

Docker Interview Questions and Answers

Intern to senior level

Docker interview questions and answers: containers vs virtual machines, images vs containers, writing a good Dockerfile, multi-stage builds, volumes and Docker Compose, with examples.

5 questions Read online: <https://www.fredevtemplates.com/interview-questions/docker>

1. What is the difference between a Docker container and a virtual machine?

Docker Easy Intern, Junior

A virtual machine runs a full guest operating system on virtualised hardware, while a container shares the host's kernel and isolates only the process, its filesystem and resources, which makes containers much lighter and faster to start.

A hypervisor gives each virtual machine its own virtual CPU, memory and disk, and each VM boots a complete operating system. That gives strong isolation and lets you run different operating systems side by side, but each VM costs gigabytes and takes time to boot.

A container is an ordinary process on the host that the Linux kernel isolates with namespaces (its own view of processes, network and filesystem) and limits with cgroups (CPU and memory). Because there is no guest kernel, containers start in milliseconds to seconds and image sizes are often tens of megabytes.

The trade-off is isolation: containers share the kernel, so a kernel vulnerability can affect every container on the host. In practice the two are combined: cloud providers run containers inside VMs.

In the interview

A VM virtualises hardware and runs its own operating system, so it is heavy but strongly isolated. A container is just an isolated process that shares the host kernel, using namespaces and cgroups, so it starts quickly and uses far fewer resources. I use containers to package applications consistently, and they usually run inside VMs in the cloud anyway.

Key points

- VM: full guest OS on a hypervisor
- Container: isolated process sharing the host kernel
- Namespaces isolate, cgroups limit resources
- Containers are lighter; VMs isolate more strongly

2. What is the difference between a Docker image and a container?

Docker Easy Intern, Junior

An image is a read-only template made of layers that contains the application and its dependencies; a container is a running (or stopped) instance of an image with its own writable layer on top.

An image is built from a Dockerfile. Each instruction creates a layer, and layers are cached and shared between images, which makes builds and pulls faster. Images are versioned with tags and stored in registries such as Docker Hub or a private registry.

Running `docker run` creates a container from an image: Docker adds a thin writable layer, sets up networking and starts the process. You can run many containers from the same image, just as you create many objects from one class.

Changes made inside a container live only in its writable layer and disappear when the container is removed. Persistent data belongs in volumes, and changes to the application belong in a new image.

Key points

- Image: read-only, layered template

- Container: running instance with a writable layer
- Many containers can share one image
- Container changes are lost on removal; use volumes for data

```
docker build -t my-api:1.2.0 . # create an image
docker run -d --name api1 my-api:1.2.0 # start a container from it
docker run -d --name api2 my-api:1.2.0 # a second container, same image
docker ps # running containers
docker images # local images
```

3. What are Docker volumes and why do you need them?

Docker Easy Junior, Mid-level

Volumes are storage managed by Docker that lives outside a container's writable layer, so data such as database files survives when the container is removed or replaced.

A container's writable layer is deleted with the container, and it is slower than native storage. Anything that must persist, such as a database, uploaded files or caches you want to keep, needs to live outside it.

There are three options. Named volumes are created and managed by Docker and are the default choice for persistent data. Bind mounts map a host directory into the container, which is handy in development for live-reloading source code. tmpfs mounts keep data in memory only.

Volumes are also how containers share data, and they can be backed by network storage in production. Back them up like any other data: removing the container does not remove a named volume, but `docker volume rm` or `docker compose down -v` does.

Key points

- Container writable layers are temporary
- Named volumes for persistent data, bind mounts for development
- Volumes outlive containers until explicitly removed

```
docker volume create pgdata
docker run -d --name db \
-e POSTGRES_PASSWORD=secret \
-v pgdata:/var/lib/postgresql/data \
postgres:16

# the data survives replacing the container
docker rm -f db
docker run -d --name db -e POSTGRES_PASSWORD=secret \
-v pgdata:/var/lib/postgresql/data postgres:16
```

4. How do you write a good Dockerfile? What are multi-stage builds?

Docker Medium Junior, Mid-level

Use a small, pinned base image, order instructions so dependencies are cached before source code, use multi-stage builds to leave build tools out of the final image, run as a non-root user and keep secrets out of the image.

Layer caching drives build speed. Copy the dependency manifest and install dependencies first, then copy the source code. Changing a source file then reuses the cached dependency layer instead of reinstalling everything.

A multi-stage build uses one stage with compilers and dev dependencies to build the app, then copies only the output into a slim runtime stage. The final image is smaller, starts faster and has a smaller attack surface.

Other habits: pin base image versions, add a `.dockerignore` (exclude `node_modules`, `.git`, local env files), run as a non-root user, use CMD in exec form so signals reach your process, and pass secrets at runtime or with build secrets rather than ENV or COPY.

Key points

- Copy dependency files before source code for caching
- Multi-stage builds keep build tools out of production images
- Pin versions, use `.dockerignore`, run as non-root
- Never bake secrets into image layers

```
# build stage
FROM node:20-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

# runtime stage
FROM node:20-alpine
WORKDIR /app
ENV NODE_ENV=production
COPY package*.json ./
RUN npm ci --omit=dev
COPY --from=build /app/dist ./dist
USER node
EXPOSE 3000
CMD ["node", "dist/server.js"]
```

5. What is Docker Compose used for?

Docker Medium Mid-level, Senior

Docker Compose defines a multi-container application (services, networks and volumes) in one YAML so you can start, stop and rebuild the whole stack with a single command, typically for local development and testing.

Most applications need more than one container: an API, a database, a cache, perhaps a worker. Starting each with long docker run commands is error-prone. Compose describes them declaratively in `compose.yaml`.

Services on the same Compose network reach each other by service name, so the API connects to `postgres://db:5432` rather than an IP address. `depends_on` with health checks controls start-up order, and environment variables can come from an `.env` file.

Compose is ideal for development, CI and small single-host deployments. For production across many machines, teams move to an orchestrator such as Kubernetes or a managed container service, which adds scheduling, self-healing and rolling updates.

Key points

- One file describes services, networks and volumes
- Services find each other by name
- Great for development, CI and single-host setups
- Kubernetes or managed services for multi-host production

```
services:
  api:
    build: .
    ports: ["3000:3000"]
    environment:
      DATABASE_URL: postgres://app:secret@db:5432/app
    depends_on:
      db:
        condition: service_healthy
    db:
    image: postgres:16
    environment:
      POSTGRES_USER: app
      POSTGRES_PASSWORD: secret
    volumes: [pgdata:/var/lib/postgresql/data]
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U app"]
      interval: 5s
    volumes:
      pgdata:
```

These answers are written for real interview conversations. Prefer explaining a trade-off out loud over reciting a definition. Full explanations and related questions: <https://www.fredevtemplates.com/interview-questions/docker>