Master Docker: From Beginner to Expert

This README is your complete guide to mastering Docker, from absolute beginner to advanced practitioner. It provides detailed explanations, practical examples, hands-on exercises, and a capstone project to help you build, deploy, and manage containerized applications with confidence. Whether you're a developer, DevOps engineer, or enthusiast, this guide equips you with the skills to use Docker in real-world scenarios, from local development to production.

Table of Contents

Introduction to Docker

Docker is an open-source platform that uses **containerization** to package applications and their dependencies into portable, isolated units called **containers**. Containers ensure your app runs identically across environments—your laptop, a colleague's machine, or a production server—eliminating issues like "it works on my machine."

Why Docker?

- **Consistency**: Same behavior in development, testing, and production.
- **Portability**: Runs anywhere Docker is installed (Windows, Mac, Linux).
- **Efficiency**: Containers share the host OS kernel, using fewer resources than virtual machines (VMs).
- **Scalability**: Easily scale with orchestration tools like Docker Swarm or Kubernetes.
- **Simplified Development**: Manages dependencies, reducing setup time.

Analogy: Containers are like standardized shipping containers. Each holds an app and its dependencies (code, libraries, configs), and Docker is the port that ensures they work anywhere, from a local dock (development) to a global hub (production).

Real-World Use Case: A Python web app with Redis can be packaged into a container, tested locally, and deployed to a cloud server without reconfiguring dependencies.

Installing Docker

Install **Docker Desktop** (Windows/Mac) or **Docker Engine** (Linux) to get started. Docker Desktop includes Docker Engine, CLI, and Compose.

Installation Steps

1. **Windows/Mac**:
   - Download Docker Desktop from docker.com.
   - **Windows**: Enable WSL 2 for better performance:

     ```
     wsl --install
     ```

   - Follow the installer prompts and launch Docker Desktop.
2. **Linux (Ubuntu)**:

   ```
   # Update package index
   sudo apt-get update
   # Install Docker
   sudo apt-get install -y docker.io
   # Start and enable Docker service
   sudo systemctl start docker
   sudo systemctl enable docker
   # Add user to docker group
   sudo usermod -aG docker $USER
   ```

   Log out and back in to apply group permissions.
3. **Verify Installation**:

   ```
   docker --version
   docker run hello-world
   ```

   The `hello-world` container pulls a small image and prints a success message.

Expected Output:

```
Hello from Docker!
This message shows that your installation appears to be working correctly.
```

Explanation: The `hello-world` image tests Docker's ability to pull images, create containers, and run them.

Troubleshooting: - **Docker not starting**: Check if virtualization is enabled in BIOS (VT-x/AMD-V). - **Permission denied**: Verify `$USER` is in the `docker` group: `groups | grep docker`. - **WSL 2 issues**: Update WSL with `wsl --update`.

Exercise: Install Docker and run `hello-world`. Confirm the output and check `docker info`.

Docker Fundamentals

Containers vs. Virtual Machines

- **Virtual Machines**:
  - Include a full guest OS, hypervisor, and app.
  - Heavyweight (GBs, slow startup).
  - Example: A Windows VM running a Python app on a Linux host.
- **Containers**:
  - Share the host OS kernel, including only the app and dependencies.
  - Lightweight (MBs, instant startup).
  - Example: A Python app container on a Linux host.

Diagram:

```
VM:                      | Container:
[App]                    | [App]
[Libs]                   | [Libs]
[Guest OS]               | [Docker Engine]
[Hypervisor]             | [Host OS]
[Host OS]                | [Hardware]
[Hardware]               |
```

Analogy: VMs are standalone houses with their own utilities. Containers are apartments sharing the building's infrastructure (host OS).

Images and Containers

- **Image**: A read-only template with the app, libraries, and dependencies, stored in a layered filesystem. Each layer represents a change (e.g., installing a package).
- **Container**: A running or stopped instance of an image, with a writable layer for runtime changes (e.g., temporary files).

Analogy: An image is a blueprint; a container is a house built from it. Multiple houses (containers) can be built from one blueprint (image).

Layered Filesystem: Docker uses a copy-on-write system. Each Dockerfile instruction creates a cached layer, making builds efficient. Containers add a writable layer on top.

Basic Docker Commands

```
# List running containers
docker ps
# List all containers
docker ps -a
# List images
docker images
# Remove a container
```

```
docker rm <container_id>
# Remove an image
docker rmi <image_id>
# View system info
docker info --format '{{.ServerVersion}}'
```

Example: Run an Nginx server:

```
docker run -d --name my-nginx nginx:1.25
docker ps
docker inspect my-nginx
```

Explanation: - `docker run`: Creates and starts a container. - `-d`: Detached mode (background). - `--name`: Custom container name. - `nginx:1.25`: Image and tag. - `docker inspect`: Shows detailed container metadata (e.g., IP, ports).

Exercise: 1. Run an `alpine` container and execute `echo "Hello, Docker!"`: `bash    docker run -it alpine sh -c 'echo "Hello, Docker!"'` 2. List all containers with `docker ps -a`.

Working with Docker Images

Pulling Images

Images are stored in registries like Docker Hub.

```
docker pull redis:7.0
```

Explanation: - `redis`: Image name. - `7.0`: Version tag. Omitting the tag pulls `latest`. - Docker caches images locally to avoid redundant downloads.

Example: Pull and run PostgreSQL:

```
docker pull postgres:15
docker run -d --name my-postgres -e POSTGRES_PASSWORD=secret postgres:15
```

Exercise: Pull `mongo:5` and verify with `docker images`.

Building Custom Images

A **Dockerfile** defines image creation steps using instructions like `FROM`, `COPY`, and `CMD`.

Example: Build a Python Flask app.

1. Create `my-flask-app` directory:

   ```
   my-flask-app/
     Dockerfile
     requirements.txt
     app.py
   ```

```
  .dockerignore
```

2. **Dockerfile**:

   ```
   # Use a lightweight base image
   FROM python:3.9-slim
   # Set working directory
   WORKDIR /app
   # Copy dependencies first for layer caching
   COPY requirements.txt .
   # Install dependencies
   RUN pip install --no-cache-dir -r requirements.txt
   # Copy app code
   COPY . .
   # Expose port (documentation)
   EXPOSE 5000
   # Set environment variable
   ENV FLASK_ENV=production
   # Define entrypoint and command
   ENTRYPOINT ["python"]
   CMD ["app.py"]
   ```

3. **requirements.txt**:

   ```
   flask==2.3.2
   ```

4. **app.py**:

   ```
   from flask import Flask
   app = Flask(__name__)
   @app.route('/')
   def home():
       return 'Hello, Dockerized Flask!'
   if __name__ == '__main__':
       app.run(host='0.0.0.0', port=5000)
   ```

5. **.dockerignore**:

   ```
   __pycache__
   *.pyc
   .git
   .env
   ```

6. Build and run:

   ```
   cd my-flask-app
   docker build -t my-flask-app:latest .
   docker run -p 5000:5000 my-flask-app
   ```

   Visit `http://localhost:5000`.

Explanation: - **FROM**: Specifies the base image. - **WORKDIR**: Sets the container's working directory (like `cd`). - **COPY**: Copies files from the host to the container. - **RUN**: Executes commands during image building. - **EXPOSE**: Documents the port (doesn't publish it). - **ENV**: Sets environment variables. - **ENTRYPOINT**: Defines the executable (fixed command). - **CMD**: Specifies arguments for `ENTRYPOINT` or the default command. - **.dockerignore**: Excludes files to reduce image size and improve security.

CMD vs. ENTRYPOINT: - `CMD`: Can be overridden when running the container (e.g., `docker run my-image bash`). - `ENTRYPOINT`: Fixed command, harder to override, ideal for defining the container's purpose. - Example: `ENTRYPOINT ["python"], CMD ["app.py"]` runs `python app.py`, but `docker run -it my-image bash` runs `python bash`.

Example 2: Build a Node.js app. 1. Create `my-node-app`: `my-node-app/`
`Dockerfile` `package.json` `app.js`

2. **Dockerfile**:

```
FROM node:18
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["node", "app.js"]
```

3. **package.json**:

```
{
  "name": "my-node-app",
  "version": "1.0.0",
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

4. **app.js**:

```
const express = require('express');
const app = express();
app.get('/', (req, res) => res.send('Hello, Dockerized Node!'));
app.listen(3000, () => console.log('Server on port 3000'));
```

5. Build and run:

```
cd my-node-app
docker build -t my-node-app:latest .
docker run -p 3000:3000 my-node-app
```

Exercise: Create a Dockerfile for a simple Go web server and run it locally.

Tagging and Pushing Images

Tag an image for a registry:

```
docker tag my-flask-app:latest yourusername/my-flask-app:latest
```

Push to Docker Hub:

```
docker login
docker push yourusername/my-flask-app:latest
```

Troubleshooting: - **Push fails**: Ensure `docker login` succeeds and the tag matches your Docker Hub username. - **Authentication error**: Verify credentials with `docker login --username yourusername`.

Exercise: Build, tag, and push an image to Docker Hub.

Managing Containers

Running Containers

Run containers in: - **Detached mode** (`-d`): Background execution. - **Interactive mode** (`-it`): Terminal access.

```
# Detached Nginx
docker run -d --name web nginx:1.25
# Interactive Ubuntu shell
docker run -it --name shell ubuntu bash
```

Explanation: - `-d`: Frees the terminal for other tasks. - `-it`: Provides an interactive shell (`-i` for input, `-t` for TTY). - `--name`: Assigns a custom name; otherwise, Docker generates a random one.

Container Lifecycle

- **Start/Stop/Restart**:

  ```
  docker stop web
  docker start web
  docker restart web
  ```

- **Inspect Logs**:

  ```
  docker logs web
  ```

- **Execute Commands**:

  ```
  docker exec -it web bash
  ```

- **Remove**:

  ```
  docker rm -f web
  ```

Example: Run Redis, check logs, and execute a command:

```
docker run -d --name my-redis redis:7.0
docker logs my-redis
docker exec -it my-redis redis-cli
```

Explanation: - `docker logs`: Shows container output (stdout/stderr). - `docker exec`: Runs commands in a running container (e.g., accessing Redis CLI).

Port Mapping

Map host ports to container ports:

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

Access `http://localhost:8080`.

Explanation: `-p host_port:container_port` forwards traffic from the host to the container.

Example: Run Flask with custom port:

```
docker run -p 5000:5000 my-flask-app
```

Environment Variables

Pass runtime configurations:

```
docker run -e MYSQL_ROOT_PASSWORD=secret -d mysql:8.0
```

In a Dockerfile:

```
ENV APP_PORT=5000
```

Example: Run a Node app with custom env vars:

```
docker run -e PORT=4000 -p 4000:4000 my-node-app
```

Exercise: Run a PostgreSQL container with custom port and environment variables, then connect using `psql`.

Docker Storage

Volumes vs. Bind Mounts

- **Volumes**: Docker-managed storage in `/var/lib/docker/volumes`. Ideal for persistent data.

  ```
  docker volume create my_volume
  docker run -v my_volume:/app -d nginx
  ```

- **Bind Mounts**: Map host directories to container paths. Useful for development.

  ```
  docker run -v /host/data:/app -d nginx
  ```

Diagram:

```
Volume:                 | Bind Mount:
[Docker Storage]        | [Host Filesystem]
  |                     |    |
[Container:/app]        | [Container:/app]
```

Analogy: Volumes are like cloud storage managed by Docker; bind mounts are like plugging in a USB drive.

Example: Persist MySQL data:

```
docker volume create mysql_data
docker run -d -v mysql_data:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=secret mysql:8.0
```

Managing Volumes

```
# List volumes
docker volume ls
# Inspect volume
docker volume inspect mysql_data
# Remove volume
docker volume rm mysql_data
```

Explanation: Volumes persist data after container removal, unlike container storage.

Exercise: Create a volume for MongoDB, remove the container, and verify data persistence in a new container.

Docker Networking

Default Networks

Docker provides: - **Bridge**: Default, isolated network for containers. - **Host**: Shares host's network stack (no isolation). - **None**: No networking.

List networks:

```
docker network ls
```

Example: Run a container on the bridge network:

```
docker run -d --name nginx1 nginx
docker network inspect bridge
```

Custom Networks

Create a custom bridge network:

```
docker network create my-app-network
docker run --network my-app-network -d --name app1 nginx
docker run --network my-app-network -d --name redis1 redis
```

Example: Test communication:

```
docker exec -it app1 ping redis1
```

Explanation: Containers on the same network resolve each other by name, simplifying communication.

Network Types

- **Overlay**: For multi-host networking in Swarm.
- **Macvlan**: Assigns a MAC address for direct network access.

Example: Create an overlay network:

```
docker swarm init
docker network create -d overlay my-overlay
```

Exercise: Create a custom network, run two containers, and verify communication by name.

Docker Logging

Logging Drivers

Docker supports drivers like: - `json-file`: Default, stores logs as JSON. - `syslog`: Sends logs to a syslog server. - `none`: Disables logging.

Configure a driver:

```
docker run --log-driver=syslog -d nginx
```

Example: Limit log size:

```
docker run --log-driver=json-file --log-opt max-size=10m --log-opt max-file=3 nginx
```

Explanation: Log options prevent disk space issues by rotating or limiting logs.

Accessing Logs

View logs:

```
docker logs --follow my-nginx
```

Tail recent logs:

```
docker logs --tail 50 my-nginx
```

Example: Run with `none` driver:

```
docker run --log-driver=none -d --name test-app my-flask-app
docker logs test-app
```

Exercise: Configure a container with `json-file` driver, limit to 5 log files, and check logs.

Docker Compose

Introduction to Docker Compose

Docker Compose orchestrates multi-container apps using a YAML file, simplifying dependency management.

Analogy: Compose is like a conductor, coordinating multiple instruments (containers) to play a symphony (your app).

Writing Compose Files

Example: Flask app with PostgreSQL and Nginx.

1. Create `my-flask-stack`:

   ```
   my-flask-stack/
     Dockerfile
     requirements.txt
     app.py
     nginx.conf
     docker-compose.yml
   ```

2. Use the previous Flask **Dockerfile**, `app.py`, and `requirements.txt`.

3. **docker-compose.yml**:

   ```yaml
   version: '3.8'
   services:
     web:
       build: .
       ports:
         - "5000:5000"
       environment:
         - DATABASE_URL=postgres://user:pass@db:5432/appdb
       depends_on:
         - db
       healthcheck:
         test: ["CMD", "curl", "-f", "http://localhost:5000"]
   ```

```
      interval: 30s
      timeout: 5s
      retries: 3
  db:
    image: postgres:15
    volumes:
      - db_data:/var/lib/postgresql/data
    environment:
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=pass
      - POSTGRES_DB=appdb
  proxy:
    image: nginx:1.25
    ports:
      - "80:80"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - web
volumes:
  db_data:
```

4. **nginx.conf**:

   ```
   events {}
   http {
     server {
       listen 80;
       location / {
         proxy_pass http://web:5000;
         proxy_set_header Host $host;
       }
     }
   }
   ```

5. Run:

   ```
   cd my-flask-stack
   docker-compose up -d
   ```

   Access `http://localhost`.

Explanation: - **services**: Defines containers (e.g., `web`, `db`). - **build**: Builds an image from a Dockerfile. - **ports**: Maps host:container ports. - **volumes**: Persists data or mounts files. - **environment**: Sets variables. - **depends_on**: Controls startup order. - **healthcheck**: Monitors container health.

Managing Multi-Container Apps

- Scale services:

  ```
  docker-compose up --scale web=3
  ```

- Stop and remove:

  ```
  docker-compose down --volumes
  ```

Example: WordPress with MySQL:

```
version: '3.8'
services:
  wordpress:
    image: wordpress:latest
    ports:
      - "8000:80"
    environment:
      - WORDPRESS_DB_HOST=db
      - WORDPRESS_DB_USER=root
      - WORDPRESS_DB_PASSWORD=password
      - WORDPRESS_DB_NAME=wp_db
    depends_on:
      - db
  db:
    image: mysql:8.0
    volumes:
      - wp_db_data:/var/lib/mysql
    environment:
      - MYSQL_ROOT_PASSWORD=password
      - MYSQL_DATABASE=wp_db
volumes:
  wp_db_data:
```

Exercise: Create a Compose file for a Node.js app with MongoDB and test locally.

Advanced Docker Concepts

Docker Security

Secure containers: - **Non-Root Execution**: `dockerfile   RUN useradd -m appuser   USER appuser` - **Image Scanning**: `bash   docker scan my-flask-app:latest` - **Resource Limits**: `bash   docker run --memory="256m" --cpus="0.5" nginx`

Example: Secure Flask app:

```
FROM python:3.9-slim
RUN useradd -m flaskuser
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
USER flaskuser
EXPOSE 5000
CMD ["python", "app.py"]
```

Exercise: Add a non-root user to a Dockerfile and scan the image.

Healthchecks

Monitor container health:

```
HEALTHCHECK --interval=30s --timeout=5s \
  CMD curl -f http://localhost:5000/ || exit 1
```

Example: Healthcheck for Node.js:

```
HEALTHCHECK --interval=10s --timeout=3s \
  CMD ["wget", "--spider", "http://localhost:3000"]
```

Exercise: Add a healthcheck and verify with `docker ps`.

Docker Secrets

Manage sensitive data in Swarm:

```
echo "my-secret-key" | docker secret create my-secret -
```

In `docker-compose.yml`:

```
services:
  web:
    build: .
    secrets:
      - my-secret
secrets:
  my-secret:
    external: true
```

Example: Access a secret:

```
docker run --rm --secret my-secret alpine cat /run/secrets/my-secret
```

Exercise: Use a secret for a database password in Compose.

Docker Swarm

Orchestrate with Swarm:

```
docker swarm init
docker stack deploy -c docker-compose.yml myapp
```

Example: Scale a service:

```
docker service scale myapp_web=3
```

Exercise: Deploy a stack to Swarm and scale it.

Multi-Stage Builds

Optimize image size:

```
FROM node:18 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build
FROM nginx:alpine
COPY --from=builder /app/build /usr/share/nginx/html
```

Exercise: Create a multi-stage build for a React app.

Docker BuildKit

Enable faster builds:

```
export DOCKER_BUILDKIT=1
docker build -t my-app .
```

Exercise: Build an image with BuildKit and compare performance.

Docker Contexts

Manage multiple Docker hosts:

```
docker context create remote --docker host=ssh://user@remote-host
docker context use remote
docker ps
```

Exercise: Create and use a remote context.

Docker Registries

Run a private registry:

```
docker run -d -p 5000:5000 --name registry registry:2
```

Push to it:

```
docker tag my-flask-app localhost:5000/my-flask-app
docker push localhost:5000/my-flask-app
```

Troubleshooting: If push fails, check registry logs: `docker logs registry`.

Exercise: Set up a private registry and push an image.

Docker Plugins

Extend Docker:

```
docker plugin install --grant-all-permissions vieux/sshfs
```

Exercise: Install and test a storage plugin.

Docker in CI/CD

Example: GitHub Actions workflow:

```yaml
name: Build and Push Docker Image
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Build and push
        run: |
          docker build -t myusername/my-flask-app:latest .
          echo ${{ secrets.DOCKER_PASSWORD }} | docker login -u ${{ secrets.DOCKER_USERNAME
          docker push myusername/my-flask-app:latest
```

Exercise: Set up a CI/CD pipeline to build and push an image.

Docker vs. Kubernetes

- **Docker Swarm**: Simpler, built-in, good for small-to-medium setups.
- **Kubernetes**: Complex, feature-rich, ideal for large-scale deployments.

Example: Kubernetes deployment:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-flask
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-flask
  template:
    metadata:
      labels:
        app: my-flask
    spec:
      containers:
      - name: my-flask
        image: my-flask-app:latest
        ports:
        - containerPort: 5000
```

Exercise: Deploy an app to Swarm and write a Kubernetes equivalent.

Troubleshooting Docker

- **Container Crashes**: Check logs:

  ```
  docker logs my-flask-app
  ```

- **Port Conflicts**: Find used ports:

  ```
  netstat -tuln
  ```

- **Disk Space**: Clean up:

  ```
  docker system prune -a --volumes
  ```

- **Permission Denied**: Ensure `docker` group:

  ```
  groups | grep docker
  ```

Example: Debug a crash:

```
docker logs my-flask-app
# If "Module not found", rebuild image
```

Exercise: Simulate a port conflict and resolve it.

Performance Optimization

- Use lightweight images:

  ```
  FROM alpine
  ```

- Cache layers:

  ```
  COPY package*.json ./
  RUN npm install
  COPY . .
  ```

- Limit resources:

  ```
  docker run --memory="512m" --cpus="1" nginx
  ```

Exercise: Optimize a Dockerfile by switching to `alpine`.

Best Practices

- Use specific tags (e.g., `nginx:1.25`).
- Include `.dockerignore`:

  ```
  node_modules
  .git
  *.log
  ```

- Run as non-root.
- Use secrets for sensitive data.
- Clean up:

  ```
  docker system prune -a --volumes
  ```

Capstone Project: Full-Stack App

Goal: Build and deploy a full-stack app with a React frontend, Node.js backend, MongoDB database, and Nginx reverse proxy.

Directory Structure

```
my-fullstack-app/
  frontend/
    Dockerfile
    package.json
    src/
      App.js
  backend/
    Dockerfile
    package.json
    server.js
  nginx/
    Dockerfile
    nginx.conf
  docker-compose.yml
```

1. Frontend (React)

frontend/Dockerfile:

```
FROM node:18 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build
FROM nginx:alpine
COPY --from=builder /app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

frontend/package.json:

```
{
  "name": "frontend",
  "version": "1.0.0",
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build"
  },
  "devDependencies": {
    "react-scripts": "^5.0.1"
  }
}
```

frontend/src/App.js:

```
import React from 'react';
function App() {
  return <h1>Hello from React Frontend!</h1>;
}
export default App;
```

Explanation: - Uses a multi-stage build to compile the React app and serve it with Nginx. - The `builder` stage installs dependencies and builds the app. - The final stage copies the static build to Nginx's web directory.

2. Backend (Node.js)

backend/Dockerfile:

```
FROM node:18
WORKDIR /app
COPY package*.json ./
RUN npm install
```

```
COPY . .
EXPOSE 3000
CMD ["node", "server.js"]
```

backend/package.json:

```
{
  "name": "backend",
  "version": "1.0.0",
  "dependencies": {
    "express": "^4.18.2",
    "mongodb": "^4.12.0"
  },
  "scripts": {
    "start": "node server.js"
  }
}
```

backend/server.js:

```
const express = require('express');
const { MongoClient } = require('mongodb');
const app = express();
const uri = process.env.MONGO_URI || 'mongodb://user:pass@db:27017/appdb';
app.get('/api', async (req, res) => {
  const client = new MongoClient(uri);
  await client.connect();
  res.json({ message: 'Connected to MongoDB!' });
  await client.close();
});
app.listen(3000, () => console.log('Backend on port 3000'));
```

Explanation: - Builds a Node.js server with Express. - Connects to MongoDB using the `MONGO_URI` environment variable. - Exposes an `/api` endpoint to verify database connectivity.

3. Nginx (Reverse Proxy)

nginx/Dockerfile:

```
FROM nginx:1.25
COPY nginx.conf /etc/nginx/nginx.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

nginx/nginx.conf:

```
events {}
http {
  server {
    listen 80;
    location / {
      proxy_pass http://frontend:80;
      proxy_set_header Host $host;
    }
    location /api/ {
      proxy_pass http://backend:3000/;
      proxy_set_header Host $host;
    }
  }
}
```

Explanation: - Configures Nginx to route `/` to the frontend and `/api/` to the backend. - Uses container names (`frontend`, `backend`) for internal routing.

4. Docker Compose

docker-compose.yml:

```yaml
version: '3.8'
services:
  frontend:
    build: ./frontend
    networks:
      - app-network
  backend:
    build: ./backend
    environment:
      - MONGO_URI=mongodb://user:pass@db:27017/appdb
    depends_on:
      - db
    networks:
      - app-network
  db:
    image: mongo:5
    volumes:
      - db_data:/data/db
    environment:
      - MONGO_INITDB_ROOT_USERNAME=user
      - MONGO_INITDB_ROOT_PASSWORD=pass
    networks:
      - app-network
  proxy:
    build: ./nginx
    ports:
      - "80:80"
    depends_on:
      - frontend
      - backend
    networks:
      - app-network
networks:
  app-network:
    driver: bridge
volumes:
  db_data:
```

5. Run the Project

```
cd my-fullstack-app
docker-compose up -d
```

- Access the frontend at `http://localhost`.
- Access the backend at `http://localhost/api`.

6. Bonus Tasks

- Push images to a private registry:

  ```
  docker tag my-fullstack-app_frontend yourusername/frontend:latest
  docker push yourusername/frontend:latest
  ```

- Deploy to Docker Swarm:

  ```
  docker swarm init
  docker stack deploy -c docker-compose.yml mystack
  ```

- Set up a CI/CD pipeline with GitHub Actions.

Exercise: 1. Build and deploy the app locally. 2. Verify the frontend (`http://localhost`) and backend (`http://localhost/api`). 3. Add a healthcheck to the backend service and test it.

Resources for Further Learning

- Docker Documentation
- Docker Hub
- Play with Docker
- Books: *Docker Deep Dive* by Nigel Poulton, *The Docker Book* by James Turnbull
- Courses: Docker Mastery (Udemy), Docker Certified Associate prep

---

This guide is now **complete**, covering all Docker topics with detailed explanations, practical examples, and a fully specified capstone project. Follow the examples and exercises to become a Docker expert!