50+ Most Asked Docker Interview Questions & Answers

Basic Docker Questions

1. What is Docker?

Answer: Docker is an open-source containerization platform that enables developers to package applications and their dependencies into lightweight, portable containers. It uses OS-level virtualization to deliver software in packages called containers.

Example:

```
# Basic Docker container
docker run hello-world

# Run Ubuntu container interactively
docker run -it ubuntu:20.04 /bin/bash
```

2. What is the difference between Docker and Virtual Machines?

Answer: Docker containers share the host OS kernel and are lightweight, while VMs have their own complete OS. Containers start faster, use fewer resources, and provide better portability.

Example:

```
# Docker container (lightweight)
docker run -d nginx:alpine    # Starts in seconds, ~5MB

# VM would require:
# - Full OS installation (~GB)
# - Hypervisor overhead
# - Minutes to boot
```

3. What is a Docker Image?

Answer: A Docker image is a read-only template containing application code, runtime, system tools, libraries, and dependencies needed to run an application. Images are used to create containers.

Example:

```
# List Docker images
docker images

# Pull an image from Docker Hub
docker pull nginx:1.21-alpine

# Build custom image
docker build -t myapp:v1.0 .
```

4. What is a Docker Container?

Answer: A Docker container is a runnable instance of a Docker image. It's an isolated process that includes the application and all its dependencies, providing a consistent runtime environment.

Example:

```
# Create and start container
docker run -d --name web-server -p 8080:80 nginx

# List running containers
docker ps

# Stop container
docker stop web-server
```

5. What is a Dockerfile?

Answer: A Dockerfile is a text file containing a series of instructions to build a Docker image automatically. It defines the base image, dependencies, configuration, and commands needed for the application.

Example:

```
FROM node:16-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
```

```
EXPOSE 3000
CMD ["npm", "start"]
```

6. Explain the difference between COPY and ADD in Dockerfile?

Answer: Both copy files from host to container, but ADD has additional features like extracting tar files and downloading from URLs. COPY is preferred for simple file copying as it's more explicit.

Example:

```
# COPY - simple file copying
COPY package.json /app/
COPY src/ /app/src/

# ADD - additional features
ADD app.tar.gz /app/          # Extracts tar file
ADD http://example.com/file.txt /app/  # Downloads from URL
```

7. What is the difference between CMD and RUN in Dockerfile?

Answer: RUN executes commands during image build and creates new image layers. CMD specifies the default command to execute when container starts and can be overridden.

Example:

```
# RUN - executes during build
RUN apt-get update && apt-get install -y curl
RUN npm install

# CMD - executes when container starts
CMD ["node", "server.js"]
CMD ["nginx", "-g", "daemon off;"]
```

8. What is Docker Hub?

Answer: Docker Hub is a cloud-based registry service for sharing Docker images. It's the default public registry where users can store, distribute, and collaborate on Docker images.

Example:

```
# Login to Docker Hub
docker login

# Push image to Docker Hub
docker tag myapp:v1.0 username/myapp:v1.0
docker push username/myapp:v1.0
```

```
# Pull from Docker Hub
docker pull username/myapp:v1.0
```

9. How do you create and run a Docker container?

Answer: Use `docker run` command to create and start a container from an image. You can specify various options like port mapping, volume mounting, and environment variables.

Example:

```
# Basic container creation
docker run nginx

# Advanced container with options
docker run -d \
  --name my-web-app \
  -p 8080:80 \
  -v /host/data:/app/data \
  -e NODE_ENV=production \
  my-web-app:latest
```

10. What are Docker Volumes?

Answer: Docker volumes are a mechanism for persisting data generated and used by Docker containers. They exist outside the container's filesystem and survive container deletion.

Example:

```
# Create named volume
docker volume create my-data

# Use volume in container
docker run -d -v my-data:/app/data nginx

# Bind mount (host directory)
docker run -d -v /host/path:/container/path nginx

# List volumes
docker volume ls
```

Intermediate Docker Questions

11. What is the difference between ENTRYPOINT and CMD?

Answer: ENTRYPOINT defines the executable that will always run when container starts and cannot be overridden. CMD provides default arguments that can be overridden from command line.

Example:

```
# ENTRYPOINT - cannot be overridden
FROM ubuntu
ENTRYPOINT ["echo", "Hello"]
CMD ["World"]

# docker run myimage          -> "Hello World"
# docker run myimage Docker   -> "Hello Docker"

# Only ENTRYPOINT
ENTRYPOINT ["nginx", "-g", "daemon off;"]
```

12. What are Docker Networks?

Answer: Docker networks provide communication between containers. Docker supports multiple network drivers including bridge, host, overlay, and none for different networking scenarios.

Example:

```
# List networks
docker network ls

# Create custom bridge network
docker network create my-network

# Run containers on custom network
docker run -d --name app1 --network my-network nginx
docker run -d --name app2 --network my-network redis

# Inspect network
docker network inspect my-network
```

13. How do you optimize Docker images for size?

Answer: Use multi-stage builds, minimal base images (Alpine), combine RUN commands, remove unnecessary files, and use .dockerignore to exclude unwanted files.

Example:

```
# Multi-stage build example
FROM node:16 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

14. What is Docker Compose?

Answer: Docker Compose is a tool for defining and running multi-container Docker applications using a YAML file. It simplifies the management of complex applications with multiple services.

Example:

```
# docker-compose.yml
version: '3.8'
services:
  web:
    build: .
    ports:
      - "8000:8000"
    depends_on:
      - db
    environment:
      - DATABASE_URL=postgresql://user:pass@db:5432/mydb

  db:
    image: postgres:13
    environment:
      - POSTGRES_DB=mydb
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=pass
    volumes:
      - postgres_data:/var/lib/postgresql/data

volumes:
  postgres_data:
```

15. How do you handle environment variables in Docker?

Answer: Environment variables can be set using the -e flag, --env-file option, or ENV instruction in Dockerfile. They provide configuration flexibility across different environments.

Example:

```
# Using -e flag
docker run -e NODE_ENV=production -e PORT=3000 myapp

# Using env file
echo "NODE_ENV=production" > .env
echo "PORT=3000" >> .env
docker run --env-file .env myapp
```

```
# In Dockerfile
ENV NODE_ENV=production
ENV PORT=3000
```

16. What are bind mounts vs volumes?

Answer: Bind mounts map a host directory to container directory, while volumes are managed by Docker. Volumes are preferred for data persistence and are more portable.

Example:

```
# Bind mount
docker run -v /host/path:/container/path nginx

# Named volume
docker run -v myvolume:/container/path nginx

# Anonymous volume
docker run -v /container/path nginx
```

17. How do you expose ports in Docker?

Answer: Use EXPOSE instruction in Dockerfile to document ports and -p flag with docker run to publish ports, mapping container ports to host ports.

Example:

```
# Dockerfile
EXPOSE 3000 8080
```

```
# Publish specific ports
docker run -p 8080:3000 myapp

# Publish all exposed ports
docker run -P myapp

# Multiple port mapping
docker run -p 8080:3000 -p 9090:8080 myapp
```

18. What is Docker layer caching?

Answer: Docker uses layer caching to speed up builds by reusing unchanged layers. Each Dockerfile instruction creates a layer, and unchanged layers are reused from cache.

Example:

```
# Optimized for caching
FROM node:16-alpine
WORKDIR /app

# Copy package files first (changes less frequently)
COPY package*.json ./
RUN npm install

# Copy source code last (changes more frequently)
COPY . .
RUN npm run build

EXPOSE 3000
CMD ["npm", "start"]
```

19. How do you debug a Docker container?

Answer: Use docker logs for output, docker exec to access running container, docker inspect for detailed information, and docker stats for resource usage monitoring.

Example:

```
# View container logs
docker logs container-name
docker logs -f container-name  # Follow logs

# Execute commands in running container
docker exec -it container-name /bin/bash
docker exec container-name ls -la /app

# Inspect container details
docker inspect container-name
```

```
# Monitor resource usage
docker stats container-name
```

20. What is the difference between docker run and docker start?

Answer: `docker run` creates a new container from an image and starts it. `docker start` starts an existing stopped container without creating a new one.

Example:

```
# Create and start new container
docker run -d --name web nginx

# Stop the container
docker stop web

# Start existing container
docker start web

# Restart container
docker restart web
```

Advanced Docker Questions

21. What are multi-stage builds in Docker?

Answer: Multi-stage builds allow using multiple FROM statements in a Dockerfile to create smaller, more secure images by copying only necessary artifacts from build stages.

Example:

```
# Build stage
FROM golang:1.19-alpine AS builder
WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -o main .

# Production stage
FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /app/main .
EXPOSE 8080
CMD ["./main"]
```

22. How does Docker ensure container isolation?

Answer: Docker uses Linux namespaces for process isolation, cgroups for resource limiting, and security features like capabilities, seccomp, and AppArmor for security isolation.

Example:

```
# Check namespaces
docker run --rm -it ubuntu unshare --help

# Limit container resources
docker run -d --name limited \
  --memory=512m \
  --cpus="1.5" \
  --pids-limit=100 \
  nginx

# Run with restricted capabilities
docker run --rm --cap-drop=ALL --cap-add=NET_BIND_SERVICE nginx
```

23. What are Docker secrets and how are they managed?

Answer: Docker secrets are encrypted data stored in Docker Swarm cluster and made available to services. They provide secure way to manage sensitive information like passwords and API keys.

Example:

```
# Create secret
echo "mypassword" | docker secret create db_password -

# Use secret in service
docker service create \
  --name web \
  --secret db_password \
  --replicas 3 \
  nginx

# Secret available at /run/secrets/db_password in container
```

24. How do you implement health checks in Docker?

Answer: Use HEALTHCHECK instruction in Dockerfile or --health-cmd flag to define container health status. Docker periodically runs health checks and marks containers as healthy or unhealthy.

Example:

```
# Dockerfile health check
FROM nginx
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
  CMD curl -f http://localhost/ || exit 1
```

```
# Command line health check
docker run -d \
  --name web \
  --health-cmd="curl -f http://localhost/ || exit 1" \
  --health-interval=30s \
  --health-retries=3 \
  --health-timeout=3s \
  nginx
```

25. What is Docker Swarm?

Answer: Docker Swarm is Docker's native clustering and orchestration solution that turns multiple Docker hosts into a single virtual Docker host for high availability and scaling.

Example:

```
# Initialize swarm
docker swarm init --advertise-addr 192.168.1.100

# Join worker node
docker swarm join --token <worker-token> 192.168.1.100:2377

# Deploy service
docker service create \
  --name web \
  --replicas 3 \
  --publish 8080:80 \
  nginx

# Scale service
docker service scale web=5
```

26. How do you secure Docker containers?

Answer: Implement security best practices including running as non-root user, using minimal base images, scanning for vulnerabilities, limiting capabilities, and using secrets management.

Example:

```
# Security best practices
FROM node:16-alpine

# Create non-root user
```

```
RUN addgroup -g 1001 -S nodejs
RUN adduser -S nodejs -u 1001

# Set working directory
WORKDIR /app

# Copy and install dependencies
COPY package*.json ./
RUN npm ci --only=production

# Copy application code
COPY . .

# Change ownership to non-root user
RUN chown -R nodejs:nodejs /app
USER nodejs

# Expose port
EXPOSE 3000

# Start application
CMD ["node", "server.js"]
```

27. What are Docker registries and how do you set up a private registry?

Answer: Docker registries store and distribute Docker images. You can set up private registries for internal use, better security, and control over image distribution.

Example:

```
# Run private registry
docker run -d \
  -p 5000:5000 \
  --name registry \
  -v /opt/registry:/var/lib/registry \
  registry:2

# Tag and push to private registry
docker tag myapp:latest localhost:5000/myapp:latest
docker push localhost:5000/myapp:latest

# Pull from private registry
docker pull localhost:5000/myapp:latest
```

28. How do you monitor Docker containers in production?

Answer: Use monitoring tools like Prometheus, Grafana, ELK stack, or commercial solutions. Monitor metrics like CPU, memory, network, disk usage, and application-specific metrics.

Example:

```
# docker-compose.yml for monitoring stack
version: '3.8'
services:
  prometheus:
    image: prom/prometheus
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml

  grafana:
    image: grafana/grafana
    ports:
      - "3000:3000"
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin

  cadvisor:
    image: gcr.io/cadvisor/cadvisor
    ports:
      - "8080:8080"
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:rw
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro
```

29. What is the difference between Docker Swarm and Kubernetes?

Answer: Docker Swarm is simpler and easier to set up, ideal for smaller deployments. Kubernetes is more complex but provides advanced features like auto-scaling, rolling updates, and extensive ecosystem.

Example:

```
# Docker Swarm service
docker service create \
  --name web \
  --replicas 3 \
  --publish 80:80 \
  nginx

# Kubernetes deployment
kubectl create deployment web --image=nginx --replicas=3
kubectl expose deployment web --port=80 --type=LoadBalancer
```

30. How do you handle container orchestration?

Answer: Container orchestration manages container lifecycle, scaling, networking, and service discovery. Use tools like Docker Swarm, Kubernetes, or cloud-managed services for production deployments.

Example:

```
# Kubernetes deployment example
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web-app
  template:
    metadata:
      labels:
        app: web-app
    spec:
      containers:
      - name: web
        image: nginx:1.21
        ports:
        - containerPort: 80
        resources:
          requests:
            memory: "64Mi"
            cpu: "250m"
          limits:
            memory: "128Mi"
            cpu: "500m"
```

Scenario-Based Docker Questions

31. How would you troubleshoot a container that keeps restarting?

Answer: Check container logs, examine exit codes, verify resource limits, check health checks, and review application configuration. Common causes include insufficient memory, failed health checks, or application errors.

Example:

```
# Check container status and restart count
docker ps -a
```

```
# Examine logs for errors
docker logs container-name
docker logs --since=1h container-name

# Check resource usage
docker stats container-name

# Inspect container configuration
docker inspect container-name

# Check system resources
docker system df
docker system events
```

32. How do you implement zero-downtime deployment with Docker?

Answer: Use rolling updates, blue-green deployment, or canary deployments. Implement health checks, load balancers, and proper orchestration to ensure seamless updates.

Example:

```
# Blue-Green deployment with Docker Compose
# docker-compose.blue.yml
version: '3.8'
services:
  web-blue:
    image: myapp:v1.0
    ports:
      - "8001:80"

# docker-compose.green.yml
version: '3.8'
services:
  web-green:
    image: myapp:v2.0
    ports:
      - "8002:80"

# Switch traffic using nginx or load balancer
# Update upstream servers from blue to green
```

33. How would you handle persistent data in containerized applications?

Answer: Use Docker volumes, bind mounts, or external storage systems. Implement backup strategies, use StatefulSets in Kubernetes, and consider data replication for high availability.

Example:

```
# Docker Compose with persistent volumes
version: '3.8'
services:
  db:
    image: postgres:13
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./backups:/backups
    environment:
      - POSTGRES_DB=myapp
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=password

  backup:
    image: postgres:13
    depends_on:
      - db
    volumes:
      - ./backups:/backups
    command: |
      bash -c '
      while true; do
        pg_dump -h db -U user myapp > /backups/backup_$$(date +%Y%m%d_%H%M%S).sql
        sleep 86400
      done'

volumes:
  postgres_data:
```

34. How do you implement service discovery in Docker?

Answer: Use Docker networks for basic discovery, implement service mesh solutions, use external service discovery tools like Consul, or leverage orchestration platform features.

Example:

```
# Docker Compose service discovery
version: '3.8'
services:
  web:
    build: .
    depends_on:
      - api
      - cache
    environment:
      - API_URL=http://api:3000
      - CACHE_URL=redis://cache:6379

  api:
    image: myapi:latest
    ports:
      - "3000:3000"
    depends_on:
      - db
    environment:
      - DB_HOST=db
      - DB_PORT=5432

  cache:
    image: redis:alpine
    ports:
      - "6379:6379"

  db:
    image: postgres:13
    environment:
      - POSTGRES_DB=myapp
```

35. How would you optimize Docker build performance?

Answer: Use layer caching, multi-stage builds, .dockerignore, parallel builds, and BuildKit features. Optimize Dockerfile instruction order and minimize layer size.

Example:

```
# syntax=docker/dockerfile:1
FROM node:16-alpine AS base
WORKDIR /app

# Dependency stage
FROM base AS deps
COPY package*.json ./
RUN --mount=type=cache,target=/root/.npm \
    npm ci --omit=dev

# Build stage
FROM base AS build
COPY package*.json ./
RUN --mount=type=cache,target=/root/.npm \
    npm ci
COPY . .
RUN npm run build

# Production stage
FROM base AS production
COPY --from=deps /app/node_modules ./node_modules
COPY --from=build /app/dist ./dist
COPY package*.json ./
EXPOSE 3000
CMD ["npm", "start"]
```

```
# Use BuildKit for advanced features
DOCKER_BUILDKIT=1 docker build --target production -t myapp .
```

```
# Build with cache mount
docker build --build-arg BUILDKIT_INLINE_CACHE=1 .
```

36. How do you handle secrets and sensitive data in Docker?

Answer: Never store secrets in images or environment variables. Use Docker secrets, external secret management systems, init containers, or sidecar patterns for secure secret injection.

Example:

```
# Using external secret management
docker run -d \
  --name myapp \
  -v /etc/ssl/certs:/etc/ssl/certs:ro \
  --env-file <(vault kv get -format=json secret/myapp | jq -r '.data.data | to_entries[]
  myapp:latest
```

```
# Multi-stage with secret mount
# syntax=docker/dockerfile:1
FROM alpine AS base
RUN --mount=type=secret,id=api_key \
    API_KEY=$(cat /run/secrets/api_key) && \
    # Use API_KEY for build process
    echo "Building with secret..."

FROM base AS final
# Secret not included in final image
COPY app .
CMD ["./app"]
```

37. How would you implement container logging strategy?

Answer: Configure logging drivers, use centralized logging systems, implement log rotation, structure logs for parsing, and monitor log volumes and performance.

Example:

```
# Configure logging driver
docker run -d \
  --name myapp \
  --log-driver=json-file \
  --log-opt max-size=10m \
  --log-opt max-file=3 \
  myapp:latest

# Use syslog driver
docker run -d \
  --log-driver=syslog \
  --log-opt syslog-address=tcp://log.example.com:514 \
  myapp:latest
```

```
# Docker Compose with ELK stack
version: '3.8'
services:
  app:
    image: myapp:latest
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"
    depends_on:
      - elasticsearch

  elasticsearch:
    image: elasticsearch:7.14.0
    environment:
      - discovery.type=single-node
    ports:
      - "9200:9200"

  logstash:
    image: logstash:7.14.0
    volumes:
      - ./logstash.conf:/usr/share/logstash/pipeline/logstash.conf

  kibana:
    image: kibana:7.14.0
    ports:
      - "5601:5601"
    depends_on:
      - elasticsearch
```

38. How do you implement auto-scaling for Docker containers?

Answer: Use orchestration platforms like Kubernetes HPA, Docker Swarm scaling, or cloud-specific auto-scaling. Monitor metrics and define scaling policies based on CPU, memory, or custom metrics.

Example:

```
# Kubernetes Horizontal Pod Autoscaler
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: web-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: web-app
  minReplicas: 3
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80
```

39. How would you migrate a legacy application to Docker?

Answer: Analyze dependencies, create Dockerfile incrementally, containerize components separately, implement data migration strategy, and plan rollback procedures.

Example:

```
# Legacy application containerization
FROM ubuntu:20.04

# Install runtime dependencies
RUN apt-get update && apt-get install -y \
    python3 \
    python3-pip \
    apache2 \
    libapache2-mod-wsgi-py3 \
    && rm -rf /var/lib/apt/lists/*

# Copy legacy application
COPY legacy-app/ /var/www/html/
COPY requirements.txt /tmp/
RUN pip3 install -r /tmp/requirements.txt

# Configure Apache
COPY apache-config.conf /etc/apache2/sites-available/000-default.conf
RUN a2enmod wsgi

# Start services
COPY start.sh /start.sh
RUN chmod +x /start.sh
CMD ["/start.sh"]
```

40. How do you handle database migrations in containerized applications?

Answer: Use init containers, migration services, or application startup scripts. Implement idempotent migrations, version control, and rollback strategies.

Example:

```yaml
# Kubernetes Job for database migration
apiVersion: batch/v1
kind: Job
metadata:
  name: db-migration
spec:
  template:
    spec:
      initContainers:
      - name: wait-for-db
        image: postgres:13
        command: ['sh', '-c', 'until pg_isready -h db-service -p 5432; do sleep 1; done']

      containers:
      - name: migrate
        image: myapp:latest
        command: ["python", "manage.py", "migrate"]
        env:
        - name: DATABASE_URL
          value: "postgresql://user:pass@db-service:5432/mydb"

      restartPolicy: OnFailure
```

Performance and Security Questions

41. How do you optimize Docker container performance?

Answer: Minimize image size, use appropriate base images, optimize resource allocation, implement proper caching, and monitor container metrics.

Example:

```bash
# Resource constraints
docker run -d \
  --name optimized-app \
  --memory=512m \
  --cpus="1.0" \
  --memory-swap=1g \
  --oom-kill-disable=false \
  myapp:latest

# Monitor performance
```

```
docker stats optimized-app

# Use Alpine for smaller images
FROM node:16-alpine  # ~110MB vs node:16 ~900MB
```

42. What are Docker security best practices?

Answer: Run containers as non-root, use minimal base images, scan for vulnerabilities, limit capabilities, use secrets management, and implement network segmentation.

Example:

```
# Security hardened Dockerfile
FROM node:16-alpine

# Create non-root user
RUN addgroup -g 1001 -S nodejs \
    && adduser -S nodejs -u 1001

# Install dependencies as root
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production && npm cache clean --force

# Copy app and change ownership
COPY --chown=nodejs:nodejs . .

# Switch to non-root user
USER nodejs

# Drop capabilities and run with security options
EXPOSE 3000
CMD ["node", "server.js"]
```

```
# Run with security options
docker run -d \
  --name secure-app \
  --user 1001:1001 \
  --read-only \
  --tmpfs /tmp \
  --cap-drop=ALL \
  --cap-add=NET_BIND_SERVICE \
  --security-opt=no-new-privileges:true \
  myapp:latest
```

43. How do you implement container image scanning?

Answer: Use tools like Trivy, Clair, Anchore, or cloud-native scanners to detect vulnerabilities in container images. Integrate scanning into CI/CD pipelines.

Example:

```
# Trivy scanning
trivy image myapp:latest

# Docker Scout (built-in)
docker scout cves myapp:latest

# Scan during build
docker build -t myapp:latest .
docker scout cves myapp:latest --exit-code
```

```
# GitHub Actions with security scanning
name: Build and Scan
on: [push]
jobs:
  build-and-scan:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2

    - name: Build image
      run: docker build -t myapp:${{ github.sha }} .

    - name: Scan image
      uses: aquasecurity/trivy-action@master
      with:
        image-ref: myapp:${{ github.sha }}
        format: 'sarif'
        output: 'trivy-results.sarif'

    - name: Upload scan results
      uses: github/codeql-action/upload-sarif@v1
      with:
        sarif_file: 'trivy-results.sarif'
```

44. How do you implement CI/CD with Docker?

Answer: Create Docker-based build pipelines, use multi-stage builds for different environments, implement automated testing, and deploy using orchestration platforms.

Example:

```
# GitLab CI/CD pipeline
stages:
  - build
  - test
  - security
  - deploy

variables:
  DOCKER_IMAGE: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA

build:
  stage: build
  script:
    - docker build -t $DOCKER_IMAGE .
    - docker push $DOCKER_IMAGE

test:
  stage: test
  script:
    - docker run --rm $DOCKER_IMAGE npm test

security:
  stage: security
  script:
    - trivy image $DOCKER_IMAGE --exit-code 1 --severity HIGH,CRITICAL

deploy:
  stage: deploy
  script:
    - kubectl set image deployment/myapp container=$DOCKER_IMAGE
    - kubectl rollout status deployment/myapp
  only:
    - main
```

45. How do you handle Docker container backup and recovery?

Answer: Implement volume backups, use consistent backup strategies, test recovery procedures, and consider using backup tools specific to your data stores.

Example:

```
# Volume backup
docker run --rm \
  -v myapp_data:/data \
  -v $(pwd):/backup \
  alpine tar czf /backup/backup.tar.gz -C /data .

# Database backup
docker exec postgres-container pg_dump -U user dbname > backup.sql

# Restore volume
docker run --rm \
  -v myapp_data:/data \
  -v $(pwd):/backup \
  alpine tar xzf /backup/backup.tar.gz -C /data
```

```
# Automated backup script
#!/bin/bash
DATE=$(date +%Y%m%d_%H%M%S)
docker run --rm \
  -v postgres_data:/data \
  -v /backups:/backup \
  postgres:13 \
  pg_dump -h db -U user -d mydb > /backup/db_backup_$DATE.sql
```

46. How do you troubleshoot Docker networking issues?

Answer: Use docker network commands, inspect container networking, check port mappings, verify DNS resolution, and use debugging tools like tcpdump or netstat.

Example:

```
# Network troubleshooting commands
docker network ls
docker network inspect bridge

# Check container networking
docker exec container-name ip addr show
docker exec container-name netstat -tuln
docker exec container-name nslookup service-name

# Test connectivity between containers
docker exec container1 ping container2
docker exec container1 curl http://container2:8080/health

# Debug network traffic
docker exec container-name tcpdump -i eth0

# Create isolated network for testing
docker network create --driver bridge test-network
docker run -d --name test1 --network test-network nginx
docker run -d --name test2 --network test-network alpine sleep 3600
```

47. How do you implement Docker resource limits and monitoring?

Answer: Set CPU, memory, and I/O constraints using docker run flags or compose files. Monitor resource usage with docker stats, cAdvisor, or monitoring solutions.

Example:

```
# Resource limits
docker run -d \
  --name resource-limited \
  --memory=1g \
  --memory-swap=2g \
  --cpus="2.0" \
  --cpu-shares=1024 \
  --blkio-weight=500 \
  --pids-limit=1000 \
  myapp:latest

# Monitor resources
docker stats --no-stream
docker system df
docker system events --filter container=myapp
```

```
# Docker Compose resource limits
version: '3.8'
services:
  web:
    image: myapp:latest
    deploy:
      resources:
        limits:
          cpus: '2.0'
          memory: 1G
        reservations:
          cpus: '1.0'
          memory: 512M
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:3000/health"]
      interval: 30s
      timeout: 10s
      retries: 3
```

48. How do you handle Docker in production environments?

Answer: Implement proper logging, monitoring, security, backup strategies, use orchestration platforms, implement health checks, and plan for disaster recovery.

Example:

```
# Production-ready Docker Compose
version: '3.8'
services:
  web:
    image: myapp:${VERSION:-latest}
    replicas: 3
    restart: unless-stopped
    environment:
      - NODE_ENV=production
    ports:
      - "80:3000"
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:3000/health"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 40s
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"
    deploy:
      resources:
        limits:
          memory: 1G
          cpus: '1.0'
        reservations:
          memory: 512M
          cpus: '0.5'
      update_config:
        parallelism: 1
        delay: 10s
        failure_action: rollback
        order: start-first
      rollback_config:
        parallelism: 1
        delay: 10s
      restart_policy:
        condition: on-failure
        delay: 5s
        max_attempts: 3
        window: 120s
```

49. How do you implement Docker container orchestration patterns?

Answer: Use service discovery, load balancing, rolling updates, circuit breakers, and health checks. Implement patterns like sidecar, ambassador, and adapter containers.

Example:

```
# Sidecar pattern with logging
version: '3.8'
services:
  app:
    image: myapp:latest
    volumes:
      - app-logs:/var/log/app
    depends_on:
      - log-collector

  log-collector:
    image: fluentd:latest
    volumes:
      - app-logs:/var/log/app:ro
      - ./fluentd.conf:/fluentd/etc/fluentd.conf
    ports:
      - "24224:24224"
```

```
# Ambassador pattern for database proxy
  db-ambassador:
    image: nginx:alpine
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
    depends_on:
      - db-primary
      - db-replica

volumes:
  app-logs:
```

50. How do you implement blue-green deployment with Docker?

Answer: Maintain two identical environments (blue/green), deploy to inactive environment, test thoroughly, then switch traffic using load balancer or DNS updates.

Example:

```
# Blue-Green deployment script
#!/bin/bash

CURRENT_ENV=$(curl -s http://lb.example.com/current)
if [ "$CURRENT_ENV" = "blue" ]; then
    NEW_ENV="green"
    OLD_ENV="blue"
else
    NEW_ENV="blue"
    OLD_ENV="green"
fi

echo "Deploying to $NEW_ENV environment"

# Deploy new version to inactive environment
docker-compose -f docker-compose.$NEW_ENV.yml up -d

# Health check
echo "Waiting for health check..."
sleep 30
HEALTH_CHECK=$(curl -s -o /dev/null -w "%{http_code}" http://$NEW_ENV.example.com/health)

if [ "$HEALTH_CHECK" = "200" ]; then
    echo "Health check passed. Switching traffic to $NEW_ENV"

    # Update load balancer to point to new environment
    curl -X POST http://lb.example.com/switch -d "env=$NEW_ENV"

    # Wait and verify
    sleep 10

    # Shutdown old environment
    echo "Shutting down $OLD_ENV environment"
    docker-compose -f docker-compose.$OLD_ENV.yml down
    echo "Deployment successful!"
else
    echo "Health check failed. Rolling back..."
    docker-compose -f docker-compose.$NEW_ENV.yml down
    exit 1
fi
```

Summary

These 50+ Docker interview questions cover comprehensive topics from basic containerization concepts to advanced production scenarios. Key areas include:

Basic Concepts:

- Docker fundamentals, images, containers, Dockerfile
- Basic commands and container lifecycle
- Volumes, networks, and port mapping

Intermediate Topics:

- Docker Compose and multi-container applications
- Optimization techniques and best practices
- Debugging and troubleshooting

Advanced Scenarios:

- Security, monitoring, and production deployments
- Orchestration patterns and CI/CD integration
- Performance optimization and resource management

Production Readiness:

- Scaling, backup strategies, and disaster recovery
- Container orchestration and deployment patterns
- Security hardening and compliance

Practice these concepts hands-on and understand real-world implementation scenarios to excel in your Docker interviews! ▯