network zones
- Employ **service mesh** or overlay networks with encryption

---

75. How do you manage Docker image vulnerabilities?

✅ Tools:

- **Trivy**
- **Clair**

- **Grype**
- **Snyk**
- **Docker Hub vulnerability scanner**

Example:

```
trivy image myapp:latest
```

> Integrate scanners into CI/CD pipeline to fail builds on vulnerabilities.

---

76. Explain Docker Content Trust (DCT)

DCT ensures only **signed images** are pulled/pushed.

```
export DOCKER_CONTENT_TRUST=1
```

- Uses **Notary** to sign images
- Ensures **authenticity and integrity**

> Useful in regulated environments (finance, healthcare) for secure delivery.

---

77. Compare Docker Swarm with Kubernetes

Feature	Docker Swarm	Kubernetes
Setup	Simple	Complex
Scaling	Easy	Very granular
Service Discovery	Built-in	CoreDNS, kube-proxy
Load Balancing	Basic (Routing Mesh)	Advanced (Services + Ingress)
Ecosystem	Limited	Large community and tooling
Production Use	Rare	Industry Standard

✅ **Pro Tip:** Use Swarm for demos, **K8s for real-world production.**

---

78. Explain Docker's service discovery mechanisms

Covered in Q48 — to summarize:

- **User-defined bridge** or **overlay networks** provide DNS
- Use **service names** instead of IPs
- In Swarm/K8s: service discovery is **built-in**

---

79. How do you deploy a Docker application using Kubernetes?

Already covered in Q49. Summary:

1. Build & push image
2. Write Kubernetes YAML (Deployment + Service)
3. Apply using `kubectl`
4. Verify pods and service exposure

Use **Helm** for templated deployments.

---

80. Describe Docker's rolling update strategy

Same as Q50. Summary:

- In **Swarm**: Use `docker service update` with flags
- In **Kubernetes**: `Deployment` handles rolling updates
- **Health checks** ensure safe rollout
- Rollbacks are automatic if failures are detected

---

81. Explain Docker's self-healing capabilities

Same as Q51. Recap:

- **Restart policies** for basic recovery
- **Swarm** & **Kubernetes** reschedule failed containers
- **Health checks** + orchestration = true self-healing

✅ You're now equipped with **full professional answers** for **81 Docker interview questions** — structured, concise, and aligned with **real-world DevOps expectations**.