WorkZone — Docker Report

Project: WorkZone — Student Part-Time Job Platform
Report Type: Dockerization & Containerisation
Prepared by: Anjana Madhushan (DevOps Engineer)
Team Members: Nimantha Vikum Kodikara (Frontend), Avishka Devananda (Backend)
Date: March 2026

Table of Contents

---

1. Executive Summary

This report documents the complete containerisation of the **WorkZone** platform using **Docker** and **Docker Compose**. WorkZone is a full-stack web application composed of three main services:

Service	Technology	Docker Image
Frontend	React 19, Vite, Nginx	Custom (multi-stage build)
Backend	Node.js 20, Express 5	Custom (multi-stage build)
Database	MongoDB 7.0	Official `mongo:7.0`

By containerising the application, the team achieves **environment consistency**, **one-command deployment**, and **clean service isolation**, eliminating the "works on my machine" problem entirely.

---

2. Project Overview

What is WorkZone?

WorkZone is a web-based job platform created to connect **students** and **companies** directly for part-time employment — removing brokers and middlemen that reduce student earnings.

Key Application Features

Feature	Description
Authentication	Email/password registration + Google OAuth 2.0 Sign-In
Student Portal	Browse jobs, submit applications, manage profile
Company Dashboard	Post jobs, review applicants, manage listings
Responsive UI	Mobile-first Tailwind CSS design
Secure API	JWT-based session management, bcrypt password hashing

Technology Stack

Layer	Technologies
Frontend	React 19, React Router 7, Vite 7, Tailwind CSS 4, Axios, `lucide-react`
Backend	Node.js 20, Express 5, Mongoose 9, `jwt-simple`, `bcryptjs`, CORS
Database	MongoDB 7.0 (NoSQL, document-based)
Auth	JWT tokens, `google-auth-library` (OAuth 2.0)
DevOps	Docker, Docker Compose, GitHub Actions, Vercel (frontend), Render (backend)

---

3. Why Docker?

Problems Before Dockerisation

Problem	Impact
Manual environment setup	New developers spent hours configuring local environments
"Works on my machine"	Differences between dev/staging/production caused bugs
MongoDB version drift	Inconsistent database behaviour across machines
Complex multi-service start	Developers had to start MongoDB, backend, and frontend manually
Dependency conflicts	Different Node.js versions caused package incompatibilities

Benefits of Docker

Benefit	Detail
Consistency	Identical environment across all developer machines and servers
One-command startup	`docker compose up` starts all three services instantly
Isolation	Services cannot interfere with each other's dependencies
Portability	Runs on any OS: Linux, macOS, Windows
Scalability	Services can be scaled independently with `--scale`
Reproducibility	Pinned image versions ensure identical builds

Benefit	Detail
CI/CD Ready	Images can be built, tested, and pushed in automated pipelines

---

4. Architecture Diagram

```
                                      Docker Host Machine


                                       workzone-net (bridge)

  Browser / Client
    Port 80 (HTTP)                 workzone-
                                   frontend    React + Nginx
                                   (Port 80)

                                         /api/* → proxy_pass

  Developer / APIs            workzone-
    Port 5000                      backend     Node.js + Express
                                   (Port 5000)

                                         mongoose

                                   workzone-
                                   mongodb     MongoDB 7.0
                                   (Port 27017  (internal only)


                                         Volume Mount
                                    mongo_data
                                   (Named Volume)
```

Service Communication Flow

```
User Request (browser)


  [Nginx - Port 80]

           Static files (HTML/JS/CSS)      Serve from /usr/share/nginx/html

           /api/* requests           [Backend - Port 5000]


                                              [MongoDB - Port 27017]
                                              (internal network only)
```

5. Docker Images & Dockerfiles

5.1 Backend Image (Node.js + Express)

File: `backend/Dockerfile`

```dockerfile
# ---- Builder Stage ----
FROM node:20-alpine AS builder

WORKDIR /app

COPY package*.json ./
RUN npm ci --omit=dev

COPY . .

# ---- Production Stage ----
FROM node:20-alpine

WORKDIR /app

RUN addgroup -S appgroup && adduser -S appuser -G appgroup

COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/package.json ./package.json
COPY --from=builder /app/index.js ./index.js
COPY --from=builder /app/routes ./routes
COPY --from=builder /app/models ./models

USER appuser

EXPOSE 5000

CMD ["node", "index.js"]
```

Key Design Decisions

Decision	Reason
`node:20-alpine` base	Minimal image size (~55 MB vs ~1 GB for full Node image)
Multi-stage build	Builder stage has full toolchain; production stage is lean
`npm ci --omit=dev`	Installs only production dependencies, excluding dev tools
Non-root user (`appuser`)	Security best practice — prevents privilege escalation
Explicit file `COPY`	Only required source files are copied, reducing attack surface

Backend `.dockerignore`

```
node_modules
npm-debug.log
```

```
.env
*.log
.git
.gitignore
```

5.2 Frontend Image (React + Vite → Nginx)

File: `frontend/Dockerfile`

```dockerfile
# ---- Builder Stage ----
FROM node:20-alpine AS builder

WORKDIR /app

COPY package*.json ./
RUN npm ci

COPY . .
RUN npm run build

# ---- Production Stage ----
FROM nginx:stable-alpine

RUN rm /etc/nginx/conf.d/default.conf

COPY nginx.conf /etc/nginx/conf.d/default.conf
COPY --from=builder /app/dist /usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

Key Design Decisions

Decision	Reason
Two-stage build	Node.js is only needed during build; the final image contains only Nginx
`nginx:stable-alpine`	Ultra-small (~25 MB) production-grade web server
Custom `nginx.conf`	Enables SPA routing (React Router) and API proxying
API proxy in Nginx	Avoids CORS issues by routing `/api/*` through the same origin (port 80)
Static asset caching	Long-lived cache headers for versioned assets (performance)

Frontend `.dockerignore`

```
node_modules
npm-debug.log
dist
.env
*.log
.git
.gitignore
```

```
server {
    listen 80;
    server_name _;

    root /usr/share/nginx/html;
    index index.html;

    location /assets/ {
        expires 1y;
        add_header Cache-Control "public, immutable";
    }

    location /api/ {
        proxy_pass http://backend:5000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }

    location / {
        try_files $uri $uri/ /index.html;
    }
}
```

Nginx Configuration (`frontend/nginx.conf`)

> **SPA Routing:** The `try_files $uri $uri/ /index.html` directive ensures that deep-linking and browser refresh work correctly with React Router.

5.3 MongoDB Image

The official `mongo:7.0` image is used directly from Docker Hub — no custom Dockerfile is required.

Property	Value
Image	`mongo:7.0`
Exposed Port	`27017` (internal network only)
Initial Database	`workzone` (via env var)
Data Persistence	Named volume `mongo_data`
Health Check	`mongosh --eval "db.adminCommand('ping')"`

> **Security:** The MongoDB port is **not** exposed to the host machine. Only the backend service (within the Docker network) can connect to it.

---

6. Docker Compose Configuration

File: `docker-compose.yml`

Docker Compose orchestrates all three services with a single command. Below is a breakdown of each service configuration.

Services Summary

Service	Image	Host Port	Internal Port	Depends On
mongodb	mongo:7.0	—	27017	—
backend	Custom (built locally)	5000	5000	mongodb
frontend	Custom (built locally)	80	80	backend

Service Start Order & Health Checks

```
mongodb (health check: mongosh ping)

       backend (waits for mongodb healthy)

                 frontend (waits for backend to start)
```

The `depends_on` with `condition: service_healthy` for MongoDB ensures the backend never attempts a database connection before MongoDB is ready.

Complete `docker-compose.yml`

```yaml
version: "3.9"

services:
  mongodb:
    image: mongo:7.0
    container_name: workzone-mongodb
    restart: unless-stopped
    environment:
      MONGO_INITDB_DATABASE: workzone
    volumes:
      - mongo_data:/data/db
    networks:
      - workzone-net
    healthcheck:
      test: ["CMD", "mongosh", "--eval", "db.adminCommand('ping')"]
      interval: 10s
      timeout: 5s
      retries: 5

  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    container_name: workzone-backend
    restart: unless-stopped
    depends_on:
      mongodb:
        condition: service_healthy
    environment:
      PORT: 5000
      MONGO_URI: mongodb://mongodb:27017/workzone
      JWT_SECRET: ${JWT_SECRET:?JWT_SECRET must be set in .env or environment}
      GOOGLE_CLIENT_ID: ${GOOGLE_CLIENT_ID:-}
      FRONTEND_URL: ${FRONTEND_URL:-http://localhost}
      NODE_ENV: ${NODE_ENV:-production}
    ports:
      - "5000:5000"
    networks:
      - workzone-net

  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    container_name: workzone-frontend
    restart: unless-stopped
    depends_on:
      - backend
    ports:
      - "80:80"
    networks:
      - workzone-net

volumes:
  mongo_data:

networks:
  workzone-net:
    driver: bridge
```

---

7. Networking

All services share a single **bridge network** called `workzone-net`.

Bridge Network Properties

Property	Value
Name	`workzone-net`
Driver	`bridge`
Scope	Local (single host)

Service Discovery

Within the `workzone-net` network, each container is reachable by its **service name** as a hostname:

From Service	To Service	Hostname Used	Port
`backend`	`mongodb`	`mongodb`	27017
`frontend`	`backend`	`backend` (via Nginx)	5000

This DNS-based service discovery is built into Docker Compose and requires no additional configuration.

Port Exposure Strategy

Service	Host Exposure	Reason
Frontend	`0.0.0.0:80`	Public-facing — accessible by end users
Backend	`0.0.0.0:5000`	Exposed for direct API testing during development
MongoDB	Not exposed	Database is internal-only for security

---

8. Volume Management & Data Persistence

Named Volume: `mongo_data`

Property	Value
Volume Name	`mongo_data`
Mount Path	`/data/db` in container
Driver	`local` (default)
Managed by	Docker Engine

The named volume `mongo_data` persists all MongoDB data **independently of the container lifecycle**. This means:

- Stopping or removing the `mongodb` container does **not** delete data
- Recreating containers with `docker compose up --build` retains existing data
- Data is stored on the host machine under Docker's volume directory

Volume Commands

```
# List all volumes
docker volume ls

# Inspect the workzone data volume
docker volume inspect workzone_mongo_data

# Remove volume (  deletes all data)
docker volume rm workzone_mongo_data
```

---

9. Environment Variables & Configuration

All sensitive configuration is managed through environment variables, never hardcoded.

Backend Environment Variables

Variable	Default Value	Description
`PORT`	`5000`	Port the Express server listens on
`MONGO_URI`	`mongodb://mongodb:27017/workzone`	MongoDB connection string
`JWT_SECRET`	**Required — no default**	Secret key for signing JWT tokens
`GOOGLE_CLIENT_ID`	*(empty)*	Google OAuth 2.0 Client ID
`FRONTEND_URL`	`http://localhost`	Allowed CORS origin for frontend
`NODE_ENV`	`production`	Node.js environment

Setting Up Environment Variables

Create a `.env` file in the project root to override defaults:

```
# .env (root level - used by docker-compose)
JWT_SECRET=your-strong-random-secret-key-here
GOOGLE_CLIENT_ID=your-google-oauth-client-id.apps.googleusercontent.com
FRONTEND_URL=http://localhost
NODE_ENV=production
```

> **Never commit `.env` files to version control.** The `.gitignore` already excludes `.env` files.

Backend `.env.example`

```
PORT=3001
MONGO_URI=mongodb://localhost:27017/workzone
JWT_SECRET=your-secret-key-change-in-production
GOOGLE_CLIENT_ID=your-google-client-id-from-console
FRONTEND_URL=https://your-frontend-domain.vercel.app
NODE_ENV=development
```

---

10. Security Considerations

Implemented Security Measures

Measure	Implementation
Non-root container user	Backend runs as `appuser` (not `root`)
MongoDB not exposed to host	Port 27017 is internal-only within Docker network
Environment variables for secrets	JWT secret and OAuth credentials are never hardcoded
`.dockerignore` files	Prevents `.env`, logs, and `node_modules` from entering images
Multi-stage builds	Build tools (npm, compilers) are excluded from production images
Alpine base images	Minimal OS surface area reduces vulnerability exposure
Password hashing	User passwords hashed with `bcryptjs` (salt rounds: 10)
JWT token expiry	Tokens expire after 7 days

Remaining Security Recommendations

Recommendation	Description
MongoDB authentication	Enable MongoDB user/password auth for production deployments
HTTPS/TLS	Place a reverse proxy (e.g., Traefik, Nginx with SSL) in front
Rate limiting	Add `express-rate-limit` middleware to auth endpoints
Docker secrets	Use Docker Swarm secrets or Kubernetes secrets for production
Image scanning	Integrate Trivy or Snyk into CI pipeline to scan for CVEs

---

11. Build & Deployment Instructions

Prerequisites

Tool	Version Required	Installation
Docker Engine	24.x or higher	https://docs.docker.com/get-docker/
Docker Compose	2.x or higher	Included with Docker Desktop
Git	Any recent	https://git-scm.com

Step 1: Clone the Repository

```
git clone https://github.com/AnjanaMadhushanaj/WorkZone.git
cd WorkZone
```

Step 2: Configure Environment Variables

```
# Copy the example and fill in real values
cp backend/.env.example .env
# Edit .env with your preferred editor
nano .env
```

> **Required:** `JWT_SECRET` **must** be set before starting the stack. Docker Compose will refuse to start if it is missing. Use a strong, random value (e.g., `openssl rand -hex 32`).

Step 3: Build and Start All Services

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

This single command will: 1. Build the backend Node.js image 2. Build the frontend React/Nginx image 3. Pull the official `mongo:7.0` image 4. Start all three containers in the correct order 5. Stream logs from all services to the terminal

Step 4: Access the Application

Service	URL
Frontend (Web)	http://localhost
Backend API	http://localhost:5000

Service	URL
API Health Check	http://localhost:5000/

Other Useful Commands

```
# Start in detached (background) mode
docker compose up -d

# View running containers
docker compose ps

# View logs for a specific service
docker compose logs backend
docker compose logs frontend
docker compose logs mongodb

# Follow logs in real time
docker compose logs -f backend

# Stop all services (keeps volumes)
docker compose down

# Stop and remove volumes (  deletes database data)
docker compose down -v

# Rebuild only the backend image
docker compose build backend

# Restart a single service
docker compose restart backend

# Scale a service (e.g., 3 backend instances)
docker compose up --scale backend=3
```

Development vs Production

Aspect	Development	Production (Docker)
Frontend serving	`vite dev` (HMR, fast refresh)	Nginx static file server
Backend start	`nodemon` (auto-restart)	`node index.js` (stable)
MongoDB	Local install or Atlas cloud	Docker container with named volume
API base URL	`http://localhost:5000`	`/api/` (proxied through Nginx)
Environment	`.env` file loaded by dotenv	Environment vars from `docker-compose`

---

12. CI/CD Integration

Current CI/CD Pipelines

The project uses **GitHub Actions** for automation.

CI Pipeline (`.github/workflows/ci.yml`) Triggers on Pull Requests to `main` or `develop`:

```
jobs:
  frontend-ci:    # npm install + vite build
  backend-ci:     # npm install + node index.js (10s timeout)
```

CD Pipelines

Workflow	Trigger	Target
`cd-frontend.yml`	Push to `main`	Vercel
`cd-backend.yml`	Push to `main`	Render

Recommended Docker Integration for CI/CD

The following additions to the CI pipeline would enable Docker image building and pushing:

```
# Example: Add to ci.yml for Docker build validation
- name: Build Docker Images
  run: docker compose build

# Example: Push to GitHub Container Registry (ghcr.io)
- name: Log in to GHCR
  uses: docker/login-action@v3
  with:
    registry: ghcr.io
    username: ${{ github.actor }}
    password: ${{ secrets.GITHUB_TOKEN }}

- name: Build and push backend image
  uses: docker/build-push-action@v5
  with:
    context: ./backend
    push: true
    tags: ghcr.io/anjanamadhushanaj/workzone-backend:latest

- name: Build and push frontend image
  uses: docker/build-push-action@v5
  with:
    context: ./frontend
    push: true
    tags: ghcr.io/anjanamadhushanaj/workzone-frontend:latest
```

13. Benefits Achieved

Image Size Comparison

Service	Without Multi-stage	With Multi-stage	Savings
Backend	~900 MB (node:20)	~150 MB (alpine)	~83%
Frontend	~900 MB (node:20)	~35 MB (nginx:alpine)	~96%
MongoDB	N/A	~800 MB (mongo:7.0)	Official

Developer Experience Improvements

Before Docker	After Docker
Install Node.js, MongoDB manually	`docker compose up --build` — done
Different Node versions per developer	All use Node 20 (pinned in Dockerfile)
Start 3 separate terminals	Single command starts everything
CORS configuration varies locally	Nginx proxy eliminates CORS for `/api/*`
MongoDB data lost on reinstall	Data persists in named Docker volume

---

14. Challenges & Solutions

Challenge	Solution
MongoDB not ready when backend starts	`healthcheck` + `depends_on: condition: service_healthy`
CORS errors in containerised setup	Nginx `/api/` proxy forwards requests to backend on same origin
Frontend React Router deep-link 404s	Nginx `try_files` fallback serves `index.html` for all routes
Secrets in environment variables	`.env` file with `.gitignore` exclusion; Docker env vars
Large Node.js images	Multi-stage builds with Alpine base images
Development vs production parity	Separate dev commands (`npm run dev`) retained; Docker is for prod

---

15. Future Improvements

Improvement	Description
Docker Swarm / Kubernetes	Orchestrate for high-availability production deployments
HTTPS with Let's Encrypt	Add Traefik or Certbot for automatic TLS certificates
MongoDB authentication	Enable `MONGO_INITDB_ROOT_USERNAME` and `MONGO_INITDB_ROOT_PASSWORD`
Environment-specific Compose	`docker-compose.dev.yml` with volume mounts for live code reloading
Image vulnerability scanning	Integrate Trivy in CI pipeline to scan for known CVEs
Container registry	Push tagged images to GHCR or Docker Hub on every release
Resource limits	Add `mem_limit` and `cpus` constraints per service in Compose

Improvement	Description
Log aggregation	Add centralized logging (ELK stack or Loki/Grafana)

---

16. Conclusion

The WorkZone platform has been successfully containerised using Docker and Docker Compose. The containerisation covers all three application services — **MongoDB**, **Node.js/Express backend**, and **React/Nginx frontend** — with production-grade practices including:

- **Multi-stage Dockerfiles** for minimal, secure images
- **Docker Compose** for one-command orchestration
- **Named volumes** for persistent database storage
- **Bridge networking** with internal-only MongoDB access
- **Health checks** for reliable service startup ordering
- **Non-root container users** for backend security
- **Nginx API proxy** for CORS-free frontend-backend communication
- **Environment variable management** with `.env` support

Any developer can now clone the repository and have a fully functional, production-equivalent WorkZone environment running locally with a single command:

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

WorkZone Docker Report — Systems Administration & Maintenance Assignment (2026)
Prepared by Anjana Madhushan (DevOps Engineer), Team: Nimantha Vikum Kodikara, Avishka Devananda