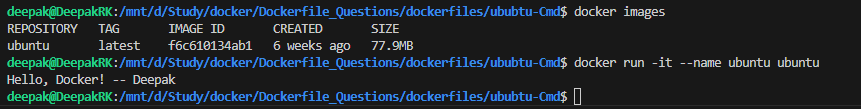
**Beginner Questions (Basic Dockerfile Concepts)**

1. **Simple Image Creation:**

Write a Dockerfile that creates a Docker image with **Ubuntu** as the base image and prints "Hello, Docker!" when run.

FROM ubuntu:22.04

CMD [ "echo", "Hello, Docker! -- Deepak" ]



1. **Installing Packages:**
   * Create a Dockerfile that installs **curl** and **git** in an Ubuntu container.
2. **Setting a Default Command (ENTRYPOINT vs CMD):**
   * What is the difference between **CMD** and **ENTRYPOINT**?
   * Create a Dockerfile that prints "Welcome to my container!" using **CMD**.
   * Modify it to use **ENTRYPOINT** instead of CMD.
3. **Using WORKDIR:**
   * Create a Dockerfile that sets **/app** as the working directory and runs a script from there.
4. **Copying Files (COPY vs ADD):**
   * What is the difference between **COPY** and **ADD**?
   * Write a Dockerfile that copies a file called index.html from your local machine into /usr/share/nginx/html/.

**🔸 Intermediate Questions (Optimization & Multi-Stage Builds)**

1. **Reducing Image Size:**
   * Why should we avoid using latest as the image tag?
   * Modify a Dockerfile to use **alpine-based images** instead of Ubuntu to reduce size.
2. **Building a Python Application:**
   * Create a Dockerfile that:
     + Uses Python 3.9 as the base image.
     + Copies requirements.txt and installs dependencies.
     + Runs app.py when the container starts.
3. **Multi-Stage Builds:**
   * Explain what **multi-stage builds** are.
   * Convert a Python Dockerfile to use multi-stage builds for reducing image size.
4. **Environment Variables (ENV vs ARG):**
   * What’s the difference between **ENV** and **ARG**?
   * Create a Dockerfile that sets an environment variable APP\_ENV=production and prints it when the container runs.
5. **Exposing Ports:**

* What does EXPOSE do?
* Create a Dockerfile for a Node.js app that exposes port 3000.

1. **Dockerfile for a Java Application:**

* Write a Dockerfile for a Java Spring Boot application that copies a JAR file and runs it using java -jar app.jar.

1. **Volume Mounting:**

* Modify a Dockerfile to ensure /data is a persistent volume.

**🔻 Advanced Questions (Security, Optimization & Production Best Practices)**

1. **Security Best Practices:**

* How can you minimize security risks in a Dockerfile?
* Implement **a non-root user** in a Dockerfile instead of running as root.

1. **Reducing Layers in a Dockerfile:**

* Explain **layer caching** and why reducing the number of layers improves efficiency.
* Refactor a multi-step Dockerfile to reduce the number of layers.

1. **Optimizing Build Speed with .dockerignore:**

* What is .dockerignore, and why is it useful?
* Create a .dockerignore file to ignore unnecessary files in a Node.js project.

1. **Healthchecks in Docker:**

* How do you define a **HEALTHCHECK** in a Dockerfile?
* Implement a HEALTHCHECK for a web server that checks if port 80 is responding.

1. **Using Labels for Metadata:**

* Add **LABEL** instructions to a Dockerfile to store metadata like maintainer, version, and description.

1. **Building Minimalist Images:**

* Convert a Dockerfile to use **distroless** images for better security.

1. **Building an Image with BuildKit:**

* Enable **BuildKit** in Docker and use it to build an image efficiently.

1. **Running Containers with Least Privileges:**

* Modify a Dockerfile to use a **non-root user** and explain why it’s important.

**🚀 Expert-Level Challenges**

1. **Multi-Architecture Docker Builds:**

* Use **Buildx** to build a multi-architecture Docker image (e.g., for both AMD64 and ARM).

1. **Advanced Multi-Stage Builds (Go Application):**

* Write a multi-stage Dockerfile for a **Go application** that compiles a binary in one stage and runs it in a minimal image in the final stage.

1. **Implementing a Reverse Proxy in Docker:**

* Create a **Dockerfile for an Nginx reverse proxy** that forwards requests to a backend service.

1. **Docker Image Signing:**

* Explain **Docker Content Trust (DCT)** and sign a Docker image before pushing it to a registry.

1. **Rootless Docker Containers:**

* Run a container **without root privileges** and explain the security benefits.

1. **Running Containers in Kubernetes with Optimized Dockerfiles:**

* Modify a Dockerfile to be **Kubernetes-friendly**, reducing attack surfaces and improving startup time.

**🔥 Bonus Questions & Challenges**

1. **Dockerfile Debugging:**

* Debug a broken Dockerfile that fails to build due to a missing package.

1. **Altering EntryPoint Dynamically:**

* Override an ENTRYPOINT at runtime and explain when this is useful.

1. **Implementing Git-based Build Caching:**

* Use **git-based caching** to optimize image builds.

1. **Creating a Private Docker Registry & Pushing an Image:**

* Set up a private Docker registry, build an image, and push it to the registry.

**Beginner Level**

1. **Basic Image Creation**
   * Create a Dockerfile that uses ubuntu:latest as the base image and runs echo "Hello, Docker!" when started.
2. **Working Directory and Commands**
   * Modify the Dockerfile to set /app as the working directory and run ls inside it.
3. **Adding Files to Image**
   * Write a Dockerfile that copies a local file (example.txt) into the container at /data/.
4. **Environment Variables**
   * Create a Dockerfile that sets an environment variable MY\_NAME="DockerUser" and prints it when the container runs.
5. **Exposing Ports**
   * Create a Dockerfile for a simple Python HTTP server (python -m http.server 8080) and expose port 8080.
6. **Running a Simple Script**
   * Write a shell script (script.sh) that prints "Container is running" and modify the Dockerfile to execute it.
7. **CMD vs ENTRYPOINT**
   * Create two versions of a Dockerfile:
     + One using CMD ["echo", "Hello from CMD"].
     + Another using ENTRYPOINT ["echo", "Hello from ENTRYPOINT"].
   * Test and observe the difference.

**Intermediate Level**

1. **Building a Custom Python App Image**
   * Create a Dockerfile that installs Python and runs a Python script (app.py).
2. **Using ARG and ENV**
   * Modify the Dockerfile to accept an argument (VERSION=3.9) and install the corresponding Python version.
3. **Optimizing Image Size**
   * Convert a multi-layer image into a smaller one using alpine as the base image instead of ubuntu.
4. **Multistage Builds**
   * Create a multistage Dockerfile that compiles a Go or Java application in one stage and runs it in a lightweight final stage.
5. **Volume Mounting**
   * Create a Dockerfile that defines a volume /app/data and persists files across container restarts.
6. **Networking Between Containers**
   * Write a Dockerfile for a database service (MySQL/PostgreSQL) and another for a frontend app that connects to it.
7. **Health Checks**
   * Modify an existing Dockerfile to include a HEALTHCHECK that pings a web server every 10 seconds.
8. **Signal Handling in Containers**
   * Create a Dockerfile that runs a long-lived process (e.g., sleep 1000) and handles termination signals correctly.

**Advanced Level**

1. **Reducing Build Context**
   * Optimize a large application’s Dockerfile by using .dockerignore to exclude unnecessary files.
2. **Implementing a Non-Root User**
   * Modify an application’s Dockerfile to run as a non-root user for security.
3. **Minimizing Layers**
   * Rewrite an inefficient Dockerfile that has multiple RUN statements into one optimized RUN block.
4. **Building for Multiple Architectures**
   * Modify a Dockerfile to build images for both amd64 and arm64 using docker buildx.
5. **Creating a Custom Base Image**
   * Build a minimal custom Linux base image with only necessary dependencies and use it in a Dockerfile.
6. **Building and Running Containers in CI/CD**
   * Write a Dockerfile that integrates with GitHub Actions or Jenkins for automated builds and deployments.
7. **Securing Docker Images**
   * Modify a Dockerfile to use best security practices, such as avoiding latest tags, scanning vulnerabilities, and using official images.
8. **Advanced Multistage Builds**
   * Build a complex application (e.g., React + Node.js + PostgreSQL) using multistage builds efficiently.
9. **Creating a Dockerfile for a Kubernetes Deployment**
   * Write a Dockerfile that is optimized for Kubernetes, including readiness and liveness probes.
10. **Debugging and Profiling Containers**
    * Create a Dockerfile that includes debugging tools like strace, tcpdump, and logs all container actions.

# 🟢 **Beginner Level (Getting Started)**

### ****Real-World Scenarios****

1. **Fixing a Slow Build for a Simple App**
   * You have a Python Flask app that takes too long to build. Modify the Dockerfile to **cache dependencies** properly to speed up the build.
2. **Handling OS Compatibility Issues**
   * Your team works on both Windows and macOS, but your current Dockerfile doesn't work on macOS due to **line endings**. Modify the Dockerfile to ensure compatibility across platforms.
3. **Building an Image for a Legacy PHP Application**
   * The company has an old PHP 5.6 app. You need to create a Dockerfile that:
     + Uses a compatible PHP version
     + Installs pdo\_mysql extension
     + Runs apache2 on startup
4. **Dockerfile for a C++ Application**
   * Your team is moving a C++ application to Docker. Write a Dockerfile that:
     + Uses gcc for compilation
     + Copies source code
     + Builds the executable
     + Runs it inside the container
5. **Fixing a “Command Not Found” Error**
   * You built an image using CMD ["python", "app.py"], but when you run the container, it says:

python: command not found

* + - Debug the issue and fix the Dockerfile.

# 🟡 **Intermediate Level (Optimization & Deployment)**

### ****Real-World Scenarios****

1. **Minimizing Image Size for a Node.js App**
   * Your **Node.js app** is 1GB after building. Optimize the Dockerfile to:
     + Use a minimal base image
     + Remove unnecessary files
     + Reduce build context size
2. **Deploying a Django App with Static Files**
   * Your Django app works fine locally but fails to serve **static files** in production. Modify the Dockerfile to:
     + Collect static files properly
     + Serve them using whitenoise or Nginx
3. **Building a Frontend + Backend Docker Image**
   * Your app has a **React frontend** and a **FastAPI backend**. You need to create a **single Dockerfile** that:
     + Builds and serves the React app
     + Runs the FastAPI backend
     + Ensures they work together in one container
4. **Fixing Permission Issues in a Mounted Volume**
   * When running a container with a mounted volume, your app throws a **"Permission Denied"** error on /app/data.
     + Modify the Dockerfile to ensure proper **file ownership** and **non-root execution**.
5. **Migrating from a Monolithic to a Microservices Docker Setup**

* You need to split a monolithic **Django + Celery** app into **separate services**:
  + One container runs Django
  + Another runs Celery workers
  + Redis is used for message passing

# 🔴 **Advanced Level (Security, Scaling & Best Practices)**

### ****Real-World Scenarios****

1. **Fixing Vulnerabilities in a Docker Image**

* Your security team flagged a **critical vulnerability** in your Dockerfile.
  + Use docker scan to find issues.
  + Fix vulnerabilities by updating dependencies and using a secure base image.

1. **Running a Container with the Least Privileges**

* Your Dockerfile runs as **root** by default. Modify it to:
  + Create a non-root user
  + Ensure the app still has necessary permissions

1. **Enabling Health Checks for a PostgreSQL Database Container**

* Your **PostgreSQL container** sometimes starts before it's ready.
  + Implement a **HEALTHCHECK** that waits for PostgreSQL to accept connections.

1. **Handling Sensitive Configuration in a Dockerfile**

* The Dockerfile currently contains **hardcoded API keys**. Fix this by:
  + Moving secrets to environment variables
  + Ensuring .env files are not copied into the image

1. **Ensuring Data Persistence for a MongoDB Container**

* Your **MongoDB database** loses all data after a container restart.
  + Modify the setup to ensure data persists across restarts using **Docker volumes**.

# ⚫ **Expert Level (Production-Grade Optimizations & Edge Cases)**

### ****Real-World Scenarios****

1. **Handling Multi-Architecture Builds for ARM & x86**

* Your company wants to support both **Raspberry Pi (ARM)** and **AWS EC2 (x86)**.
  + Modify the Dockerfile to build a **multi-architecture image** using **Buildx**.

1. **Using Distroless Images for Security**

* Your security team requires that all images be **distroless**.
  + Convert an existing Dockerfile to use Google’s **distroless** base image for a secure, minimal footprint.

1. **Fixing a Memory Leak in a Dockerized Java App**

* Your **Java Spring Boot container** runs out of memory after 24 hours.
  + Modify the Dockerfile to set proper **JVM memory limits** (-XX:+UseContainerSupport).

1. **Implementing a Reverse Proxy with Docker & Nginx**

* You need to deploy multiple services (app, admin, api) under **one domain** using Nginx.
  + Configure an **Nginx reverse proxy** in Docker to route requests correctly.

1. **Building and Pushing to a Private Docker Registry**

* Your company has a private **Docker registry**.
  + Modify the CI/CD pipeline to build a Docker image, tag it, and push it to the private registry.