MEAN Stack CRUD Application

DevOps Deployment Documentation

Deployed on AWS EC2 with Docker, Nginx & CI/CD

---

1. Introduction

This document describes the implementation, containerization, cloud deployment, and automation of a full-stack MEAN (MongoDB, Express, Angular, Node.js) CRUD application using DevOps best practices.

The primary objective of this project was to demonstrate practical knowledge in:

- Docker containerization
- Multi-container orchestration using Docker Compose
- Cloud deployment on AWS EC2
- Reverse proxy configuration using Nginx
- CI/CD automation using GitHub Actions

This project follows a production-level deployment architecture.

---

2. Project Objectives

The objectives of this assignment were:

1. Containerize backend and frontend applications.
2. Use Docker Compose to orchestrate multiple services.
3. Deploy the application on a cloud virtual machine.
4. Configure Nginx as a reverse proxy.
5. Implement a CI/CD pipeline to automate deployment.
6. Ensure secure and production-ready architecture.

---

3. Technology Stack

Frontend:

- Angular

Backend:

- Node.js
- Express.js

Database:

- MongoDB

DevOps Tools:

- Docker
- Docker Compose
- AWS EC2
- Nginx
- GitHub Actions

Container Registry:

- Docker Hub

---

4. Application Architecture

The deployed architecture follows this flow:

Internet
↓
AWS Security Group
↓
Nginx (Port 80)
↓
Frontend Container (Angular)
↓
Backend Container (Node.js / Express)
↓
MongoDB Container

Explanation:

- Nginx listens on port 80 and handles incoming HTTP requests.
- Requests to / are routed to the frontend container.
- Requests to /api are routed to the backend container.
- Backend communicates internally with MongoDB container.
- MongoDB is not exposed to the public network.

This architecture ensures proper separation of concerns and security.

---

5. Docker Implementation

5.1 Backend Dockerfile

The backend Dockerfile performs the following steps:

- Uses Node 18 base image.
- Sets working directory.
- Copies package.json.
- Installs dependencies.
- Copies application source code.
- Exposes port 3000.
- Starts the application using npm start.

This ensures the backend runs inside an isolated container.

5.2 Frontend Dockerfile

The frontend Dockerfile uses multi-stage build:

Stage 1:

- Uses Node image.
- Installs Angular dependencies.
- Builds production-ready Angular app.

Stage 2:

- Uses Nginx base image.
- Copies built Angular files into Nginx directory.
- Exposes port 80.

This reduces image size and improves performance.

---

6. Docker Compose Configuration

Docker Compose was used to manage three services:

- mongo
- backend
- frontend

Key features:

- Services communicate using Docker network.
- MongoDB uses a persistent volume.
- Backend depends on mongo.
- Frontend depends on backend.
- Restart policies enabled for resilience.

Environment variable used:

MONGO_URI=mongodb://mongo:27017/ddtask

This ensures backend connects to MongoDB container internally.

---

7. AWS EC2 Deployment

7.1 EC2 Configuration

- Instance Type: t2.micro
- OS: Ubuntu 22.04 LTS

- Key-based SSH authentication
- Security Group configuration

Open Ports:

- 22 (SSH)
- 80 (HTTP)

Ports 3000, 4200, and 27017 were not exposed publicly in production.

7.2 Deployment Steps

1. Installed Docker and Docker Compose on EC2.
2. Logged into Docker Hub.
3. Pulled latest backend and frontend images.
4. Ran docker-compose up -d.
5. Verified containers using docker ps.
6. Installed and configured Nginx.
7. Configured reverse proxy.
8. Restarted Nginx.
9. Verified application through public IP.

Application is accessible at:

http://107.20.44.254/

---

8. Nginx Reverse Proxy Configuration

Nginx was configured to:

- Listen on port 80.
- Route frontend requests to port 4200.
- Route API requests to port 3000.

Configuration example:

```
server { listen 80;

location / {

   proxy_pass http://localhost:4200;

}


location /api/ {

   proxy_pass http://localhost:3000/;

}

}
```

This ensures a clean production URL without exposing internal ports.

---

9. CI/CD Pipeline Implementation

GitHub Actions was used to automate deployment.

Trigger:

- On push to main branch.

Pipeline Steps:

1. Checkout repository.
2. Login to Docker Hub.
3. Build backend Docker image.
4. Push backend image to Docker Hub.
5. Build frontend Docker image.
6. Push frontend image to Docker Hub.
7. SSH into EC2.
8. Pull latest images.

9. Restart containers.

Benefits:

- Fully automated deployment.
- No manual intervention required.
- Production-ready DevOps workflow.

---

10. Security Considerations

- MongoDB not exposed publicly.
- Only port 80 exposed in production.
- SSH restricted via key-based authentication.
- Containers use restart policy.
- Environment variables used for sensitive configuration.

---

11. Challenges Faced

1. Nginx serving default page instead of project.
2. SSL errors due to HTTP/HTTPS confusion.
3. Security group misconfiguration.
4. Docker image permission issues.
5. Memory constraints during Angular build.

All issues were resolved using systematic debugging.

---

12. Production Best Practices Followed

- Multi-stage Docker builds.
- Image registry usage (Docker Hub).
- Reverse proxy instead of exposing internal ports.

- CI/CD automation.
- Cloud-based deployment.
- Separation of application layers.

---

13. Conclusion

This project successfully demonstrates real-world DevOps implementation by combining:

- Application containerization
- Cloud infrastructure deployment
- Reverse proxy configuration
- CI/CD automation

The deployment architecture follows production-level standards and reflects practical DevOps workflow.

---

14. Future Improvements

- Add HTTPS using Let’s Encrypt.
- Add custom domain mapping.
- Implement monitoring (Prometheus/Grafana).
- Add container security scanning.
- Use AWS Load Balancer for scalability.
- Deploy using Kubernetes for high availability.