BOOTS ON THE GROUND AI

AI Solutions for Small Business Owners

DOCKER PLAYBOOK

Never Build Without Containers Again

Your complete guide to Docker for Next.js & React web apps.
Rules, checklists, templates, and workflows that ensure
every build is consistent, portable, and production-ready.

Built for builders who want it done right, every time.

D

DOCKER

BOOTS APPROVED

Version 1.0 | February 12, 2026
Boots On The Ground AI | bootsagentai.com

TABLE OF CONTENTS

Your roadmap to Docker mastery

01 WHY DOCKER?

The Boots Philosophy on Building Right

If you've ever said **'but it works on my machine'**, Docker is your answer. At Boots On The Ground AI, we believe every build should be predictable, portable, and production-ready from day one.

BOOTS SAYS:

BOOTS SAYS: Think of Docker like a shipping container for your code. No matter what truck (server) carries it, everything inside stays exactly the same. Your app, its dependencies, its config - all locked in and ready to deploy anywhere.

Without Docker, you face:

X "Works on my machine" syndrome - code breaks on the server
X Dependency conflicts - different Node versions, package mismatches
X Inconsistent environments - dev looks nothing like production
X Painful onboarding - new devs spend days setting up
X Deployment roulette - every deploy is a gamble

With Docker, you get:

+ Identical environments everywhere - dev, staging, production
+ One command startup - docker compose up and you are running
+ Dependency isolation - each app gets exactly what it needs
+ Easy scaling - need more? Spin up another container
+ Reliable deploys - if it runs locally, it runs in production

THE BOOTS STANDARD:

THE BOOTS STANDARD: Every project at Boots On The Ground AI ships with a Dockerfile and docker-compose.yml. No exceptions. If it does not have a container, it does not leave the shop.

02 THE 10 GOLDEN RULES OF DOCKER

Memorize these. Live by them. Never break them.

1 One Container, One Process

Each container runs ONE thing. Your Next.js app is one container. Your database is another. Your Redis cache is another. Never bundle multiple services into one container.

2 Always Use Multi-Stage Builds

Your Dockerfile should have a 'builder' stage and a 'runner' stage. Build your app in the first stage, copy only the production output to the second. This cuts image size by 60-80%.

3 Never Store Data in Containers

Containers are disposable. Use Docker volumes for databases, uploads, and any data that needs to persist. If a container dies, your data survives.

4 Pin Your Base Image Versions

Use node:20-alpine, NOT node:latest. Latest changes without warning. Pinned versions mean your build works the same today, tomorrow, and next year.

5 Use .dockerignore Like .gitignore

Always create a .dockerignore file. Exclude node_modules, .git, .env files, and build artifacts. This speeds up builds and keeps secrets out of images.

6 Environment Variables for Config

Never hardcode URLs, API keys, or ports. Use environment variables passed through docker-compose.yml or .env files. Different values for dev vs. production.

7 Health Checks Are Mandatory

Every container needs a HEALTHCHECK instruction. Docker uses this to know if your app is actually working, not just running. Auto-restart broken containers.

8 Tag Every Image You Build

Use semantic versioning: myapp:1.0.0, myapp:1.0.1. Never deploy 'latest' to production. Tags let you roll back instantly if something breaks.

9

Keep Images Small (Alpine First)

Start with Alpine-based images (node:20-alpine). They are 5x smaller than full images. Smaller images = faster deploys, less storage, reduced attack surface.

10

docker-compose.yml Is Your Blueprint

Your docker-compose.yml defines your entire application stack. Every service, every network, every volume. A new developer should be able to run 'docker compose up' and have everything working.

03 FILE STRUCTURE

Where everything lives in a Dockerized Next.js project

Every Dockerized project at Boots On The Ground AI follows this exact structure. No guessing, no improvising.

File / Folder	Purpose
`my-nextjs-app/`	
`Dockerfile`	Multi-stage build file (required)
`Dockerfile.dev`	Development-only Dockerfile (optional)
`docker-compose.yml`	Full stack definition (required)
`docker-compose.dev.yml`	Dev overrides (hot reload, debug)
`docker-compose.prod.yml`	Production overrides
`.dockerignore`	Files to exclude from Docker builds
`.env.example`	Template for env vars (committed to git)
`.env`	Actual env vars (NEVER commit this)
`package.json`	Node dependencies
`next.config.js`	Next.js configuration
`src/`	Your application source code
`public/`	Static assets
`nginx/`	Reverse proxy config (production)
`nginx.conf`	Nginx configuration file

WARNING:

CRITICAL: The .env file with real secrets must NEVER be committed to git or included in Docker images. Add it to both .gitignore and .dockerignore. Use .env.example as a template with placeholder values.

04 THE DOCKERFILE

Annotated production template for Next.js

Copy this template for every new project. Each line is explained.

STAGE 1: Dependencies

```
# Stage 1: Install dependencies only
FROM node:20-alpine AS deps

# Security: run as non-root user
RUN addgroup --system --gid 1001 nodejs
RUN adduser --system --uid 1001 nextjs

WORKDIR /app

# Copy package files first (cache optimization)
COPY package.json package-lock.json* ./

# Install dependencies
RUN npm ci --only=production
```

BOOTS EXPLAINS:

WHY npm ci? Unlike npm install, 'npm ci' uses the exact versions from package-lock.json. This means your container gets the identical dependencies every single time. No surprises.

STAGE 2: Build

```
# Stage 2: Build the application
FROM node:20-alpine AS builder
WORKDIR /app

# Copy deps from Stage 1
COPY --from=deps /app/node_modules ./node_modules
COPY . .

# Set build-time env vars
ARG NEXT_PUBLIC_API_URL
ENV NEXT_PUBLIC_API_URL=$NEXT_PUBLIC_API_URL

# Build Next.js
RUN npm run build
```

STAGE 3: Production Runner

```
# Stage 3: Production image (minimal)
FROM node:20-alpine AS runner
WORKDIR /app

ENV NODE_ENV=production

# Create non-root user
RUN addgroup --system --gid 1001 nodejs
RUN adduser --system --uid 1001 nextjs

# Copy only production files
COPY --from=builder /app/public ./public
COPY --from=builder /app/.next/standalone ./
COPY --from=builder /app/.next/static ./.next/static

# Run as non-root
USER nextjs

# Expose port
EXPOSE 3000
ENV PORT=3000
```

```
# Health check
HEALTHCHECK --interval=30s --timeout=3s --retries=3 \
CMD wget --no-verbose --tries=1 --spider \
http://localhost:3000/api/health || exit 1

# Start the app
CMD ["node", "server.js"]
```

SIZE SAVINGS:

MULTI-STAGE RESULT: The final image contains ONLY the compiled app and Node runtime. No source code, no dev dependencies, no build tools. Typical size: 150-200MB vs 1.2GB for a single-stage build.

05 DOCKER COMPOSE

Your entire application stack in one file

docker-compose.yml defines every service your app needs. One command starts everything.

Production docker-compose.yml

```
version: "3.8"


services:

# Your Next.js application

app:

build:

context: .

dockerfile: Dockerfile

args:

- NEXT_PUBLIC_API_URL=${API_URL}

container_name: boots-app

restart: unless-stopped

ports:

- "3000:3000"

environment:

- DATABASE_URL=${DATABASE_URL}

- REDIS_URL=redis://cache:6379

depends_on:

db:

condition: service_healthy

cache:

condition: service_started

networks:

- boots-network
```

```
# PostgreSQL database

db:

image: postgres:16-alpine

container_name: boots-db

restart: unless-stopped

volumes:

- postgres_data:/var/lib/postgresql/data

environment:

- POSTGRES_DB=${DB_NAME}

- POSTGRES_USER=${DB_USER}

- POSTGRES_PASSWORD=${DB_PASSWORD}

healthcheck:

test: ["CMD-SHELL", "pg_isready -U ${DB_USER}"]

interval: 10s

timeout: 5s

retries: 5

networks:

- boots-network


# Redis cache

cache:

image: redis:7-alpine

container_name: boots-cache

restart: unless-stopped

volumes:

- redis_data:/data

networks:

- boots-network


# Nginx reverse proxy (production)

nginx:

image: nginx:alpine
```

```
container_name: boots-nginx

restart: unless-stopped

ports:

- "80:80"

- "443:443"

volumes:

- ./nginx/nginx.conf:/etc/nginx/nginx.conf

depends_on:

- app

networks:

- boots-network


volumes:

postgres_data:

redis_data:


networks:

boots-network:

driver: bridge
```

BOOTS TIP:

BOOTS TIP: Use 'depends_on' with 'condition: service_healthy' so your app waits for the database to be READY, not just started. A started container is not the same as a ready database.

Development Override (docker-compose.dev.yml)

Run with: **docker compose -f docker-compose.yml -f docker-compose.dev.yml up**

```
version: "3.8"


services:

app:

build:

dockerfile: Dockerfile.dev

volumes:
```

```
# Hot reload: mount source code
- ./src:/app/src
- ./public:/app/public
# Exclude node_modules from mount
- /app/node_modules
environment:
- NODE_ENV=development
- NEXT_TELEMETRY_DISABLED=1
command: npm run dev
```

06 ENVIRONMENT VARIABLES & SECRETS

Keep your secrets safe, keep your config flexible

The Rules of Env Vars

NEVER hardcode secrets - No API keys, passwords, or tokens in code or Dockerfiles

Use .env files locally - docker compose reads .env automatically

Use .env.example as template - Commit this to git with placeholder values

Production uses real secrets - Use your cloud provider's secrets manager (AWS Secrets Manager, etc.)

NEXT_PUBLIC_ prefix - Only vars with this prefix are exposed to the browser in Next.js

.env.example (commit this)

```
# Database
DB_NAME=myapp
DB_USER=postgres
DB_PASSWORD=CHANGE_ME
DATABASE_URL=postgresql://${DB_USER}:${DB_PASSWORD}@db:5432/${DB_NAME}

# API
API_URL=http://localhost:3000
NEXT_PUBLIC_API_URL=http://localhost:3000

# Redis
REDIS_URL=redis://cache:6379

# Security
JWT_SECRET=CHANGE_ME
NEXTAUTH_SECRET=CHANGE_ME
NEXTAUTH_URL=http://localhost:3000
```

.dockerignore (required)

```
node_modules
.next
```

```
.git
.env
.env.local
.env.*.local
Dockerfile*
docker-compose*
README.md
.gitignore
.dockerignore
npm-debug.log*
coverage/
```

SECURITY ALERT:

SECURITY ALERT: If your .env file ends up in your Docker image, anyone with access to that image can extract your secrets. Always verify .env is in your .dockerignore. Run 'docker history' on your image to audit what is inside.

07 DEVELOPMENT WORKFLOW

Your daily Docker routine

Follow this workflow every day. It becomes muscle memory.

1. **Start Your Day**

   docker compose -f docker-compose.yml -f docker-compose.dev.yml up -d | This starts all services in the background with hot reload enabled.

2. **Write Code**

   Edit files normally in your editor (VS Code, etc.). | Changes auto-reload inside the container. No rebuild needed.

3. **Check Logs**

   docker compose logs -f app | Follow your app's logs in real time. Ctrl+C to stop watching.

4. **Run Tests**

   docker compose exec app npm test | Runs tests inside the container (same environment as production).

5. **Install New Package**

   docker compose exec app npm install some-package | Then rebuild: docker compose build app

6. **Stop for the Day**

   docker compose down | Stops all containers. Your data persists in volumes.

7. **Full Rebuild (when needed)**

   docker compose build --no-cache | Forces a fresh build. Use after major dependency changes.

BOOTS TIP:

BOOTS TIP: Create a Makefile or scripts in your package.json for these commands. Example: 'npm run docker:dev' instead of typing the full compose command every time.

08 PRODUCTION DEPLOYMENT CHECKLIST

Before you push the big button

Run through this checklist before EVERY production deployment. No shortcuts.

Image Build

- ☐ Dockerfile uses multi-stage build
- ☐ Base image is pinned (node:20-alpine, NOT node:latest)
- ☐ npm ci used instead of npm install
- ☐ Final stage runs as non-root user
- ☐ HEALTHCHECK instruction is present
- ☐ Image size is under 300MB
- ☐ .dockerignore excludes .env, node_modules, .git

Configuration

- ☐ All secrets use environment variables (no hardcoded values)
- ☐ NODE_ENV is set to 'production'
- ☐ Database connection string uses production credentials
- ☐ CORS origins are restricted to your domain
- ☐ Rate limiting is configured
- ☐ Logging is set to appropriate level

Security

- ☐ No secrets in Docker image layers (verify with docker history)
- ☐ Container runs as non-root user
- ☐ Unnecessary ports are NOT exposed
- ☐ SSL/TLS is configured (via nginx or cloud provider)
- ☐ Security headers are set in nginx config
- ☐ Docker image scanned for vulnerabilities

Data & Persistence

- ☐ Database uses a named volume (not a bind mount)
- ☐ Backup strategy is in place for volumes
- ☐ Database migrations run before app starts
- ☐ Redis persistence is configured if needed

Networking

- ☐ Services communicate via Docker network (not localhost)
- ☐ Only nginx/app ports are exposed to host
- ☐ Database port is NOT exposed to host
- ☐ DNS names use service names from docker-compose

09 COMMAND QUICK REFERENCE

The commands you will use daily

BUILDING

Command	What It Does
`docker compose build`	Build all services
`docker compose build app`	Build just the app service
`docker compose build --no-cache`	Force rebuild from scratch
`docker build -t myapp:1.0.0 .`	Build with a specific tag

RUNNING

Command	What It Does
`docker compose up`	Start all services (foreground)
`docker compose up -d`	Start all services (background)
`docker compose down`	Stop and remove containers
`docker compose restart app`	Restart just the app
`docker compose stop`	Stop without removing

DEBUGGING

Command	What It Does
`docker compose logs -f app`	Follow app logs
`docker compose logs --tail 100 app`	Last 100 log lines
`docker compose exec app sh`	Open shell inside container
`docker compose ps`	List running containers
`docker stats`	Live resource usage

CLEANUP

Command	What It Does
`docker system prune`	Remove unused containers/images

`docker volume prune`	Remove unused volumes
`docker image prune -a`	Remove all unused images
`docker compose down -v`	Stop and remove volumes too

INSPECTION

Command	What It Does
`docker compose config`	Validate compose file
`docker inspect boots-app`	Full container details
`docker history myapp:1.0.0`	Image layer history
`docker compose exec app env`	View env vars in container

10 TROUBLESHOOTING

When things go sideways

Container keeps restarting

Check logs: docker compose logs app. Usually a crash at startup - missing env var, bad database URL, or port conflict.

```
docker compose logs --tail 50 app
```

Port already in use

Another process is using port 3000 (or whatever port). Either stop that process or change the port mapping in docker-compose.yml.

```
lsof -i :3000 # Find what is using the port
```

Changes not showing up (dev mode)

Your volume mount might not be working. Check that your source folder is correctly mounted and that node_modules is excluded.

```
docker compose exec app ls -la /app/src
```

npm install fails in build

Usually a network issue or package-lock.json is out of sync. Delete package-lock.json, run npm install locally, then rebuild.

```
rm package-lock.json && npm install && docker compose build --no-cache
```

Out of disk space

Docker images and volumes accumulate. Prune unused resources to free space.

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

Cannot connect to database

Container networking issue. Make sure you use the service name (db) as the hostname, not localhost. Check that the database container is healthy.

```
docker compose exec app ping db
```

Image is too large

You are probably not using multi-stage builds, or node_modules is being copied to the final stage. Check your Dockerfile against our template.

```
docker images myapp # Check the size
```

Permission denied errors

File ownership mismatch between host and container. Make sure your Dockerfile creates the correct user and sets permissions.

```
docker compose exec app ls -la /app
```

11 PRE-BUILD CHECKLIST

Print this. Pin it to your wall. Use it every time.

BOOTS APPROVED

BOOTS ON THE GROUND AI - DOCKER PRE-BUILD CHECKLIST. Complete every item before starting a new project or deploying to production. No exceptions.

PROJECT SETUP

- ☐ Created Dockerfile with multi-stage build
- ☐ Created docker-compose.yml with all services defined
- ☐ Created .dockerignore (excludes .env, node_modules, .git)
- ☐ Created .env.example with all required variables
- ☐ Created .env with actual values (added to .gitignore)
- ☐ Base images are pinned to specific versions
- ☐ Health checks defined for all services

SECURITY

- ☐ No secrets hardcoded anywhere in code
- ☐ Container runs as non-root user
- ☐ Only necessary ports exposed
- ☐ Docker image scanned for vulnerabilities
- ☐ .env file is in .gitignore AND .dockerignore

TESTING

- ☐ docker compose build completes without errors
- ☐ docker compose up starts all services
- ☐ Application responds on expected port
- ☐ Health check endpoint returns OK
- ☐ Database connection works from app container
- ☐ Hot reload works in development mode

DOCUMENTATION

- ☐ README includes Docker setup instructions
- ☐ All environment variables documented in .env.example
- ☐ docker-compose.yml has comments for each service
- ☐ Deployment steps documented

THE BOOTS STANDARD

IF IT DOES NOT HAVE A DOCKERFILE, IT DOES NOT LEAVE THE SHOP. Every project. Every time. No exceptions. This is the Boots On The Ground AI standard.