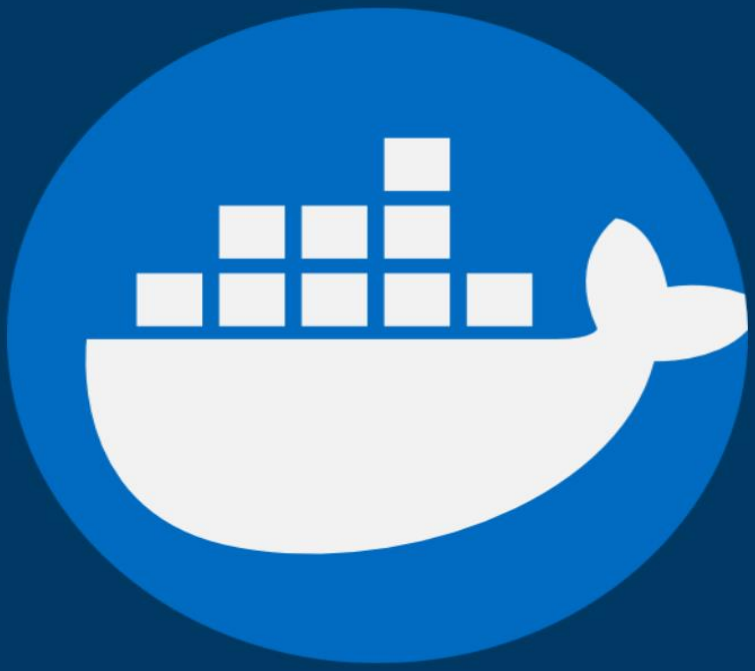
Devops Shack

DOCKER PROJECTS

Basic to Advanced



By DevOps Shack

2025

www.devopsshack.com

10 Projects

[Click here for DevSecOps & Cloud DevOps Course]

DevOps Shack

10 Docker Projects to Master Docker:

Basic to Advanced

Table of Contents

(Beginner)

(Advanced)

6. Secure Docker Deployment with Traefik and Let's Encrypt

- ✅ Configuring Traefik as a reverse proxy
- ✅ Enabling HTTPS with Let's Encrypt
- ✅ Adding Basic Authentication for security

7. Scalable Microservices Architecture with Docker and Kubernetes

- ✅ Creating multiple microservices
- ✅ Deploying services using Kubernetes
- ✅ Setting up an API Gateway for routing

8. Continuous Integration & Deployment (CI/CD) for Dockerized Applications

- ✅ Automating builds with GitHub Actions / Jenkins
- ✅ Pushing images to Docker Hub
- ✅ Deploying to a remote server or Kubernetes

9. Multi-Stage Docker Build for Optimized Production Deployment

- ✅ Using multi-stage builds to reduce image size
- ✅ Running the app with Gunicorn for better performance
- ✅ Setting up an Nginx reverse proxy

10. Optimized Docker Image Deployment with Kubernetes

- ✅ Deploying the optimized Docker image to Kubernetes
- ✅ Using ConfigMaps and Secrets for environment variables
- ✅ Scaling services efficiently

Introduction

Docker has revolutionized the way applications are developed, deployed, and managed. By leveraging containerization, developers can ensure consistency across different environments, improve application scalability, and enhance overall deployment efficiency. Whether you are a beginner looking to get started or an experienced developer aiming to refine your skills, hands-on projects are the best way to gain a deep understanding of Docker.

This document presents **10 practical projects** that will help you master Docker, covering everything from basic containerization to advanced deployment strategies. Starting with fundamental concepts such as running a simple web application in a container, you will progress through multi-container applications, persistent storage, networking, scaling, security, and automation. Each project is carefully designed to introduce key Docker features and best practices while providing real-world applications for practical learning.

By the end of this document, you will have built a **solid foundation in Docker**, learned how to efficiently manage containerized applications, and gained experience with orchestration tools like Docker Swarm and Kubernetes. These projects will not only enhance your technical expertise but also prepare you for real-world challenges in DevOps and cloud-native application development.

Let's dive into the projects and begin our journey to mastering Docker!

1. Containerized Web Application (Beginner)

Goal: Learn the basics of Docker by containerizing a simple web application.

Tech Stack:

- Python (Flask) or Node.js (Express)
- Docker

Implementation Steps:

1. **Set Up Your Web Application**
    - Create a simple Flask or Node.js application.
    - Example (Flask app in app.py):

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def home():
    return "Hello, Docker!"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

2. **Write a Dockerfile**
    - Create a Dockerfile in the project root:

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

3. **Build & Run the Docker Image**
    - Build the Docker image:

```
docker build -t flask-app .
```

    - Run the container:

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

4. **Verify the Application**
    - Open http://localhost:5000 in your browser.

2. Multi-Container Application with Docker Compose

Goal: Learn how to manage multi-container applications using Docker Compose.

Tech Stack:

- Flask/Node.js
- PostgreSQL/MySQL
- Docker Compose

Implementation Steps:

1. **Set Up Web Application & Database**
    - Create a simple web application with a database connection (Flask + PostgreSQL).
    - Example app.py:

```
import psycopg2
from flask import Flask

app = Flask(__name__)

def connect_db():
    return psycopg2.connect(
        dbname="mydb",
        user="user",
        password="password",
        host="db"
    )

@app.route("/")
def home():
```

```
    conn = connect_db()
    return "Connected to DB!"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

2. **Create a Dockerfile**

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

3. **Create docker-compose.yml**

```
version: '3.8'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    depends_on:
      - db

  db:
    image: postgres
    environment:
      POSTGRES_USER: user
```

POSTGRES_PASSWORD: password

POSTGRES_DB: mydb

4. **Run with Docker Compose**

```
docker-compose up --build
```

5. **Verify Connection**
    - Open http://localhost:5000 in your browser.

3. Custom Docker Images & CI/CD Pipeline

Goal: Automate Docker image builds and deployments with CI/CD.

Tech Stack:

- GitHub Actions/GitLab CI
- Docker Hub
- Flask/Node.js

Implementation Steps:

1. **Set Up a GitHub Repository**
    - Push your Dockerized app to GitHub.
2. **Create a GitHub Actions Workflow (.github/workflows/docker.yml)**

```
name: Docker Build & Push

on:
  push:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Repository
        uses: actions/checkout@v2

      - name: Login to Docker Hub
```

```
      run: echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u "${{
secrets.DOCKER_USERNAME }}" --password-stdin

    - name: Build Docker Image
      run: docker build -t myusername/myapp:latest .

    - name: Push to Docker Hub
      run: docker push myusername/myapp:latest
```

3. **Configure Secrets in GitHub**
    - Add DOCKER_USERNAME and DOCKER_PASSWORD as repository secrets.
4. **Push to GitHub & Trigger CI/CD Pipeline**
    - Every push to main will build and push the Docker image.

4. Docker Swarm & Load Balancing

Goal: Deploy a scalable, load-balanced web app using Docker Swarm.

Tech Stack:

- Docker Swarm
- Nginx
- Flask/Node.js

Implementation Steps:

1. **Initialize Swarm**

```
docker swarm init
```

2. **Create docker-compose.yml for Swarm**

```
yaml
CopyEdit
version: '3.8'
services:
  web:
    image: myusername/myapp:latest
    deploy:
      replicas: 3
      restart_policy:
        condition: any
    ports:
      - "5000:5000"

  nginx:
    image: nginx
    ports:
```

```
    - "80:80"
  volumes:
    - ./nginx.conf:/etc/nginx/nginx.conf
```

3. **Deploy to Swarm**

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

4. **Verify Load Balancing**
    - Access http://localhost and see requests being balanced.

5. Kubernetes with Docker

Goal: Orchestrate Docker containers using Kubernetes.

Tech Stack:

- Kubernetes
- Flask/Node.js
- Minikube

Implementation Steps:

1. **Start Minikube**

```
minikube start
```

2. **Create a Kubernetes Deployment (deployment.yaml)**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web-app
```

```
        image: myusername/myapp:latest
        ports:
         - containerPort: 5000
```

3. **Create a Service (service.yaml)**

```
apiVersion: v1
kind: Service
metadata:
  name: web-service
spec:
  selector:
    app: web
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
  type: LoadBalancer
```

4. **Deploy to Kubernetes**

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

5. **Verify Deployment**

```
kubectl get pods
kubectl get services
```

6. **Access the App**
    - Run minikube service web-service to get the URL.

Final Thoughts

These projects cover all major Docker concepts:

- ✅ **Basic Containerization**
- ✅ **Multi-Container Applications**
- ✅ **CI/CD Automation**
- ✅ **Swarm for Scaling**
- ✅ **Kubernetes for Orchestration**

By the end of these projects, you'll be confident in real-world Docker deployments! 🚀

6. Monitoring and Logging with Docker (Advanced)

Goal: Set up monitoring and logging for Docker containers using Prometheus, Grafana, and the ELK stack (Elasticsearch, Logstash, Kibana).

Tech Stack:

- **Prometheus** (Monitoring)
- **Grafana** (Visualization)
- **ELK Stack** (Logging)
- **Docker Compose**

Implementation Steps:

Step 1: Set Up a Sample Web Application

- Create a simple Flask or Node.js application that logs messages.

Flask Example (app.py)

```
from flask import Flask
import logging

app = Flask(__name__)

# Configure logging
logging.basicConfig(filename='app.log', level=logging.INFO)

@app.route("/")
def home():
    app.logger.info("Home route accessed")
    return "Monitoring Docker App"
```

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

- Create a requirements.txt file:

```
Flask
```

- Create a Dockerfile:

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

Step 2: Set Up docker-compose.yml with Prometheus, Grafana, and ELK Stack

Create a docker-compose.yml file:

```
version: '3.8'
services:
  web:
    build: .
    ports:
      - "5000:5000"

  prometheus:
    image: prom/prometheus
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
    ports:
```

```
    - "9090:9090"

grafana:
  image: grafana/grafana
  ports:
    - "3000:3000"

elasticsearch:
  image: docker.elastic.co/elasticsearch/elasticsearch:7.10.1
  environment:
    - discovery.type=single-node
  ports:
    - "9200:9200"

logstash:
  image: docker.elastic.co/logstash/logstash:7.10.1
  volumes:
    - ./logstash.conf:/usr/share/logstash/pipeline/logstash.conf
  depends_on:
    - elasticsearch

kibana:
  image: docker.elastic.co/kibana/kibana:7.10.1
  ports:
    - "5601:5601"
  depends_on:
```

```
    - elasticsearch
```

Step 3: Configure Prometheus for Monitoring

Create a prometheus.yml file:

```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'docker-metrics'
    static_configs:
      - targets: ['web:5000']
```

Step 4: Configure Logstash for Logging

Create a logstash.conf file:

```
input {
  file {
    path => "/var/log/app.log"
    start_position => "beginning"
  }
}

output {
  elasticsearch {
    hosts => ["elasticsearch:9200"]
    index => "docker-logs"
  }
```

```
}
```

Step 5: Run the Services

Run the entire setup using:

```
docker-compose up --build
```

Step 6: Access the Monitoring & Logging Dashboards

- **Flask App:** http://localhost:5000
- **Prometheus UI:** http://localhost:9090
- **Grafana UI:** http://localhost:3000
- **Kibana UI:** http://localhost:5601

Step 7: Visualize Logs & Metrics

- **Grafana:**
  - Add Prometheus as a data source.
  - Create dashboards for container metrics.
- **Kibana:**
  - View logs stored in Elasticsearch.
  - Create visualizations for container logs.

Final Outcome

✅ **Real-time monitoring of container metrics with Prometheus & Grafana**
✅ **Centralized logging using ELK stack (Elasticsearch, Logstash, Kibana)**
✅ **Hands-on experience with observability tools in Docker environments**

This project will teach you **production-grade monitoring and logging** for Dockerized applications! 🚀

7. Secure Docker Deployment with Traefik and Let's Encrypt

Goal: Deploy a secure, production-ready Docker application with **Traefik (reverse proxy), SSL (Let's Encrypt), and authentication**.

Tech Stack:

- **Docker & Docker Compose**
- **Traefik (Reverse Proxy & Load Balancer)**
- **Let's Encrypt (SSL Certificates)**
- **Basic Authentication**

Implementation Steps:

Step 1: Set Up a Sample Web Application

Create a simple **Flask** or **Node.js** app (app.py for Flask):

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def home():
    return "Secure Docker Deployment with Traefik!"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

- Create a requirements.txt file:

```
Flask
```

- Create a Dockerfile:

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

Step 2: Configure Traefik as a Reverse Proxy

Create a traefik.yml configuration file:

```
global:
  checkNewVersion: true
  sendAnonymousUsage: false

entryPoints:
  web:
    address: ":80"
  websecure:
    address: ":443"

providers:
  docker:
    exposedByDefault: false

certificatesResolvers:
  myresolver:
    acme:
```

```
    email: your-email@example.com
    storage: acme.json
    httpChallenge:
      entryPoint: web
```

NOTE: Change your-email@example.com to your actual email to request an SSL certificate.

Step 3: Create a Secure docker-compose.yml Setup

```
version: '3.8'
services:
  traefik:
    image: traefik:v2.9
    command:
      - "--api.insecure=true"
      - "--providers.docker=true"
      - "--entrypoints.web.address=:80"
      - "--entrypoints.websecure.address=:443"
      - "--certificatesresolvers.myresolver.acme.email=your-email@example.com"
      - "--certificatesresolvers.myresolver.acme.storage=/letsencrypt/acme.json"
      - "--certificatesresolvers.myresolver.acme.httpChallenge.entryPoint=web"
    ports:
      - "80:80"
      - "443:443"
      - "8080:8080"
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
      - "./letsencrypt:/letsencrypt"
```

```
  web:
    build: .
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.web.rule=Host(`yourdomain.com`)"
      - "traefik.http.routers.web.entrypoints=websecure"
      - "traefik.http.routers.web.tls.certresolver=myresolver"
      -
"traefik.http.middlewares.auth.basicauth.users=admin:$$apr1$$yKXHkH7y$$O
tlv1bqxlFbT8Kl0KzmkQ/" # Replace with your hashed password
    networks:
      - webnet

networks:
  webnet:
```

IMPORTANT: Replace yourdomain.com with your actual domain.
Generate a **Basic Auth Password** using:

```
echo $(htpasswd -nb admin mypassword) | sed -e s/\\$/\\$\\$/g
```

Then, replace the hashed password in the traefik.http.middlewares.auth.basicauth.users label.

Step 4: Run the Secure Deployment

Start the services:

```
docker-compose up --build -d
```

Step 5: Verify SSL & Authentication

1. Open https://yourdomain.com.
2. Check the **SSL certificate** (issued by Let's Encrypt).
3. A **login prompt** should appear (Basic Auth).

Final Outcome

✅ **Secure HTTPS deployment with Let's Encrypt SSL**
✅ **Reverse proxy with Traefik for managing multiple services**
✅ **Basic Authentication for added security**

This project ensures **a production-ready, secure Docker deployment**! 🚀

8. Scalable Microservices Architecture with Docker and Kubernetes

Goal: Deploy a **microservices-based** architecture using **Docker, Kubernetes, and an API Gateway** for efficient scaling and management.

Tech Stack:

- **Docker** (Containerization)
- **Kubernetes** (Orchestration)
- **NGINX or Traefik** (API Gateway)
- **Flask/Node.js** (Microservices)
- **MongoDB/PostgreSQL** (Database)

Implementation Steps:

Step 1: Define the Microservices

Let's create three microservices:

1. **User Service** (Handles user authentication)
2. **Product Service** (Manages product information)
3. **Order Service** (Processes orders)

Example **User Service** (user_service/app.py):

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/users", methods=["GET"])
def get_users():
    return jsonify({"users": ["Alice", "Bob", "Charlie"]})

if __name__ == "__main__":
```

```
app.run(host="0.0.0.0", port=5001)
```

Create a requirements.txt:

```
Flask
```

Create a Dockerfile for the service:

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

Repeat this process for **Product** and **Order** services, changing the API endpoints accordingly.

Step 2: Define Kubernetes Deployment for Each Microservice

Create k8s/user-deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-service
spec:
  replicas: 3
  selector:
    matchLabels:
      app: user-service
  template:
    metadata:
      labels:
```

```
      app: user-service
  spec:
    containers:
     - name: user-service
       image: myusername/user-service:latest
       ports:
        - containerPort: 5001
---
apiVersion: v1
kind: Service
metadata:
 name: user-service
spec:
 selector:
  app: user-service
 ports:
  - protocol: TCP
    port: 80
    targetPort: 5001
 type: ClusterIP
```

Repeat this for product-deployment.yaml and order-deployment.yaml, changing the ports accordingly.

Step 3: Set Up API Gateway (NGINX Ingress Controller)

Create k8s/api-gateway.yaml:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
```

```
metadata:
  name: api-gateway
spec:
  rules:
    - host: myapp.local
      http:
        paths:
          - path: /users
            pathType: Prefix
            backend:
              service:
                name: user-service
                port:
                  number: 80
          - path: /products
            pathType: Prefix
            backend:
              service:
                name: product-service
                port:
                  number: 80
          - path: /orders
            pathType: Prefix
            backend:
              service:
                name: order-service
```

```
      port:
        number: 80
```

Step 4: Deploy Services to Kubernetes

1. Start Minikube:

```
minikube start
```

2. Deploy each microservice:

```
bash
CopyEdit
kubectl apply -f k8s/user-deployment.yaml
kubectl apply -f k8s/product-deployment.yaml
kubectl apply -f k8s/order-deployment.yaml
```

3. Deploy the API Gateway:

```
bash
CopyEdit
kubectl apply -f k8s/api-gateway.yaml
```

Step 5: Access the Microservices

Get the Minikube IP:

```
minikube ip
```

Then test the API endpoints:

```
perl
CopyEdit
curl http://<minikube-ip>/users
curl http://<minikube-ip>/products
curl http://<minikube-ip>/orders
```

Final Outcome

- ✅ **Fully containerized microservices running on Kubernetes**
- ✅ **API Gateway for service routing**
- ✅ **Scalability with Kubernetes deployments**

This project gives you **hands-on experience with Kubernetes for managing microservices**! 🚀

9. Continuous Integration & Deployment (CI/CD) for Dockerized Applications

Goal: Automate the **building, testing, and deployment** of a Dockerized application using **GitHub Actions & Jenkins** with Docker and Kubernetes.

Tech Stack:

- **Docker** (Containerization)
- **GitHub Actions / Jenkins** (CI/CD Automation)
- **Kubernetes / Docker Compose** (Deployment)
- **Flask/Node.js** (Sample Application)

Implementation Steps:

Step 1: Create a Sample Web Application

Let's use a simple **Flask** app (app.py):

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def home():
    return "Automated CI/CD with Docker!"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

Create a requirements.txt:

```
Flask
```

Create a Dockerfile:

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

Step 2: Set Up GitHub Actions for CI/CD

Create a .github/workflows/docker-ci.yml file:

```
name: CI/CD Pipeline

on:
  push:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Repository
        uses: actions/checkout@v2

      - name: Set up Docker Buildx
```

```
      uses: docker/setup-buildx-action@v2

    - name: Log in to Docker Hub
      run: echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u "${{
secrets.DOCKER_USERNAME }}" --password-stdin

    - name: Build and Push Docker Image
      run: |
        docker build -t myusername/myapp:latest .
        docker push myusername/myapp:latest

  deploy:
    needs: build
    runs-on: ubuntu-latest
    steps:
    - name: Deploy to Server
      uses: appleboy/ssh-action@v0.1.10
      with:
        host: ${{ secrets.SERVER_IP }}
        username: ${{ secrets.SERVER_USER }}
        key: ${{ secrets.SSH_PRIVATE_KEY }}
        script: |
          docker pull myusername/myapp:latest
          docker stop myapp || true
          docker rm myapp || true
          docker run -d -p 80:5000 --name myapp myusername/myapp:latest
```

NOTE: Replace myusername/myapp with your **Docker Hub repo name**.
Store your **Docker Hub credentials** & **server SSH key** as GitHub Secrets.

Step 3: Set Up Jenkins as an Alternative CI/CD Tool

If you prefer **Jenkins**, install plugins for **Docker & GitHub Webhooks** and configure a pipeline:

1. **Install Jenkins & Required Plugins**

```
sudo apt update && sudo apt install -y docker.io
sudo usermod -aG docker jenkins
```

2. **Create a Jenkins Pipeline** (Jenkinsfile):

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        sh 'docker build -t myusername/myapp:latest .'
      }
    }
    stage('Push') {
      steps {
        withDockerRegistry([credentialsId: 'docker-hub', url: '']) {
          sh 'docker push myusername/myapp:latest'
        }
      }
    }
    stage('Deploy') {
      steps {
```

```
            sh '''
            ssh user@server "docker pull myusername/myapp:latest && \
            docker stop myapp || true && \
            docker rm myapp || true && \
            docker run -d -p 80:5000 --name myapp myusername/myapp:latest"
            '''
        }
      }
    }
}
```

Add your **Docker Hub credentials** in Jenkins under **Manage Credentials**.

Step 4: Deploy Application to Kubernetes (Optional)

If using **Kubernetes**, deploy the app using:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 2
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
```

```
      app: myapp
  spec:
    containers:
     - name: myapp
       image: myusername/myapp:latest
       ports:
        - containerPort: 5000
---
apiVersion: v1
kind: Service
metadata:
 name: myapp
spec:
 selector:
  app: myapp
 ports:
  - protocol: TCP
    port: 80
    targetPort: 5000
 type: LoadBalancer
```

Apply the deployment:

bash

CopyEdit

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

Step 5: Test the CI/CD Pipeline

1. **Push code to GitHub** → Triggers CI/CD workflow.
2. **GitHub Actions/Jenkins builds and deploys the app**.
3. **Visit http://your-server-ip to see the deployed app**.

Final Outcome

✅ **Automated Docker image builds & deployments**
✅ **CI/CD with GitHub Actions OR Jenkins**
✅ **Seamless Kubernetes deployment (optional)**

This project **automates DevOps workflows** for Docker applications! 🚀

10. Multi-Stage Docker Build for Optimized Production Deployment

Goal: Use **multi-stage builds** to create a lightweight, optimized Docker image for a production-ready application.

Tech Stack:

- **Docker (Multi-Stage Builds)**
- **Nginx (Reverse Proxy)**
- **Flask/Node.js** (Sample Application)

Implementation Steps:

Step 1: Create a Sample Web Application

Let's use a simple **Flask** app (app.py):

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def home():
    return "Optimized Docker Multi-Stage Build!"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

Create a requirements.txt:

```
nginx
CopyEdit
Flask
```

```
gunicorn
```

Step 2: Create a Multi-Stage Dockerfile

Instead of creating a large image, we will:

✅ Use **one stage for building dependencies**

✅ Use **another stage for running the application**

```
# STAGE 1: Build Stage

FROM python:3.9 AS builder

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt


# STAGE 2: Production Stage

FROM python:3.9-slim

WORKDIR /app

COPY --from=builder /usr/local/lib/python3.9/site-packages /usr/local/lib/python3.9/site-packages

COPY . .

EXPOSE 5000

CMD ["gunicorn", "-b", "0.0.0.0:5000", "app:app"]
```

Why Multi-Stage Build?

- The **builder stage** installs dependencies **without bloating** the final image.
- The **final image** is lightweight, using only necessary files.

Step 3: Create an Nginx Reverse Proxy for Better Performance

Create an nginx.conf file:

```
server {
    listen 80;
    location / {
        proxy_pass http://web:5000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

Step 4: Set Up docker-compose.yml for Multi-Container Deployment

```
version: '3.8'

services:
  web:
    build: .
    container_name: web_app
    ports:
      - "5000:5000"

  nginx:
    image: nginx:latest
    container_name: nginx_proxy
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    ports:
      - "80:80"
```

```
    depends_on:
      - web
```

Step 5: Build and Run the Application

```
docker-compose up --build -d
```

Step 6: Verify Optimization

Check image size:

bash

```
docker images
```

A multi-stage build **reduces the final image size** significantly!

Final Outcome

✅ **Optimized production-ready Docker images**
✅ **Lightweight, efficient deployment using multi-stage builds**
✅ **NGINX reverse proxy for better performance**

This project **teaches advanced Docker image optimization techniques**!

Conclusion

Mastering Docker requires a hands-on approach, and the **10 projects outlined in this document** provide a structured learning path that takes you from **basic containerization to advanced deployment and orchestration**. By working through these projects, you have gained experience with:

✅ **Containerizing applications** using Docker and Docker Compose. ✅ **Managing multi-container environments** and networking between services. ✅ **Ensuring data persistence** and securing deployments. ✅ **Scaling applications efficiently** with Docker Swarm and Kubernetes. ✅ **Automating workflows** using CI/CD pipelines with GitHub Actions or Jenkins. ✅ **Optimizing production deployments** using multi-stage builds and reverse proxies.

With this knowledge, you are well-equipped to handle real-world challenges in **DevOps, cloud computing, and software deployment**. The ability to deploy scalable, efficient, and secure applications is a critical skill in today's tech industry. Continue experimenting, explore new Docker features, and integrate them into your projects to further refine your expertise.

By applying these skills, you can confidently **build and deploy containerized applications**, making you a valuable asset in the world of modern software development. 🚀