AI VOID PROJECT LAB

# Docker Projects 2026: Ideas & Code

Explore practical Docker project ideas for 2026, complete with problem statements, architecture, and concepts. Perfect for beginners and developers to master containerization.

AI VOID

2026-08-06 15:47 IST

# Contents

CHAPTER 01

# Setting Up Your Docker Development Environment (2026 Edition)

Welcome to the foundation of your Docker journey. In this chapter, we'll establish a robust and consistent Docker development environment, preparing your system to build and run containerized applications effectively. This isn't just about installing software; it's about configuring a workspace that mirrors production, ensuring reliable application behavior from your local machine to deployment.

By the end of this guide, you will have Docker Engine and Docker Compose installed, configured, and verified on your system, ready to tackle complex multi-service projects. This foundational setup is critical for preventing common development headaches and ensuring a smooth progression through the advanced Docker concepts we'll explore.

---

## Chapter Project Overview: Establishing Your Docker Foundation

Our initial "project" is your development machine itself. We aim to transform it into a reliable Docker host capable of running isolated, containerized applications.

### The Problem: Inconsistent Environments

Developers often face issues where an application works perfectly on one machine but fails on another due to differing dependencies, operating system configurations, or software versions. This inconsistency slows down development and introduces bugs that are hard to reproduce.

### The Solution: A Standardized Docker Environment

Docker solves this by packaging applications and their dependencies into portable containers. To leverage this, we need a properly configured local Docker environment. This chapter focuses on:

- **Installing Docker Engine**: The core runtime for managing containers.
- **Installing Docker Compose**: The tool for defining and running multi-container applications.

- **Verifying Functionality**: Ensuring everything works as expected with standard tests.

### Success Criteria

By the end of this chapter, you should be able to:

1. Run `docker --version` and see an output indicating Docker Engine `29.1.x`.
2. Run `docker compose version` and see an output indicating Docker Compose `v2.40.3`.
3. Successfully execute `docker run hello-world` and receive the "Hello from Docker!" message.

---

## Core Components & Architecture

A Docker development environment typically consists of several key components that work together. Understanding their roles is crucial.

### Docker Engine

This is the heart of Docker. The Docker Engine comprises:

- **Docker Daemon (`dockerd`)**: A persistent background process that manages Docker objects like images, containers, networks, and volumes. It listens for Docker API requests.
- **Docker CLI (Client)**: The command-line interface (`docker`) that allows you to interact with the Docker Daemon. When you type `docker run`, the CLI sends this command to the daemon.
- **REST API**: The interface through which the CLI (and other tools) communicate with the daemon.

### Docker Compose

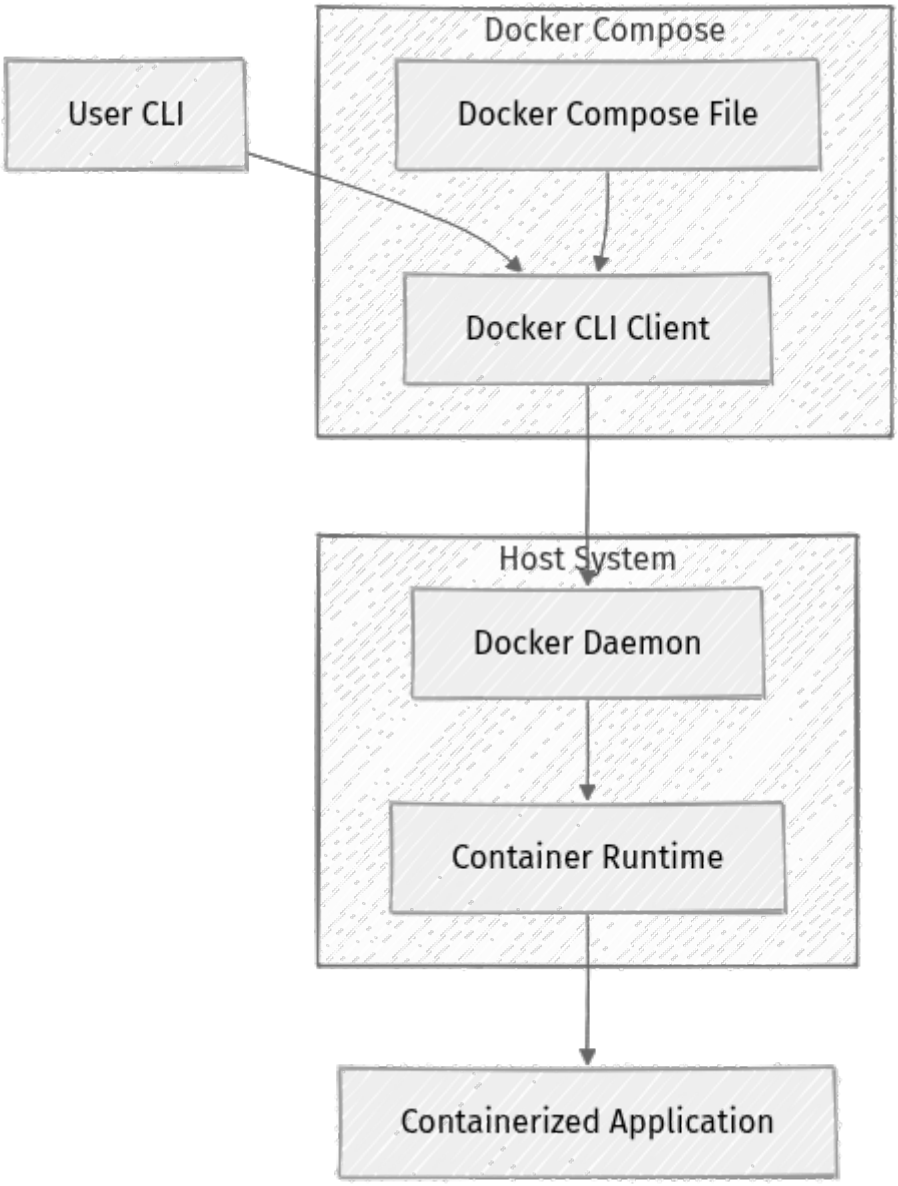
Docker Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services, making it easy to orchestrate complex setups with a single command. It's essential for microservices architectures.

### Docker Desktop

For Windows and macOS users, Docker Desktop bundles the Docker Engine, Docker CLI, Docker Compose, and other utilities (like Kubernetes and a GUI dashboard) into a single, easy-to-install package. It provides the necessary virtualization (WSL 2 on Windows, HyperKit on macOS) to run Linux containers natively.

For Linux users, Docker Engine and Docker Compose are typically installed as separate packages, offering more control, especially in server environments.

The architectural flow for a local Docker setup looks like this:



**Explanation:** Your commands go through the `Docker CLI Client` to the `Docker Daemon`, which then orchestrates the `Container Runtime` to manage your `Containerized Application`. `Docker Compose` simplifies this for multi-service applications by reading a configuration file.

---

## Tech Stack & Version Targets

To ensure consistency and leverage the latest features, we will target specific versions for our core tools.

### Core Development Tools

- **Operating System**: Windows 10/11 (64-bit), macOS (Intel or Apple Silicon), or a modern Linux distribution (e.g., Ubuntu, Fedora).
- **Git**: Essential for version control and cloning project repositories.
    - Installation: Download from the official Git website.
- **Code Editor**: Visual Studio Code (VS Code) is highly recommended for its excellent Docker integration, extensions, and integrated terminal.
    - Installation: Download from the official VS Code website.

### Docker Components (as of 2026-08-06)

We prioritize the latest stable releases for security, performance, and modern features.

- **Docker Engine**: `29.1.x`
- **Docker Compose**: `2.40.3` (as a `docker compose` plugin for Docker Engine)

These versions are current as of 2026-08-06 and provide a stable, feature-rich platform.

---

## Milestones for Environment Setup

Our plan for setting up the environment involves three clear milestones:

1. **Install Prerequisites**: Ensure Git and a code editor are ready.
2. **Install Docker**: Choose the appropriate method (Docker Desktop or native Linux installation).
3. **Verify Installation**: Confirm Docker Engine and Compose are running correctly using built-in commands and a test container.

## Step-by-Step Installation

Follow the instructions specific to your operating system.

### Option 1: Docker Desktop (Windows & macOS)

Docker Desktop offers the most integrated experience for Windows and macOS.

1. **Download Docker Desktop**:
    - For Windows: Visit the official Docker Desktop for Windows installation guide.
    - For macOS: Visit the official Docker Desktop for Mac installation guide.
2. **Run the Installer**:
    - **Windows**: Double-click the downloaded `.exe` file. Follow the installation wizard. Ensure "Install required Windows components for WSL 2" is checked, as WSL 2 provides the best performance for Docker on Windows. A system restart is typically required.
    - **macOS**: Open the `.dmg` file and drag the Docker icon to your Applications folder. Launch Docker Desktop from your Applications folder. You may need to grant system permissions.
3. **Start Docker Desktop**: Once installed, launch Docker Desktop. You should see the Docker whale icon appear in your system tray (Windows) or menu bar (macOS). Wait for the status indicator to show that Docker Engine is running. This may take a few moments.

    🧠 **Important**: On Windows, Docker Desktop relies on WSL 2. Ensure WSL 2 is installed and updated for optimal performance. You can check your WSL version with `wsl -l -v` in PowerShell. On macOS, it uses HyperKit for virtualization.

### Option 2: Docker Engine and Docker Compose (Linux)

For Linux, we'll install Docker Engine and the Docker Compose plugin separately. This guide uses Ubuntu as an example, but steps are similar for other Debian-based distributions.

1. **Uninstall Older Versions (if any)**: It's good practice to remove any conflicting older Docker packages first.

```
    for pkg in docker.io docker-doc docker-compose docker-compose-v2 podman-
docker containerd runc; do sudo apt remove $pkg; done
```

```
This command iterates through common Docker-related package names and removes
them.
```

1. **Set up the Docker Repository**: This step adds Docker's official GPG key and repository to your system, ensuring you download authentic and up-to-date Docker packages.

```
    # Add Docker's official GPG key:
    sudo apt update
    sudo apt install ca-certificates curl gnupg -y
    sudo install -m 0755 -d /etc/apt/keyrings
    curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --
dearmor -o /etc/apt/keyrings/docker.gpg
    sudo chmod a+r /etc/apt/keyrings/docker.gpg

    # Add the repository to Apt sources:
    echo \
      "deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/
docker.gpg] https://download.docker.com/linux/ubuntu \
      "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
      sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
    sudo apt update
```

```
`curl` downloads the GPG key, `gpg --dearmor` converts it to a format `apt`
can use, and `tee` adds the repository URL to your sources list. Finally,
`sudo apt update` refreshes your package lists.
```

1. **Install Docker Engine and Docker Compose Plugin**: This command installs the core Docker Engine, its CLI, `containerd.io` (a container runtime), `docker-buildx-plugin` (for building multi-platform images), and the `docker-compose-plugin`.

```
    sudo apt install docker-ce docker-ce-cli containerd.io docker-buildx-
plugin docker-compose-plugin -y
```

```
After installation, the Docker daemon should start automatically. You can
check its status with `sudo systemctl status docker`.
```

1. **Add Your User to the `docker` Group (Recommended)**: By default, only the `root` user or users with `sudo` privileges can execute Docker commands. To run `docker` commands without `sudo`, add your user to the `docker` group.

```
    sudo usermod -aG docker $USER
    newgrp docker # Apply group changes immediately for the current shell
session
```

```
For the changes to take full effect globally, you should log out and log back
into your system.
```

---

## Testing & Verification

After installation, it's critical to verify that Docker Engine and Docker Compose are correctly installed and fully operational. This ensures you're ready for the next steps.

### Verify Docker Engine Version

Open your terminal or command prompt and run the following command to check the Docker Engine version:

```
docker --version
```

#### Expected Output (example):

```
Docker version 29.1.0, build 5f7c357
```

The version number should be `29.1.x` or very close.

### Verify Docker Compose Version

For installations using the `docker-compose-plugin` (which is the modern approach and what Docker Desktop provides):

```
docker compose version
```

**Expected Output (example):**

```
Docker Compose version v2.40.3
```

The version should be `v2.40.3` or very close. Note the command is `docker compose` (without a hyphen). If you see `command not found`, ensure you installed the `docker-compose-plugin` or that Docker Desktop is running.

### Run a Test Container: Hello World

The most definitive test is to run a simple, pre-built container. Docker's `hello-world` image is specifically designed for this.

```
docker run hello-world
```

This command performs several actions:

1. **Image Pull**: Docker checks if the `hello-world:latest` image exists locally. If not, it pulls it from Docker Hub (Docker's public image registry).
2. **Container Creation**: It creates a new container instance from that image.
3. **Execution**: The container runs its default command, which prints a greeting message.
4. **Exit**: The container then exits as its task is complete.

**Expected Output:** You should see a message similar to this, detailing the steps Docker took and ending with "Hello from Docker!":

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest:
sha256:4d872c3d0c2666a014a51e5e6e8460655e2d6332156828859ff14a275f67a21f
Status: Downloaded newer image for hello-world:latest

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

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub (if
it was not already locally available).
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output back to the Docker client, which
sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash
```

```
Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

If you see this output, your Docker environment is correctly set up and ready for development.

---

## Production Awareness: The "Build Once, Run Anywhere" Principle

While we've focused on setting up your local development environment, it's crucial to understand that Docker's core value extends directly to production. The same Dockerfile and Docker Compose configuration that runs your application locally can be used in staging, testing, and production environments.

This "build once, run anywhere" philosophy minimizes discrepancies, reduces "it works on my machine but not in production" scenarios, and streamlines deployment workflows. It's a cornerstone of modern DevOps practices, ensuring consistency and predictability across the entire software delivery lifecycle.

---

## Common Issues & Troubleshooting

Even with robust installers, you might encounter some common hurdles. Here's how to address them:

1. **Virtualization Not Enabled (Windows/macOS)**: Docker Desktop requires hardware virtualization (Intel VT-x or AMD-V) to be enabled in your computer's BIOS/UEFI settings.
    - **Symptom**: Docker Desktop fails to start, or containers don't run.
    - **Solution**: Restart your computer, enter BIOS/UEFI settings (often by pressing F2, F10, F12, or Del during boot), and enable virtualization technology (sometimes called "Intel VT-d," "AMD-V," or "Virtualization Technology").

2. **Permission Denied (Linux `docker` group)**: If you attempt to run `docker` commands without `sudo` and haven't added your user to the `docker` group, you'll see permission errors.
    - **Symptom**: `docker: Got permission denied while trying to connect to the Docker daemon socket.`
    - **Solution**: Run `sudo usermod -aG docker $USER` and then `newgrp docker`. Remember to log out and log back in for persistent changes.
3. **Docker Daemon Not Running**: The Docker daemon (the background service) might not start correctly or could crash.
    - **Symptom**: `Cannot connect to the Docker daemon. Is the docker daemon running on this host?`
    - **Solution**:
        - **Docker Desktop**: Check the Docker Desktop application (whale icon) in your system tray/menu bar. If it's not running or shows an error, restart it from its menu.
        - **Linux**: Check the service status with `sudo systemctl status docker`. If it's inactive, start it with `sudo systemctl start docker`.
4. **Network Conflicts**: Rarely, Docker's default network ranges might conflict with your existing local network configuration.
    - **Symptom**: Containers cannot communicate with each other or the internet, or specific ports are unreachable.
    - **Solution**: This is an advanced scenario. In Docker Desktop, you can often adjust the default network ranges via Settings -> Resources -> Network. For Linux, you might need to configure daemon options in `/etc/docker/daemon.json`.

---

## Summary & What's Next

Congratulations! You have successfully set up your Docker development environment. You've installed Docker Engine `29.1.x` and Docker Compose `2.40.3`, and you've verified their functionality by running a `hello-world` container. This is a crucial first step that provides a consistent, isolated, and reproducible foundation for all future projects.

With your environment ready, we're now prepared to dive into our first practical project. In the next chapter, we'll build a static website served by Nginx, incorporating volume management and a reverse proxy, to solidify your understanding of basic Docker concepts with a real-world application.

---

## References


- Docker Desktop for Windows Installation Guide
- Docker Desktop for Mac Installation Guide
- Install Docker Engine on Ubuntu
- Install Docker Compose
- Git Official Website
- Visual Studio Code Official Website


---

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 02

# Project 1: Secure Static Website with Nginx, Certbot, and Docker Compose

Serving a website securely over HTTPS is a fundamental requirement for any modern web presence. This chapter guides you through deploying a static website using Nginx as the web server, securing it with HTTPS via Certbot and Let's Encrypt, and orchestrating both services with Docker Compose. This isn't just about getting a site online; it's about doing it with modern containerization best practices, preparing you for robust production deployments.

By the end of this project, you will have a fully functional, secure static website accessible over HTTPS, demonstrating essential Docker concepts like multi-service orchestration, volume management, and network configuration. You'll also learn how to automate SSL certificate provisioning and renewal, a critical aspect of web security that often trips up new developers.

This milestone is important because it lays the groundwork for understanding how to containerize and manage robust web services. A secure, performant Nginx server is often the first line of defense and traffic manager for more complex applications. You'll gain practical, hands-on experience with tools widely used in production environments, setting a strong foundation for future projects.

---

## Project Overview: Secure Static Site

Our objective is to host static content (HTML, CSS, JS) and ensure all traffic is encrypted with HTTPS. This setup requires two primary components: a web server (Nginx) to serve content and handle requests, and an SSL certificate management tool (Certbot) to automate certificate acquisition and renewal. Docker Compose will define and run these services together.

### Problem Statement

We need to serve a static website, `example.com`, securely over HTTPS. The solution must be containerized, easy to deploy, and capable of handling automatic SSL certificate renewal to maintain continuous security without manual intervention. This approach prioritizes security, maintainability, and resource efficiency.

## Tech Stack

This project leverages the following core technologies:

- **Docker Engine (v29.1.x, checked 2026-08-06):** The underlying platform for building and running containers.
- **Docker Compose (v2.40.3, checked 2026-08-06):** For defining and running multi-container Docker applications.
- **Nginx (v1.26.x, checked 2026-08-06):** A high-performance HTTP and reverse proxy server, used here to serve static content and manage SSL/TLS.
- **Certbot (v2.10.0, checked 2026-08-06):** A free, open-source tool that automates the process of obtaining and renewing SSL/TLS certificates from Let's Encrypt.
- **Let's Encrypt:** A free, automated, and open certificate authority (CA) that issues SSL/TLS certificates.

## Required Docker Concepts

This project will solidify your understanding of several core Docker concepts:

- **Dockerfiles:** Creating custom images for Nginx, focusing on optimized builds for smaller, more secure production images.
- **Docker Compose:** Defining and running multi-container applications as a single unit.
- **Volumes:** Persisting Nginx configuration, website content, and Certbot certificates across container restarts and updates.
- **Networks:** Enabling secure and isolated communication between Nginx and Certbot containers.
- **Environment Variables:** Passing configuration data, like the domain name, to containers dynamically.
- **Image Building & Tagging:** Creating and managing your custom images.

## High-Level Architecture

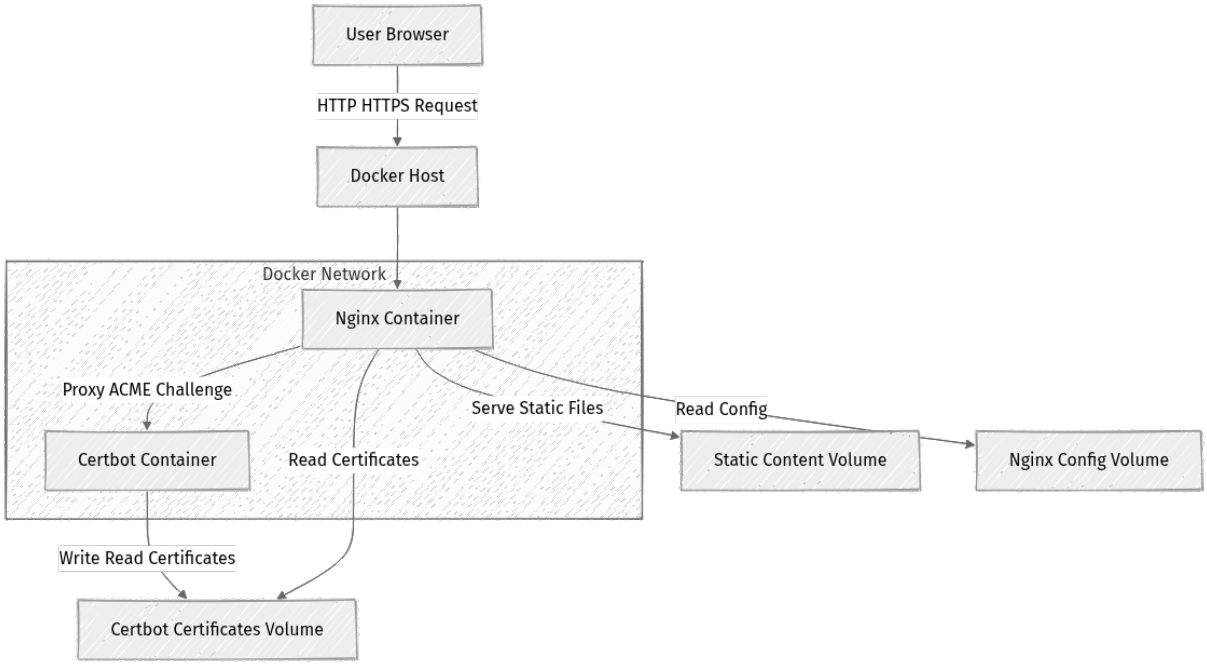
Our setup will involve three main logical layers working together:

1. **Static Content:** Your website files (HTML, CSS, JS) that Nginx will serve.
2. **Nginx Container:** Serves the static content, acts as a reverse proxy, and handles HTTP-to-HTTPS redirection. It also temporarily exposes a path for Certbot's domain verification challenge.

3. **Certbot Container:** Obtains and renews SSL certificates from Let's Encrypt. It uses the HTTP-01 challenge, which requires Nginx to temporarily proxy requests to Certbot's designated volume.

These containers will share critical data through Docker volumes, ensuring persistent storage for configuration and SSL certificates.

## Architecture Diagram




This diagram illustrates how user requests first reach the Docker host. They are then routed to the Nginx container, which either serves static files, redirects HTTP traffic to HTTPS, or proxies requests for Certbot's domain verification challenge. All critical data—static files, Nginx configuration, and Certbot certificates—is stored in named Docker volumes, ensuring data persistence even if containers are stopped or replaced.

## Build Plan

We will implement this project in the following incremental steps:

1. **Initialize Project Structure:** Set up the necessary directories.
2. **Create Static Content:** Add a simple `index.html` file.
3. **Define Nginx Dockerfile:** Create a custom image for Nginx.
4. **Configure Nginx:** Write the Nginx configuration for HTTP, HTTPS, and Certbot challenges.
5. **Configure Docker Compose:** Orchestrate Nginx and Certbot services, volumes, and networks.

6. **Set Environment Variables:** Define the domain name in a `.env` file.
7. **Generate Initial Certificates:** Use Certbot to acquire SSL certificates. This involves a temporary Nginx configuration.
8. **Deploy Final Configuration:** Start all services with the permanent Nginx configuration.

---

## Step-by-Step Implementation

Let's build this project incrementally, starting with our project structure and static content.

### 1. Initialize Project Directory Structure

First, create a clean directory for our project. This structure helps organize configuration, static assets, and Docker-related files.

```
mkdir -p secure-static-site/nginx/conf.d
mkdir -p secure-static-site/www
cd secure-static-site
```

This command creates the following:

- `secure-static-site/`: The root directory for our entire project.
- `secure-static-site/nginx/conf.d/`: This is where our Nginx server configuration files will be stored.
- `secure-static-site/www/`: This directory will hold our static website content (HTML, CSS, JavaScript, images).

### 2. Create Static Website Content

For demonstration purposes, let's create a very simple `index.html` file in the `www` directory. This will be the content Nginx serves.

File: `secure-static-site/www/index.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Secure Static Site</title>
    <style>
        body { font-family: sans-serif; text-align: center; margin-top: 50px;
background-color: #f4f4f4; color: #333; }
        h1 { color: #007bff; }
        p { font-size: 1.1em; }
```

```
    </style>
</head>
<body>
    <h1>Welcome to Your Secure Static Website!</h1>
    <p>This site is served by Nginx and secured with Let's Encrypt via
Certbot.</p>
    <p>If you see this over HTTPS, it's working!</p>
</body>
</html>
```

This basic HTML file provides visual confirmation that our Nginx server is successfully serving content.

### 3. Define Nginx Dockerfile

We'll create a custom Docker image for Nginx. Using a custom `Dockerfile` allows us to precisely control the Nginx environment and integrate our static files and configuration.

File: `secure-static-site/nginx/Dockerfile`

```
# Use the official Nginx stable image as a base for our web server.
# The `alpine-slim` variant is chosen for its minimal size, enhancing security
and reducing download times.
# Docker Engine version 29.1.x is current as of 2026-08-06.
# Nginx stable version 1.26.x is current as of 2026-08-06.
FROM nginx:1.26-alpine-slim

# Copy our custom Nginx configuration files into the container.
# These files will override or supplement the default Nginx configuration.
COPY conf.d/ /etc/nginx/conf.d/

# Copy the static website content into Nginx's default serving directory.
# The --chown=nginx:nginx flag ensures the files are owned by the 'nginx' user
and group inside the container.
# This adheres to the principle of least privilege, preventing Nginx from
running with unnecessary root permissions.
COPY --chown=nginx:nginx ../www/ /usr/share/nginx/html/

# Expose ports 80 (HTTP) and 443 (HTTPS).
# This informs Docker that the container listens on these ports, serving as
documentation.
# It does not automatically open these ports on the host system's firewall.
EXPOSE 80
EXPOSE 443

# Define the command to run Nginx when the container starts.
# "daemon off;" ensures Nginx runs in the foreground, which is essential for
Docker containers
# as Docker expects the main process to remain active to keep the container
running.
CMD ["nginx", "-g", "daemon off;"]
```

**Explanation of the Nginx Dockerfile:**

- `FROM nginx:1.26-alpine-slim`: We base our image on the official Nginx 1.26 stable release, specifically the `alpine-slim` variant. Alpine Linux is known for its small footprint, which translates to smaller, more secure Docker images. The version `1.26` is stable as of our checked date, 2026-08-06.
- `COPY conf.d/ /etc/nginx/conf.d/`: This instruction copies our custom Nginx configuration files from the host's `nginx/conf.d` directory into the container's `/etc/nginx/conf.d/`. These files will define how Nginx handles requests.
- `COPY --chown=nginx:nginx ../www/ /usr/share/nginx/html/`: Here, we copy our static website content from the host's `www` directory into the container's `/usr/share/nginx/html/`. The `--chown` flag is a crucial security measure, ensuring that the files are owned by the `nginx` user and group inside the container. This prevents Nginx from needing to run as `root` to access its content, adhering to the principle of least privilege.
- `EXPOSE 80` / `EXPOSE 443`: These lines indicate that the Nginx container is configured to listen on ports 80 (HTTP) and 443 (HTTPS). This is primarily for documentation and network introspection within Docker; actual port mapping to the host is handled in `docker-compose.yml`.
- `CMD ["nginx", "-g", "daemon off;"]`: This specifies the command that will be executed when the container starts. `nginx -g "daemon off;"` runs Nginx in the foreground. In Docker, a container typically runs only as long as its primary process is active. Running Nginx in the foreground ensures the container stays alive.

## 4. Configure Nginx for Certbot and HTTPS

We need a single Nginx configuration file that handles both HTTP and HTTPS traffic, and importantly, allows Certbot to perform its domain ownership challenges.

File: `secure-static-site/nginx/conf.d/default.conf`

```
# Server block for HTTP traffic (port 80)
server {
    listen 80;
    listen [::]:80; # Listen on IPv6 as well

    server_name ${DOMAIN}; # Dynamically set domain from environment variable

    # Configuration to handle Certbot's ACME challenge requests.
    # Certbot will place temporary files in /var/www/certbot, and Nginx needs
```

```
to serve them.
    location /.well-known/acme-challenge/ {
        root /var/www/certbot;
    }

    # All other HTTP traffic is redirected to HTTPS.
    # This is a security best practice to ensure all communication is
encrypted.
    location / {
        return 301 https://$host$request_uri;
    }
}

# Server block for HTTPS traffic (port 443)
server {
    listen 443 ssl http2;
    listen [::]:443 ssl http2; # Listen on IPv6

    server_name ${DOMAIN};

    # SSL certificates and key provided by Certbot.
    # These paths point to the shared volume where Certbot stores them.
    ssl_certificate /etc/letsencrypt/live/${DOMAIN}/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/${DOMAIN}/privkey.pem;

    # Essential SSL/TLS security settings for robust encryption and
performance.
    # These settings prioritize strong ciphers and modern protocols.
    ssl_session_cache shared:SSL:10m;
    ssl_session_timeout 1h;
    ssl_protocols TLSv1.2 TLSv1.3; # Only allow modern, secure TLS protocols
    ssl_prefer_server_ciphers on;
    ssl_ciphers "EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH";
    ssl_ecdh_curve secp384r1; # Strong elliptic curve, requires OpenSSL >=
1.0.2
    ssl_stapling on; # Enable OCSP stapling for faster certificate validation
    ssl_stapling_verify on; # Verify OCSP response

    # HTTP Strict Transport Security (HSTS) header.
    # Tells browsers to only connect via HTTPS for a long period, preventing
downgrade attacks.
    add_header Strict-Transport-Security "max-age=63072000;
includeSubDomains; preload";

    # Security headers to prevent common web vulnerabilities.
    add_header X-Frame-Options DENY; # Prevents clickjacking
    add_header X-Content-Type-Options nosniff; # Prevents MIME-sniffing
attacks

    # Root directory for serving static files over HTTPS.
    root /usr/share/nginx/html;
    index index.html index.htm; # Default files to serve if a directory is
requested

    # Custom error pages (optional, but good for user experience)
    error_page 404 /404.html;
    location = /404.html {
        internal;
    }
}
```

**Explanation of Nginx `default.conf`:**

- **HTTP Server Block (`listen 80`):**
  - `server_name ${DOMAIN};`: Nginx will use the `DOMAIN` environment variable (which we'll define in `docker-compose.yml` and a `.env` file) to match incoming requests. This makes the configuration flexible.
  - `location /.well-known/acme-challenge/`: This block is crucial for Certbot. It instructs Nginx to serve files from the `/var/www/certbot` directory whenever a request comes in for the `/.well-known/acme-challenge/` path. This allows Certbot to place temporary files here to prove domain ownership to Let's Encrypt.
  - `location / { return 301 <https://$host$request_uri>; }`: This is a fundamental security best practice. It ensures that all unencrypted HTTP traffic arriving on port 80 is automatically and permanently redirected (301 Moved Permanently) to the secure HTTPS equivalent.

- **HTTPS Server Block (`listen 443 ssl http2`):**
    - `ssl_certificate` and `ssl_certificate_key`: These directives point to the full certificate chain and the private key, respectively. These files will be generated by Certbot and stored in a shared Docker volume, making them accessible to Nginx.
    - `ssl_*` directives: These lines define robust SSL/TLS security settings:
        - `ssl_session_cache`, `ssl_session_timeout`: Optimize SSL session handling for performance.
        - `ssl_protocols TLSv1.2 TLSv1.3`: Restrict Nginx to only use modern and secure TLS protocols, disabling older, vulnerable versions.
        - `ssl_prefer_server_ciphers on`, `ssl_ciphers`, `ssl_ecdh_curve`: Prioritize strong, secure cryptographic ciphers and elliptic curves.
        - `ssl_stapling on`, `ssl_stapling_verify on`: Enable OCSP stapling, which improves SSL handshake performance and privacy by allowing the server to provide certificate revocation status directly.
    - `add_header Strict-Transport-Security "max-age=63072000; includeSubDomains; preload";`: This sets the HSTS header. HSTS tells compliant browsers to only connect to your domain over HTTPS for the specified duration (two years in this case), even if a user types `http://`. This is a powerful defense against downgrade attacks.
    - `add_header X-Frame-Options DENY;`, `add_header X-Content-Type-Options nosniff;`: These are additional security headers to mitigate common web vulnerabilities like clickjacking and MIME-sniffing.
    - `root /usr/share/nginx/html;`: This specifies the directory from which Nginx will serve our static files when accessed over HTTPS.
    - `index index.html index.htm;`: Defines the default files Nginx should look for if a directory is requested (e.g., `<https://yourdomain.com/>`).

### 5. Configure Docker Compose

Now, let's define our `nginx` and `certbot` services, along with their shared networks and volumes, using a `docker-compose.yml` file. This file orchestrates our multi-container application.

File: `secure-static-site/docker-compose.yml`

```
# Specify the Docker Compose file format version.
# Version '3.8' is a recent and widely supported standard as of 2026-08-06.
# Docker Compose version 2.40.3 is current as of 2026-08-06.
version: '3.8'

services:
  nginx:
    # Build the Nginx image from the Dockerfile located in the './nginx'
directory.
    build:
      context: ./nginx
      dockerfile: Dockerfile
    container_name: nginx # Assign a memorable name to the Nginx container.
    restart: unless-stopped
# Automatically restart the container if it stops, unless explicitly stopped.

    # Map host ports to container ports.
    # "80:80" maps host port 80 to container port 80.
    # "443:443" maps host port 443 to container port 443.
    ports:
      - "80:80"
      - "443:443"

    # Define volumes for data persistence and sharing.
    volumes:
      - static_www:/usr/share/nginx/html:ro  # Mount static content as read-
only.
      - nginx_conf:/etc/nginx/conf.d:ro     # Mount Nginx configuration as
read-only.
      - certbot_certs:/etc/letsencrypt      # Mount Certbot certificates
(read-write for Certbot, read-only for Nginx).
      - certbot_www:/var/www/certbot        # Mount Certbot challenge files
(read-write for Certbot, read-only for Nginx).

    # Pass the DOMAIN environment variable from the .env file into the
container.
    environment:
      - DOMAIN=${DOMAIN}

    # Nginx needs certificates, so it logically depends on Certbot.
    # However, for initial certificate generation, Certbot needs Nginx to
serve the challenge.
    # We'll manage this "chicken-and-egg" scenario with specific commands
later.
    depends_on:
      - certbot

    # Connect Nginx to our custom 'webnet' network for inter-container
communication.
    networks:
      - webnet

  certbot:
    # Use the official Certbot Docker image. Pinning to a specific version
(v2.10.0)
    # ensures reproducibility and avoids unexpected changes from 'latest'.
    # Certbot version v2.10.0 is current as of 2026-08-06.
    image: certbot/certbot:v2.10.0
```

```
    container_name: certbot # Assign a memorable name to the Certbot
container.
    restart: unless-stopped # Automatically restart the container if it stops.

    # Define volumes for Certbot to store certificates and handle challenges.
    # These volumes are shared with the Nginx container.
    volumes:
      - certbot_certs:/etc/letsencrypt
      - certbot_www:/var/www/certbot

    # Define the entrypoint for Certbot. This script runs 'certbot renew'
every 12 hours.
    # 'trap exit TERM; while :; do ...' ensures the script gracefully handles
termination signals,
    # and 'sleep 12h & wait $!' allows the sleep to be interrupted for faster
shutdown.
    entrypoint: "/bin/sh -c 'trap exit TERM; while :; do certbot renew; sleep
12h & wait $!; done;'"

    # Pass the DOMAIN environment variable to the Certbot container.
    environment:
      - DOMAIN=${DOMAIN}

    # Connect Certbot to our custom 'webnet' network.
    networks:
      - webnet

# Define custom networks. 'webnet' is a bridge network, ideal for
communication
# between services on the same Docker host.
networks:
  webnet:
    driver: bridge

# Define named volumes for persistent data storage.
# Named volumes are the recommended way to manage data in Docker,
# as they are managed by Docker and easier to back up.
volumes:
  static_www: # Stores our static website files.
  nginx_conf: # Stores our Nginx configuration files.
  certbot_certs: # Stores SSL certificates and keys generated by Certbot.
  certbot_www: # Stores temporary files for Certbot's ACME challenges.
```

### Explanation of `docker-compose.yml`:

- `version: '3.8'`: Specifies the Compose file format version. Using a specific version (like `3.8`) ensures compatibility and predictable behavior.

- **`nginx` service:**
  - `build: ./nginx`: Instructs Docker Compose to build the Nginx image using the `Dockerfile` found in the `./nginx` directory.
  - `container_name: nginx`: Assigns a fixed, human-readable name to the container, making it easier to manage.
  - `restart: unless-stopped`: This policy ensures that the Nginx container automatically restarts if it crashes or if the Docker daemon itself restarts, enhancing uptime.
  - `ports: - "80:80" - "443:443"`: These lines map host machine ports 80 and 443 to the Nginx container's internal ports 80 and 443, respectively. This makes your web server accessible from outside the Docker host.
  - `volumes`:
    - `static_www:/usr/share/nginx/html:ro`: Mounts a named volume called `static_www` to the Nginx container's `/usr/share/nginx/html` directory. The `:ro` (read-only) flag is a security measure, preventing Nginx from accidentally modifying your static content.
    - `nginx_conf:/etc/nginx/conf.d:ro`: Mounts the `nginx_conf` volume to hold our Nginx configuration, also read-only.
    - `certbot_certs:/etc/letsencrypt`: This volume is shared with Certbot and will store the SSL certificates. Nginx reads from this.
    - `certbot_www:/var/www/certbot`: This volume is also shared with Certbot and is used for the ACME challenge files.
  - `environment: - DOMAIN=${DOMAIN}`: Passes the value of the `DOMAIN` variable (defined in our `.env` file) into the Nginx container as an environment variable. Nginx uses this in its configuration.
  - `depends_on: - certbot`: This tells Docker Compose that the `nginx` service depends on the `certbot` service. While `depends_on` ensures `certbot` is started before `nginx`, it doesn't wait for Certbot to actually generate certificates. We'll address this initial certificate generation process manually.
  - `networks: - webnet`: Connects the `nginx` container to our custom `webnet` bridge network.

- **`certbot` service:**
  - `image: certbot/certbot:v2.10.0`: Uses the official Certbot Docker image. Pinning to a specific version (`v2.10.0` as of 2026-08-06) is critical for consistent and reproducible deployments.
  - `volumes`: Shares the same `certbot_certs` and `certbot_www` volumes with Nginx. This is how Certbot places certificates for Nginx to use and receives challenge requests.
  - `entrypoint`: This is a custom shell command that instructs the Certbot container to run `certbot renew` every 12 hours. Let's Encrypt certificates are valid for 90 days, so this frequent check ensures renewal well before expiration. The `trap exit TERM; ... & wait $!` pattern handles graceful shutdown.
  - `environment: - DOMAIN=${DOMAIN}`: Passes the domain name to Certbot.
  - `networks: - webnet`: Connects Certbot to the `webnet` network.
- `networks`: Defines our custom bridge network named `webnet`. This provides a private, isolated network for our containers to communicate with each other securely using their service names (e.g., `nginx` can talk to `certbot` by hostname).
- `volumes`: Declares the named volumes used by our services. Named volumes are the preferred method for persistent data storage in Docker as they are managed by Docker and are easier to back up and restore than host-mounted bind mounts.

## 6. Set Environment Variables

Create a `.env` file in the `secure-static-site` root directory to store your domain name. This keeps sensitive or environment-specific values out of your `docker-compose.yml`.

**CRITICAL:** Replace `yourdomain.com` with your actual domain name. You must own this domain and point its DNS A record to your Docker host's public IP address. Without correct DNS, Certbot cannot verify domain ownership, and your site will not be accessible.

File: `secure-static-site/.env`

```
DOMAIN=yourdomain.com
```

## 7. Generate Initial Certificates

This is a crucial step that resolves the "chicken-and-egg" problem: Nginx needs certificates to run HTTPS, but Certbot needs Nginx to be running (and serving the `.well-known` path) to obtain those certificates. We'll temporarily configure Nginx to serve the challenge, get the certificates, then revert Nginx to its full HTTPS configuration.

**Step 7.1: Temporarily Modify Nginx Configuration** First, we need to modify `secure-static-site/nginx/conf.d/default.conf` to serve the Certbot challenge directly on port 80 without attempting to redirect to HTTPS, and without requiring SSL certificates.

**IMPORTANT:** After successfully obtaining certificates, you will revert this file to the original `default.conf` content shown in Step 4.

**Temporary `secure-static-site/nginx/conf.d/default.conf` (for initial cert generation):**

```
server {
    listen 80;
    listen [::]:80;

    server_name ${DOMAIN};

    # This block is essential for Certbot to verify domain ownership.
    location /.well-known/acme-challenge/ {
        root /var/www/certbot;
    }


# During initial certificate generation, we serve static files directly on
HTTP.
    # The HTTP-to-HTTPS redirect is disabled temporarily.
    location / {
        root /usr/share/nginx/html; # Serve static files directly
        index index.html index.htm;
    }
}
```

### Step 7.2: Build and Start Nginx (Temporarily)

Now, build the Nginx image with this temporary configuration and start only the Nginx service.

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

### Step 7.3: Request Certificates with Certbot

Once Nginx is running and serving the challenge path, request the certificates using Certbot.

**CRITICAL:** Ensure your domain's DNS A record points to your server's public IP address before running this command.

```
docker compose run --rm certbot certonly --webroot --webroot-path=/var/www/
certbot -d ${DOMAIN} --email your_email@example.com --agree-tos --no-eff-email
```

**Explanation of the Certbot command:**

- `docker compose run --rm certbot`: This executes the `certbot` service as a one-off command. The `--rm` flag ensures the container is removed immediately after it finishes, keeping your system clean.
- `certonly`: This option tells Certbot to only obtain certificates and not to install them into a web server configuration (as Nginx is already configured).
- `--webroot`: Specifies that Certbot should use the webroot plugin, which places challenge response files in a designated directory.
- `--webroot-path=/var/www/certbot`: This specifies the directory inside the Certbot container where challenge files will be placed. This path corresponds to the shared `certbot_www` volume, which Nginx is configured to serve.
- `-d ${DOMAIN}`: Specifies the domain for which you are requesting certificates. Certbot will read this from your `.env` file.
- `--email your_email@example.com`: Provides your email address for urgent notices from Let's Encrypt regarding certificate expiration or issues.
- `--agree-tos`: Automatically agrees to Let's Encrypt's Terms of Service.
- `--no-eff-email`: Opts out of sharing your email with the Electronic Frontier Foundation (EFF).

If successful, you will see a message similar to: `Congratulations! Your certificate and chain have been saved at: /etc/letsencrypt/live/yourdomain.com/fullchain.pem`.

### Step 7.4: Stop Nginx and Revert Configuration

Now that certificates are obtained, stop the temporarily running Nginx container.

```
docker compose stop nginx
```

**CRITICAL:** Revert the `secure-static-site/nginx/conf.d/default.conf` file to its original content (the one with both HTTP and HTTPS server blocks, including the HTTP-to-HTTPS redirect, as shown in Step 4). This is vital for Nginx to serve HTTPS and automatically redirect HTTP traffic.

### 8. Deploy Final Configuration

With the certificates in place and the Nginx configuration reverted to handle HTTPS and redirects, we can now start both services together.

```
docker compose up -d
```

This command will bring up both the `nginx` and `certbot` containers in detached mode (`-d`). Nginx will now correctly use the newly acquired SSL certificates and enforce HTTPS redirects. The Certbot container will run in the background, automatically renewing certificates before they expire, ensuring your site remains secure.

---

## Testing & Verification

It's crucial to verify that everything is working as expected after deployment.

1. **Check Container Status:** Confirm both containers are running.

```
    docker compose ps
```

```
You should see both `nginx` and `certbot` containers listed with a `running`
status.
```

1. **Access the Website (HTTP and HTTPS):**
    - Open your web browser and navigate to `<http://yourdomain.com>`. You should observe an automatic redirection to `<https://yourdomain.com>`.
    - Navigate directly to `<https://yourdomain.com>`.
    - **Verify Secure Connection:** Look for a padlock icon in your browser's address bar. Click on it to inspect the certificate details. It should be issued by "Let's Encrypt" and valid for your domain.
2. **Inspect Nginx Logs:** Review Nginx logs to confirm traffic is being handled correctly.

```
docker compose logs nginx
```

```
You should see Nginx access logs for your requests. Look for `301` status
codes for successful HTTP-to-HTTPS redirects and `200` status codes for
successful HTTPS requests serving your `index.html`.
```

1. **Inspect Certbot Logs (Optional):** While Certbot mostly runs in the background for renewals, you can check its logs for initial setup messages and renewal attempts.

```
docker compose logs certbot
```

```
You'll see output from Certbot's initial run confirming certificate
acquisition and subsequent messages indicating it's waiting for renewal
schedules.
```

---

## Production Considerations

When deploying to a production environment, several factors beyond basic functionality become critical for reliability, security, and maintainability.

- **Non-Root User:** Our `Dockerfile` copies static content with `chown=nginx:nginx`, and the official Nginx image runs Nginx processes as a non-root user by default. This is a fundamental security best practice, limiting potential damage if the Nginx process were compromised.
- **Resource Limits:** To prevent a single container from monopolizing host resources, define CPU and memory limits in your `docker-compose.yml`. This is vital for host stability and multi-service deployments.

```
services:
  nginx:
    # ... existing config ...
    deploy:
      resources:
        limits:
          cpus: '0.5' # Limit to 50% of one CPU core
          memory: 128M # Limit to 128 MB RAM
        reservations:
          cpus: '0.1' # Reserve 10% of one CPU core
          memory: 32M # Reserve 32 MB RAM
```

```
`limits` define the maximum, while `reservations` guarantee a minimum amount
of resources.
```

- **Firewall Configuration:** Ensure your host machine's firewall (e.g., `ufw` on Linux, Windows Firewall) explicitly allows incoming traffic on TCP ports 80 (HTTP) and 443 (HTTPS). Without these ports open, users cannot reach your website.
- **DNS Configuration:** Verify that your domain's A record (and AAAA record for IPv6) correctly points to the public IP address of your Docker host. This is non-negotiable for Certbot validation and user access.
- **Automated Renewal:** The Certbot container's `entrypoint` ensures certificates are renewed every 12 hours. Let's Encrypt certificates are valid for 90 days, so this frequent check provides ample buffer against expiration, preventing service outages.
- **Backup Volumes:** For real production systems, regularly back up your `certbot_certs` volume. This volume contains your SSL certificates and private keys. Losing them would necessitate re-issuing certificates and potential downtime.

## Common Issues & Solutions

Even with careful planning, issues can arise. Here are some common problems and their solutions:

- **Certbot Challenge Failure:**
  - **Issue:** Certbot fails with an error message like "Failed to connect to host" or "Invalid response from [yourdomain.com]".
  - **Solution:**
    1. **DNS Check:** Confirm your `DOMAIN`'s A record (and AAAA record if applicable) correctly points to your server's public IP address. Use `dig +short yourdomain.com` or `nslookup yourdomain.com` to verify.
    2. **Firewall:** Ensure TCP ports 80 and 443 are open on your server's firewall. Let's Encrypt must be able to reach your server on port 80 for the HTTP-01 challenge.
    3. **Nginx Config (Temporary):** Double-check that the temporary Nginx configuration (used for initial cert generation) is correctly configured to serve files from `/var/www/certbot` for the `/.well-known/acme-challenge/` path.
    4. **Nginx Running:** Make sure the `nginx` service is actually running when you execute the `certbot certonly` command (`docker compose ps` should show it as `running`).
- **Nginx Fails to Start with HTTPS:**
  - **Issue:** The Nginx container exits immediately or its logs show errors like "No such file or directory" related to SSL certificate paths (`fullchain.pem`, `privkey.pem`).
  - **Solution:** This almost always means Certbot did not successfully generate the certificates, or Nginx's configuration points to incorrect paths. Carefully re-run the Certbot initial generation step. Verify that the paths specified in `default.conf` for `ssl_certificate` and `ssl_certificate_key` exactly match where Certbot saves certificates (`/etc/letsencrypt/live/${DOMAIN}/`). Ensure you reverted to the correct `default.conf` after generating certificates.

- **Port Conflicts:**
  - **Issue:** `docker compose up` fails with an error such as "port is already allocated" or "Bind for 0.0.0.0:80 failed: port is already in use".
  - **Solution:** Another process on your host machine is already using port 80 or 443. This could be another web server (e.g., Apache, a system Nginx instance) or a previous Docker container that wasn't properly shut down. Stop any conflicting services or change the port mapping in your `docker-compose.yml` (e.g., change `"80:80"` to `"8080:80"` if you want to access it on `<http://yourdomain.com:8080>`).

---

## Summary & Next Step

Congratulations! You've successfully designed, implemented, and deployed a secure static website using Nginx, Certbot, and Docker Compose. This project provided hands-on experience with several critical concepts:

- Building custom Docker images and understanding the benefits of minimal base images.
- Orchestrating multiple interdependent services using `docker-compose.yml`.
- Managing persistent data and configuration through Docker volumes.
- Implementing HTTPS with automated SSL certificate acquisition and renewal from Let's Encrypt.
- Applying basic container security principles and considering production deployment aspects like resource limits and firewalls.

This project establishes a solid, reusable pattern for serving static content and securing web services, forming a foundational piece for more complex architectures.

Next, we'll dive into building a more intricate full-stack web application. This will introduce database services, demonstrate more advanced inter-service communication, and further refine your Docker Compose skills for multi-tier applications.

---

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

## References


- Docker Engine Documentation (v29.1.x)
- Docker Compose Documentation (v2.40.3)
- Nginx Official Docker Image
- Certbot Official Documentation
- Let's Encrypt Documentation
- Best Practices for Using Azure Container Registry - Azure Container Registry | Microsoft Learn

CHAPTER 03

# Project 2: Full-Stack Application - Node.js API with PostgreSQL Persistence

Building a full-stack application often means orchestrating multiple services: a backend API, a database, and potentially a frontend. Managing these components, especially across different environments, can introduce significant complexity. This chapter addresses that challenge by demonstrating how to containerize a Node.js API that interacts with a PostgreSQL database, all orchestrated with Docker Compose.

By the end of this project, you will have a fully functional, containerized "Todo" API, complete with robust data persistence. You will gain practical experience in defining multi-service applications, configuring inter-container communication, and ensuring your data remains safe and accessible across container lifecycles. This project is a critical step towards understanding more complex microservice architectures and preparing applications for production.

---

## Project Overview: The Todo API

This project focuses on building a simple but complete "Todo" application backend. The core problem it solves is providing a persistent storage layer and an API interface for managing user tasks.

**Problem Statement:** Develop a backend service that can store, retrieve, and manage a list of todo items, ensuring data persistence even if the application components are restarted or scaled. The service should be easy to deploy and manage in a containerized environment.

**Solution:** We will build a Node.js Express API that connects to a PostgreSQL database. Both services will be containerized using Docker and orchestrated with Docker Compose. Data persistence for PostgreSQL will be handled by a Docker named volume.

**Project Outcome:** A fully functional, containerized RESTful API for managing todo items, demonstrating robust inter-service communication and data persistence within a Docker Compose setup. This system will be ready for integration with a separate frontend or for use by other services.

## Core Technologies

To achieve our project goals, we will leverage the following primary technologies:

- **Docker Engine (v29.1.x, checked 2026-08-06):** The core platform for building, shipping, and running containerized applications.
- **Docker Compose (v2.40.3, checked 2026-08-06):** A tool for defining and running multi-container Docker applications. It uses YAML files to configure application services.
- **Node.js (LTS v20.10.0, checked 2026-08-06):** A JavaScript runtime environment for building our backend API.
- **Express.js (v4.x, checked 2026-08-06):** A fast, unopinionated, minimalist web framework for Node.js, used to build our API endpoints.
- **PostgreSQL (v16.2, checked 2026-08-06):** A powerful, open-source relational database system chosen for its reliability and advanced features, used for data persistence.
- **`pg` (v8.x, checked 2026-08-06):** The non-blocking PostgreSQL client for Node.js.

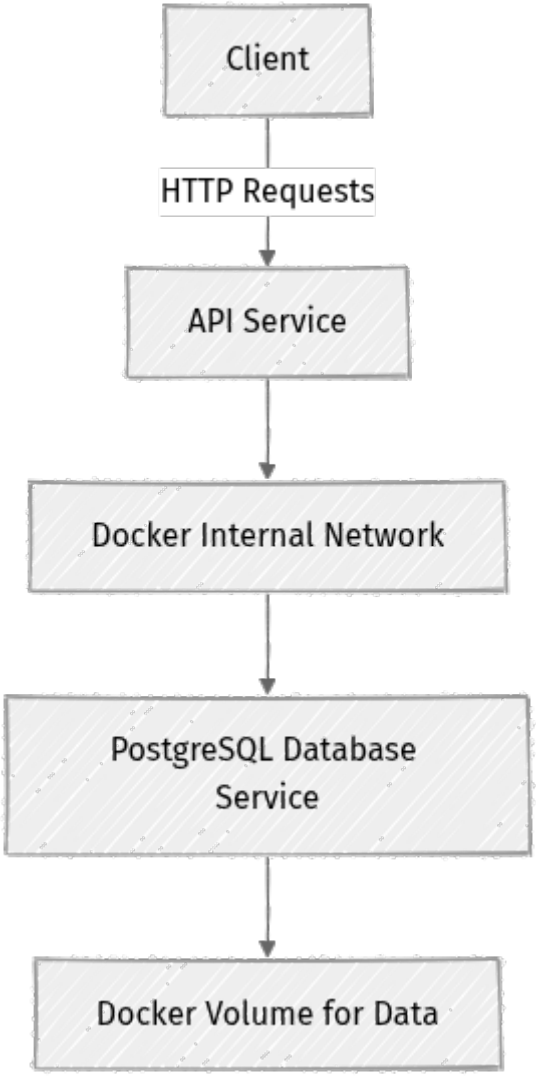
## Build Plan and Milestones

We'll tackle this project in logical, verifiable steps:

1. **Project Setup:** Initialize the project directory and the Node.js API application structure.
2. **Node.js API Development:** Implement the core Express API logic for managing todo items and connecting to PostgreSQL.
3. **PostgreSQL Service Definition:** Create the `docker-compose.yaml` configuration for our PostgreSQL database, including data persistence.
4. **Database Verification:** Start and test the PostgreSQL service in isolation, including schema initialization.
5. **Node.js API Containerization:** Create a `Dockerfile` for the Node.js API, focusing on multi-stage builds and security.
6. **Full Stack Orchestration:** Integrate the Node.js API service into `docker-compose.yaml` with inter-service dependencies and health checks.
7. **End-to-End Verification:** Start the complete application stack and test all API endpoints.

## Project Architecture

Our full-stack application adopts a simple microservices-like architecture, separating the API from its database. Docker Compose provides the network and volume management to connect these services seamlessly.




- **Client:** Your development machine's browser or command-line tools (like `curl`) will interact with the Node.js API.
- **Node.js API Service:** This container hosts our Express.js application. It listens for HTTP requests and communicates with the database. It is designed to be stateless.
- **Docker Internal Network:** Docker Compose automatically creates a private network. Services within this network can communicate using their service names as hostnames, ensuring secure and simplified routing.
- **PostgreSQL Database Service:** This container runs the PostgreSQL database, storing all todo item data. It is a stateful service.

- **Docker Volume for Data:** A named Docker volume (e.g., `postgres_data`) is mounted into the PostgreSQL container. This ensures that the database's data persists across container restarts, removals, or upgrades.

---

## Step-by-Step Implementation

Let's get started by setting up our project and building the individual components.

### 1. Initialize Project Structure and Node.js API

First, we'll create the project directories and set up the basic Node.js application.

1. **Create the project root directory:** Open your terminal and create the main project folder.

```
mkdir project-2-fullstack
cd project-2-fullstack
```

1. **Create the Node.js application directory:** This directory will house our API's source code.

```
mkdir api
cd api
```

1. **Initialize Node.js project and install dependencies:** We need `express` for the web server and `pg` to interact with PostgreSQL.

```
npm init -y
npm install express pg@8.11.3 # Using pg v8.11.3, a stable release as of 2026-08-06
```

```
`npm init -y` creates a default `package.json` file. `npm install` adds the required libraries.
```

1. **Create `api/app.js` - The Node.js API:** This file contains our Express.js application, defining the API endpoints and database interactions.

```
// api/app.js
const express = require('express');
const { Pool } = require('pg');

const app = express();
```

```
    const port = process.env.PORT || 3000;

    // Middleware to parse JSON request bodies
    app.use(express.json());

    // PostgreSQL connection pool configuration.
    // Connection details are loaded from environment variables.
    const pool = new Pool({
      user: process.env.DB_USER,
      host: process.env.DB_HOST,
      database: process.env.DB_NAME,
      password: process.env.DB_PASSWORD,
      port: process.env.DB_PORT,
      // Increase connection timeout to allow DB to start up
      connectionTimeoutMillis: 10000, // 10 seconds
    });

    // Test database connection on startup
    // This provides early feedback if the API cannot reach the DB.
    pool.connect((err, client, release) => {
      if (err) {
        return console.error('Error acquiring PostgreSQL client on startup:',
err.stack);
      }
      client.query('SELECT NOW()', (err, result) => {
        release(); // Release the client back to the pool
        if (err) {
          return console.error('Error executing initial database query:',
err.stack);
        }
        console.log('PostgreSQL database connected successfully at:', result.r
ows[0].now);
      });
    });

    // Basic health check endpoint
    app.get('/', (req, res) => {
      res.send('Node.js API is running!');
    });

    // GET /todos: Fetch all todo items
    app.get('/todos', async (req, res) => {
      try {
        const result = await pool.query('SELECT id, title, completed FROM
todos ORDER BY id ASC');
        res.json(result.rows);
      } catch (err) {
        console.error('Error fetching todos:', err.message);
        res.status(500).send('Server Error fetching todos');
      }
    });

    // POST /todos: Add a new todo item
    app.post('/todos', async (req, res) => {
      const { title } = req.body;
      if (!title) {
        return res.status(400).send('Title is required to add a todo item.');
      }
      try {
        const result = await pool.query(
          'INSERT INTO todos (title) VALUES ($1) RETURNING id, title,
completed',
```

```
          [title]
        );
        res.status(201).json(result.rows[0]);
      } catch (err) {
        console.error('Error adding todo:', err.message);
        res.status(500).send('Server Error adding todo');
      }
    });

    // Start the API server
    app.listen(port, () => {
      console.log(`Node.js API listening on port ${port}`);
    });
```

```
**Explanation:**
- This Express application defines three routes: a root health check (`/`), a
GET endpoint to retrieve all todos (`/todos`), and a POST endpoint to add a
new todo (`/todos`).
- Database connection parameters are read from environment variables, which
Docker Compose will provide.
- A `connectionTimeoutMillis` is added to the `pg` pool configuration to give
the database more time to start up before the API gives up trying to connect.
- An initial `pool.connect` call verifies the database connection when the API
starts, providing immediate feedback.
```

1. **Create `api/.dockerignore`:** This file tells Docker which files and directories to exclude when building the image. It's crucial for keeping image sizes small and build times fast. Place it in the `api/` directory.

```
    # api/.dockerignore
    node_modules
    npm-debug.log
    .env
    Dockerfile
```

```
**Explanation:** We exclude `node_modules` because we'll install them inside
the Docker container. `.env` contains sensitive information and should never
be copied into the image.
```

1. **Navigate back to the project root:**

```
    cd ..
```

## 2. Define PostgreSQL Service with Docker Compose

Now, we'll configure our `docker-compose.yaml` file to define the PostgreSQL database service and its associated named volume for data persistence.

1. **Create `project-2-fullstack/docker-compose.yaml`:** This file orchestrates our services.

```
    # project-2-fullstack/docker-compose.yaml
    version: '3.8'

    services:
      postgresql:
        image: postgres:16.2-alpine # Using specific, stable PostgreSQL v16.2
(checked 2026-08-06)
        restart: always
        environment:
          POSTGRES_USER: ${DB_USER}
          POSTGRES_PASSWORD: ${DB_PASSWORD}
          POSTGRES_DB: ${DB_NAME}
        volumes:
          - postgres_data:/var/lib/postgresql/data # Mount a named volume for
data persistence
        ports:
          - "5432:5432" # Optional: Expose PostgreSQL port to the host for
direct access
        healthcheck:
          test: ["CMD-SHELL", "pg_isready -U ${DB_USER} -d ${DB_NAME}"]
          interval: 10s
          timeout: 5s
          retries: 5
          start_period: 10s # Give the database time to initialize before
health checks start

    volumes:
      postgres_data: # Define the named volume, managed by Docker
```

```
**Explanation:**
- **`version: '3.8'`**: Specifies the Docker Compose file format.
- **`postgresql`**: This is the service name. Our API will use this name to
connect.
- **`image: postgres:16.2-alpine`**: Uses the official PostgreSQL `16.2`
image. The `alpine` variant is chosen for its minimal size, which reduces
download times and image attack surface.
- **`restart: always`**: Ensures the container automatically restarts if it
crashes or Docker restarts.
- **`environment`**: These variables (`POSTGRES_USER`, `POSTGRES_PASSWORD`,
`POSTGRES_DB`) configure the PostgreSQL instance. Values are read from a
`.env` file.
- **`volumes: - postgres_data:/var/lib/postgresql/data`**: **Crucial for data
persistence.** `postgres_data` is a *named volume* managed by Docker. `/var/
lib/postgresql/data` is PostgreSQL's default data directory. This mapping
ensures your data is saved on the host and survives container lifecycle
events.
- **`ports: - "5432:5432"`**: Maps the container's PostgreSQL port (5432) to
the host's port 5432. This is useful for connecting with external tools like
```

```
DBeaver or `psql` from your host, but not strictly required for inter-
container communication.
- **`healthcheck`**: Defines how Docker Compose determines if the `postgresql`
service is healthy. `pg_isready` is a utility provided by PostgreSQL to check
its readiness. `start_period` gives the database enough time to fully
initialize before the health checks begin, preventing false negatives.
- **`volumes: postgres_data:`**: This top-level block explicitly declares the
`postgres_data` named volume.
```

1. **Create `project-2-fullstack/.env`:** This file holds environment variables for our services. Docker Compose automatically reads this file when `docker compose up` is executed.

   🧠 **Important:** For production, **never commit this `.env` file to version control** as it contains sensitive credentials.

```
    # project-2-fullstack/.env
    DB_USER=devuser
    DB_PASSWORD=devpassword
    DB_NAME=todos_db
    DB_HOST=postgresql # This is the service name in docker-compose.yaml
    DB_PORT=5432
```

```
**Explanation:**
- `DB_HOST=postgresql`: This is vital. Inside the Docker Compose network,
services resolve each other by their service names. So, our Node.js API will
connect to the hostname `postgresql`, not `localhost` or an IP address.
```

## 3. Verification: Starting and Initializing PostgreSQL

Let's start just the PostgreSQL service to ensure it's functioning and then set up our database schema.

1. **Start the PostgreSQL service:** Ensure you are in the `project-2-fullstack` directory.

```
    docker compose up -d postgresql
```

```
The `-d` flag runs the container in detached mode (background).
```

1. **Check container status:**

```
    docker compose ps
```

```
You should see `postgresql` listed with a `healthy` status eventually. It
might take a few seconds for the health check to pass as `start_period` is
10s.
```

1. **View logs for the database service:**

```
    docker compose logs postgresql
```

```
Look for messages indicating PostgreSQL has started successfully and is ready
for connections, such as `database system is ready to accept connections`.
```

1. **Connect to the database and initialize schema:** We'll use `docker compose exec` to run a command inside the running `postgresql` container.

```
    docker compose exec postgresql psql -U devuser -d todos_db
```

```
Once inside the `psql` prompt, create the `todos` table:
```

```
    CREATE TABLE todos (
        id SERIAL PRIMARY KEY,
        title VARCHAR(255) NOT NULL,
        completed BOOLEAN DEFAULT FALSE
    );
    \dt -- List tables to confirm 'todos' exists
    \q  -- Quit psql
```

```
**Verification:** This step confirms that our PostgreSQL container is running,
accessible, and we can successfully execute SQL commands to create our
application's schema.
```

## 4. Containerize Node.js API Service

Now we'll create a `Dockerfile` for our Node.js API and integrate it into `docker-compose.yaml`.

1. **Create `api/Dockerfile`:** This Dockerfile will use a multi-stage build to create a small, secure production image. Place it in the `api/` directory.

```
    # api/Dockerfile
    # Stage 1: Build the application dependencies
    FROM node:20.10.0-alpine AS builder # Using specific Node.js LTS v20.10.0
(checked 2026-08-06)
```

```
    WORKDIR /app

    # Copy package.json and package-lock.json first to leverage Docker cache
    # This layer only changes if dependencies change, speeding up rebuilds.
    COPY package*.json ./
    RUN npm install --omit=dev # Install production dependencies only

    # Copy the rest of the application files
    COPY . .

    # Stage 2: Create the final, lightweight, and secure production image
    FROM node:20.10.0-alpine

    WORKDIR /app

    # Create a non-root user for security best practices
    # Run application with least privileges.
    RUN addgroup --system appgroup && adduser --system --ingroup appgroup appu
ser
    USER appuser

    # Install curl for health checks defined in docker-compose.yaml
    # Alpine uses 'apk' for package management. --no-cache optimizes image siz
e.
    RUN apk add --no-cache curl

    # Copy only necessary artifacts from the builder stage
    # This keeps the final image minimal and free of build tools/dev dependenc
ies.
    COPY --from=builder /app/node_modules ./node_modules
    COPY --from=builder /app/app.js .

    EXPOSE 3000 # Inform Docker that the container listens on port 3000
    CMD ["node", "app.js"] # Command to run when the container starts
```

```
**Explanation:**
- **Multi-stage build (`FROM ... AS builder` then `FROM ...` again):** This is
a best practice. The first stage (`builder`) handles dependency installation.
The second stage then copies *only* the essential `node_modules` and
application code from the `builder` stage, resulting in a significantly
smaller and more secure final image. It prevents build tools and development
dependencies from ending up in the production image.
- **`FROM node:20.10.0-alpine`**: Uses the official Node.js `20.10.0` LTS
image, specifically the `alpine` variant for its small size.
- **`WORKDIR /app`**: Sets the working directory inside the container.
- **`COPY package*.json ./`**: Copies the package files early. If these don't
change, Docker can cache the `npm install` step, speeding up subsequent
builds.
- **`RUN npm install --omit=dev`**: Installs only production dependencies,
further reducing image size and potential vulnerabilities.
- **`USER appuser`**: This is a critical security measure. The application
will run as a non-root user (`appuser`), minimizing the impact if the
container is compromised.
- **`RUN apk add --no-cache curl`**: Installs `curl` (Alpine's package manager
is `apk`) for the health check we'll define in `docker-compose.yaml`. `--no-
cache` prevents caching package indexes, saving space.
- **`EXPOSE 3000`**: Documents that the container listens on port 3000. It
doesn't publish the port; `ports` in `docker-compose.yaml` does that.
```

```
- **`CMD ["node", "app.js"]`**: The default command to execute when the
container starts, launching our Node.js API.
```

1. **Update `project-2-fullstack/docker-compose.yaml` to include the API service:** Add the `api` service definition alongside the `postgresql` service.

```
    # project-2-fullstack/docker-compose.yaml (updated)
    version: '3.8'

    services:
      postgresql:
        image: postgres:16.2-alpine
        restart: always
        environment:
          POSTGRES_USER: ${DB_USER}
          POSTGRES_PASSWORD: ${DB_PASSWORD}
          POSTGRES_DB: ${DB_NAME}
        volumes:
          - postgres_data:/var/lib/postgresql/data
        ports:
          - "5432:5432"
        healthcheck:
          test: ["CMD-SHELL", "pg_isready -U ${DB_USER} -d ${DB_NAME}"]
          interval: 10s
          timeout: 5s
          retries: 5
          start_period: 10s

      api:
        build: ./api # Instruct Docker Compose to build the image from the
Dockerfile in the 'api' directory
        restart: always
        environment:
          PORT: ${API_PORT}
          DB_USER: ${DB_USER}
          DB_PASSWORD: ${DB_PASSWORD}
          DB_NAME: ${DB_NAME}
          DB_HOST: postgresql # Crucial: connects to the 'postgresql' service
by its service name
          DB_PORT: ${DB_PORT}
        ports:
          - "${API_PORT}:${API_PORT}" # Map API port from container to host
        depends_on:
          postgresql:
            condition: service_healthy # Ensure PostgreSQL is healthy before
starting the API
        healthcheck:
          test: ["CMD", "curl", "-f", "http://localhost:${API_PORT}/"] # Use
curl to check API health
          interval: 30s
          timeout: 10s
          retries: 3
          start_period: 20s # Give the API more time to start up before
checking health

    volumes:
```

```
      postgres_data:
```

```
**Explanation:**
- **`api`**: The service name for our Node.js application.
- **`build: ./api`**: Tells Docker Compose to build the image for this service
using the `Dockerfile` located in the `api` directory.
- **`environment`**: Passes environment variables from our `.env` file to the
Node.js container. `DB_HOST` is set to `postgresql`, leveraging Docker's
internal DNS.
- **`ports: - "${API_PORT}:${API_PORT}"`**: Maps the container's API port
(e.g., 3000) to the host's `API_PORT` (defined in `.env`).
- **`depends_on: postgresql: condition: service_healthy`**: This is a powerful
orchestration feature. It ensures the `api` service *only starts* after the
`postgresql` service is reported as `healthy` by its health check. This
prevents the API from crashing while waiting for the database to become
available.
- **`healthcheck`**: Defines a health check for the API using `curl` to hit
its root endpoint. `start_period` is important here, giving the Node.js app
sufficient time to boot up before the health checks begin.
```

1. **Update `project-2-fullstack/.env` with API port:** Add the `API_PORT` variable to your `.env` file.

```
    # project-2-fullstack/.env (updated)
    DB_USER=devuser
    DB_PASSWORD=devpassword
    DB_NAME=todos_db
    DB_HOST=postgresql
    DB_PORT=5432
    API_PORT=3000 # New: Port for our Node.js API
```

## 5. End-to-End Verification: Starting the Full Stack

Now we can bring up both services together and test the full application.

1. **Stop and remove previous containers and volumes:** It's good practice to start clean, especially when making significant configuration changes. Ensure you are in the `project-2-fullstack` directory.

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

```
This command stops and removes all containers, networks, and the
`postgres_data` volume, ensuring a fresh start with an empty database. If you
wished to preserve your data, you would omit the `--volumes` flag.
```

1. **Start all services defined in `docker-compose.yaml`:**

```
    docker compose up -d
```

```
This command will build the `api` image (if not already built), then start
both `postgresql` and `api` services in detached mode. Docker Compose will
respect the `depends_on` condition, ensuring PostgreSQL is healthy before the
API starts.
```

1. **Check container status:**

```
    docker compose ps
```

```
You should see both `postgresql` and `api` services running and eventually
transitioning to a `healthy` status. This might take a moment due to
`start_period` and `interval` settings for the health checks.
```

1. **View logs for both services:**

```
    docker compose logs -f
```

```
You should observe the PostgreSQL startup messages, followed by the Node.js
API connecting to the database and logging `PostgreSQL database connected
successfully`.
```

1. **Test the API:** Once both services are healthy, you can interact with your API using `curl` from your terminal.
    - **Health Check:** Verify the API is responsive.

```
        curl http://localhost:3000/
```

```
    Expected output: `Node.js API is running!`

- **Add a Todo:** Send a POST request to create a new todo item.
```

```
        curl -X POST -H "Content-Type: application/json" -d '{"title": "Learn
Docker Compose"}' http://localhost:3000/todos
```

```
    Expected output: `{"id":1,"title":"Learn Docker
Compose","completed":false}`

- **Add another Todo:**
```

```
        curl -X POST -H "Content-Type: application/json" -d '{"title": "Build
a Dockerized App"}' http://localhost:3000/todos
```

```
    Expected output: `{"id":2,"title":"Build a Dockerized
App","completed":false}`

- **List Todos:** Retrieve all todo items.
```

```
        curl http://localhost:3000/todos
```

```
    Expected output: `[{"id":1,"title":"Learn Docker
Compose","completed":false},{"id":2,"title":"Build a Dockerized
App","completed":false}]`
```

Congratulations! You've successfully built and verified a full-stack application with a Node.js API and PostgreSQL database, all containerized and orchestrated by Docker Compose. This is a significant milestone in your Docker journey.

---

## Production Considerations: Hardening and Operations

As a project mentor, I always emphasize incorporating production thinking early in the development cycle. Here's what we've already considered and what more you might add.

### Non-Root User in Dockerfile

Running processes as `root` inside containers is a significant security risk. If an attacker compromises your application, they immediately gain root privileges within the container, potentially enabling them to escape the container sandbox or cause more extensive damage to the host system.

- **Our Solution:** In `api/Dockerfile`, we explicitly create a non-root user (`appuser`) and switch to it using the `USER appuser` instruction. This strictly adheres to the principle of least privilege, reducing the severity of a potential compromise.

- **Impact:** If your application needs to write to specific directories, ensure `appuser` has the necessary permissions for those directories within the container image.

### Resource Limits

In a production environment, it's crucial to prevent one container from consuming all available host resources, which could destabilize other services or the host itself.

- **Implementation (Example in `docker-compose.yaml`):** You can add `deploy.resources.limits` to your services in `docker-compose.yaml` (under the `api` and `postgresql` service definitions) to set CPU and memory constraints.

```
    # ... inside a service definition, e.g., 'api'
        deploy:
          resources:
            limits:
              cpus: '0.5' # Limit to 50% of one CPU core
              memory: 512M # Limit to 512 MB of RAM
            reservations: # Minimum resources guaranteed for the service
              cpus: '0.25'
              memory: 128M
```

```
- **Trade-offs:** Setting resource limits too low can cause performance
bottlenecks, `Out Of Memory` errors, or service crashes. It's essential to
monitor your application's actual resource usage in a representative
environment to find optimal and safe values.
```

### Health Checks and depends_on

Health checks (as implemented in `docker-compose.yaml` for both services) are vital for reliable deployments and robust service orchestration.

- **Purpose:** They go beyond simply checking if a container has started; they tell Docker when a service is truly ready to receive traffic and, crucially, if it remains functional over time.
- **Orchestration:** The `depends_on: condition: service_healthy` feature leverages these checks to ensure services start in the correct order, preventing common startup failures (e.g., the API trying to connect to a database that is still initializing).

- **Deployment:** In more advanced production orchestration systems like Kubernetes, similar health checks (liveness and readiness probes) are extensively used for rolling updates, auto-healing of failing services, and intelligent traffic routing.

### Environment Variable Management with `.env`

The `.env` file is an excellent and convenient solution for local development to manage configuration. However, for production deployments:

- **Security:** As mentioned, never commit `.env` files to version control repositories. They often contain sensitive credentials and API keys.
- **Production Deployment Best Practices:** In production environments, you should use more secure and robust methods for managing secrets:
    - **Container Orchestrators:** Kubernetes secrets, Docker Swarm secrets.
    - **Cloud Secret Management Services:** AWS Secrets Manager, Azure Key Vault, Google Secret Manager.
    - **CI/CD Pipelines:** Inject variables directly into the build/deploy process as secure pipeline variables.

## Troubleshooting Common Issues

Even with careful planning, issues can arise when working with multi-service containerized applications. Here are some common pitfalls and effective debugging strategies.

1. **Database Connection Refused/Timeout:**
    - **Symptom:** Your Node.js API logs show errors like `Error acquiring PostgreSQL client:` or `connect ECONNREFUSED`.
    - **Cause:** The PostgreSQL service might not be running, isn't yet healthy, or the `DB_HOST` configuration in your API service is incorrect.
    - **Solution:**
        - Run `docker compose ps` to check the `postgresql` service's status. Is it `healthy`?
        - Inspect `docker compose logs postgresql` for any startup errors or indications that the database is not ready.
        - Verify that `DB_HOST` in your `project-2-fullstack/.env` file is correctly set to `postgresql`.
        - Ensure `depends_on: postgresql: condition: service_healthy` is correctly configured for the `api` service in `docker-compose.yaml` to guarantee proper startup order.
2. **Port Conflicts:**
    - **Symptom:** `docker compose up` fails with an error message like `Bind for 0.0.0.0:3000 failed: port is already allocated`.
    - **Cause:** Another process on your host machine is already using the specified port (e.g., port 3000 for the API or 5432 for PostgreSQL).
    - **Solution:**
        - Identify and stop the conflicting process. On Linux/macOS, use `lsof -i :<PORT_NUMBER>`. On Windows, use `netstat -ano | findstr :<PORT_NUMBER>` followed by `taskkill /PID <PID> /F`.
        - Alternatively, change `API_PORT` (or `DB_PORT`) in your `.env` file to an unused port (e.g., `3001` for the API) and restart Docker Compose.

3. **Volume Permission Issues (less common with official images):**
    - **Symptom:** PostgreSQL logs show errors indicating `could not open file "/var/lib/postgresql/data/..."` or `permission denied`.
    - **Cause:** The user running PostgreSQL inside the container (often `postgres` with UID/GID 999 in official images) does not have the necessary write permissions to the mounted volume directory on the host. This issue is more common with bind mounts or when using custom Docker images that don't correctly set user permissions.
    - **Solution:**
        - For named volumes, Docker usually manages permissions correctly. If an issue occurs, it might indicate a deeper Docker daemon configuration problem or host filesystem permissions.
        - For bind mounts, you might need to manually adjust permissions on the host directory that is mounted into the container. For example, `sudo chown -R 999:999 /path/to/host/data` (where 999 is a common UID/GID for the `postgres` user in official images) might resolve it.

---

## Summary & Next Steps

In this chapter, you've significantly advanced your Docker skills by moving beyond single-container applications to orchestrating a multi-service full-stack system. You now have:

- A Node.js API container capable of interacting with a PostgreSQL database.
- A PostgreSQL database container with robust, persistent data storage via Docker volumes.
- A `docker-compose.yaml` file that effectively defines and orchestrates both services, their network, and crucial startup dependencies using health checks.
- A practical understanding of multi-stage Dockerfiles for creating secure and efficient production-ready images.
- Hands-on experience with essential Docker Compose commands like `up`, `down`, `ps`, and `logs`.

This project is now a runnable, self-contained full-stack application. For further refinement and to continue your learning, consider these next steps:

- **Frontend Integration:** Add a simple frontend application (e.g., built with React, Vue, or even a static HTML file) in its own container. You could serve it using Nginx (similar to what you learned in Project 1) and configure it to connect to your Node.js API.
- **Database Migrations:** Integrate a database migration tool like `Knex.js` or `TypeORM` to manage schema changes systematically and version-control your database structure.
- **Automated Testing:** Implement unit and integration tests for your Node.js API, and configure them to run within a Docker container as part of your development workflow.
- **Environment-Specific Configuration:** Explore using multiple Docker Compose files (e.g., `docker-compose.dev.yaml` for development, `docker-compose.prod.yaml` for production) to manage different configurations and optimize for each environment.

Your ability to containerize and orchestrate such systems is a critical and highly sought-after skill in modern software development and DevOps practices.

---

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

## References


- Docker Compose documentation
- PostgreSQL Docker Official Image
- Node.js Docker Official Image
- Best practices for writing Dockerfiles
- Docker volumes documentation
- Alpine Linux `apk` package manager documentation
- Node.js `pg` module documentation

CHAPTER 04

# Project 3: Resilient Background Processing with Python, Redis, and Health Checks

## Introduction: Building Resilient Background Services

In modern applications, many tasks are too time-consuming to execute synchronously within a user's request. Think about sending email notifications, processing large image uploads, generating complex reports, or performing data analytics. Blocking the user interface for these operations leads to poor user experience and can cause timeouts. This is where background processing comes in.

This chapter guides you through building a resilient background worker system using Docker Compose. We'll set up a Python worker that processes jobs from a Redis queue, ensuring data persistence and implementing health checks for service reliability. By the end, you'll have a robust, containerized system capable of offloading heavy tasks, significantly improving your application's responsiveness and fault tolerance.

## Planning & Design: Asynchronous Task Execution

The core problem we're solving is how to execute long-running or resource-intensive tasks without directly impacting the main application's performance or user experience. Our solution involves a classic message queue pattern: a producer pushes tasks onto a queue, and one or more consumers (workers) pull tasks from the queue and process them asynchronously.

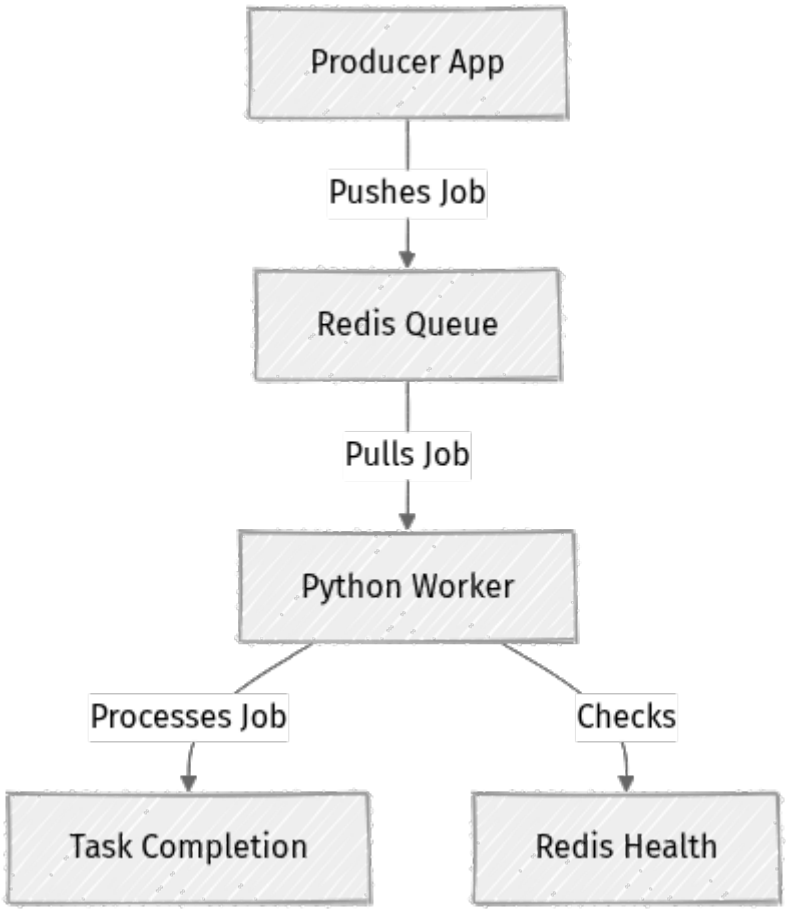
### Architecture Overview

Our system will consist of three main logical components, all orchestrated by Docker Compose:

1. **Producer (Simulated):** A simple Python script that simulates a web application pushing jobs (e.g., "process image", "send email") onto a Redis queue.

2. **Redis:** An in-memory data store used as our message broker. It will store the job queue and ensure jobs persist even if the worker temporarily fails.
3. **Python Worker:** A Python application that continuously monitors the Redis queue, fetches pending jobs, performs a simulated "heavy computation," and marks the job as complete.

This setup decouples the task creation from task execution, making the system more scalable and fault-tolerant.




## Key Docker Concepts Applied

In this project, we'll leverage several critical Docker concepts:

- **Docker Compose:** To define and run our multi-service application (Redis and Python Worker) as a single unit.
- **Custom Dockerfile:** For our Python worker, enabling us to install dependencies and configure the environment. We'll use multi-stage builds for efficiency.
- **Docker Volumes:** To ensure Redis data (our job queue) persists even if the Redis container is stopped or restarted.

- **Docker Networks:** Docker Compose automatically sets up a default network, allowing services to communicate by their service names (e.g., `redis`).
- **Environment Variables:** To configure our worker (e.g., the Redis host) without hardcoding values.
- **Health Checks:** To monitor the operational status of our worker service, allowing Docker Compose to react if a service becomes unhealthy.
- **Non-root User:** Running containers with a dedicated, non-root user for enhanced security.

---

## Step-by-Step Implementation

Let's start building our background processing system. We'll create a new directory for this project.

```
mkdir project3-background-worker
cd project3-background-worker
```

### Step 1: Set up the Python Worker Application

First, we'll create the Python application that will act as our worker.

Create a directory named `worker` inside `project3-background-worker`:

```
mkdir worker
```

Inside the `worker` directory, create a file named `requirements.txt`:

**`project3-background-worker/worker/requirements.txt`**

```
redis==5.0.1
```

⚡ **Quick Note:** We're using `redis==5.0.1`. The `redis-py` library is the official Python client for Redis, providing a straightforward API for interacting with Redis servers. We specify a version to ensure reproducibility.

Next, create the main worker script, `app.py`:

**`project3-background-worker/worker/app.py`**

```
import os
import time
import redis
```

```
import json

# Retrieve Redis connection details from environment variables
REDIS_HOST = os.getenv('REDIS_HOST', 'localhost')
REDIS_PORT = int(os.getenv('REDIS_PORT', 6379))
REDIS_DB = int(os.getenv('REDIS_DB', 0))

# Connect to Redis
try:
    r = redis.Redis(host=REDIS_HOST, port=REDIS_PORT, db=REDIS_DB, decode_resp
onses=True)
    r.ping()
    print(f"Connected to Redis at {REDIS_HOST}:{REDIS_PORT}")
except redis.exceptions.ConnectionError as e:
    print(f"Could not connect to Redis: {e}")
    exit(1)

# Function to simulate a heavy task
def process_job(job_data):
    print(f"Processing job: {job_data['id']} - Type: {job_data['type']}")
    # Simulate work by sleeping for a random duration
    time.sleep(job_data.get('duration', 5))
    print(f"Job {job_data['id']} completed after {job_data.get('duration',
5)} seconds.")
    return f"Job {job_data['id']} processed."

# Main worker loop
if __name__ == "__main__":
    print("Python Worker started. Waiting for jobs...")
    while True:
        # Blocking pop from the 'task_queue' list in Redis
        # This will wait indefinitely until a job is available
        _, job_json = r.brpop('task_queue', timeout=60) # timeout is for
graceful shutdown in real apps

        if job_json:
            job_data = json.loads(job_json)
            try:
                process_job(job_data)
            except Exception as e:
                print(f"Error processing job {job_data.get('id', 'unknown')}:
{e}")
        else:
            print("No job in queue, waiting...")
        time.sleep(1) # Small delay to prevent busy-waiting if brpop times out
```

**Explanation of `app.py`:**

- **Environment Variables:** `REDIS_HOST`, `REDIS_PORT`, `REDIS_DB` are loaded from environment variables, making the worker configurable without code changes. Defaults are provided for local testing.
- **Redis Connection:** It attempts to connect to Redis and pings it to verify the connection. If connection fails, the worker exits.

- **`process_job`:** This function simulates a long-running task using `time.sleep()`. In a real application, this would be where your actual business logic (e.g., image resizing, API calls) resides.
- **Worker Loop:** The `while True` loop continuously fetches jobs.
- **`r.brpop('task_queue', timeout=60)`:** This is a crucial Redis command. `BRPOP` (blocking right pop) removes and returns the last element from the `task_queue` list. If the list is empty, it blocks the connection until an element is available or the `timeout` (60 seconds here) is reached. This is efficient as it doesn't busy-wait.
- **JSON Handling:** Jobs are expected to be JSON strings, which are then parsed.
- **Error Handling:** Basic `try-except` block around job processing to prevent a single bad job from crashing the worker.

### Step 2: Create the Dockerfile for the Python Worker

Now, let's containerize our Python worker. We'll use a multi-stage build to create a lean production image.

Create a file named `Dockerfile` inside the `worker` directory:

**`project3-background-worker/worker/Dockerfile`**

```
# Stage 1: Build stage
# Use a specific Python version for consistency.
# Python 3.12-slim-bookworm is a good balance of features and size for 2026.
FROM python:3.12-slim-bookworm AS builder

# Set working directory
WORKDIR /app

# Install build dependencies if needed (e.g., for some pip packages)
# For 'redis-py', often no specific build dependencies are strictly required
# but it's good practice to include them if your project grows.
# RUN apt-get update && apt-get install -y --no-install-recommends \
#     build-essential \
#     && rm -rf /var/lib/apt/lists/*

# Copy requirements file and install Python dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Stage 2: Production stage
# Use a minimal base image for the final runtime
FROM python:3.12-slim-bookworm

# Set working directory
WORKDIR /app

# Create a non-root user for security
RUN adduser --system --group appuser
```

```
USER appuser

# Copy only the installed dependencies and application code from the builder
stage
COPY --from=builder /usr/local/lib/python3.12/site-packages /usr/local/lib/
python3.12/site-packages
COPY app.py .

# Command to run the worker
CMD ["python", "app.py"]
```

**Explanation of `Dockerfile`:**

- **`FROM python:3.12-slim-bookworm AS builder`**: Starts with a slim Python image (based on Debian Bookworm) for the build stage. This is a current, lightweight base image as of 2026-08-06.
- **`WORKDIR /app`**: Sets the working directory inside the container.
- **`COPY requirements.txt .` & `RUN pip install ...`**: Copies the `requirements.txt` and installs Python dependencies. Using `--no-cache-dir` reduces image size.
- **`FROM python:3.12-slim-bookworm`**: The second stage uses the same minimal base image.
- **`RUN adduser --system --group appuser` & `USER appuser`**: **Crucially**, this creates a non-root `appuser` and switches to it. Running containers as non-root is a fundamental security best practice, preventing potential privilege escalation if vulnerabilities are exploited.
- **`COPY --from=builder ...`**: This copies only the installed Python packages from the builder stage to the production stage.
- **`COPY app.py .`**: Copies our worker application script. Since the build context is `./worker` and `WORKDIR` is `/app`, `app.py` refers to `project3-background-worker/worker/app.py`.
- **`CMD ["python", "app.py"]`**: Defines the command that runs when the container starts.

### Step 3: Define Services with Docker Compose

Now, let's orchestrate our Redis and Python worker services using Docker Compose.

Create a file named `docker-compose.yaml` in the root `project3-background-worker` directory:

**`project3-background-worker/docker-compose.yaml`**

```yaml
# Use a recent Compose file format for Docker Compose v2.40.3
version: '3.8'

services:
  redis:
    image: redis:7.2.5-alpine # A lightweight and stable Redis version as of
2026-08-06
    container_name: project3_redis
    ports:
      - "6379:6379" # Expose Redis port for local testing/debugging
    volumes:
      - redis_data:/data # Persist Redis data to a named volume
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 5s
      timeout: 5s
      retries: 5
      start_period: 10s

  worker:
    build:
      context: ./worker # Build context is the 'worker' directory
      dockerfile: Dockerfile
    container_name: project3_worker
    environment:
      # Pass Redis connection details to the worker
      REDIS_HOST: redis # Use the service name 'redis' for inter-container
communication
      REDIS_PORT: 6379
      REDIS_DB: 0
    depends_on:
      redis:
        condition: service_healthy # Ensure Redis is healthy before starting
the worker
    healthcheck:
      test: ["CMD-SHELL", "pgrep -f app.py || exit 1"] # Check if the Python
process is running
      interval: 10s
      timeout: 5s
      retries: 3
      start_period: 20s
    restart: on-failure # Automatically restart if the worker crashes

volumes:
  redis_data: # Define the named volume for Redis persistence
```

### Explanation of `docker-compose.yaml`:

- **`version: '3.8'`**: Specifies the Compose file format version. `3.8` is a common and robust choice compatible with Docker Compose 2.40.3.

- **`redis` service:**
  - **`image: redis:7.2.5-alpine`**: Uses the official `redis` image, specifically a lightweight Alpine Linux variant for a smaller footprint. Version 7.2.5 is a stable release as of 2026-08-06.
  - **`ports: - "6379:6379"`**: Maps the container's Redis port to the host's port, allowing you to connect to Redis directly from your host if needed.
  - **`volumes: - redis_data:/data`**: Mounts a named Docker volume (`redis_data`) to the `/data` directory inside the Redis container. This is where Redis persists its data (RDB snapshots, AOF logs), ensuring that our job queue is not lost if the container restarts.
  - **`healthcheck`**: Defines how Docker Compose can determine if the Redis service is healthy. It uses `redis-cli ping` to check connectivity. `start_period` gives Redis time to initialize.
- **`worker` service:**
  - **`build: context: ./worker`**: Tells Docker Compose to build the image for this service using the `Dockerfile` found in the `./worker` directory.
  - **`environment`**: Passes the `REDIS_HOST`, `REDIS_PORT`, and `REDIS_DB` environment variables to the worker container. Notice `REDIS_HOST: redis` – Docker Compose automatically sets up internal DNS resolution, so containers can reach each other using their service names.
  - **`depends_on: redis: condition: service_healthy`**: This is a critical feature. It ensures that the `worker` service will only start once the `redis` service is reported as `healthy` by its health check. This prevents the worker from trying to connect to a Redis instance that isn't fully ready.
  - **`healthcheck`**: For the worker, we use `pgrep -f app.py` to check if our Python script is actively running. If `pgrep` fails (returns a non-zero exit code), the health check fails.
  - **`restart: on-failure`**: Configures the worker container to automatically restart if it exits with a non-zero status (i.e., crashes). This adds resilience.
- **`volumes: redis_data:`**: Declares the `redis_data` named volume, which Docker will manage.

### Step 4: Create a Job Producer (for Testing)

To test our worker, we need a way to put jobs into the Redis queue.

Create a file named `producer.py` in the root `project3-background-worker` directory:

**`project3-background-worker/producer.py`**

```
import os
import redis
import json
import time
import uuid
import random

# Retrieve Redis connection details from environment variables
# For local execution, assume Redis is on localhost if not containerized
REDIS_HOST = os.getenv('REDIS_HOST', 'localhost')
REDIS_PORT = int(os.getenv('REDIS_PORT', 6379))
REDIS_DB = int(os.getenv('REDIS_DB', 0))

try:
    r = redis.Redis(host=REDIS_HOST, port=REDIS_PORT, db=REDIS_DB, decode_resp
onses=True)
    r.ping()
    print(f"Producer connected to Redis at {REDIS_HOST}:{REDIS_PORT}")
except redis.exceptions.ConnectionError as e:
    print(f"Producer could not connect to Redis: {e}")
    exit(1)

def publish_job(job_type, duration=None):
    job_id = str(uuid.uuid4())
    job_data = {
        'id': job_id,
        'type': job_type,
        'timestamp': time.time()
    }
    if duration:
        job_data['duration'] = duration

    job_json = json.dumps(job_data)
    r.lpush('task_queue', job_json) # Push job to the left of the list
    print(f"Published job {job_id} of type '{job_type}' to 'task_queue'.")

if __name__ == "__main__":
    print("Starting job producer...")
    while True:
        # Publish different types of jobs with varying durations
        job_types = ["image_resize", "send_email", "report_generation", "data_
cleanup"]
        selected_type = random.choice(job_types)
        job_duration = random.randint(2, 10) # Simulate 2-10 seconds of work

        publish_job(selected_type, job_duration)
        time.sleep(random.randint(1, 3)) # Wait 1-3 seconds before publishing
next job
```

**Explanation of `producer.py`:**

- **Redis Connection:** Similar to the worker, it connects to Redis. For this script, we'll run it from our host machine, so `REDIS_HOST` defaults to `localhost`.
- **`publish_job`:** Creates a unique job ID, constructs a JSON payload, and uses `r.lpush('task_queue', job_json)` to add the job to the left side of the Redis list. `LPUSH` is the counterpart to `BRPOP` for pushing items to the queue.
- **Loop:** Continuously publishes random job types with random durations every few seconds.

---

## Testing & Verification

Now that all our files are in place, let's bring up our services and verify everything is working.

1. **Build and Run Services:** Navigate to the `project3-background-worker` directory in your terminal and run:

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

```
- `up`: Starts the services defined in `docker-compose.yaml`.
- `--build`: Forces Docker Compose to rebuild images if their definitions
(like our `worker/Dockerfile`) have changed.
- `-d`: Runs the containers in detached mode (in the background).

You should see output indicating that the services are being created and
started. Pay attention to the `condition: service_healthy` for `redis` before
`worker` starts.
```

1. **Monitor Worker Logs:** To see the worker processing jobs, stream its logs:

```
    docker compose logs -f worker
```

```
Initially, the worker will start and print "Python Worker started. Waiting for
jobs..." and "No job in queue, waiting...". This is expected because we
haven't sent any jobs yet.
```

1. **Send Jobs with the Producer:** Open a new terminal window, navigate to `project3-background-worker`, and run the producer script directly on your host machine:

```
    python producer.py
```

```
You should see output in this terminal indicating jobs being published to
Redis.
```

1. **Observe Worker Processing:** Switch back to the terminal where you're watching the `worker` logs. You should now see the worker picking up jobs from the queue, printing "Processing job..." and "Job ... completed..." messages.

```
    project3_worker_1  | Python Worker started. Waiting for jobs...
    project3_worker_1  | Connected to Redis at redis:6379
    project3_worker_1  | Processing job: e1b2c3d4-e5f6-7890-a1b2-c3d4e5f67890
- Type: image_resize
    project3_worker_1  | Job e1b2c3d4-e5f6-7890-a1b2-c3d4e5f67890 completed
after 5 seconds.
    project3_worker_1  | Processing job: f1g2h3i4-j5k6-7890-a1b2-c3d4e5f67890
- Type: send_email
    project3_worker_1  | Job f1g2h3i4-j5k6-7890-a1b2-c3d4e5f67890 completed
after 8 seconds.
```

1. **Check Service Health:** In another terminal, you can check the health status of your services:

```
    docker compose ps
```

```
You should see `(healthy)` next to both `project3_redis` and
`project3_worker`. This confirms our health checks are working.
```

```
    NAME                IMAGE                     COMMAND
SERVICE    CREATED          STATUS                    PORTS
    project3_redis      redis:7.2.5-alpine        "docker-entrypoint.s…"
redis      10 seconds ago   running (healthy)         0.0.0.0:6379->6379/tcp
    project3_worker     project3-background-wo…   "python app.py"
```

```
worker     8 seconds ago    running (healthy)
```

1. **Verify Redis Data Persistence (Optional):** Stop the `redis` service:

```
    docker compose stop redis
```

```
Wait a few seconds, then start it again:
```

```
    docker compose start redis
```

```
The `worker` should reconnect to `redis` (after its own restart or connection
retry logic). If you stop the `producer.py` and let the worker clear the
queue, then restart `redis`, any jobs still in the queue before the stop would
still be there after the restart. This demonstrates the volume's
effectiveness.

When you're done, stop and remove the containers:
```

```
    docker compose down
```

```
This command will stop and remove the containers and networks. **Note that by
default, `docker compose down` does not remove named volumes.** To also remove
the `redis_data` volume (and thus clear any persistent Redis data), use:
```

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

## Production Considerations

Building a background processing system for production requires more than just functional code.

- **Scaling Workers:** Docker Compose allows you to easily scale your workers. To run multiple worker instances, use:

```
    docker compose up --scale worker=3 -d
```

```
This will start three `worker` containers, all consuming jobs from the same
Redis queue. This is a fundamental way to increase throughput for background
tasks.
```

- **Robustness and Retries:** Our current worker processes a job once. In a real system, jobs might fail due to transient issues (e.g., external API downtime). Implement retry mechanisms (e.g., exponential backoff) and potentially a dead-letter queue (DLQ) for jobs that consistently fail after multiple retries.
- **Monitoring and Alerting:** Integrate logging (e.g., structured JSON logs for easier parsing), metrics (e.g., job processing rates, queue length), and alerting (e.g., notify if queue length is too high or workers are unhealthy). Tools like Prometheus and Grafana are commonly used for this.
- **Resource Limits:** Define CPU and memory limits for your worker containers in `docker-compose.yaml` to prevent a runaway worker from consuming all host resources:

```
    worker:
      # ... other configurations
      deploy:
        resources:
          limits:
            cpus: '0.5' # 50% of a CPU core
            memory: 512M # 512 MB of RAM
```

```
`🧠 Important:` These `deploy` limits are typically honored by orchestrators
like Swarm or Kubernetes, but Docker Desktop and `docker compose` can apply
them locally as well.
```

- **Security for Redis:** While we exposed Redis on `6379` for convenience, in production, Redis should generally not be exposed directly to the public internet. Use a private network, strong passwords (with the `requirepass` directive in Redis config), or TLS encryption. Docker secrets can be used to pass sensitive passwords securely.

## Common Issues & Solutions

1. **Worker Cannot Connect to Redis:**
    - **Issue:** You see `Could not connect to Redis` errors in the worker logs.
    - **Solution:**
        - Verify the `REDIS_HOST` environment variable in `docker-compose.yaml` for the `worker` service. It should be `redis` (the service name), not `localhost`.
        - Check if the `redis` service is actually running and healthy (`docker compose ps`).
        - Ensure no firewall is blocking internal Docker network communication (less common for Docker Compose default networks).
        - Check Redis logs for any startup errors.
2. **Jobs Aren't Being Processed (Worker is Idle):**
    - **Issue:** `producer.py` is pushing jobs, but the worker logs show "No job in queue, waiting...".
    - **Solution:**
        - Confirm `producer.py` is connected to the correct Redis instance. If running `producer.py` on the host, ensure Redis is exposed via `ports: "6379:6379"` in `docker-compose.yaml` and `producer.py` is configured to connect to `localhost:6379`.
        - Check for typos in the queue name (`task_queue`) in both `producer.py` and `worker/app.py`.
        - Verify the worker container is running and healthy (`docker compose ps`).
3. **Redis Data Loss After Container Restart:**
    - **Issue:** Jobs disappear from the queue if you stop and restart the `redis` container.
    - **Solution:** Ensure the `volumes: - redis_data:/data` line is correctly configured for the `redis` service in `docker-compose.yaml`, and the `volumes: redis_data:` declaration exists at the top level. This named volume ensures data persistence.

## Summary & Next Steps

In this chapter, you've successfully built a resilient background processing system using Python, Redis, and Docker Compose. You've learned how to:

- Containerize a multi-service application.
- Implement data persistence using Docker volumes.
- Configure inter-container communication.
- Use environment variables for flexible configuration.
- Add health checks for robust service management.
- Apply multi-stage Docker builds and non-root users for security and efficiency.

This project demonstrates a common pattern for building scalable and reliable microservices. The concepts of message queues, workers, and health checks are fundamental in distributed systems.

Next, we'll explore more advanced Docker concepts, including custom networking, volume types, and image optimization techniques, to further enhance your containerization skills.

## References


- Docker Compose documentation (v2.40.3)
- Redis official documentation
- Official Python Docker Images
- docker/docker-bench-security: Docker Bench for Security
- Best practices for writing Dockerfiles


---

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 05

# Advanced Docker Concepts: Networking, Volumes, and Image Optimization

## Deeper Dive into Docker: Networking, Volumes, and Image Optimization

You've successfully containerized several applications, using Docker and Docker Compose to orchestrate multi-service environments. You've seen the power of isolation and portability firsthand. But as you move towards shipping real-world projects, you'll encounter critical questions: How do your containers securely communicate? Where does application data live if containers are ephemeral? And how can you ensure your deployed images are as small and secure as possible?

This chapter is your deep dive into these fundamental questions. We'll elevate your Docker skills by exploring advanced concepts crucial for robust, efficient, and production-ready applications. By the end, you'll be able to design more resilient, performant, and maintainable containerized systems, ready for deployment.

### Planning and Design

Before we jump into code, let's frame what we'll achieve in this chapter and why these concepts are so vital for any serious Docker project.

#### Project Overview

This chapter focuses on mastering three core advanced Docker functionalities:

1. **Custom Networking:** Beyond the default, we'll configure isolated networks for secure and efficient inter-container communication.
2. **Persistent Volumes:** We'll ensure critical application data survives container lifecycles by properly managing Docker volumes.
3. **Image Optimization:** We'll dramatically reduce image sizes and improve security using multi-stage builds and non-root users.

Each of these elements is a cornerstone of building reliable, scalable, and secure containerized applications.

### Tech Stack

For our practical examples, we'll continue to leverage the core Docker tooling:

- **Docker Engine:** Version `29.1.x` (as of 2026-08-06)
- **Docker Compose:** Version `2.40.3` (as of 2026-08-06)
- **Base Images:** We'll use lightweight Alpine-based images for Nginx, Redis, and Node.js (`nginx:1.25.3-alpine`, `redis:7.2.4-alpine`, `node:20.10.0-alpine3.18`).

### Milestones for This Chapter

We'll tackle these advanced topics incrementally, each building on the last:

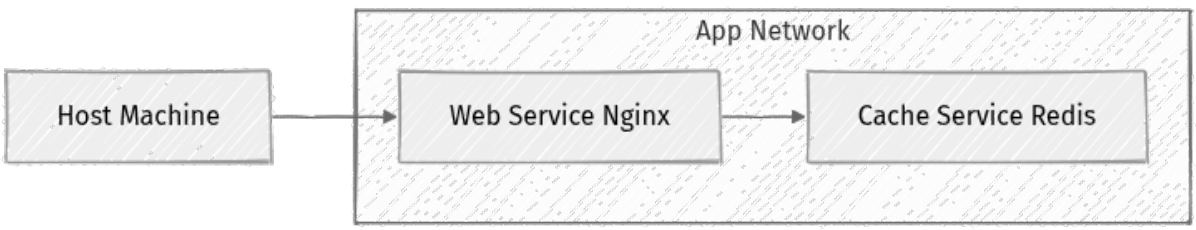
1. **Networking Setup:** Create a custom bridge network for a web and cache service.
2. **Volume Integration:** Add a named volume to persist data for a Redis cache.
3. **Image Optimization:** Implement a multi-stage Dockerfile for a Node.js application, reducing image size and enhancing security.

### Architectural Principles

Our approach to these advanced concepts will emphasize:

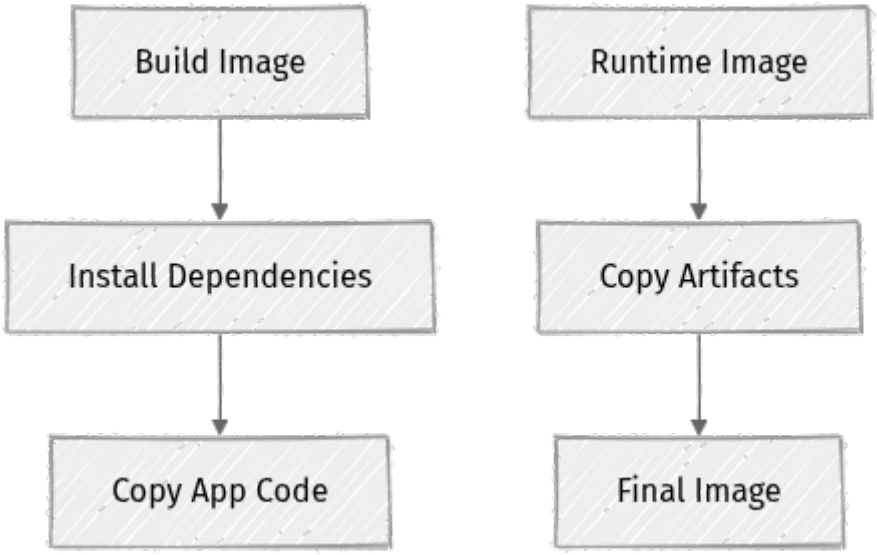
- **Loose Coupling:** Services should communicate via well-defined interfaces over networks, without direct host dependencies.
- **Stateless Containers:** Application containers should ideally be stateless, offloading data persistence to dedicated volumes or external services.
- **Security by Design:** Minimal privilege, reduced attack surface, and clear separation of concerns are paramount.

Here's a high-level view of how a custom network facilitates communication between services:




The *`app_network`* isolates communication between the *`WebServer`* and *`CacheService`*, while the *`Host_OS`* can access the *`WebServer`* via exposed ports.

And here's the conceptual flow of a multi-stage build:




The `BuildStage` handles compilation and dependency installation, passing only essential artifacts to the `ProductionStage` to create a smaller, more secure `FinalImage`.

## Understanding the Interconnected World of Docker Networking

When you run multiple containers, they often need to talk to each other. Docker provides sophisticated networking capabilities to facilitate this communication securely and efficiently. By default, Docker Compose creates a default network for your services, but custom networks offer more control and isolation.

### Why Custom Networks Matter

While the default bridge network works for basic setups, custom networks provide significant advantages for production environments:

- **Isolation:** Services on different custom networks cannot communicate unless explicitly configured, enhancing security and preventing unintended interactions.
- **Service Discovery:** Containers on the same custom network can discover each other by their service names (defined in `docker-compose.yml`), eliminating the need to hardcode IP addresses or manage complex DNS. Docker's internal DNS handles this seamlessly.
- **Network Scoping:** You can define specific networks for different application tiers (e.g., `frontend_net`, `backend_net`, `database_net`), improving organization, security, and making firewall rules easier to implement.

Let's illustrate this by setting up a custom network for a simple web application and a database. We'll use a basic `nginx` service and a `redis` instance, ensuring they can communicate over a dedicated network.

### Implementation: Custom Docker Network

First, let's create a directory for our example.

```
mkdir -p docker-advanced/networking-example
cd docker-advanced/networking-example
```

Now, create a `docker-compose.yml` file:

**`docker-advanced/networking-example/docker-compose.yml`**

```
# docker-compose.yml
version: '3.8'

services:
  web:
    image: nginx:1.25.3-alpine # Using a specific, recent Alpine-based Nginx
for small size
    ports:
      - "80:80" # Expose host port 80 to container port 80
    networks:
      - app_network # Attach to our custom network
    healthcheck: # Basic health check for Nginx
      test: ["CMD", "curl", "-f", "http://localhost"]
      interval: 30s
      timeout: 10s
      retries: 3

  cache:
    image: redis:7.2.4-alpine # Specific Alpine-based Redis for small size
    networks:
      - app_network # Attach to our custom network
    healthcheck: # Basic health check for Redis
      test: ["CMD", "redis-cli", "ping"]
      interval: 30s
      timeout: 10s
      retries: 3

networks:
  app_network:
    driver: bridge # Explicitly define a bridge network
```

### Explanation of the `docker-compose.yml`:

- `version: '3.8'`: Specifies the Docker Compose file format version, ensuring compatibility with Docker Compose `2.40.3`.
- `services`: Defines our `web` (Nginx) and `cache` (Redis) services.

- `image`: We're using `nginx:1.25.3-alpine` and `redis:7.2.4-alpine`. Using `alpine` variants is a widely adopted best practice for smaller, more secure images in production.
- `ports`: The `web` service exposes port 80 of the container to port 80 on the host machine. The `cache` service (Redis) does not expose its port to the host, as it's only meant for internal communication within `app_network`. This is a crucial security measure.
- `networks: - app_network`: Both services are explicitly attached to our custom `app_network`. This means they will be part of the same isolated network segment.
- `healthcheck`: We've added basic `healthcheck` configurations for both services. This is a vital production practice, allowing Docker to monitor the operational status of your containers and restart unhealthy ones.
- `networks: app_network:`: This top-level key defines our custom network named `app_network`. We explicitly set its `driver` to `bridge`, which is the most common and suitable driver for single-host applications needing to communicate.

### Verification: Network Connectivity

Now, let's bring up the services and verify their network configuration.

```
docker compose up -d
```

This command will create the `app_network` and then start the `web` and `cache` containers, attaching them to the newly created network.

1. **Inspect the network:** To confirm the network was created and containers are attached, use the Docker CLI:

```
    docker network inspect networking-example_app_network
```

```
You'll see a detailed JSON output including the network's configuration,
attached containers, and their assigned IP addresses within that network. Look
for the `Containers` section to confirm `networking-example-web-1` and
`networking-example-cache-1` are listed.
```

1. **Test inter-container communication:** We can connect to the `web` container and try to `ping` the `cache` container using its service name. This tests Docker's internal DNS resolution.

```
    docker exec -it networking-example-web-1 ping cache
```

```
You should see `ping` responses, indicating that the `web` container can
successfully resolve and reach the `cache` container by its service name
(`cache`). This confirms that Docker's internal DNS is functioning correctly
within the `app_network`. Press `Ctrl+C` to stop the ping.

**📌 Key Idea:** Using service names for inter-container communication is a
fundamental Docker Compose best practice. It decouples your application from
specific IP addresses, making your setup more robust and portable.
```

Once verified, clean up the resources:

```
docker compose down
```

## Ensuring Data Persistence with Docker Volumes

Containers are designed to be ephemeral. When a container is removed, any data written inside its filesystem is lost. This is unacceptable for applications that need to store state, like databases, message queues, or user-uploaded files. Docker volumes solve this problem by providing a mechanism to persist data independently of the container's lifecycle.

### Volume Types: Bind Mounts vs. Named Volumes

Docker offers two primary ways to manage persistent data, each with its own use cases and tradeoffs:

1. **Bind Mounts:**
    - **What it is:** Mounts a file or directory from the host machine directly into a container.
    - **Pros:** Very flexible, easy to use for development (e.g., live-reloading code where you edit files on the host and see changes in the container), host path is explicit.
    - **Cons:** Host machine structure dependence (less portable), potential security risks (container can potentially modify or corrupt host files if not careful), less manageable by Docker itself.
    - **Real-world insight:** Excellent for local development and configuration files, but generally avoided for critical production data.

2. **Named Volumes (Docker Managed Volumes):**
    - **What it is:** Docker manages the creation, location, and lifecycle of the volume. You reference it by a symbolic name (e.g., `redis_data`). Docker handles the underlying host path internally.
    - **Pros:** More portable (Docker handles the underlying host path, making your `docker-compose.yml` work across different hosts without path changes), easier to back up and manage with Docker CLI commands, better security (container only sees the data within the volume, not arbitrary host directories).
    - **Cons:** Less transparent where data resides on the host, can be slightly more complex to set up initially than a simple bind mount.
    - **Real-world insight: Preferred for most production scenarios** involving databases, message queue data, or other critical application state due to their portability, manageability, and security benefits.

🧠 **Important:** For production-grade applications, especially those relying on databases or critical application data, **named volumes are the go-to solution**. They offer a superior balance of portability, data integrity, and security compared to bind mounts.

### Implementation: Named Volumes

Let's enhance our previous `redis` example to use a named volume for data persistence. This ensures that even if the Redis container is removed, its data (e.g., cached items, session information) remains intact and can be re-attached to a new container.

Create a new directory for this example:

```
mkdir -p docker-advanced/volumes-example
cd docker-advanced/volumes-example
```

**`docker-advanced/volumes-example/docker-compose.yml`**

```
# docker-compose.yml
version: '3.8'

services:
  cache:
    image: redis:7.2.4-alpine # Specific Alpine-based Redis for small size
    command: redis-server --appendonly yes # Enable AOF persistence for Redis
    volumes:
      - redis_data:/data # Mount the named volume into the container's /data
directory
    healthcheck: # Basic health check for Redis
```

```
      test: ["CMD", "redis-cli", "ping"]
      interval: 30s
      timeout: 10s
      retries: 3

volumes:
  redis_data: # Define the named volume at the top level
    driver: local # Explicitly use the local driver (default and generally
sufficient)
```

**Explanation of the `docker-compose.yml`:**

- `command: redis-server --appendonly yes`: We've added a custom command to the Redis service to enable AOF (Append Only File) persistence. This is a Redis-specific feature that writes every command received by the server to a log file, making data recoverable upon restart. Without this, Redis data is usually only in memory.
- `volumes: - redis_data:/data`: This line is the key to persistence. It instructs Docker to mount a named volume called `redis_data` into the `/data` directory inside the Redis container. Redis stores its persistence files (like `appendonly.aof`) in `/data` by default, so this ensures they are written to our persistent volume.
- `volumes: redis_data:`: At the top level of the `docker-compose.yml` file, we explicitly define the `redis_data` named volume. Docker will automatically create and manage this volume for us. `driver: local` specifies that the volume should be stored on the local filesystem of the Docker host, which is the default and suitable for most single-host setups.

## Verification: Volume Persistence

Let's demonstrate that data persists across container restarts and removals, proving the value of named volumes.

1. **Bring up the service:**

```
    docker compose up -d
```

```
This will create the `redis_data` named volume (if it doesn't already exist)
and start the `cache` container.
```

1. **Add some data to Redis:** Connect to the Redis container's CLI and set a key-value pair.

```
    docker exec -it volumes-example-cache-1 redis-cli
```

```
Inside the `redis-cli` prompt, type:
```

```
    SET mykey "Hello, Docker Volumes!"
    GET mykey
```

```
You should see `OK` after `SET` and then `"Hello, Docker Volumes!"` returned
for `GET`. This confirms data is in Redis. Type `exit` to leave the `redis-
cli`.
```

1. **Remove the container (but keep the volume):** Now, let's stop and remove the container. Crucially, we will not remove the volume yet.

```
    docker compose stop
    docker compose rm -f
```

```
This stops and removes the `cache` container. The `redis_data` volume,
however, remains untouched and managed by Docker. You can verify its existence
with `docker volume ls`.
```

1. **Start a new container and check data:** Bring up a new `cache` container. Docker will reuse the existing `redis_data` volume.

```
    docker compose up -d
    docker exec -it volumes-example-cache-1 redis-cli
```

```
Inside the `redis-cli` prompt, type:
```

```
    GET mykey
```

```
You should still see `"Hello, Docker Volumes!"`. This definitively confirms
that the data persisted across the removal and recreation of the container,
thanks to the named volume!
```

1. **Clean up:** When you are completely done with the project and its data, remember to remove the volume.

```
docker compose down -v # The -v flag removes named volumes as well
```

```
**⚠️ What can go wrong:** Forgetting the `-v` flag with `docker compose down`
will leave unused volumes on your system. While sometimes desired (e.g., for
debugging), these can accumulate and consume significant disk space over time.
Regularly prune them using `docker volume prune` or explicitly use `-v` when
you intend to remove data.
```

## Optimizing Docker Images for Production

The size and content of your Docker images directly impact deployment speed, security, and resource consumption. Large images are slow to pull, consume more storage, and often contain unnecessary tools or dependencies that increase the attack surface. Image optimization is a critical production best practice.

### Multi-Stage Builds: The Game Changer

Multi-stage builds are a powerful Dockerfile feature introduced in Docker Engine 17.05 (well before our 29.1.x version). They allow you to use multiple `FROM` statements in a single Dockerfile. Each `FROM` instruction can use a different base image, and each stage can copy only the necessary artifacts from previous stages.

#### Why are multi-stage builds so important?

- **Smaller Images:** You can compile/build your application in an "intermediate" stage with all necessary build tools (e.g., compilers, Node.js development dependencies, large SDKs) and then copy only the essential compiled artifacts or runtime code to a much smaller, "final" stage. This drastically reduces the final image size.
- **Improved Security:** The final image contains only what's needed to run the application, reducing the number of packages, libraries, and potential vulnerabilities from build-time dependencies.
- **Cleaner Dockerfiles:** It cleanly separates build logic from runtime logic, making Dockerfiles easier to read, understand, and maintain.

- **Faster Deployment:** Smaller images mean faster pulls from registries, leading to quicker deployments and scaling operations.

### Implementation: Multi-Stage Build Example (Node.js)

Let's create a simple Node.js application and demonstrate how to optimize its Dockerfile using multi-stage builds.

Create a new directory:

```
mkdir -p docker-advanced/multi-stage-example
cd docker-advanced/multi-stage-example
```

**`docker-advanced/multi-stage-example/app.js`**

```
// app.js
const http = require('http');

const hostname = '0.0.0.0'; // Listen on all network interfaces
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello from a multi-stage Docker container!\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

This is a basic Node.js HTTP server.

**`docker-advanced/multi-stage-example/package.json`**

```
{
  "name": "multi-stage-app",
  "version": "1.0.0",
  "description": "A simple Node.js app for multi-stage build demo",
  "main": "app.js",
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    # No specific external dependencies for this simple app, but in a real app
lication, they would be listed here.
    # For example: "express": "^4.18.2"
  }
}
```

This file defines our Node.js project and the `start` script.

Now, the optimized Dockerfile using a multi-stage build:

**`docker-advanced/multi-stage-example/Dockerfile`**

```
# Dockerfile

# --- Stage 1: Build Stage ---
# Use a Node.js image with build tools. We name this stage 'builder'.
FROM node:20.10.0-alpine3.18 AS builder

WORKDIR /app

# Copy package.json and package-lock.json first.
# This optimizes Docker's cache: if only app.js changes, npm install won't re-
run.
COPY package*.json ./

# Install Node.js dependencies.
RUN npm install

# Copy the rest of the application code.
COPY . .

# --- Stage 2: Production Stage ---
# Start a new, much smaller base image for the production runtime.
# We use the same Alpine base for consistency and minimal footprint.
FROM node:20.10.0-alpine3.18

WORKDIR /app

# Copy only the necessary files from the 'builder' stage.
# This is the core of multi-stage builds: we discard all build tools.
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/app.js .
COPY --from=builder /app/package*.json . # Copy package.json for the 'npm
start' script

# Expose the application port.
EXPOSE 3000

# Run the application as a non-root user - a critical security best practice.
# Create a dedicated non-root user and group.
RUN addgroup --system appgroup && adduser --system --ingroup appgroup appuser
USER appuser # Switch to the non-root user

# Define the command to run the application when the container starts.
CMD ["npm", "start"]
```

**Explanation of the `Dockerfile`:**

- **Stage 1 (`builder`):**
  - `FROM node:20.10.0-alpine3.18 AS builder`: We start with a specific Node.js image based on Alpine Linux and name this stage `builder`. This image includes `npm` and all necessary tools for installing dependencies.
  - `WORKDIR /app`: Sets the working directory inside the container.
  - `COPY package*.json ./`: Copies `package.json` and `package-lock.json`. This is a Docker caching optimization: if only source code changes (not dependencies), this layer (and `npm install`) can be reused from the cache, speeding up builds.
  - `RUN npm install`: Installs all Node.js dependencies.
  - `COPY . .`: Copies the rest of the application code into the builder stage.

- **Stage 2 (Production):**
  - `FROM node:20.10.0-alpine3.18`: This is the start of a new, clean build stage. We use the same minimal Node.js Alpine image, but this new stage is completely independent of the `builder` stage's filesystem. All the build tools from the first stage are discarded.
  - `COPY --from=builder /app/node_modules ./node_modules`: This is the magic of multi-stage builds. It copies only the `node_modules` directory from the `builder` stage into our new, lean production image. All the intermediate build artifacts and development dependencies are left behind.
  - `COPY --from=builder /app/app.js .`, `COPY --from=builder /app/package*.json .`: Copies the application entry point and `package.json` (which is needed for the `npm start` command) from the `builder` stage.
  - `EXPOSE 3000`: Declares that the container listens on port 3000. This is documentation; it doesn't actually publish the port.
  - `RUN addgroup --system appgroup && adduser --system --ingroup appgroup appuser`: **Security Best Practice!** We explicitly create a dedicated system group (`appgroup`) and a non-root system user (`appuser`). Running containers as `root` is a significant security risk.
  - `USER appuser`: **Security Best Practice!** Switches the user for subsequent commands and the `CMD` to `appuser`. This ensures the application runs with the principle of least privilege.
  - `CMD ["npm", "start"]`: Defines the command to run the application when the container starts.

### Building and Verifying the Image

1. **Build the image:** Navigate to the `multi-stage-example` directory and build your optimized image:

```
docker build -t my-multi-stage-app:1.0.0 .
```

```
Docker will execute both stages, but only the artifacts copied to the second
stage will be part of the final image.
```

1. **Inspect image size:** Compare the size of this image. If you were to build a single-stage Dockerfile that installed all dependencies and then ran the app, the image would be larger due to leftover build tools and caches.

```
    docker images my-multi-stage-app:1.0.0
```

```
You'll notice the image size is significantly smaller than a naive single-
stage build. For simple Node.js apps, `node:alpine` is already small, but in
complex applications with many build tools (e.g., C++ compilers, large SDKs,
frontend build processes), the size savings are dramatic.
```

1. **Run the container:**

```
    docker run -p 8080:3000 my-multi-stage-app:1.0.0
```

```
This command runs the container, mapping host port `8080` to container port
`3000`. Open your browser to `<http://localhost:8080`>. You should see "Hello
from a multi-stage Docker container!".

**⚡ Quick Note:** You can inspect the running user inside the container by
running `docker exec -it <container_id> whoami`. It should output `appuser`,
confirming our security best practice is in place.
```

1. **Clean up:**

```
    docker stop $(docker ps -q --filter ancestor=my-multi-stage-app:1.0.0)
    docker rm $(docker ps -aq --filter ancestor=my-multi-stage-app:1.0.0)
    docker rmi my-multi-stage-app:1.0.0
```

### Production Considerations

Implementing advanced Docker concepts isn't just about making things work; it's about making them work reliably, securely, and efficiently in production.

- **Network Security:**
  - **Principle of Least Privilege:** Configure your networks and firewalls to restrict communication between containerized services only to what's absolutely necessary. For example, your database container should only be accessible from your application containers, not directly from the host or public internet.
  - **Container Network Policies:** For more advanced scenarios in orchestrators like Kubernetes, implement network policies to explicitly define which pods can communicate with each other.
- **Volume Backup Strategy:**
  - For critical data stored in named volumes, establish a robust backup and recovery strategy. This might involve regularly stopping services to ensure data consistency, creating snapshots of volumes, or using cloud-provider specific volume management tools (e.g., AWS EBS snapshots, Azure Disk Backup).
  - ⚡ **Real-world insight:** Never rely solely on Docker volumes for long-term data durability in production without a separate backup solution.
- **Image Scanning:**
  - Integrate image scanning tools (e.g., Trivy, Clair, Grype) into your CI/CD pipeline. These tools analyze your images for known vulnerabilities in their layers and dependencies, helping you catch security issues before deployment.
  - **Reference:** GitHub - docker/docker-bench-security provides a script to check for common best practices.
- **Resource Limits:**
  - In `docker-compose.yml` (or your orchestrator), always define resource limits (CPU, memory) for your services using the `deploy: resources:` key. This prevents a misbehaving or runaway container from consuming all host resources, potentially leading to system instability or denial of service for other applications.

```
# Example for resource limits in docker-compose.yml
services:
  my_service:
```

```
        image: myapp:latest
        deploy:
          resources:
            limits:
              cpus: '0.5' # Limit to 50% of one CPU core
              memory: 512M # Limit to 512 MB of memory
            reservations:
              cpus: '0.25' # Reserve 25% of one CPU core (guaranteed minimum)
              memory: 128M # Reserve 128 MB of memory (guaranteed minimum)
```

```
-   **Reference:** [Docker Docs: Runtime options with Memory, CPUs, and GPUs]
(https://docs.docker.com/config/containers/resource_constraints/)
```

### Common Issues & Solutions

Even with careful planning, you might encounter issues. Here are some common pitfalls related to advanced Docker concepts and how to debug them:

1. **"Host not found" or "Cannot connect" errors between containers:**
    - **Issue:** Your containers are attempting to communicate but are on different networks, or one container is trying to connect to another using the wrong hostname/IP.
    - **Solution:**
        - **Verify Network:** Ensure all communicating services are on the same custom network in `docker-compose.yml`.
        - **Use Service Names:** Always use the service name (e.g., `cache`, `db`) as the hostname for inter-container communication. Docker's internal DNS will resolve this.
        - **Inspect:** Use `docker network inspect <network_name>` to see which containers are attached and their internal IPs.
        - **Test Connectivity:** Use `docker exec -it <source_container> ping <destination_service_name>` to confirm network reachability and DNS resolution.

2. **Data loss after container restart/removal:**
    - **Issue:** No volume was mounted, or the wrong path was specified, leading to data being stored only within the container's ephemeral filesystem.
    - **Solution:**
        - **Use Named Volumes:** For all persistent data, always use named volumes.
        - **Verify Mount Paths:** Double-check the `volumes` mapping in `docker-compose.yml` to ensure the correct internal container path is used. For example, PostgreSQL stores data in `/var/lib/postgresql/data`, Redis in `/data`, MySQL in `/var/lib/mysql`. Consult official image documentation for default data paths.
        - **Inspect Volume:** Use `docker inspect <container_id>` and look under the `Mounts` section to see what volumes are actually mounted and where.
3. **Docker images are unexpectedly large:**
    - **Issue:** Forgetting to use multi-stage builds, not cleaning up build artifacts, or using a large base image (e.g., `ubuntu:latest` instead of `alpine`).
    - **Solution:**
        - **Implement Multi-Stage Builds:** This is the most effective way to reduce image size by separating build-time dependencies from runtime.
        - **Use Small Base Images:** Prefer `alpine` variants of official images, or even `scratch` for static binaries.
        - **`.dockerignore`:** Use a `.dockerignore` file to exclude unnecessary files (like `.git` directories, `node_modules` for the builder stage, documentation, temporary files) from being copied into the build context.
        - **Clean Up Layers:** Within `RUN` commands, combine multiple commands into a single `RUN` instruction and clean up temporary files or caches immediately (e.g., `RUN apk add --no-cache some-pkg && rm -rf /var/cache/apk/*` for Alpine).

### Summary & Next Steps

This chapter has equipped you with essential advanced Docker skills, moving you firmly into the realm of production-minded containerization:

- You've learned to create and manage **custom Docker networks** for isolated, secure, and efficient inter-container communication.
- You've mastered **Docker named volumes** to ensure your application data persists beyond the life of individual containers, a non-negotiable for stateful services.
- You've implemented **multi-stage builds** to create dramatically smaller, more secure, and faster-to-deploy Docker images, adhering to the principle of "least privilege" for image content.

These concepts are fundamental to building any robust, scalable, and secure containerized application. You are now better prepared to handle the complexities of real-world deployments and optimize your container workflows.

In the next chapter, we'll shift our focus entirely to **Security & Production Best Practices**. We'll delve deeper into hardening your Docker environments, implementing observability, and discussing deployment considerations, bringing you closer to shipping truly robust applications.

---

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

## References


- Docker Docs: Networking overview
- Docker Docs: Use volumes
- Docker Docs: Build with multi-stage Dockerfiles
- Docker Docs: Healthchecks
- Docker Docs: Runtime options with Memory, CPUs, and GPUs
- GitHub - docker/docker-bench-security

CHAPTER 06

# Container Security Hardening and Production Best Practices

## Introduction: Fortifying Your Dockerized Applications

You've successfully built several containerized applications using Docker Engine 29.1.x and Docker Compose 2.40.3. Now, it's time to shift our focus from functionality to resilience and security, crucial aspects for any production deployment. Running applications in containers introduces new considerations for security, performance, and operational stability that differ from traditional VM or bare-metal deployments.

This chapter guides you through hardening your Docker images and containers, applying production best practices to minimize attack surfaces, manage resources effectively, and ensure your applications run securely and reliably. We'll cover everything from optimizing Dockerfiles to runtime security and preparing for deployment in a production environment.

By the end of this chapter, you will understand how to:

- Build smaller, more secure Docker images using multi-stage builds.
- Run containers with non-root users to reduce privilege escalation risks.
- Implement resource limits and read-only filesystems for improved stability.
- Utilize tools like `docker-bench-security` to audit your container configurations.
- Apply best practices for managing secrets and pushing images to registries.

## Planning & Design: A Layered Security Mindset

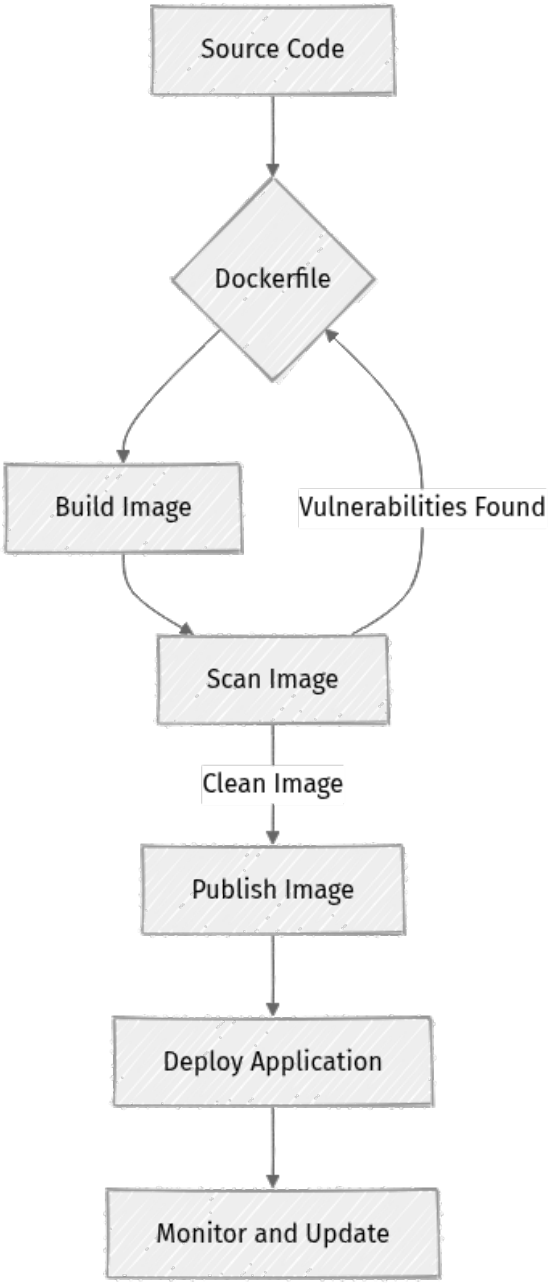
Securing containerized applications requires a layered approach, often referred to as "defense in depth." This means implementing security measures at every stage of the container lifecycle: from image creation, through deployment, to runtime. Relying on a single security control is never sufficient.

Our strategy will involve:

1. **Image Hardening:** Reducing the attack surface of the Docker image itself. This is the first line of defense.

2. **Runtime Security:** Configuring containers to run with minimal privileges and resources, isolating them from the host and other containers.
3. **Deployment Best Practices:** Ensuring secure handling of sensitive data and efficient operations in a production environment.

Consider the following high-level flow for a secure container build and deployment:




This flow emphasizes iterative improvements, scanning, and continuous monitoring. Every step provides an opportunity to enhance security.

## Step-by-Step Implementation: Hardening Our Containers

Let's apply these principles to our existing projects, starting with the Dockerfile and moving up to Docker Compose configurations.

### 1. Dockerfile Hardening: Building Lean and Secure Images

The Dockerfile is the blueprint for your container image. A well-crafted Dockerfile is the foundation of a secure container.

#### a. Use Non-Root Users

Running container processes as `root` is a significant security risk. If an attacker compromises your application, they gain root privileges inside the container, which can potentially be escalated to the host. The `USER` instruction in a Dockerfile sets the user and group for subsequent commands and the container's runtime.

**Why it matters:** Principle of least privilege. If a process doesn't need root, it shouldn't have it.

Let's modify a sample `Dockerfile` (e.g., from Project 2: Full-stack Web Application) to use a non-root user.

**File:** `project2/backend/Dockerfile`

```
# Stage 1: Build the application
FROM node:20-alpine AS build-stage

WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .
RUN npm run build

# Stage 2: Create the production image
FROM node:20-alpine AS production-stage

# Create a non-root user
# The 'node' user often exists in official Node.js images.
# If not, you'd create one: RUN addgroup --system appgroup && adduser --system
--ingroup appgroup appuser
# For 'node:20-alpine', the 'node' user already exists with UID 1000 and GID
1000.
# We'll use this existing user.
# ⚡ Quick Note: Always verify the UID/GID for the user you intend to use.
# You can check this by running `id node` inside a temporary container from
the base image.

WORKDIR /app
```

```
COPY --from=build-stage /app/node_modules ./node_modules
COPY --from=build-stage /app/dist ./dist # Assuming your build output is in 'd
ist'
COPY --from=build-stage /app/package*.json ./

# Expose the application port
EXPOSE 3000

# Switch to the non-root user
USER node # Use the 'node' user that comes with the base image

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

**Explanation:**

- We explicitly switch to the `node` user (which is typically a non-root user with UID/GID 1000 in official Node.js images) using `USER node`.
- All subsequent commands and the final `CMD` will run as this user.

### b. Multi-Stage Builds for Smaller Images

Large images increase the attack surface, take longer to pull, and consume more disk space. Multi-stage builds are a powerful technique to create smaller, more secure production images by separating build-time dependencies from runtime dependencies.

**Why it matters:** Reduced image size means fewer potential vulnerabilities, faster deployments, and less storage overhead.

The `Dockerfile` above already demonstrates a multi-stage build:

- `build-stage`: Installs development dependencies (`npm install`), builds the application.
- `production-stage`: Starts from a fresh, minimal base image, copies only the necessary build artifacts and runtime dependencies from the `build-stage`, discarding everything else.

**Verification:** After building your image, compare its size to an image built without multi-stage builds. Run the following command in your `project2/backend` directory:

```
docker build -t myapp-backend:latest .
docker images myapp-backend:latest
```

You should see a significantly smaller image compared to a single-stage build that includes all `node_modules` from the `build-stage`.

### c. Minimizing Attack Surface (Alpine Linux)

Using a minimal base image like Alpine Linux (e.g., `node:20-alpine`) is another way to reduce image size and attack surface. Alpine images are much smaller because they use Musl libc instead of Glibc and contain fewer pre-installed packages.

**Why it matters:** Fewer packages mean fewer potential vulnerabilities that could be exploited.

When adding packages, always use the `--no-cache` flag with `apk` to prevent caching package lists, further reducing image size and avoiding stale data.

**Example:**

```
# ...
FROM alpine:3.18 # Or node:20-alpine

RUN apk add --no-cache curl openssl
# ...
```

## 2. Docker Compose Security: Configuring Runtime Behavior

Docker Compose allows you to define runtime security configurations for your services.

### a. Resource Limits

Uncontrolled containers can consume excessive CPU or memory, impacting other services on the host or even crashing the host itself. Setting resource limits prevents "noisy neighbor" issues and provides resilience.

**Why it matters:** Prevents resource exhaustion attacks and ensures fair resource distribution.

**File:** `project2/docker-compose.yml` (Example for a Node.js backend)

```
version: '3.8'

services:
  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    environment:
      NODE_ENV: production
      DATABASE_URL: postgres://user:password@db:5432/mydb
    # Add resource limits
    deploy:
      resources:
        limits:
```

```
          cpus: '0.5' # Limit to 50% of one CPU core
          memory: 256M # Limit to 256 MB of RAM
        reservations:
          cpus: '0.25' # Reserve 25% of one CPU core
          memory: 128M # Reserve 128 MB of RAM
    depends_on:
      - db

  db:
    image: postgres:16-alpine # Using a specific, minimal version
    environment:
      POSTGRES_DB: mydb
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    volumes:
      - db_data:/var/lib/postgresql/data
    deploy:
      resources:
        limits:
          cpus: '1.0'
          memory: 1G
        reservations:
          cpus: '0.5'
          memory: 512M

volumes:
  db_data:
```

**Explanation:**

- `deploy.resources.limits`: Specifies the maximum CPU and memory a container can consume.
- `deploy.resources.reservations`: Guarantees a minimum amount of CPU and memory for the container.

### b. Read-Only Filesystem

Many applications only need to write data to specific, explicitly mounted volumes. By default, container filesystems are writable. Setting a container's root filesystem to read-only (`read_only: true`) prevents unauthorized writes and makes it harder for attackers to persist changes or install malware.

**Why it matters:** Prevents accidental or malicious modification of the container's root filesystem.

**File:** `project2/docker-compose.yml` (Example)

```
version: '3.8'

services:
  backend:
    # ... other configurations ...
    read_only: true # Set the root filesystem to read-only
    volumes:
```

```
      - /app/logs # If your app needs to write logs, mount a specific volume
for it
    # ...

  db:
    # ... other configurations ...
    read_only: true # Databases need to write to their data directory, so
ensure it's a volume
    volumes:
      - db_data:/var/lib/postgresql/data # This volume will be writable
    # ...
```

**Explanation:**

- `read_only: true` makes the container's root filesystem immutable.
- If your application needs to write data (e.g., logs, uploads), you must explicitly mount a volume for those specific paths, as shown for `/app/logs` or `db_data`.

### c. Environment Variables vs. Docker Secrets

Environment variables are easy to use but are not secure for sensitive data like API keys or database credentials, as they can be easily inspected (`docker inspect`) and might persist in shell history or logs. Docker provides a built-in `secrets` mechanism for Swarm mode, but for standalone Docker Compose, a common pattern is to use `.env` files for non-sensitive configuration and secure external secret management solutions for sensitive data.

**Why it matters:** Prevents exposure of sensitive information.

For Docker Compose, for truly sensitive data, consider external secret management tools like HashiCorp Vault, AWS Secrets Manager, or Azure Key Vault, or use the Swarm secrets feature if deploying to a Swarm cluster. For development, `.env` files are acceptable for local convenience.

**File:** `.env` (Example, not for production secrets)

```
DATABASE_URL=postgres://user:password@db:5432/mydb
API_KEY=my_dev_api_key_123 # For development only
```

**File:** `project2/docker-compose.yml` (Example using `.env`)

```
version: '3.8'

services:
  backend:
    # ...
    env_file:
```

```
      - ./.env # Load environment variables from .env file
    # ...
```

🧠 **Important:** Never commit `.env` files containing sensitive production secrets to version control. Use `.gitignore` to exclude them. For production, environment variables should be injected securely by your deployment system or orchestrator.

### 3. Runtime Security: Auditing with Docker Bench for Security

The Docker Bench for Security is a script that checks for dozens of common best practices around deploying Docker containers in production. It automates many of the checks recommended by the CIS Docker Benchmark.

**Why it matters:** Provides an automated, objective assessment of your Docker daemon, images, and container configurations against industry best practices.

Let's run it on your Docker host.

**Steps:**

1. **Clone the repository:**

```
git clone https://github.com/docker/docker-bench-security.git
cd docker-bench-security
```

1. **Run the bench script:**

```
sudo sh docker-bench-security.sh
```

```
(On Windows with WSL2, you'd run this within your WSL2 distribution.)
```

**Verification:** The script will output a detailed report in your terminal, indicating `INFO`, `WARN`, and `PASS` results for various checks. Pay close attention to `WARN` messages, as these highlight areas where your Docker setup or container configurations can be improved.

**Example Output Snippet:**

```
[INFO] 1  - Host Configuration
[INFO] 1.1 - Linux Host Configuration
[INFO] 1.1.1 - Ensure a separate partition for containers has been created
(Automated)
[WARN] 1.1.1 - /var/lib/docker is not on a separate partition.
[INFO] 1.1.2 - Ensure the container host has been hardened (Manual)
...
```

```
[INFO] 4  - Container Images and Build File
[INFO] 4.1 - Ensure a user for the container has been created (Automated)
[PASS] 4.1  - User for the container has been created
[WARN] 4.2  - Ensure that containers use a HEALTHCHECK to detect and act upon
hung containers (Automated)
```

This output helps you identify specific areas for improvement, such as creating a separate partition for Docker data or adding `HEALTHCHECK` instructions to your Dockerfiles.

---

## Testing & Verification

Throughout the implementation, we've included verification steps. Let's consolidate them:

1. **Non-Root User Verification:**
    - Build your image with the `USER` instruction.
    - Run a container from it:

```
docker run -it --rm myapp-backend:latest whoami
```

```
- Expected output: `node` (or whatever user you specified), confirming the
container process is not running as root.
```

1. **Image Size Verification:**
    - After building with multi-stage builds:

```
docker images myapp-backend:latest
```

```
- Compare the size to previous builds. A multi-stage build should be
significantly smaller.
```

1. **Read-Only Filesystem Verification:**
    - Start your service with `read_only: true` in `docker-compose.yml`.
    - Try to write a file to the container's root filesystem (e.g., `/tmp/test.txt`) from within the running container:

```
        docker compose up -d
        docker exec -it project2-backend-1 touch /tmp/test.txt
```

```
- Expected output: A "Permission denied" error, confirming the filesystem is
read-only.
```

1. **Resource Limits Verification:**
    - While running `docker compose up`, observe resource usage using `docker stats`:

```
        docker stats
```

```
- You should see your container's CPU and memory usage, and it should not
exceed the limits you set.
```

1. **Docker Bench Security Report Review:**
    - Carefully read the output of `sudo sh docker-bench-security.sh`.
    - Prioritize `WARN` messages and address them systematically. This tool is a continuous improvement resource.

---

## Production Considerations

Beyond the immediate configurations, several broader production concerns are vital for secure and maintainable Docker deployments.

### 1. Secure Container Registries

When pushing your images to a registry (like Docker Hub, Azure Container Registry, AWS ECR, or Google Container Registry), ensure you:

- **Use private repositories:** Never push proprietary images to public registries.
- **Implement strong access control:** Use specific credentials (e.g., service principles, IAM roles) with least privilege for pushing and pulling images.
- **Enable image scanning:** Many registries offer built-in vulnerability scanning (e.g., Azure Container Registry scan, AWS ECR scan). Use these to automatically detect known vulnerabilities in your images before deployment.

## 2. Logging and Monitoring

Containerized applications can be ephemeral, making traditional logging challenging.

- **Centralized logging:** Configure your containers to send logs to a centralized logging system (e.g., ELK Stack, Splunk, Datadog) rather than writing to the local filesystem. Docker's logging drivers (e.g., `json-file`, `syslog`, `gelf`) can facilitate this.
- **Health checks:** Implement `HEALTHCHECK` instructions in your Dockerfiles to allow orchestrators to automatically detect and restart unhealthy containers.

## 3. Secrets Management at Scale

For production, relying solely on `.env` files is insufficient.

- **Orchestrator-native secrets:** If using Kubernetes, leverage Kubernetes Secrets. For Docker Swarm, use Docker Secrets.
- **Dedicated secret managers:** For more complex scenarios or multi-cloud environments, integrate with tools like HashiCorp Vault, AWS Secrets Manager, Azure Key Vault, or Google Secret Manager. These tools provide secure storage, versioning, and access control for secrets.

## 4. Regular Updates and Patching

Container security is not a one-time setup.

- **Base image updates:** Regularly update your base images to benefit from security patches and bug fixes. Automate this process where possible.
- **Application dependencies:** Keep your application's libraries and frameworks up-to-date to mitigate known vulnerabilities.
- **Docker Engine updates:** Keep your Docker Engine (currently 29.1.x) and Docker Compose (currently 2.40.3) updated to the latest stable versions to benefit from security fixes and new features.

## Common Issues & Solutions

Here are some common pitfalls in container security and how to address them:

1. **Running Containers as Root:**
    - **Issue:** Processes inside the container run with root privileges, increasing the blast radius if compromised.
    - **Solution:** Always define a non-root `USER` in your Dockerfile. If your application needs to bind to privileged ports (e.g., 80, 443), consider using an Nginx or Caddy reverse proxy running on port 80/443 and forwarding traffic to your application running on a non-privileged port (e.g., 3000, 8080) as a non-root user.
2. **Large, Bloated Images:**
    - **Issue:** Images contain unnecessary tools, libraries, or build artifacts, increasing size and potential vulnerabilities.
    - **Solution:** Implement multi-stage builds. Use minimal base images (e.g., Alpine variants). Clean up temporary files (`rm -rf /var/cache/apk/*`) and build caches during image creation.
3. **Exposing Sensitive Ports Directly:**
    - **Issue:** Directly exposing database ports or internal API ports to the host network or public internet.
    - **Solution:** Only expose ports that need to be accessible from outside the Docker network. Use Docker's internal networking for inter-container communication. Use a reverse proxy (like Nginx) to handle external traffic and forward it to your application's internal port. Limit network access with firewall rules on the host.
4. **Improper Volume Mapping:**
    - **Issue:** Mounting host directories with excessive permissions, or not using named volumes for persistent data.
    - **Solution:** Use named volumes for persistent data for databases and other stateful services. For host mounts, ensure the permissions are restricted to what the container absolutely needs. Avoid mounting sensitive host directories into containers unless absolutely necessary and carefully restricted.

## Summary & Next Steps

This chapter equipped you with essential strategies for hardening your Docker images and containers, moving your projects closer to production readiness. We covered:

- **Dockerfile best practices:** Multi-stage builds, non-root users, and minimal base images.
- **Docker Compose configurations:** Resource limits and read-only filesystems.
- **Runtime security:** Using `docker-bench-security` to audit your setup.
- **Production considerations:** Secure registries, logging, and secrets management.

Security is an ongoing process, not a destination. Regularly review your Dockerfiles, update your base images, and keep abreast of new security practices and tools. The principles learned here are fundamental to building robust, production-grade containerized applications.

In the next chapter, we'll explore advanced Docker concepts, diving deeper into networking, advanced volume management, and image optimization techniques to further refine your Docker expertise.

---

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

## References


- Docker Engine documentation (version 29.1.x)
- Docker Compose documentation (version 2.40.3)
- GitHub - docker/docker-bench-security
- Best practices for writing Dockerfiles
- Run a container as a non-root user
- Configure resource constraints for containers

CHAPTER 07

# Automating Builds: Integrating Docker into a CI Pipeline

Building a Docker image manually is a common starting point, but in any professional development environment, it's a critical bottleneck. Manual builds introduce inconsistencies, slow down delivery, and are prone to human error. This chapter guides you through automating the core of Continuous Integration (CI) for Dockerized applications: consistently building your images and pushing them to a secure container registry.

By the end of this chapter, you will have implemented a robust CI pipeline using GitHub Actions. This pipeline will automatically build your Dockerized application, tag the image appropriately, and securely publish it to an Azure Container Registry (ACR) whenever code is pushed. This level of automation is foundational for reliable software delivery, ensuring that every deployable artifact is derived from your latest, validated codebase.

---

## Project Overview: Automating Docker CI

This project focuses on establishing a Continuous Integration (CI) pipeline for a simple Dockerized Node.js application. The goal is to automate the entire process from code commit to a securely stored, versioned Docker image in a private container registry. This automation ensures consistency, reduces manual errors, and speeds up the delivery of new features or bug fixes.

Upon completion, you will have a working GitHub Actions workflow that:

- Triggers automatically on code pushes to the `main` branch.
- Checks out the latest code from your GitHub repository.
- Securely authenticates with Azure Container Registry (ACR).
- Builds a Docker image using a multi-stage `Dockerfile`.
- Tags the image with both a `latest` tag and a unique Git commit SHA.
- Pushes the tagged images to your private ACR instance.

## Tech Stack Choices

To build this automated CI pipeline, we will leverage the following core technologies:

- **Docker Engine (via GitHub Actions Runner):** The underlying technology for building and managing containers. GitHub Actions runners come pre-configured with a compatible Docker Engine. As of 2026-08-06, Docker Engine typically runs version 29.1.x or later on these runners.
- **Docker Compose:** While not directly used within the CI build process itself in this chapter, Docker Compose (version 2.40.3 as of 2026-08-06) is a key tool for local multi-service development and is assumed for testing the application locally before CI.
- **GitHub Actions:** The CI/CD platform native to GitHub. We use it to orchestrate the build, test, and push steps. Its declarative YAML syntax and extensive marketplace of actions simplify pipeline creation.
- **Azure Container Registry (ACR):** A managed, private Docker image registry provided by Microsoft Azure. It offers robust security features, scalability, and integration with other Azure services, making it an excellent choice for production image storage.
- **Node.js & Express:** Our sample application stack. The principles learned here are transferable to any language or framework.

## Build Plan: Automating Image Builds

This chapter will guide you through the following incremental milestones:

1. **Prepare the Application & Dockerfile:** Set up a basic Node.js Express application and create an efficient multi-stage `Dockerfile`.
2. **Provision Azure Container Registry (ACR):** Create a dedicated, private registry instance in Azure to store your Docker images.
3. **Secure Azure Authentication:** Create an Azure Service Principal with minimal necessary permissions for GitHub Actions to interact with ACR.
4. **Configure GitHub Secrets:** Store sensitive ACR credentials securely within your GitHub repository.
5. **Define GitHub Actions Workflow:** Create the YAML file that specifies the steps for building and pushing your Docker image.

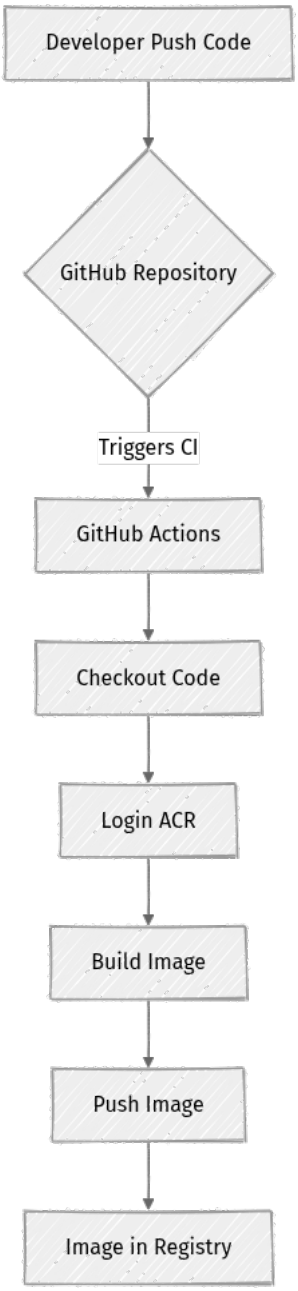
6. **Execute and Verify:** Trigger the CI pipeline and confirm that images are correctly built and pushed to ACR.

---

## Architecting the CI Pipeline

Continuous Integration is a development practice where developers frequently integrate their code changes into a central repository. Each integration then triggers an automated build and test process. For containerized applications, this workflow typically involves several key steps that are orchestrated by the CI system.

Here's a high-level architectural overview of the CI pipeline we will build:



**Why this specific architecture?**

- **GitHub Actions:** It's natively integrated with GitHub repositories, simplifying workflow setup and management. GitHub Actions offers a vast marketplace of pre-built actions, accelerating pipeline development.
- **Azure Container Registry (ACR):** A fully managed, secure, and scalable Docker image registry provided by Microsoft Azure. ACR is a best practice for production environments, offering robust access control, image scanning, and integration with other Azure services. Using a private registry significantly enhances security and control compared to relying solely on public alternatives.
- **Secure Credential Management:** Sensitive information, such as cloud service credentials, will be handled securely using GitHub Secrets and Azure Service Principals. This prevents credentials from being exposed directly in your repository or workflow files, adhering to the principle of least privilege.

---

## Step-by-Step Implementation

For this practical example, we'll use a simple Node.js web application. We assume you've already created a GitHub repository for your project.

### 1. Project Setup and Dockerfile

First, ensure your project structure and `Dockerfile` are ready. Create a new directory, e.g., `docker-ci-example`, and set up the following files. This structure assumes your application code is within an `app/` subdirectory.

**`app/index.js`**

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Hello from Dockerized CI! Version 1.0');
});

app.listen(port, () => {
  console.log(`App listening at http://localhost:${port}`);
});
```

This is a basic Express.js application that serves a "Hello World" message.

**`app/package.json`**

```
{
  "name": "docker-ci-example",
  "version": "1.0.0",
  "description": "A simple Node.js app for Docker CI demo",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "dependencies": {
    "express": "^4.19.2"
  }
}
```

This defines the Node.js project and its `express` dependency.

**`Dockerfile` (in the root of your repository)**

```
# Stage 1: Build the application
# Use a lightweight Node.js base image for building
FROM node:20-alpine AS builder

WORKDIR /app

# Copy package.json and package-lock.json to install dependencies
COPY app/package*.json ./
RUN npm install

# Copy the rest of the application source code
COPY app/ .

# Stage 2: Create the final production image
# Use a fresh, minimal Node.js runtime image for production
FROM node:20-alpine

WORKDIR /app

# Copy only the necessary files from the builder stage
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/index.js .
COPY --from=builder /app/package.json .

# Expose the port the application listens on
EXPOSE 3000
# Define the command to run the application
CMD ["npm", "start"]

# 🔥 Optimization / Pro tip: Run as a non-root user
# For production deployments, always create and use a dedicated non-root user
# to mitigate potential security risks and adhere to the principle of least
privilege.
# Example for a Node.js app:
# RUN addgroup --system appgroup && adduser --system --ingroup appgroup
```

```
appuser
# USER appuser
```

- **Decision:** We employ a multi-stage `Dockerfile`. This is a crucial optimization technique. The first stage (`builder`) handles dependency installation, creating a temporary environment. The second stage then copies only the essential build artifacts (compiled code, `node_modules`, etc.) into a clean, minimal base image.
    - **Tradeoff:** This slightly increases the `Dockerfile`'s complexity but drastically reduces the final image size and attack surface, improving security and deployment speed.
- `node:20-alpine`: This specifies Node.js version 20 on an Alpine Linux base. Alpine images are significantly smaller than their Debian-based counterparts, leading to faster downloads and smaller storage footprints.
- **Non-root User:** The commented-out section highlights a critical security best practice. Running containers as a non-root user limits the potential damage if an attacker compromises the container.

## 2. Set up Azure Container Registry (ACR)

You will need an active Azure subscription for this step.

1. **Create an ACR instance:**
    - Navigate to the Azure portal (portal.azure.com).
    - Search for "Container Registries" and select "Create".
    - Fill in the required details:
        - **Registry name:** Choose a globally unique name (e.g., `mydockerciacr2026`).
        - **Resource group:** Select an existing one or create a new one.
        - **Location:** Choose a region close to your deployment target.
        - **SKU:** "Basic" is sufficient for this tutorial. For production, consider "Standard" or "Premium" for features like geo-replication and content trust.
    - Click "Review + create", then "Create".
    - 📌 **Key Idea:** A dedicated, private registry like ACR is essential for managing your images securely and reliably in production.

2. **Create a Service Principal (Recommended for Production):** For production scenarios, using an Azure Service Principal with specific permissions is the most secure method for authenticating CI/CD pipelines.
    - Open Azure Cloud Shell or your local Azure CLI (ensure you're logged in with `az login`).
    - Retrieve your ACR's resource ID:

```
        ACR_NAME="mydockerciacr2026" # Replace with your ACR name
        ACR_ID=$(az acr show --name $ACR_NAME --query id --output tsv)
        echo $ACR_ID
```

```
- Create a Service Principal and grant it the `acrPush` role to your ACR. This role allows it to push images but not delete or modify the registry configuration.
```

```
        SP_NAME="github-actions-sp-2026"
        SP_CREDENTIALS=$(az ad sp create-for-rbac --name $SP_NAME --scopes $ACR_ID --role acrPush --query "{clientId: appId, clientSecret: password, subscriptionId: subscription, tenantId: tenant}" --output json)
        echo $SP_CREDENTIALS
```

```
- **🧠 Important:** Save the entire JSON output from `SP_CREDENTIALS`. This JSON string contains your `clientId` (which acts as a username) and `clientSecret` (password), along with `subscriptionId` and `tenantId`. You will use this entire JSON string as a secret in GitHub.
- **⚡ Quick Note:** While ACR also offers an "Admin user" with username/password, it grants broad permissions and is less secure for automated pipelines. Service Principals offer granular control and are preferred.
```

### 3. Configure GitHub Secrets

To securely provide your ACR credentials to GitHub Actions, we'll store them as encrypted secrets in your GitHub repository.

1. Navigate to your GitHub repository in your web browser.
2. Go to "Settings" > "Secrets and variables" > "Actions".
3. Click "New repository secret".

4. Create the following secrets:
    - `ACR_LOGIN_SERVER`: The login server for your ACR instance (e.g., `mydockerciacr2026.azurecr.io`). You can find this on the "Overview" page of your ACR in the Azure portal.
    - `AZURE_CREDENTIALS`: Paste the complete JSON string you obtained when creating the Service Principal (e.g., `{ "clientId": "...", "clientSecret": "...", "subscriptionId": "...", "tenantId": "..." }`).

    ⚠️ **What can go wrong:** Ensure the JSON string for `AZURE_CREDENTIALS` is valid and copied completely without extra spaces or characters. Invalid JSON will cause authentication failures.

## 4. Create GitHub Actions Workflow

Now, let's define the CI workflow in your repository. This YAML file will instruct GitHub Actions on how to build and push your Docker image.

1. In the root of your repository, create a directory named `.github/workflows`.
2. Inside this directory, create a new file named `docker-ci.yml`.

**`.github/workflows/docker-ci.yml`**

```
name: Docker CI for Web App

on:
  push:
    branches:
      - main # Trigger on pushes to the main branch
  pull_request:
    branches:
      - main # Trigger on pull requests targeting the main branch
  workflow_dispatch: # Allows manual triggering of the workflow from GitHub UI

env:
  REGISTRY: ${{ secrets.ACR_LOGIN_SERVER }} # Azure Container Registry login
server
  IMAGE_NAME: docker-ci-example             # Name for your Docker image
  IMAGE_PATH: .                             # Context path for Dockerfile
(repository root)

jobs:
  build-and-push-image:
    runs-on: ubuntu-latest # Use the latest Ubuntu runner provided by GitHub
Actions
    permissions:
      contents: read # Allow reading repository content
      id-token: write # Required for secure OIDC login to Azure

    steps:
```

```
    - name: Checkout repository code
      uses: actions/checkout@v4.1.7 # As of 2026-08-06, checks out the
repository

    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v3.3.0 # As of 2026-08-06, enables
advanced Docker build features

    - name: Log in to Azure Container Registry
      uses: azure/login@v1.6.1 # As of 2026-08-06, securely logs in to Azure
      with:
        creds: ${{ secrets.AZURE_CREDENTIALS }} # Use the Service Principal
credentials from GitHub Secrets
        enable-AzPSSession: false # Not strictly needed for ACR login, set to
false for minimal permissions

    - name: Build and push Docker image
      uses: docker/build-push-action@v5.3.0 # As of 2026-08-06, builds and
pushes the Docker image
      with:
        context: ${{ env.IMAGE_PATH }} # Set the build context to the
repository root
        push: true                     # Enable pushing the image to the
registry
        tags: |
          ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}:${{ github.sha }} # Tag
with full Git commit SHA for immutability

${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}:latest            # Tag as 'latest'
for convenience
        cache-from: type=gha            # Enable caching from GitHub Actions
cache
        cache-to: type=gha,mode=max     # Store build cache to GitHub Actions
cache (max mode for aggressive caching)
```

- **`name`**: A human-readable name for your workflow, visible in the GitHub Actions UI.
- **`on`**: Defines the events that trigger this workflow. Here, it runs on `push` or `pull_request` to the `main` branch, or it can be triggered manually via `workflow_dispatch`.
- **`env`**: Sets environment variables available to all jobs in the workflow.
  - `REGISTRY`: Dynamically retrieves the ACR login server from GitHub Secrets.
  - `IMAGE_NAME`: The chosen name for your Docker image within the registry.
  - `IMAGE_PATH`: **Crucially, set to `.` (the repository root).** This ensures that the Docker build context includes both the `Dockerfile` and the `app/` directory, allowing `COPY app/` commands in the `Dockerfile` to function correctly.

- `jobs`: Workflows are composed of one or more jobs.
  - `build-and-push-image`: Our single job for this pipeline.
    - `runs-on: ubuntu-latest`: Specifies the virtual machine environment where the job will execute.
    - `permissions`: This block is vital for security. `id-token: write` enables OpenID Connect (OIDC) authentication with Azure, a more secure, token-based approach that avoids long-lived secrets when possible.
    - `actions/checkout@v4.1.7`: This action clones your repository's code onto the runner.
    - `docker/setup-buildx-action@v3.3.0`: Initializes Docker Buildx, which provides enhanced build capabilities, including multi-platform builds and improved layer caching.
    - `azure/login@v1.6.1`: Logs into Azure using the `AZURE_CREDENTIALS` secret (your Service Principal JSON). This step authenticates the runner to interact with Azure services, including ACR.
    - `docker/build-push-action@v5.3.0`: This powerful action handles both building your Docker image and pushing it to the specified registry.
      - `context: ${{ env.IMAGE_PATH }}`: Specifies the root directory for the Docker build.
      - `push: true`: Instructs the action to push the built image.
      - `tags`: Defines the tags for the image. We're applying two tags: one with the unique `github.sha` (the full Git commit hash) for traceability, and a `latest` tag for convenience during development.
      - `cache-from`, `cache-to`: These leverage GitHub Actions' built-in cache. Docker layers are cached between workflow runs, significantly accelerating subsequent builds by reusing unchanged layers.

**Important Version Notes (checked 2026-08-06):**

- `actions/checkout`: Latest stable is `v4.1.7`.
- `docker/setup-buildx-action`: Latest stable is `v3.3.0`.
- `azure/login`: Latest stable is `v1.6.1`.

- `docker/build-push-action`: Latest stable is `v5.3.0`. Always consult the GitHub Marketplace or official documentation for the absolute latest stable versions of these actions, as they are frequently updated.

---

## Testing & Verification

Once your workflow file is committed, it's time to see the automation in action.

1. **Commit and Push:** Save all your changes (`app/`, `Dockerfile`, and `.github/workflows/docker-ci.yml`) and push them to your `main` branch on GitHub.

```
git add .
git commit -m "feat: Implement Docker CI pipeline with GitHub Actions"
git push origin main
```

1. **Monitor GitHub Actions:**
    - Go to your GitHub repository in your browser.
    - Click on the "Actions" tab.
    - You should see a new workflow run initiated by your push (it will have the commit message as its title). Click on this run.
    - Observe the progress of each step: `Checkout repository code`, `Set up Docker Buildx`, `Log in to Azure Container Registry`, and `Build and push Docker image`.
    - Each step should eventually show a green checkmark, indicating success. If any step fails, click on it to expand the logs and review the error messages for debugging clues.

2. **Verify Image in ACR:**
    - Once the GitHub Actions workflow successfully completes, navigate back to your Azure Container Registry in the Azure portal.
    - In your ACR instance, go to "Repositories" under the "Services" section.
    - You should now see a new repository named `docker-ci-example`. Click on it.
    - Inside, you will find your newly pushed images, tagged with both the `latest` tag and the full Git commit SHA (e.g., `abcdef123`).

    Alternatively, you can use the Azure CLI to verify:

```
    # Ensure you are logged in to Azure CLI
    az login

    # List repositories in your ACR
    az acr repository list --name <your-acr-name> --output tsv

    # Show tags for your specific image
    az acr repository show-tags --name <your-acr-name> --repository docker-ci-
example --output tsv
```

```
You should see output similar to `latest` and a long string representing the
Git commit SHA.
```

## Production Considerations

Implementing CI for Dockerized applications introduces several critical considerations for production environments:

- **Secure Credential Handling:** The use of GitHub Secrets and Azure Service Principals with specific roles (like `acrPush`) is paramount. Never embed credentials directly in your workflow files. The `id-token: write` permission for OIDC-based authentication is the modern, secure approach to avoid static secrets where possible.

- **Image Versioning Strategy:** While `latest` is convenient for development, in production, always rely on immutable tags like the Git commit SHA (`github.sha`) or semantic versioning (e.g., `v1.0.0`, `v1.0.1`). This ensures that deployments are always tied to a specific, reproducible codebase.
    - ⚡ **Real-world insight:** Many teams use `latest` for development/staging and then promote specific SHA-tagged images or manually apply semantic version tags for production releases.
- **Automated Image Scanning:** Azure Container Registry offers integrated vulnerability scanning through Microsoft Defender for Cloud. Enable this feature to automatically scan all new images for known security vulnerabilities upon push. This is a critical line of defense before deployment.
- **Multi-Platform Builds:** Docker Buildx, enabled by `docker/setup-buildx-action`, allows you to build images for multiple architectures (e.g., `linux/amd64`, `linux/arm64`) within a single workflow. This is crucial for supporting diverse deployment targets, from cloud VMs to edge devices.
- **Efficient Build Caching:** The `cache-from` and `cache-to` parameters in `docker/build-push-action` are vital for reducing CI build times and costs. By caching Docker layers, only changed layers need to be rebuilt, leading to significant performance improvements for frequent pushes.
- **CI Runner Resource Management:** Be mindful of the resource limits provided by your CI service (e.g., GitHub-hosted runners). For very large or complex Docker builds, you might need to consider self-hosted runners with more powerful hardware to avoid timeouts or performance bottlenecks.
- **Container Health Checks:** While not directly part of the build pipeline, ensure your application's `Dockerfile` or orchestration configuration includes health checks (e.g., `HEALTHCHECK` instruction in Dockerfile, readiness/liveness probes in Kubernetes). This ensures that deployed containers are truly ready to serve traffic.

## Common Issues and Troubleshooting

Even with a well-designed pipeline, issues can arise. Here are some common problems and their debugging strategies:

1. **`azure/login` failure: "Permissions for the credential could not be verified."**
    - **Issue:** The `AZURE_CREDENTIALS` secret is incorrect, incomplete, or the associated Service Principal lacks the necessary permissions (e.g., `acrPush` role on the ACR).
    - **Solution:** Carefully re-verify the `AZURE_CREDENTIALS` JSON string in your GitHub Secret for any typos or missing fields (`clientId`, `clientSecret`, `tenantId`, `subscriptionId`). Ensure the Service Principal has been granted the `acrPush` role on your specific ACR instance. If using OIDC (`id-token: write`), confirm the `permissions` block is correctly configured in your workflow.
2. **`docker/build-push-action` failure: "denied: requested access to the resource is denied"**
    - **Issue:** The authenticated identity (Service Principal) does not have permission to push images to the specified Azure Container Registry.
    - **Solution:** Confirm that the Service Principal used in `AZURE_CREDENTIALS` has the `acrPush` role assigned to your ACR. Also, double-check that the `REGISTRY` environment variable in your workflow (`${{ secrets.ACR_LOGIN_SERVER }}`) correctly points to your ACR's login server.
3. **Image build failure: "Cannot find module 'express'" or similar application errors.**
    - **Issue:** The `Dockerfile` or the application code itself has an error, or dependencies were not correctly installed or copied during the build process.
    - **Solution:** Review the build logs in GitHub Actions carefully. Check your `Dockerfile` for correct paths, `COPY` commands, and ensure `npm install` (or equivalent for your language) runs successfully. Verify that the `context` parameter in `docker/build-push-action` (`${{ env.IMAGE_PATH }}`) is correctly set to the directory containing your `Dockerfile` and source code.

4. **Incorrect Image Tagging or Image Not Appearing in ACR.**
    - **Issue:** The image is built but doesn't appear in ACR with the expected tags, or it's missing entirely.
    - **Solution:** Inspect the `tags` parameter within your `docker/build-push-action` in the workflow file. Verify that the environment variables (`${{ env.REGISTRY }}`, `${{ env.IMAGE_NAME }}`, `${{ github.sha }}`) are resolving to the correct values by checking the workflow run logs. Ensure `push: true` is set.

---

## Summary & Next Steps

You've successfully implemented a Continuous Integration pipeline for your Dockerized application using GitHub Actions and Azure Container Registry. This is a foundational achievement in modern software development, transitioning from manual, error-prone builds to an automated, consistent process.

With this setup, you can now:

- Automatically build Docker images for your application on every code commit or pull request.
- Securely store these versioned images in a private, managed container registry.
- Ensure a high level of consistency and reliability in your build artifacts.

This automated build and registry push lays the groundwork for Continuous Delivery (CD). The natural progression from here is to automate the deployment of these newly built and pushed images to your staging, testing, or production environments. This next phase typically involves integrating with orchestration platforms like Kubernetes, Azure Container Apps, or other cloud-native deployment services that consume images from your ACR. This is where your journey into full CI/CD truly begins, transforming your development process into a streamlined, efficient, and robust system.

## References


- GitHub Actions Documentation
- Azure Container Registry Documentation
- Best Practices for Using Azure Container Registry - Azure Container Registry | Microsoft Learn
- docker/build-push-action GitHub Marketplace
- azure/login GitHub Marketplace


This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 08

# Monitoring, Logging, and Operational Readiness for Containerized Apps

## Introduction: Beyond Deployment to Operational Excellence

Deploying a containerized application is a significant achievement, but the true test of engineering lies in its operational readiness. This chapter shifts our focus from "making it run" to "keeping it running reliably." We'll dive into the critical aspects of observability—logging and metrics—and explore essential operational practices that ensure your Dockerized applications are robust, maintainable, and resilient in production environments.

By the end of this chapter, you will have built a basic, yet functional, observability stack for your sample application. You'll understand how to instrument your code for effective monitoring, aggregate logs from distributed containers, and visualize key performance indicators (KPIs) and health metrics using industry-standard open-source tools. This foundation will empower you to diagnose issues quickly, prevent outages, and ensure a smooth operational experience.

## Why Observability is Non-Negotiable

In modern, distributed systems, especially those leveraging microservices and containers, understanding the internal state of your applications becomes incredibly complex. Services interact asynchronously, failures can cascade, and traditional debugging methods often fall short. Observability is the capability to understand what's happening inside a system by examining the data it produces. It's typically categorized into three pillars:

1. **Logs:** Discrete, timestamped records of events. These are crucial for understanding specific actions, errors, or state changes within an application's lifecycle. Think of them as the detailed forensic evidence of your system.

2. **Metrics:** Numerical measurements collected over time, representing the health, performance, or resource utilization of a service. Metrics provide aggregated views, trends, and enable proactive monitoring and alerting. Examples include CPU usage, request latency, error rates, and active connections.
3. **Traces:** End-to-end representations of requests as they flow through multiple services in a distributed system. Tracing helps pinpoint bottlenecks and latency issues across service boundaries.

For our projects, we will build a strong foundation in structured logging and metrics. Without these, operating a system at scale is like navigating a ship in dense fog—you're reacting to collisions rather than proactively steering clear.

---

## Project Setup: A Centralized Observability Stack

Before we dive into implementation, let's establish a clear project structure for our observability stack. We'll create a root directory, `docker-observability-project`, and place our `docker-compose.yaml` file there. Each component (the application, Fluentd, Prometheus, Grafana) will reside in its own subdirectory.

```
docker-observability-project/
├── docker-compose.yaml
├── my-log-app/
│   ├── app.py
│   ├── Dockerfile
│   └── requirements.txt
├── fluentd/
│   ├── fluent.conf
│   └── Dockerfile
├── prometheus/
│   └── prometheus.yml
└── grafana/
    # (No custom files needed in 'grafana' for this setup, but the directory
is good practice)
```

**Action:** Create the `docker-observability-project` directory and the subdirectories listed above.

```
mkdir docker-observability-project
cd docker-observability-project
mkdir my-log-app fluentd prometheus grafana
touch docker-compose.yaml
```

Now, let's start by creating our example application within `my-log-app/`.

# Structured Logging: Making Logs Machine-Readable

Traditional plain-text logs are difficult to parse and analyze at scale. Structured logging, typically in JSON format, embeds rich metadata with each log entry, making it machine-readable, queryable, and highly valuable for analytics.

## Emitting Structured Logs from Applications

The best practice for containerized applications is to write logs to `stdout` (standard output) and `stderr` (standard error). Docker's logging drivers can then capture these streams, decoupling your application from the underlying logging infrastructure.

Let's create a simple Python application that emits structured JSON logs.

**1. Create a Python application:** Inside `my-log-app/`, create `app.py`:

```
# docker-observability-project/my-log-app/app.py
import logging
import json
import time
import os
import random

# Configure basic logging to stdout. We'll let the application format JSON.
logging.basicConfig(level=logging.INFO, format='%(message)s')
logger = logging.getLogger(__name__)

def log_event(event_type, message, **kwargs):
    """
    Constructs and logs a structured JSON event to stdout.
    """
    log_entry = {
        "timestamp": time.time(),
        "level": "INFO",
        "service": os.getenv("SERVICE_NAME", "unknown-service"),
        "event_type": event_type,
        "message": message,
        **kwargs # Add any additional keyword arguments as context
    }
    logger.info(json.dumps(log_entry))

if __name__ == "__main__":
    count = 0
    while True:
        count += 1
        request_id = f"req-{count:03d}"
        user_id = f"user-{random.randint(100, 999)}"

        log_event(
            "request_processed",
            f"Processing request {request_id}",
            request_id=request_id,
            user_id=user_id,
```

```
            duration_ms=round(random.uniform(10, 200), 2)
        )
        if count % 5 == 0:
            log_event(
                "warning",
                f"High load detected for request {request_id}",
                request_id=request_id,
                threshold=0.8,
                current_load=random.uniform(0.7, 0.95)
            )
        time.sleep(1)
```

This Python script uses the `logging` module to print JSON strings to `stdout`. Each log entry includes core fields like `timestamp`, `level`, `service`, `event_type`, and a `message`, along with dynamic, context-specific fields like `request_id` and `duration_ms`.

**2. Create a Dockerfile for the application:** Inside `my-log-app/`, create `Dockerfile`:

```
# docker-observability-project/my-log-app/Dockerfile
# Use a multi-stage build for smaller, more secure images
FROM python:3.11-slim-bookworm AS builder

# Install build dependencies
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Final stage for the runtime image
FROM python:3.11-slim-bookworm

WORKDIR /app
# Copy only the installed packages from the builder stage
COPY --from=builder /usr/local/lib/python3.11/site-packages /usr/local/lib/
python3.11/site-packages
COPY app.py .

# Set environment variable for service identification in logs
ENV SERVICE_NAME="my-log-service"

# Command to run the application
CMD ["python", "app.py"]
```

This `Dockerfile` sets up a Python environment and runs `app.py`. The multi-stage build ensures our final image is lean, containing only necessary runtime components and not build tools.

**3. Create `requirements.txt`:** Inside `my-log-app/`, create `requirements.txt`. For this basic logging app, it's empty, but we include it for good practice and future expansion.

```
# docker-observability-project/my-log-app/requirements.txt
# No external dependencies for basic logging, but good practice to include.
```

### Running the Application with Docker Compose

Now, let's define our service in `docker-compose.yaml` to run our `my-log-app`.

**1. Create `docker-compose.yaml`:** At the project root (`docker-observability-project/`), create `docker-compose.yaml`:

```
# docker-observability-project/docker-compose.yaml
version: '3.8'

services:
  log-app:
    build:
      context: ./my-log-app # Build from the my-log-app directory
      dockerfile: Dockerfile
    container_name: log-app-instance
    environment:
      SERVICE_NAME: "my-log-service"
    logging:
      driver: "json-file" # This is the default, explicitly shown for clarity
      options:
        max-size: "10m" # Rotate log files at 10MB
        max-file: "3"   # Keep a maximum of 3 log files
```

Here, we explicitly specify the `json-file` logging driver and its options for log rotation. This prevents log files from consuming all disk space on the host.

**2. Build and run the container:**

```
cd docker-observability-project
docker compose up -d --build
```

**Verification:** To view the logs:

```
docker logs log-app-instance
```

You should see JSON log lines streaming to your terminal, confirming the application is running and emitting structured logs. Press `Ctrl+C` to stop `docker logs`.

```
{"timestamp": 1678886400.123, "level": "INFO", "service": "my-log-service", "e
vent_type": "request_processed", "message": "Processing request req-001", "req
uest_id": "req-001", "user_id": "user-123", "duration_ms": 55.23}
```

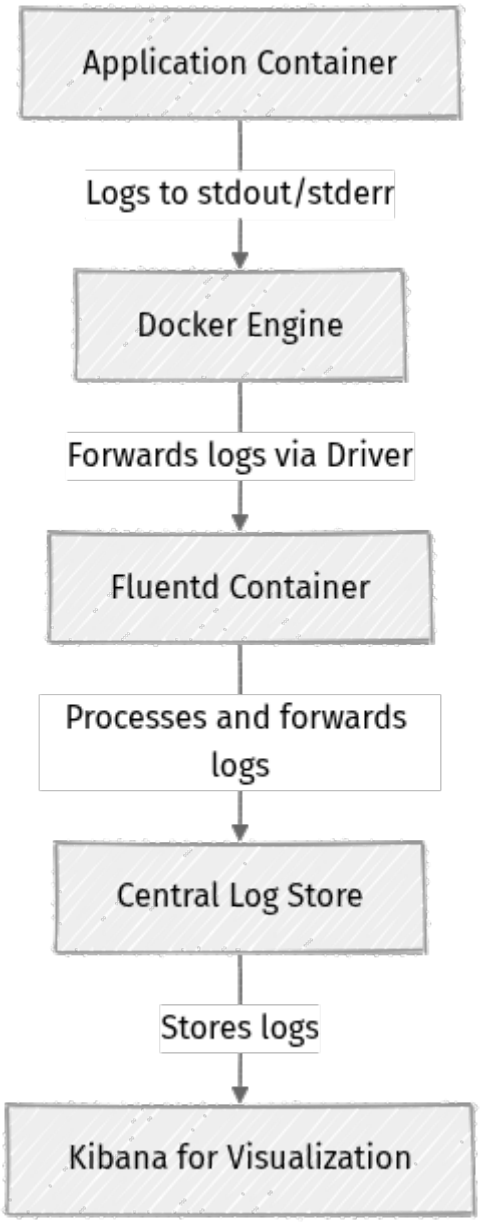
This output is clean and ready for ingestion by a log aggregator.

---

## Centralized Logging with Fluentd

While `docker logs` is useful for immediate inspection, production systems require a centralized log aggregation system. This system collects logs from all containers, stores them efficiently, and makes them searchable and analyzable. Fluentd is a popular open-source data collector for unified logging.

### Logging Architecture Overview



### Setting up Fluentd with Docker Compose

We'll use Fluentd as a log shipper. For this example, Fluentd will collect logs from our `log-app` service and print them to its own `stdout`, simulating forwarding to a centralized system like Elasticsearch.

**1. Create a `fluentd` configuration:** Inside `fluentd/`, create `fluent.conf`:

```
# docker-observability-project/fluentd/fluent.conf
<source>
  @type forward
  port 24224
  bind 0.0.0.0
</source>

# This match rule captures all incoming logs (**)
<match **>
  @type stdout
  # For demonstration, we're outputting to stdout.
  # In a production setup, this would typically be an output plugin like:
  # @type elasticsearch
  # @type s3
  # @type kafka
  <format>
    @type json # Ensure output is JSON for easier parsing
  </format>
</match>
```

This `fluent.conf` configures Fluentd to listen for forwarded logs on port 24224 and then output all received logs to its own `stdout` in JSON format.

**2. Create a Dockerfile for Fluentd:** Inside `fluentd/`, create `Dockerfile`:

```
# docker-observability-project/fluentd/Dockerfile
FROM fluent/fluentd:v1.16-debian-1.0 # Using a stable Fluentd image as of
2026-08-06

# Install necessary plugins if needed.
# For this basic example, no extra plugins are required.
# Example: RUN gem install fluent-plugin-elasticsearch
```

We're using an official Fluentd image from Docker Hub.

**3. Update `docker-compose.yaml` to include Fluentd:** Modify your `docker-compose.yaml` (at the project root) to add the `fluentd` service and configure `log-app` to use the `fluentd` logging driver.

```
# docker-observability-project/docker-compose.yaml
version: '3.8'

services:
  log-app:
    build:
```

```
      context: ./my-log-app
      dockerfile: Dockerfile
    container_name: log-app-instance
    environment:
      SERVICE_NAME: "my-log-service"
    logging:
      driver: "fluentd" # Use the fluentd logging driver
      options:
        fluentd-address: fluentd:24224 # Point to the fluentd service within
the Docker network
        tag: "docker.log-app" # A tag for Fluentd to identify logs from this
service

  fluentd:
    build:
      context: ./fluentd # Build Fluentd from the 'fluentd' directory
      dockerfile: Dockerfile
    container_name: fluentd-aggregator
    volumes:
      - ./fluentd/fluent.conf:/fluentd/etc/fluent.conf:ro # Mount Fluentd
config as read-only
    ports:
      - "24224:24224" # Expose Fluentd's forward port to the host for
potential external shippers
    command: ["fluentd", "-c", "/fluentd/etc/fluent.conf"]
    # Fluentd needs to be accessible by log-app, Docker Compose handles this
via service names
```

Notice the `fluentd-address: fluentd:24224`. Within a Docker Compose network, services can refer to each other by their service names (e.g., `fluentd`). This is a key feature of Docker's internal DNS.

**4. Stop existing containers and run the full stack:**

```
docker compose down # Stop and remove previous containers
docker compose up -d --build
```

**Verification:** Check the `fluentd` container logs:

```
docker logs fluentd-aggregator
```

You should now see the structured JSON logs from `log-app-instance` being received and printed by the `fluentd-aggregator` container. This confirms your log forwarding is working correctly.


```
{"json":{"timestamp":1678886400.123,"level":"INFO","service":"my-log-
service","event_type":"request_processed","message":"Processing request
req-001","request_id":"req-001","user_id":"user-123","duration_ms":55.23},"con
tainer_id":"...","container_name":"/log-app-instance","source":"stdout","log_t
ag":"docker.log-
app","stream":"stdout","time":"2026-03-15T10:40:00.123456789Z"}
```

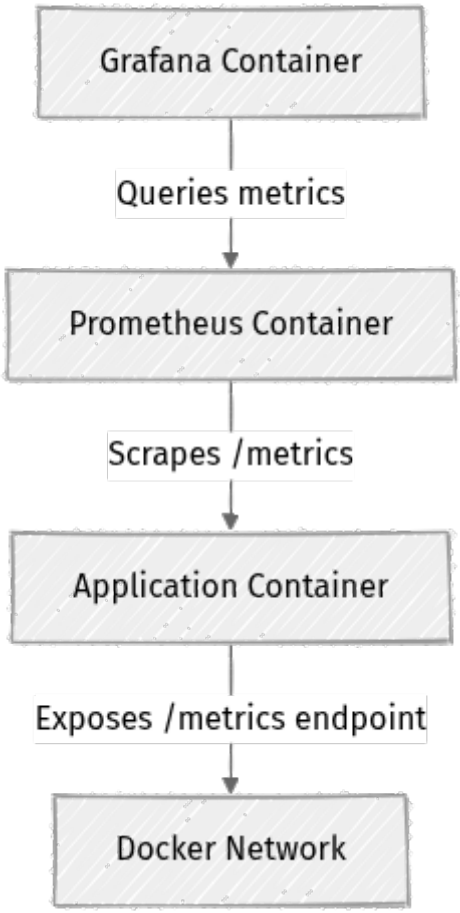
Fluentd adds its own metadata (like `container_id`, `container_name`, `log_tag`, `time`) around your application's JSON log, enriching the data for further analysis.

---

## Metrics Collection with Prometheus

Metrics provide numerical insights into your application's health and performance over time. Prometheus is an open-source monitoring system that collects metrics from configured targets by "scraping" their HTTP endpoints at regular intervals. It uses a "pull" model, where Prometheus initiates the data collection.

### Metrics Architecture Overview




### Instrumenting an Application for Prometheus

Applications need to expose a `/metrics` endpoint in a Prometheus-compatible text format. For Python, the `prometheus_client` library simplifies this.

**1. Update `my-log-app/app.py` to expose metrics:** Modify `my-log-app/app.py` to include Prometheus instrumentation.

```
# docker-observability-project/my-log-app/app.py
import logging
```

```
import json
import time
import os
import random
from prometheus_client import start_http_server, Counter, Gauge, Histogram

# Configure basic logging to stdout
logging.basicConfig(level=logging.INFO, format='%(message)s')
logger = logging.getLogger(__name__)

# Prometheus Metrics
# Counter: A cumulative metric that represents a single monotonically
increasing counter.
REQUESTS_TOTAL = Counter('http_requests_total', 'Total HTTP requests', ['metho
d', 'endpoint'])
# Histogram: Samples observations (e.g., request durations) and counts them in
configurable buckets.
REQUEST_DURATION_SECONDS = Histogram('http_request_duration_seconds', 'HTTP
Request duration in seconds', ['method', 'endpoint'])
# Gauge: A metric that represents a single numerical value that can
arbitrarily go up and down.
ACTIVE_REQUESTS = Gauge('active_requests', 'Number of active requests')
SERVICE_HEALTH = Gauge('service_health', 'Service health status (1=healthy,
0=unhealthy)')

def log_event(event_type, message, **kwargs):
    """
    Constructs and logs a structured JSON event to stdout.
    """
    log_entry = {
        "timestamp": time.time(),
        "level": "INFO",
        "service": os.getenv("SERVICE_NAME", "unknown-service"),
        "event_type": event_type,
        "message": message,
        **kwargs
    }
    logger.info(json.dumps(log_entry))

if __name__ == "__main__":
    # Start up the Prometheus HTTP server to expose the metrics.
    metrics_port = int(os.getenv("METRICS_PORT", 8000))
    start_http_server(metrics_port)
    logger.info(f"Prometheus metrics exposed on port {metrics_port}")

    count = 0
    SERVICE_HEALTH.set(1) # Initialize service as healthy

    while True:
        count += 1
        method = random.choice(['GET', 'POST', 'PUT'])
        endpoint = random.choice(['/', '/data', '/status', '/users'])

        # Increment active requests gauge
        ACTIVE_REQUESTS.inc()
        # Measure request duration using the histogram context manager
        with REQUEST_DURATION_SECONDS.labels(method, endpoint).time():
            # Simulate processing work
            time.sleep(random.uniform(0.01, 0.2))
            # Increment total requests counter
            REQUESTS_TOTAL.labels(method, endpoint).inc()
        # Decrement active requests gauge
```

```
        ACTIVE_REQUESTS.dec()

        request_id = f"req-{count:03d}"
        user_id = f"user-{random.randint(100, 999)}"

        log_event(
            "request_processed",
            f"Processing request {request_id}",
            request_id=request_id,
            user_id=user_id,
            duration_ms=round(random.uniform(10, 200), 2)
        )
        if count % 10 == 0:
            log_event(
                "warning",
                f"High load detected for request {request_id}",
                request_id=request_id,
                threshold=0.8,
                current_load=random.uniform(0.7, 0.95)
            )
            if random.random() < 0.15: # 15% chance to become unhealthy
                SERVICE_HEALTH.set(0)
                log_event("health_status_change", "Service became
unhealthy!", status="unhealthy")
            else:
                SERVICE_HEALTH.set(1) # Back to healthy (if it was unhealthy)

        time.sleep(1)
```

This updated `app.py` initializes various Prometheus metrics (`Counter`, `Gauge`, `Histogram`), starts an HTTP server to expose them on a configurable port (`METRICS_PORT`), and then updates these metrics as it simulates processing requests.

**2. Update `my-log-app/requirements.txt`:** Add the Prometheus client library dependency.

```
# docker-observability-project/my-log-app/requirements.txt
prometheus_client==0.20.0 # Latest stable as of 2026-08-06
```

**3. Update `my-log-app/Dockerfile`:** Add the `METRICS_PORT` environment variable and `EXPOSE` instruction.

```
# docker-observability-project/my-log-app/Dockerfile
FROM python:3.11-slim-bookworm AS builder

WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

FROM python:3.11-slim-bookworm

WORKDIR /app
COPY --from=builder /usr/local/lib/python3.11/site-packages /usr/local/lib/
```

```
python3.11/site-packages
COPY app.py .

ENV SERVICE_NAME="my-log-service"
ENV METRICS_PORT=8000 # Metrics server will listen on this port
EXPOSE 8000 # Inform Docker that the container listens on this port

CMD ["python", "app.py"]
```

### Setting up Prometheus with Docker Compose

**1. Create Prometheus configuration:** Inside `prometheus/`, create `prometheus.yml`:

```
# docker-observability-project/prometheus/prometheus.yml
global:
  scrape_interval: 15s # How frequently Prometheus scrapes targets
  evaluation_interval: 15s # How frequently Prometheus evaluates alerting
rules

scrape_configs:
  - job_name: 'docker_services'
    # metrics_path defaults to /metrics
    # scheme defaults to http
    static_configs:
      - targets: ['log-app:8000'] # Scrape our log-app service on its metrics
port
```

Here, `log-app` is the service name defined in `docker-compose.yaml`. Docker Compose's internal DNS resolves `log-app` to the container's IP address, allowing Prometheus to discover and scrape its metrics endpoint.

**2. Update `docker-compose.yaml` to include Prometheus:** Modify your `docker-compose.yaml` (at the project root) to add the `prometheus` service and expose the `log-app`'s metrics port within the Docker network.

```
# docker-observability-project/docker-compose.yaml
version: '3.8'

services:
  log-app:
    build:
      context: ./my-log-app
      dockerfile: Dockerfile
    container_name: log-app-instance
    environment:
      SERVICE_NAME: "my-log-service"
      METRICS_PORT: 8000
    # Expose metrics port for external access (e.g., for direct testing via
localhost:8000/metrics)
    ports:
      - "8000:8000"
    logging:
      driver: "fluentd"
```

```
      options:
        fluentd-address: fluentd:24224
        tag: "docker.log-app"

  fluentd:
    build:
      context: ./fluentd
      dockerfile: Dockerfile
    container_name: fluentd-aggregator
    volumes:
      - ./fluentd/fluent.conf:/fluentd/etc/fluent.conf:ro
    ports:
      - "24224:24224"
    command: ["fluentd", "-c", "/fluentd/etc/fluent.conf"]

  prometheus:
    image: prom/prometheus:v2.49.1 # Latest stable as of 2026-08-06
    container_name: prometheus-server
    volumes:
      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml:ro # Mount
config read-only
      - prometheus_data:/prometheus # Persistent storage for metrics data
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--web.console.libraries=/usr/share/prometheus/console_libraries'
      - '--web.console.templates=/usr/share/prometheus/consoles'
    ports:
      - "9090:9090" # Expose Prometheus UI to host port 9090

volumes:
  prometheus_data: # Define a named volume for Prometheus data persistence
```

We've added the `prometheus` service, mapped its configuration file and a persistent data volume, and exposed its UI on port 9090.

**3. Stop existing containers and run all services:**

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

**Verification:**

1. **Access `log-app` metrics directly:** Open your browser to `<http://localhost:8000/metrics>`. You should see a page with Prometheus-formatted metrics, confirming the application is exposing them correctly.
2. **Access Prometheus UI:** Open your browser to `<http://localhost:9090>`.
3. Navigate to "Status" -> "Targets". You should see `log-app:8000` listed as "UP", indicating Prometheus is successfully scraping metrics.

4. Go to the "Graph" tab. In the expression bar, type `http_requests_total` and click "Execute". You should see a graph of the total request count increasing over time. Try other metrics like `active_requests` or `service_health`.

---

## Visualization with Grafana

Grafana is an open-source analytics and interactive visualization web application. It connects to various data sources (like Prometheus) and allows you to create powerful dashboards to monitor and analyze your application's metrics and logs.

### Setting up Grafana with Docker Compose

**1. Update `docker-compose.yaml` to add the Grafana service:** Modify your `docker-compose.yaml` (at the project root) to include the `grafana` service.

```
# docker-observability-project/docker-compose.yaml
version: '3.8'

services:
  log-app:
    build:
      context: ./my-log-app
      dockerfile: Dockerfile
    container_name: log-app-instance
    environment:
      SERVICE_NAME: "my-log-service"
      METRICS_PORT: 8000
    ports:
      - "8000:8000"
    logging:
      driver: "fluentd"
      options:
        fluentd-address: fluentd:24224
        tag: "docker.log-app"

  fluentd:
    build:
      context: ./fluentd
      dockerfile: Dockerfile
    container_name: fluentd-aggregator
    volumes:
      - ./fluentd/fluent.conf:/fluentd/etc/fluent.conf:ro
    ports:
      - "24224:24224"
    command: ["fluentd", "-c", "/fluentd/etc/fluent.conf"]

  prometheus:
    image: prom/prometheus:v2.49.1
    container_name: prometheus-server
    volumes:
      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml:ro
      - prometheus_data:/prometheus
    command:
```

```
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--web.console.libraries=/usr/share/prometheus/console_libraries'
      - '--web.console.templates=/usr/share/prometheus/consoles'
    ports:
      - "9090:9090"

  grafana:
    image: grafana/grafana:10.3.3 # Latest stable as of 2026-08-06
    container_name: grafana-dashboard
    volumes:
      - grafana_data:/var/lib/grafana # Persistent storage for Grafana data
(dashboards, config)
    environment:
      - GF_SECURITY_ADMIN_USER=admin
      - GF_SECURITY_ADMIN_PASSWORD=admin # IMPORTANT: Change this in
production!
    ports:
      - "3000:3000" # Expose Grafana UI to host port 3000

volumes:
  prometheus_data:
  grafana_data: # Define a named volume for Grafana data persistence
```

We've added the `grafana` service, mounted a persistent volume for its data, and set up initial admin credentials via environment variables.

**2. Stop existing containers and run all services:**

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

**Verification:**

1. **Access Grafana UI:** Open your browser to `<http://localhost:3000>`.
2. **Log in:** Use `admin` for both username and password (as configured). You'll likely be prompted to change the password immediately.
3. **Add Prometheus as a data source:**
    - From the left-hand menu, click "Connections" (or the cog icon for "Configuration") -> "Data sources".
    - Click "Add data source" -> Select "Prometheus".
    - Set "Name": `Prometheus` (or any descriptive name).
    - Set "URL": `<http://prometheus:9090>`. Remember, `prometheus` is the service name within our Docker Compose network.
    - Scroll down and click "Save & test". You should see a green "Data source is working" message.

4. **Create a dashboard:**
    - From the left-hand menu, click "Dashboards" -> "New dashboard" -> "Add a new panel".
    - In the "Query" tab, ensure your "Prometheus" data source is selected.
    - In the "PromQL" field, enter a metric like `http_requests_total`.
    - You should see a graph of your application's total requests.
    - Experiment with other metrics (`active_requests`, `service_health`) and different visualization types (Graph, Gauge, Singlestat).

---

## Operational Readiness Checklist: Beyond Observability

While logging and metrics are foundational, a truly production-ready application requires attention to several other operational concerns.

### 1. Health Checks

Health checks allow Docker and orchestrators (like Kubernetes or Docker Swarm) to determine if a container is truly healthy and ready to serve requests, not just if its main process is running. This is critical for reliable deployments and self-healing systems.

**In `my-log-app/Dockerfile` (Add a `HEALTHCHECK` instruction):** While Docker Compose `healthcheck` block overrides this, it's good practice to include it in the Dockerfile for standalone deployments.

```
# docker-observability-project/my-log-app/Dockerfile (excerpt)
...
EXPOSE 8000

# Define a health check that pings the metrics endpoint
HEALTHCHECK --interval=30s --timeout=10s --retries=3 --start-period=5s \
  CMD curl --fail http://localhost:8000/metrics || exit 1
```

This check attempts to `curl` the `/metrics` endpoint every 30 seconds. If it fails (returns a non-zero exit code) three consecutive times, Docker considers the container unhealthy. `start-period` gives the application time to initialize before the first check.

**In `docker-compose.yaml` (Use the `healthcheck` block):** This provides more control and overrides the Dockerfile's `HEALTHCHECK`.

```
# docker-observability-project/docker-compose.yaml (excerpt for log-app)
services:
  log-app:
    # ... other configurations ...
    healthcheck:
      test: ["CMD", "curl", "--fail", "http://localhost:8000/metrics"]
      interval: 30s # Check every 30 seconds
      timeout: 10s  # Fail if check takes longer than 10 seconds
      retries: 3    # Mark unhealthy after 3 consecutive failures
      start_period: 5s # Give the app 5 seconds to warm up before starting
checks
```

**Verification:** After running `docker compose up -d`, execute `docker ps`. You'll see `(healthy)` or `(unhealthy)` in the `STATUS` column for your `log-app-instance`.

## 2. Resource Limits

Setting CPU and memory limits is crucial to prevent "noisy neighbor" issues, where one misbehaving container consumes excessive resources and impacts other services on the same host. It also helps in capacity planning.

**In `docker-compose.yaml` (Add to `log-app` service):**

```
# docker-observability-project/docker-compose.yaml (excerpt for log-app)
services:
  log-app:
    # ... other configurations ...
    deploy: # Used for swarm mode, but also respected by Docker Compose for
resource limits
      resources:
        limits:
          cpus: '0.5' # Max 0.5 CPU core (e.g., 50% of one core)
          memory: 128M # Max 128MB RAM
        reservations:
          cpus: '0.25' # Reserve 0.25 CPU core (guaranteed minimum)
          memory: 64M # Reserve 64MB RAM (guaranteed minimum)
```

`limits` are hard caps; if a container tries to exceed them, it might be throttled (CPU) or killed (memory

- OOMKilled). `reservations` are guaranteed minimums. ⚠️ `What can go wrong:` Under-resourcing can lead to performance degradation and crashes. Over-resourcing wastes expensive infrastructure. Start with reasonable estimates and refine based on observed metrics.

### 3. Graceful Shutdowns

When a container is stopped or restarted, Docker sends a `SIGTERM` signal. Applications should be designed to catch this signal, gracefully complete any ongoing tasks, release resources (e.g., close database connections, flush logs), and then exit. If the application doesn't exit within a grace period, Docker sends a `SIGKILL` (forceful termination).

**In `docker-compose.yaml` (Add to `log-app` service):**

```
# docker-observability-project/docker-compose.yaml (excerpt for log-app)
services:
  log-app:
    # ... other configurations ...
    stop_grace_period: 30s # Give the container up to 30 seconds to shut down
gracefully
```

Your application code (e.g., `app.py`) would need to implement signal handling, but `stop_grace_period` ensures Docker waits before forceful termination.

### 4. Backup Strategies

For any service with persistent data (like Prometheus and Grafana, which use named volumes), a robust backup strategy is non-negotiable. While Docker volumes provide persistence across container restarts, they don't inherently protect against host failure or accidental deletion.

⚡ `Real-world insight:` Backup strategies typically involve:

- **Volume backups:** Using host-level tools to snapshot or copy the volume data.
- **Cloud-specific solutions:** If deployed to a cloud provider, leveraging their managed backup services (e.g., AWS EBS snapshots, Azure Disk Backup).
- **Configuration backups:** Keeping `prometheus.yml`, `grafana.ini`, and dashboard JSON definitions in version control.

### 5. Alerting

Monitoring tells you what's happening; alerting tells you when something needs attention. Prometheus integrates with Alertmanager, a separate component that handles deduplicating, grouping, and routing alerts to various notification channels (email, Slack, PagerDuty, etc.).

This is a more advanced topic, but in production, you would configure rules in Prometheus to detect anomalous behavior (e.g., `service_health == 0` for more than 5 minutes, or `http_requests_total` dropping unexpectedly) and have Alertmanager notify the relevant team.

## Common Issues & Solutions for Observability

1. **Log Volume Too High / PII in Logs:**
    - **Issue:** Applications logging excessively (e.g., DEBUG level in production) or inadvertently including sensitive information (Personally Identifiable Information - PII, secrets). This can overwhelm logging systems, incur high costs, and create security/compliance risks.
    - **Solution:**
        - Implement proper log levels (DEBUG, INFO, WARN, ERROR) and configure your application to use appropriate levels for production (e.g., `INFO` or `WARN`).
        - Sanitize or redact sensitive data before logging. Use dedicated logging libraries that support structured logging and redaction.
        - `⚡ Real-world insight:` For extremely high-volume events, consider log sampling to reduce ingestion costs while retaining statistical data.
2. **Prometheus Scrape Failures (`log-app` target down):**
    - **Issue:** Prometheus cannot reach the `/metrics` endpoint of a target service, showing it as "DOWN" in the UI.
    - **Solution:**
        - Verify the target service (`log-app-instance`) is running: `docker ps`.
        - Ensure its metrics port (8000) is correctly exposed and listening within the Docker network.
        - Check the `targets` configuration in `prometheus/prometheus.yml` for correct service names and ports. Remember to use the Docker Compose service name (`log-app`) for internal communication, not `localhost`.
        - Inspect the target container's logs (`docker logs log-app-instance`) for errors related to network binding or metrics server startup.
        - Ensure no firewall rules on the host are blocking internal Docker network communication (uncommon in default Docker setups but possible with custom network configurations).

3. **Resource Exhaustion (OOMKilled):**
    - **Issue:** A container attempts to use more memory than its allocated `limits`, leading to it being forcefully terminated by the operating system (Out Of Memory Killed - OOMKilled). This results in service instability and restarts.
    - **Solution:**
        - Monitor `container_memory_usage_bytes` and `container_cpu_usage_seconds_total` metrics in Prometheus/Grafana to identify resource hogs and understand typical usage patterns.
        - Adjust `deploy.resources.limits` in `docker-compose.yaml` based on observed peak usage, leaving a small buffer.
        - Optimize application code for memory efficiency or CPU usage, if resource consumption is genuinely excessive.
        - ⚠️ `What can go wrong:` Repeated OOMKills indicate a fundamental issue with resource allocation or application design. It's a critical alert.

---

## Summary & Next Steps

In this chapter, you've moved beyond simply running containers to building a robust, observable, and operationally ready containerized application. You've gained hands-on experience with:

- **Structured Logging:** Emitting machine-readable JSON logs from your application.
- **Log Aggregation:** Using Docker's `fluentd` driver and a `fluentd` service to centralize logs.
- **Metrics Instrumentation:** Adding Prometheus-compatible metrics to your application.
- **Metrics Collection:** Setting up Prometheus to scrape and store these metrics.
- **Visualization:** Configuring Grafana to connect to Prometheus and create insightful dashboards.
- **Operational Practices:** Implementing health checks, resource limits, and understanding graceful shutdowns and backup strategies.

This foundational observability stack is indispensable for understanding the behavior of your applications in the wild, enabling proactive problem-solving, and ensuring system reliability. While we've used basic configurations, real-world systems often involve more sophisticated setups, including dedicated log storage (e.g., Elasticsearch, Loki), advanced alerting (Prometheus Alertmanager), and distributed tracing with tools like OpenTelemetry.

**Next Steps for Your Docker Journey:**

- **Explore Cloud Deployment:** Learn how to deploy your Docker Compose applications to managed container services in the cloud (e.g., AWS ECS, Azure Container Instances, Google Cloud Run). These platforms often provide integrated logging and monitoring solutions.
- **Advanced Observability:** Dive deeper into distributed tracing with OpenTelemetry to track requests across multiple services. Investigate advanced log analysis with tools like Loki for cost-effective log management.
- **CI/CD Integration:** Automate the build, test, and deployment of your containerized applications using Continuous Integration/Continuous Delivery (CI/CD) pipelines. This is the next logical step to truly operationalize your projects.

By continuously refining your observability and operational practices, you'll build more resilient, maintainable, and ultimately, more successful applications that reliably serve your users.

---

## References


- Docker Engine Docs: Configure logging drivers
- Prometheus Official Documentation
- Grafana Official Documentation
- Fluentd Official Documentation
- Prometheus Python Client Documentation


This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 09

# Building Production-Ready Docker Applications (2026 Edition)

Modern software deployment hinges on consistency and efficiency. Docker provides the tools to package applications and their dependencies into portable, isolated containers, simplifying development, testing, and production operations. This guide is designed to help you build practical, production-minded Docker applications from the ground up, equipping you with the skills to confidently containerize your projects.

## Why Docker Matters in 2026

In today's complex software landscape, ensuring that your application behaves identically across different environments—from your local machine to a staging server and finally to production—is paramount. Docker solves this by providing a standardized unit of software development, the container. By 2026, containerization has become a fundamental skill for developers and operations engineers alike, enabling faster deployment cycles, improved scalability, and robust system architectures. Mastering Docker means you can ship software more reliably and manage infrastructure more effectively.

This guide moves beyond theoretical concepts. You'll build several distinct, real-world applications, each designed to highlight specific Docker capabilities and best practices. We'll focus on practical implementation, explaining the "why" behind every decision, from Dockerfile instructions to multi-service orchestration with Docker Compose.

## What You'll Build

You will construct three complete, containerized applications, each demonstrating a different facet of modern application architecture and Docker's capabilities:

1. **A Secure Static Website with Nginx and Certbot**: You'll containerize a static web application, serve it with Nginx, and automatically provision and renew HTTPS certificates using Certbot. This project introduces core concepts like Docker volumes for persistent data and inter-container networking for service communication.

2. **A Full-Stack Application with Node.js and PostgreSQL**: This project involves building a stateless Node.js API backend connected to a persistent PostgreSQL database. You'll learn how to orchestrate multiple services, manage environment variables, and handle data persistence for stateful applications using Docker Compose.
3. **A Resilient Background Processing System with Python and Redis**: You'll develop a system where Python workers process tasks asynchronously using a Redis message queue. This project emphasizes building robust, scalable microservices, implementing health checks, and setting resource limits for containers.

By the end of this guide, you won't just know what Docker is; you'll understand how to use it to build, deploy, and secure applications ready for production environments.

## Architectural Approach

Throughout these projects, we will adhere to a microservices-like architecture where each application component (web server, API, database, worker) runs in its own container. Key architectural principles include:

- **Stateless Application Containers**: Application logic will reside in containers that can be easily scaled horizontally and replaced without data loss.
- **Persistent Data with Docker Volumes**: Databases and other stateful components will leverage Docker volumes to ensure data persists independently of container lifecycles.
- **Internal Networking**: Services will communicate securely and efficiently over Docker's internal DNS and custom networks, avoiding direct host port exposure where possible.
- **Environment-Specific Configuration**: We'll use `.env` files and Docker Compose's environment variable management to handle configurations for different deployment stages.

### Prerequisites

To get the most out of this guide, you'll need the following installed on your system:

- **Docker Desktop**: This includes Docker Engine and Docker Compose.
  - **Docker Engine**: Version `29.1.x` (checked 2026-08-06).
  - **Docker Compose**: Version `2.40.3` (checked 2026-08-06).
  - Note: If you are on Linux or macOS, you can also install Docker Engine and Docker CLI separately.
- **Git**: For cloning the project repository and version control.
- **Code Editor**: Such as VS Code, for writing and editing code.

Familiarity with basic command-line operations and a programming language (like Python or JavaScript) will be beneficial, but the guide will explain Docker-specific concepts thoroughly.

### Setting Up Your Development Environment

Before diving into the projects, we'll ensure your Docker environment is correctly set up and ready for action. The first chapter will walk you through the installation and initial verification steps.

### Learning Path

This guide is structured into a series of incremental milestones, each building upon the last to deepen your understanding of Docker and containerization best practices.

#### Setting Up Your Docker Development Environment (2026 Edition)

Install Docker Engine 29.1.x, Docker Compose 2.40.3, and essential tools, then run your first container to verify the setup.

#### Project 1: Secure Static Website with Nginx, Certbot, and Docker Compose

Containerize a static website using Nginx and automate HTTPS certificate provisioning with Certbot, demonstrating volumes and inter-container networking.

#### [Project 2: Full-Stack Application

- Node.js API with PostgreSQL Persistence]( /docker-projects-2026/project-2-fullstack-node-postgres/ ) Develop and containerize a stateless Node.js API backend connected to a persistent PostgreSQL database using Docker Compose, focusing on multi-service applications and data volumes.

### Project 3: Resilient Background Processing with Python, Redis, and Health Checks

Build a robust asynchronous task processing system using Python workers and a Redis message queue, incorporating container health checks and resource limits.

### Advanced Docker Concepts: Networking, Volumes, and Image Optimization

Deep dive into Docker's networking models, advanced volume configurations for data management, and techniques for creating smaller, more efficient container images.

### Container Security Hardening and Production Best Practices

Learn to apply essential security practices, including multi-stage builds, non-root users, and using tools like Docker Bench for Security, to harden your Docker images and containers.

### Automating Builds: Integrating Docker into a CI Pipeline

Set up a conceptual continuous integration pipeline to automate the building, testing, and pushing of Docker images to a container registry.

### Monitoring, Logging, and Operational Readiness for Containerized Apps

Implement basic logging, monitoring considerations, and robust health checks to ensure your Dockerized applications are production-ready and observable.

---

## References


- Docker Engine Documentation
- Docker Compose Documentation
- Docker Bench for Security GitHub Repository
- Best Practices for Using Azure Container Registry


This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.