AI VOID PROJECT LAB

# Build a Production Docker Stack Guide

Master building a production-ready Docker stack in 13 steps. Learn best practices for deployment, scaling, and securing modern applications with Docker Compose.

AI VOID

2026-05-22 15:52 IST

# Contents


| | | |
|---|---|---|
| 01 | Project Setup and Docker Engine Installation | 3 |
| 02 | Containerizing a Simple Web Application | 12 |
| 03 | Building and Running Your First Container Image | 22 |
| 04 | Orchestrating Services with Docker Compose | 33 |
| 05 | Integrating a Database Service (PostgreSQL) | 49 |
| 06 | Establishing Secure Inter-Service Networking | 63 |
| 07 | Handling Configuration and Secrets Securely | 75 |
| 08 | Implementing Health Checks for Service Robustness | 93 |
| 09 | Optimizing Docker Images with Multi-Stage Builds | 108 |
| 10 | Securing Containers with Non-Root Users and Resource Limits | 122 |
| 11 | Auditing Docker Host and Containers with docker-bench-security | 136 |
| 12 | Finalizing the Production Stack and Deployment Considerations | 147 |
| 13 | Building a Production-Ready Docker Compose Stack | 161 |

CHAPTER 01

# Project Setup and Docker Engine Installation

Embarking on a journey to build a production-ready application stack requires a solid foundation. This first chapter focuses on establishing that foundation: setting up your local development environment and installing **Docker Engine**. This crucial step enables you to run, build, and manage containers, which are the atomic units of modern cloud-native applications.

By the end of this chapter, you will have a fully functional Docker Engine installation on your system, verified and ready to execute your first container. This ensures consistency and reproducibility from your local machine to future deployment environments.

---

## Project Overview: Building a Production-Ready Docker Stack

Our overarching project aims to build a fully functional, multi-service web application stack deployed using Docker Compose. This guide will walk you through key production-ready practices, including image optimization, secure data persistence, health checks, and basic scaling. The target audience includes developers and DevOps engineers who want to deploy robust applications using Docker. Success means having a deployable, resilient, and observable application stack.

### Tech Stack at a Glance

For this project, our primary tools are:

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

**Version Information (Checked 2026-05-22):**

- **Docker Engine:** The exact stable version of Docker Engine is continuously updated. We will always recommend installing the latest stable release available via official channels. The installation instructions below are geared towards this.

- **Docker Compose:** We will leverage the `docker compose` CLI plugin, which adheres to the [Compose Specification](). This modern approach means we generally avoid specifying a `version` field in `docker-compose.yml` files, letting the Compose CLI automatically use the latest specification.

### Build Plan: Chapter 1 Milestones

This chapter is entirely dedicated to setting up your local Docker environment. Specifically, we will:

1. **Install Docker Engine:** Install the core Docker daemon and CLI on your chosen operating system.
2. **Verify Installation:** Confirm that Docker Engine is running and accessible by executing a test container.

---

## Core Concept: Understanding Docker Engine

At its heart, Docker Engine is a client-server application that consists of three main components:

1. **Docker Daemon (`dockerd`):** A persistent background process that manages Docker objects like images, containers, networks, and volumes. It listens for Docker API requests.
2. **Docker REST API:** An interface that programs can use to talk to the daemon and instruct it on what to do.
3. **Docker CLI (Command Line Interface):** The `docker` command you'll use in your terminal to interact with the Docker daemon via the API.

**Why Docker Engine exists:** It provides a consistent environment to package and run applications. By isolating applications into containers, Docker solves the "it works on my machine" problem, ensuring that an application behaves the same way regardless of the underlying infrastructure. This isolation also enhances security and resource management.

---

## Planning the Docker Engine Installation

A successful Docker Engine installation is the non-negotiable first step. It ensures that the `docker` command is available, the daemon is active, and your user has the necessary permissions to interact with it.

### Prerequisites

Before you start, ensure your system meets these basic requirements:

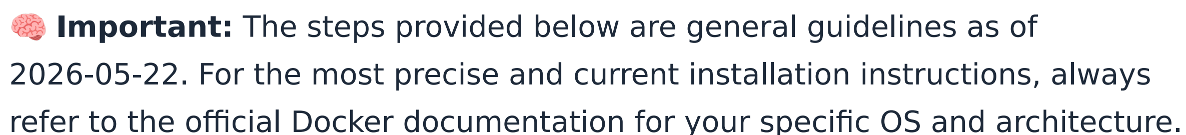
- **Operating System:** A recent, stable version of Linux (e.g., Ubuntu, Debian), macOS, or Windows 10/11.
- **Text Editor:** Visual Studio Code, Sublime Text, or similar, for future code editing.
- **Command-Line Proficiency:** Basic familiarity with running commands in a terminal or PowerShell.
- **Internet Connection:** Essential for downloading Docker packages and images.

The installation approach differs significantly based on your operating system. For Linux, we'll install directly from Docker's official repositories. For macOS and Windows, the recommended path is Docker Desktop, which bundles Docker Engine, Docker Compose, and other utilities.

---

## Step-by-Step Docker Engine Installation

Follow the instructions below for your specific operating system. Always prioritize the official Docker documentation for the most up-to-date and architecture-specific instructions.



🧠 **Important:** The steps provided below are general guidelines as of 2026-05-22. For the most precise and current installation instructions, always refer to the official Docker documentation for your specific OS and architecture.

### 1. For Linux (Ubuntu/Debian Example)

Installing Docker Engine on Linux from its official repositories is the preferred method for stability and updates.

1. **Uninstall Older Versions (if present):** Ensure a clean slate by removing any existing or conflicting Docker packages.

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

1. **Set Up Docker's Official Repository:** This involves installing necessary utility packages, adding Docker's GPG key for package verification, and then configuring the stable repository.

```
    # Update package index and install prerequisites
    sudo apt update
    sudo apt install ca-certificates curl gnupg lsb-release -y

    # Add Docker's official GPG key
    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 Docker 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

    # Update the package index again to include Docker packages
    sudo apt update
```

1. **Install Docker Engine Components:** Install the Docker Engine, CLI, and the `containerd.io` runtime, along with the `docker-buildx-plugin` and `docker-compose-plugin`.

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

1. **Add Your User to the `docker` Group:** This is a critical step to run Docker commands without prepending `sudo`. After running this command, you **must log out and log back in** (or reboot your system) for the changes to take effect.

```
    sudo usermod -aG docker $USER
```

```
**⚡ Quick Note:** Simply opening a new terminal might not be sufficient. A
full re-login ensures your user's group memberships are correctly re-
evaluated.
```

## 2. For macOS and Windows (using Docker Desktop)

For macOS and Windows environments, **Docker Desktop** is the recommended and most straightforward solution. It bundles Docker Engine, Docker Compose, Kubernetes, and other developer tools into a single, easy-to-manage application.

1. **Download Docker Desktop:** Navigate to the official Docker Desktop download page: [https://docs.docker.com/desktop/install/](https://docs.docker.com/desktop/install/) Download the installer appropriate for your operating system.
2. **Install Docker Desktop:**
    - **macOS:** Open the downloaded `.dmg` file and drag the Docker icon into your Applications folder. Launch Docker Desktop from Applications.
    - **Windows:** Run the downloaded `Docker Desktop Installer.exe`. Follow the on-screen instructions. **Crucially, ensure "Use WSL 2 based engine" is enabled** during installation if you are on Windows, as it offers superior performance and compatibility for Linux containers.
3. **Windows Specific - Enable WSL2 (if needed):** Docker Desktop on Windows relies on **Windows Subsystem for Linux 2 (WSL2)**. If you haven't already, enable and update WSL2:
    - Open PowerShell as an Administrator and run:

```
        wsl --install
        wsl --update
        wsl --set-default-version 2
```

```
- You may also need to install a Linux distribution (e.g., Ubuntu) from the Microsoft Store if you don't have one already, and then run `wsl --set-default-version 2` again after the distribution is installed.
```

1. **Start Docker Desktop:** Launch Docker Desktop from your Applications (macOS) or Start Menu (Windows). It will start the Docker Engine in the background. Wait for the Docker icon in your system tray (Windows) or menu bar (macOS) to indicate that Docker Engine is running and ready.

## Verification: Confirming Your Setup

After installation, it's vital to confirm Docker Engine is correctly installed and operational.

1. **Check Docker Version:** Open your terminal or command prompt and execute these commands:

```
    docker --version
    docker compose version
```

```
You should see output similar to this, confirming the Docker CLI and Compose
CLI are installed:
```

```
    Docker version 25.0.3, build 4de43a0
    Docker Compose version v2.24.5
```

```
*(Note: The specific version numbers will vary depending on the latest stable
release at the time of your installation.)*
```

1. **Run a Test Container:** The most definitive way to verify the Docker Engine daemon is running and responsive is to run the official `hello-world` image.

```
    docker run hello-world
```

```
**Expected Output:**
A successful execution will display output similar to this:
```

```
    Unable to find image 'hello-world:latest' locally
    latest: Pulling from library/hello-world
    ... (download progress) ...
    Digest:
sha256:f5233545e43561888496a7989027956529ce578fe166c36dc2b11617a69b406b
    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.
     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.
```

```
This message signifies that Docker Engine is fully operational, capable of
pulling images, creating containers, and executing commands within them.
```

## Production Awareness: Beyond Development Setup

While this chapter focuses on your local development environment, it's important to briefly touch upon production considerations related to Docker Engine:

- **Host Security:** The underlying operating system running Docker Engine (the "Docker host") is a critical attack surface. Secure it by limiting access, regularly patching, and following least-privilege principles. In production, tools like `docker-bench-security` can audit your Docker host and container configurations for common vulnerabilities.
- **User Permissions:** Granting a user access to the `docker` group (allowing `docker` commands without `sudo`) is convenient for development but bestows root-level privileges over the host. In production, this practice is generally avoided. Instead, orchestration platforms (like Kubernetes) or dedicated CI/CD systems interact with the Docker daemon with appropriate permissions.
- **Resource Management:** Docker Engine and its containers consume system resources (CPU, memory, disk I/O). On a development machine, ensure you have sufficient resources to avoid performance bottlenecks. In production, careful resource allocation and monitoring are essential for stability and performance.

## Troubleshooting Common Issues

Encountering issues during installation is common. Here are some frequent problems and their solutions:

1. **`docker: command not found`:**
    - **Cause:** The Docker CLI might not be installed correctly, or its executable path isn't included in your system's `PATH` environment variable. On Linux, this often means the installation process failed or was incomplete.
    - **Solution:** For Linux, meticulously re-follow the installation steps, paying close attention to repository setup. For macOS/Windows, ensure Docker Desktop is installed and running, and that its bin directory is in your system's PATH (Docker Desktop usually handles this automatically).
2. **`Got permission denied while trying to connect to the Docker daemon socket` (Linux only):**
    - **Cause:** Your user account is not a member of the `docker` group, or the group membership hasn't been activated since you added it.
    - **Solution:** Verify you ran `sudo usermod -aG docker $USER`. Then, critically, **log out and log back in** to your user session (or reboot your machine). A simple `newgrp docker` might work for the current terminal but is less reliable for a permanent fix.
3. **Docker Desktop not starting / WSL2 issues (Windows only):**
    - **Cause:** WSL2 might not be properly installed, updated, or enabled on your Windows system, or there could be a conflict with virtualization settings in your BIOS/UEFI.
    - **Solution:** Ensure WSL2 is fully installed and updated by running `wsl --update` and `wsl --set-default-version 2` in an administrative PowerShell. Check your BIOS/UEFI settings to confirm hardware virtualization (e.g., Intel VT-x or AMD-V) is enabled. Consult the official Docker Desktop troubleshooting guide for specific error messages or advanced diagnostics.

## Summary & Next Step

Congratulations! You have successfully installed Docker Engine and verified its functionality by running your first container. This foundational environment setup is now complete, equipping your machine for containerized application development.

**Key Takeaways:**

- Docker Engine is the indispensable core for building and running containers.
- Installation methods are OS-specific, with official documentation being the most reliable source for current versions.
- Correct user permissions (especially on Linux) and proper WSL2 setup (on Windows) are crucial for smooth operation.
- Running the `hello-world` container is the quickest and most effective way to confirm your Docker setup is working.

With Docker Engine ready, we are now prepared to containerize our first application. In the next chapter, we will take a simple web application and package it into a Docker image, laying the groundwork for our multi-service deployment.

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

## References


- Docker Documentation: [https://docs.docker.com/](https://docs.docker.com/)
- Install Docker Engine on Ubuntu: [https://docs.docker.com/engine/install/ubuntu/](https://docs.docker.com/engine/install/ubuntu/)
- Install Docker Desktop for Mac: [https://docs.docker.com/desktop/install/mac-install/](https://docs.docker.com/desktop/install/mac-install/)
- Install Docker Desktop for Windows: [https://docs.docker.com/desktop/install/windows-install/](https://docs.docker.com/desktop/install/windows-install/)
- Compose Specification Versioning: [https://github.com/jamesatdocker/docker-docs/blob/main/compose/compose-file/compose-versioning.md](https://github.com/jamesatdocker/docker-docs/blob/main/compose/compose-file/compose-versioning.md)

CHAPTER 02

# Containerizing a Simple Web Application

## Introduction

In the previous chapter, we set up our Docker development environment. Now, it's time to put Docker to work by containerizing our first application. This chapter guides you through taking a simple web application and packaging it into a Docker image, making it portable and isolated.

By the end of this milestone, you will have a functional Python Flask web application running inside a Docker container. You'll understand the fundamental components of a `Dockerfile` and how to build and run your custom images. This is a critical step towards building complex, multi-service applications, as it establishes the core pattern for isolating individual services.

## Planning & Design: Our First Container

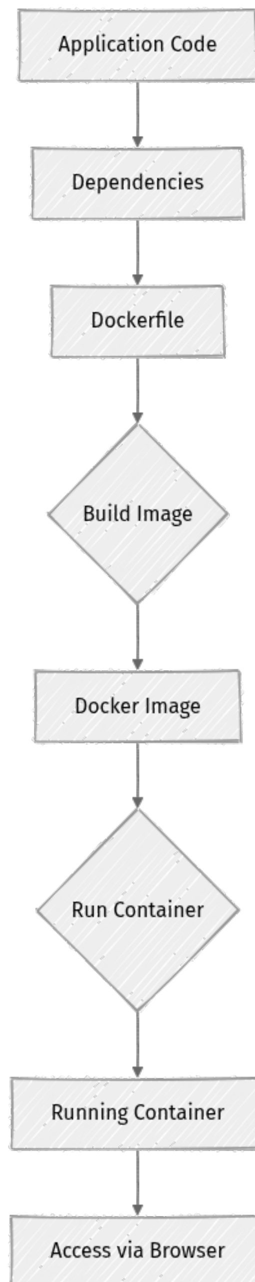
Our goal is to create a minimal web application and define its environment using a `Dockerfile`. This `Dockerfile` will serve as a blueprint for building an immutable Docker image. The image will then be used to launch a container, which is an isolated process running our application.

For this chapter, we'll use a very simple Python Flask application that returns "Hello, Docker!". This choice allows us to focus purely on Docker concepts without getting bogged down in application-specific complexities.

The file structure will be straightforward:

```
.
├── app.py
├── Dockerfile
└── requirements.txt
```

Here's the conceptual flow we'll implement:




This diagram illustrates how our application code and dependencies are combined with a `Dockerfile` to create a Docker image. This image is then run as a container, which we can access via our browser.

---

## Step-by-Step Implementation

Let's get started by creating the necessary files for our simple web application.

### 1. Create the Project Directory

First, create a new directory for our project.

```
mkdir docker-web-app
cd docker-web-app
```

### 2. Write the Web Application Code (`app.py`)

Inside the `docker-web-app` directory, create a file named `app.py` and add the following Python code.

```
# docker-web-app/app.py
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return "Hello, Docker! This is a containerized web app."

if __name__ == '__main__':
    # Listen on all public IPs (0.0.0.0) and port 8000
    # This is crucial for Docker containers to be accessible externally.
    app.run(host='0.0.0.0', port=8000)
```

**Explanation:**

- `from flask import Flask`: Imports the Flask framework.
- `app = Flask(__name__)`: Initializes a Flask application.
- `@app.route('/')`: Defines a route for the root URL (`/`).
- `def hello(): return "Hello, Docker! ..."`: The function executed when the root URL is accessed.
- `app.run(host='0.0.0.0', port=8000)`: Starts the Flask development server.
    - `host='0.0.0.0'` is critical inside a Docker container. It tells the server to listen on all available network interfaces, making it accessible from outside the container. If you used `127.0.0.1` or `localhost`, the application would only be accessible within the container's loopback interface, not from the host machine.
    - `port=8000`: The port our application will listen on inside the container.

### 3. Define Application Dependencies (`requirements.txt`)

Next, create a file named `requirements.txt` in the same directory. This file lists the Python packages our application needs.

```
# docker-web-app/requirements.txt
Flask==2.3.3
```

**Explanation:**

- `Flask==2.3.3`: Specifies that our application requires Flask version 2.3.3. Pinning versions is a good practice for reproducibility, ensuring your application runs with the exact dependencies it was developed and tested with.

### 4. Create the Dockerfile

Now, create the `Dockerfile` in the `docker-web-app` directory. This file contains the instructions Docker will use to build your image.

```
# docker-web-app/Dockerfile

# Stage 1: Build Stage (using a larger image for build tools if needed, though
not strictly for Flask)
# This uses the official Python image, version 3.9, based on Debian Buster
slim.
# The 'slim-buster' variant is preferred over 'latest' or full images
# because it contains only the minimal packages needed, resulting in a smaller
# and more secure final image.
FROM python:3.9-slim-buster AS base

# Set the working directory inside the container.
# All subsequent commands will run from this directory.
WORKDIR /app

# Copy the requirements.txt file into the container at /app.
# This step is intentionally separated from copying the rest of the
application
# code. Docker layers are cached. If only app.py changes, but requirements.txt
# does not, Docker can reuse the cached layer for dependency installation,
# speeding up subsequent builds.
COPY requirements.txt .

# Install any Python dependencies specified in requirements.txt.
# The --no-cache-dir flag helps keep the image size down by not storing pip's
cache.
# The -r flag tells pip to install from the specified requirements file.
RUN pip install --no-cache-dir -r requirements.txt

# Copy the rest of the application code into the container.
# The '.' indicates copying everything from the current host directory (where
Dockerfile is)
# to the current working directory in the container (/app).
COPY . .

# Expose port 8000. This informs Docker that the container listens on this
port
# at runtime. It's metadata; it doesn't actually publish the port.
EXPOSE 8000

# Define the command to run when the container starts.
# Use the exec form (CMD ["executable", "param1", "param2"]) for better signal
handling.
# This command starts our Flask application.
CMD ["python", "app.py"]
```

**Explanation of Dockerfile Instructions:**

- `FROM python:3.9-slim-buster AS base`:
    - `FROM`: Specifies the base image for our build. We're using `python:3.9-slim-buster`, which is a light-weight, official Python image based on Debian Buster. Using `slim` variants is a best practice for production images as they reduce image size and attack surface.
    - `AS base`: Assigns a name to this build stage. While not strictly a multi-stage build yet, it's good practice for future expansion.
- `WORKDIR /app`:
    - `WORKDIR`: Sets the working directory for subsequent instructions. All commands like `COPY` and `RUN` will operate relative to `/app` inside the container.
- `COPY requirements.txt .`:
    - `COPY`: Copies files from the host machine (where you run `docker build`) into the Docker image.
    - This copies `requirements.txt` from our project directory to `/app` inside the image.
    - **Caching Strategy**: By copying `requirements.txt` and installing dependencies before copying the rest of the application, Docker can cache this layer. If only `app.py` changes, Docker won't re-run `pip install`, significantly speeding up builds.
- `RUN pip install --no-cache-dir -r requirements.txt`:
    - `RUN`: Executes commands during the image build process.
    - This command installs the Python packages listed in `requirements.txt`.
    - `--no-cache-dir`: Prevents `pip` from storing downloaded packages in a cache, further reducing the final image size.
- `COPY . .`:
    - Copies all remaining files from the current directory (`.`) on the host into the `/app` directory in the image. This includes `app.py`.

- `EXPOSE 8000`:
  - `EXPOSE`: Documents that the container listens on the specified network port(s) at runtime. It's purely informational and doesn't publish the port to the host. Port publishing is done with the `docker run -p` command.
- `CMD ["python", "app.py"]`:
  - `CMD`: Specifies the default command to execute when a container is launched from this image.
  - Using the "exec form" (`["executable", "param1", "param2"]`) is generally recommended as it allows Docker to handle signals (like `SIGTERM` for graceful shutdown) correctly.

### 5. Build the Docker Image

With the `Dockerfile` and application code in place, navigate to your `docker-web-app` directory in your terminal and build the Docker image.

```
docker build -t my-web-app:1.0 .
```

**Explanation:**

- `docker build`: The command to build a Docker image.
- `-t my-web-app:1.0`:
  - `-t` (or `--tag`): Tags the image with a name and optional tag (version).
  - `my-web-app`: The chosen name for our image.
  - `1.0`: The version tag. It's good practice to tag images for version control.
- `.`: Specifies the build context (the directory containing the `Dockerfile` and application files). Docker will send all files in this directory to the Docker daemon for the build process.

You should see output indicating each step of your `Dockerfile` being executed, followed by a successful build message.

### 6. Run the Docker Container

Now that the image is built, let's run a container from it.

```
docker run -p 8000:8000 --name my-flask-app my-web-app:1.0
```

**Explanation:**

- `docker run`: The command to create and start a container from an image.
- `-p 8000:8000`:
    - `-p` (or `--publish`): Publishes (maps) a container's port to a host's port.
    - The format is `HOST_PORT:CONTAINER_PORT`.
    - Here, we map port `8000` on our host machine to port `8000` inside the container. This means you can access the application from your host's browser at `<http://localhost:8000`>.
- `--name my-flask-app`: Assigns a human-readable name to the container. This makes it easier to refer to the container later (e.g., for stopping or logging).
- `my-web-app:1.0`: The name and tag of the image we want to run.

Your terminal will now display the Flask application's logs, indicating it's running.

```
 * Serving Flask app 'app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production
deployment. Use a production WSGI server instead.
 * Running on http://0.0.0.0:8000
Press CTRL+C to quit
```

---

## Testing & Verification

With the container running, let's verify that our web application is accessible.

1. **Access the Application**: Open your web browser and navigate to `<http://localhost:8000`>. You should see the message: "Hello, Docker! This is a containerized web app."
2. **Inspect Container Logs**: While the container is running in your current terminal, you'll see its output. If you had run it in detached mode (`-d`), you could check logs with:

```
docker logs my-flask-app
```

```
This command shows the standard output and error streams from your running
container, which is invaluable for debugging.
```

1. **Stop and Remove the Container**: To stop the running container, press `Ctrl+C` in the terminal where `docker run` is active. To remove the container (if it's stopped), use:

```
docker rm my-flask-app
```

```
If it's still running, you'd first stop it:
```

```
docker stop my-flask-app
docker rm my-flask-app
```

```
`docker ps -a` will show all containers, including stopped ones. Use it to
confirm `my-flask-app` is gone.
```

---

## Production Considerations

Even with this simple application, we can apply some production-minded thinking.

- **Image Size**: Using `python:3.9-slim-buster` significantly reduces image size compared to a full `python:3.9` image, which includes many development tools and libraries not needed at runtime. Smaller images mean faster downloads, less storage, and a reduced attack surface.
- **Non-Root User**: For enhanced security, containers should ideally run processes as a non-root user. Our current `Dockerfile` implicitly runs `python app.py` as root (the default user in most base images). In later chapters, we'll implement explicit user creation and switching using the `USER` instruction.
- **Resource Limits**: In a production environment, you would configure CPU and memory limits for your containers to prevent a single misbehaving application from consuming all host resources. This is typically done at runtime with `docker run --cpus` and `--memory`.

- **Development vs. Production Servers**: The Flask development server (`app.run()`) is not suitable for production. It's single-threaded and lacks robustness. In a real application, you would use a production-ready WSGI server like Gunicorn or uWSGI to serve your Flask app. We'll introduce this in a future chapter.

## Common Issues & Solutions

- **"Hello, Docker!" not appearing in browser**:
  - **Check `docker run -p`**: Ensure you correctly mapped the host port to the container port (`-p 8000:8000`).
  - **Check `app.run(host='0.0.0.0')`**: Verify your Flask app is listening on `0.0.0.0` inside the container. If it's `127.0.0.1`, it won't be reachable from the host.
  - **Check container logs**: Use `docker logs my-flask-app` to see if the application started successfully or if there are any errors.
- **`ModuleNotFoundError: No module named 'flask'`**:
  - This indicates Flask wasn't installed inside the container. Double-check your `requirements.txt` file and the `RUN pip install` command in your `Dockerfile`. Ensure `requirements.txt` is copied before the `RUN` instruction.
- **`docker build` fails with "no such file or directory"**:
  - Ensure your `Dockerfile`, `app.py`, and `requirements.txt` are all in the `docker-web-app` directory and that you are running `docker build` from that directory (`docker build -t ... .`). The `.` at the end is crucial.

## Summary & Next Step

You've successfully containerized your first web application! You now have a solid understanding of:

- Creating a basic `Dockerfile` to define an application's environment.
- Using `FROM`, `WORKDIR`, `COPY`, `RUN`, `EXPOSE`, and `CMD` instructions.
- Building a Docker image with `docker build`.
- Running a Docker container and mapping ports with `docker run`.

- Basic verification steps and initial production considerations.

This isolated, portable application is now ready to be integrated with other services. In the next chapter, we will introduce a database service and learn how to orchestrate multiple containers using Docker Compose, moving us closer to a full-stack, production-ready environment.

---

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

---

## References


- Docker Documentation
- Dockerfile reference
- Python Docker Official Images
- Flask Documentation

CHAPTER 03

# Building and Running Your First Container Image

In this chapter, we'll take our first concrete step towards a production-ready application stack: containerizing a simple web application. You'll learn how to define a Docker image using a `Dockerfile`, build that image, and then run it as a Docker container. This is the foundational skill for all subsequent containerized deployments and is essential for achieving consistent, isolated environments.

By the end of this milestone, you will have a working "Hello World" web server running inside its own isolated Docker container, accessible from your host machine. This demonstrates the core Docker workflow of packaging an application and its dependencies into a portable unit, a critical step for modern deployments.

---

## Project Overview for This Chapter

Our objective for this chapter is straightforward: transform a basic Node.js Express web application into a Docker image and run it as a container. This involves:

1. Creating a minimal Node.js Express application.
2. Writing a `Dockerfile` to define the container environment and application build steps.
3. Building a Docker image from the `Dockerfile`.
4. Running a Docker container from the newly built image.
5. Verifying the application is running and accessible.

This process lays the groundwork for understanding how individual services are packaged and executed in a containerized world.

---

## Tech Stack

For this chapter, we're focusing on:

- **Docker Engine**: The core platform for building and running containers. (Version unknown, checked 2026-05-22)

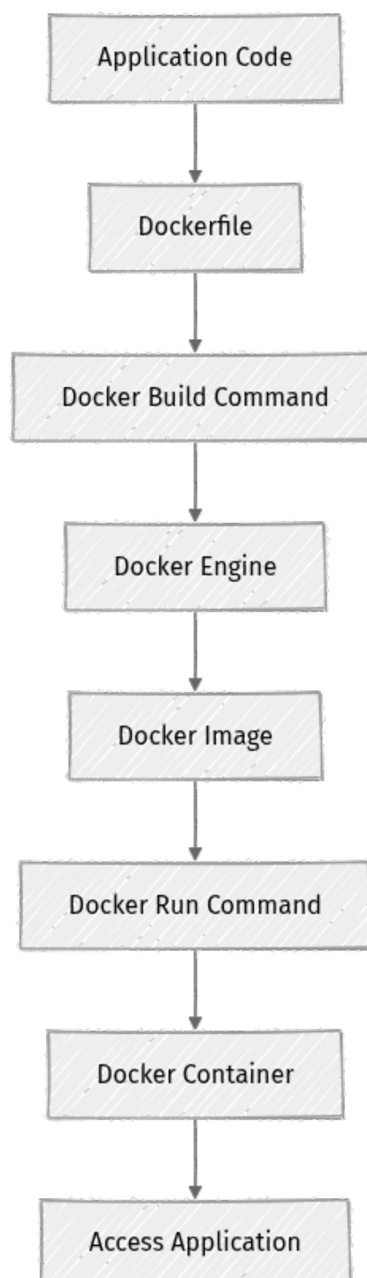
- **Node.js**: The JavaScript runtime for our simple web application. We'll use Node.js 20, which is the current LTS release as of 2026-05-22.
- **Express.js**: A minimal and flexible Node.js web application framework.

---

## Architecture: Docker Image Build and Run Flow

The `Dockerfile` acts as a blueprint, specifying everything needed to run our application, from the base operating system to the application code itself and its dependencies. This ensures that our application runs consistently across different environments.

The process of creating and running our container will follow these steps:



## Build Plan: Milestones for Containerization

We'll break down the containerization process into these concrete steps:

1. **Project Setup**: Create the root directory for our application.
2. **Application Development**: Write a simple Node.js Express server.
3. **Dockerfile Creation**: Define the image build instructions.
4. **Image Building**: Execute the `docker build` command.
5. **Container Execution**: Run the image as a container with port mapping.
6. **Verification**: Test the running application and inspect container status.

### Initial Project Structure

We'll start with a clean directory, then add our application files and `Dockerfile` to it.

```
my-docker-app/
├── app.js
├── package.json
└── Dockerfile
```

## Step-by-Step Implementation

Let's get hands-on and build our first container.

### 1. Set Up Your Project Directory

First, create a new directory for our project and navigate into it. This will be the "build context" for our Docker image.

```
mkdir my-docker-app
cd my-docker-app
```

### 2. Create a Simple Node.js Application

We'll create a basic Express application that listens on port `3000` and responds with "Hello from Docker!".

**File: `package.json`**

Start by initializing a `package.json` file. This manages our project's metadata and dependencies.

```
npm init -y
```

The `-y` flag accepts all default prompts, creating a basic `package.json` file. Next, install Express:

```
npm install express
```

This command adds `express` as a dependency to your `package.json` and installs it into `node_modules`.

**File: `app.js`**

Create `app.js` in your `my-docker-app` directory. This file contains our simple web server logic.

```
// my-docker-app/app.js
const express = require('express');
const app = express();
const port = 3000; // The port our app will listen on inside the container

app.get('/', (req, res) => {
  console.log('Received request for /'); // Log requests for visibility
  res.send('Hello from Docker!');
});

app.listen(port, () => {
  console.log(`Web server listening on port ${port}`);
});
```

This is a straightforward Express server. It's designed to be minimal to focus on the Docker concepts without application complexity.

### 3. Craft Your Dockerfile

The `Dockerfile` contains instructions for Docker on how to build your image. Create a file named `Dockerfile` (no extension) in the `my-docker-app` directory.

**File: `Dockerfile`**

```
# my-docker-app/Dockerfile

# Stage 1: Use a minimal Node.js base image for building
# Node.js 20 is the current LTS as of 2026-05-22.
# 'alpine' variants are much smaller and more secure than full Debian images.
FROM node:20-alpine AS base

# Set the working directory inside the container
# All subsequent commands will execute relative to this directory.
WORKDIR /app

# Copy package.json and package-lock.json to leverage Docker's build cache.
# This crucial step ensures npm install only runs if dependency definitions
change,
# speeding up subsequent builds.
```

```
COPY package*.json ./

# Install application dependencies
# The --omit=dev flag ensures only production dependencies are installed,
# significantly reducing the image size and potential attack surface.
RUN npm install --omit=dev

# Copy the rest of the application code into the container's working
directory.
COPY . .

# Expose the port the app runs on.
# This is documentation; it informs Docker that the container listens on this
port
# but does not actually publish it to the host.
EXPOSE 3000

# Define the command to run when the container starts.
# This uses the default Node.js executable from the base image to start our
app.
CMD ["node", "app.js"]
```

#### Explanation of Dockerfile Instructions:

- `FROM node:20-alpine AS base`:
    - **What it does**: Specifies the base image for our build.
    - **Why it's used**: Provides a pre-configured environment with Node.js 20.
    - **Tradeoffs/Decisions**: We choose `alpine` for its small size and minimal footprint, which translates to faster downloads, less disk space, and a reduced attack surface compared to larger Debian-based images. `AS base` names this build stage, a practice that becomes powerful with multi-stage builds (covered later).
- `WORKDIR /app`:
    - **What it does**: Sets the working directory inside the container for any subsequent `RUN`, `CMD`, `ENTRYPOINT`, `COPY`, or `ADD` instructions.
    - **Why it's used**: Keeps our application files organized within the container's filesystem.

- `COPY package*.json ./`:
    - **What it does**: Copies `package.json` and `package-lock.json` (if it exists) from your host machine's current directory (the build context) into the `/app` directory inside the container.
    - **Why it's used**: This is a key optimization. By copying only the dependency manifest files before installing dependencies, Docker can cache the `npm install` layer. If `package.json` doesn't change, Docker reuses the cached layer, speeding up subsequent builds.
- `RUN npm install --omit=dev`:
    - **What it does**: Executes the `npm install` command inside the container.
    - **Why it's used**: Installs all the production dependencies listed in `package.json`. The `--omit=dev` flag is crucial for production images, as it prevents development dependencies from being installed, significantly reducing image size and potential vulnerabilities.
    - `📌 Key Idea:` Always minimize dependencies in production images.
- `COPY . .`:
    - **What it does**: Copies all remaining files from your current host directory (`.`, the build context) into the `/app` directory inside the container (`.`).
    - **Why it's used**: Brings your application source code into the image. This step is placed after `npm install` to benefit from build caching if only code changes, not dependencies.
- `EXPOSE 3000`:
    - **What it does**: Informs Docker that the container listens on port `3000` at runtime.
    - **Why it's used**: It's documentation for anyone inspecting the image or for tools like `docker inspect`. It does not actually publish the port to the host system.
- `CMD ["node", "app.js"]`:
    - **What it does**: Specifies the default command that will be executed when the container starts.
    - **Why it's used**: This is how our application is launched when a container is created from this image. This command can be overridden when running the container.

### 4. Build the Docker Image

Now, let's build the image using the `docker build` command. Make sure you are in the `my-docker-app` directory.

```
docker build -t my-web-app:1.0.0 .
```

#### Command Explanation:

- `docker build`: The command to build a Docker image.
- `-t my-web-app:1.0.0`: The `-t` (tag) flag names and optionally tags your image. `my-web-app` is the image name, and `1.0.0` is the tag (version). It's good practice to tag images with meaningful, version-controlled names.
- `.`: The build context. This tells Docker to look for the `Dockerfile` and associated files in the current directory. All files in this directory are sent to the Docker daemon for the build process.

You will see output as Docker executes each step in your `Dockerfile`. Each `RUN` command creates a new layer in the image. If successful, the last line will indicate that the image was built and tagged.

### 5. Run Your Container

With the image built, we can now run it as a container.

```
docker run -p 3000:3000 my-web-app:1.0.0
```

#### Command Explanation:

- `docker run`: The command to create and start a new container from an image.
- `-p 3000:3000`: The `-p` (publish) flag is critical. It maps a port from your host machine to a port inside the container. The format is `HOST_PORT:CONTAINER_PORT`. Here, we're mapping host port `3000` to container port `3000`. This is essential for accessing your web application from your browser or other clients on your host machine.
- `my-web-app:1.0.0`: The name and tag of the image you want to run.

You should see output similar to: `Web server listening on port 3000`. This confirms your Node.js application is running inside the container and listening on its internal port.

## Testing & Verification

Now that the container is running, let's verify that our application is accessible and behaving as expected.

1. **Access the Application**: Open your web browser and navigate to `<http://localhost:3000`>. You should see the text "Hello from Docker!". This confirms your application is serving content through the mapped port.
2. **Inspect Running Containers**: Open a new terminal window (keep the one running your container open). Use `docker ps` to see currently running containers.

```
docker ps
```

```
You should see an entry for `my-web-app:1.0.0`, showing its `CONTAINER ID`,
`IMAGE`, `COMMAND`, `CREATED`, `STATUS` (e.g., `Up X seconds`), `PORTS` (e.g.,
`0.0.0.0:3000->3000/tcp`), and `NAMES`. Note the `PORTS` column, which
explicitly shows the host-to-container port mapping.
```

1. **Check Container Logs**: You can also inspect the logs generated by your application inside the container. In your new terminal, replace `<CONTAINER_ID_FROM_DOCKER_PS>` with the actual ID from the `docker ps` command.

```
docker logs <CONTAINER_ID_FROM_DOCKER_PS>
```

```
You should see the output from your `app.js` like `Web server listening on
port 3000` and `Received request for /` if you accessed it via the browser.
This is crucial for debugging and understanding container behavior.
```

## Production Considerations

Even for a simple "Hello World," thinking about production best practices from the start is crucial. These early habits prevent major issues down the line.

- **Image Size**:
    - **Why it matters**: Smaller images mean faster image pulls, less disk usage, and a reduced attack surface. `node:20-alpine` is a prime example of this.
    - **Real-world insight**: In CI/CD pipelines, smaller images drastically reduce build and deployment times. In cloud environments, they can lower storage costs.
- **Security (Least Privilege)**:
    - **What can go wrong**: Our current `Dockerfile` runs `npm install` and the `node app.js` command as the `root` user by default inside the container. Running as root is a security risk because if an attacker compromises your application, they gain root privileges within the container.
    - **Optimization / Pro tip**: In production, you should ideally run containers with a non-root user. This is achieved using the `USER` instruction in the Dockerfile. We will explore this in a later chapter to keep this introduction focused.
- **Resource Management**:
    - **What can go wrong**: By default, Docker containers can consume all available CPU and memory on the host. One misbehaving container (e.g., a memory leak) could starve other containers or even crash the host system.
    - **Real-world insight**: In production, it's critical to set resource limits (CPU, memory) for containers using `docker run --cpus` and `--memory`. When we introduce Docker Compose, we'll see how to define these limits declaratively.

- **Port Exposure**:
  - **Why it matters**: We only exposed port `3000` using `-p 3000:3000`. Only expose ports that are absolutely necessary for your application to function.
  - **What can go wrong**: Exposing unnecessary ports is a significant security risk, as it opens up potential attack vectors to services that shouldn't be publicly accessible.

---

## Common Issues & Solutions

Here are some common issues you might encounter and how to troubleshoot them:

- **"Cannot connect to the Docker daemon"**:
  - **Cause**: The Docker Engine is not running on your machine.
  - **Solution**: Start your Docker Desktop application (on Windows/macOS) or ensure the Docker service is running (on Linux: `sudo systemctl start docker`).
- **"Error response from daemon: driver failed programming external connectivity on endpoint..." or "port is already allocated"**:
  - **Cause**: Port `3000` on your host machine is already in use by another process.
  - **Solution**: Stop the other process, or choose a different host port to map (e.g., `docker run -p 8080:3000 my-web-app:1.0.0`).
- **"No such image: my-web-app:1.0.0"**:
  - **Cause**: You might have a typo in the image name or tag, or the image didn't build successfully.
  - **Solution**: Run `docker images` to see a list of all locally available images and verify the name and tag. Re-run the `docker build` command if the image is missing or the tag is incorrect.

- **Application not starting/crashing**:
    - **Cause**: The container starts but immediately exits, or you can't access it. This often means there's an error in your `app.js` or the `CMD` instruction.
    - **Solution**: Check the container logs using `docker logs <CONTAINER_ID>`. This will show you any errors or output from your Node.js application, helping you pinpoint the problem.

---

## Summary & Next Step

Congratulations! You've successfully built your first Docker image and run it as a container. You now understand the fundamental process of:

- Defining application dependencies and execution steps in a `Dockerfile`.
- Using `docker build` to create a reusable, versioned image.
- Using `docker run` to launch an isolated container, mapping necessary ports for external access.

This containerized web application is a standalone, portable unit. It can now run consistently across any environment with Docker installed. In the next chapter, we'll introduce another service—a database—and learn how to make these services communicate with each other, moving us closer to a full multi-service application.

---

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

---

## References


- Docker Documentation
- Dockerfile reference
- Node.js Docker Official Image
- Docker `CMD` instruction
- Docker `EXPOSE` instruction
- Node.js Long Term Support (LTS) Schedule
- Express.js Official Website

CHAPTER 04

# Orchestrating Services with Docker Compose

## Orchestrating Services with Docker Compose

Modern applications rarely consist of a single, monolithic service. Instead, they are typically composed of multiple interconnected components: a web frontend, a backend API, a database, perhaps a caching layer, and other auxiliary services. Manually managing the lifecycle, networking, and configuration of these interconnected containers can quickly become complex, time-consuming, and prone to error.

This chapter introduces Docker Compose, a powerful command-line tool designed to simplify the definition and management of multi-container Docker applications. By using a single YAML file, you can declaratively define your entire application stack, ensuring consistency and reproducibility across development, testing, and even production environments.

By the end of this milestone, you will have a fully functional, two-service web application stack—a Python Flask web application and a PostgreSQL database—all orchestrated and managed by Docker Compose. This setup will demonstrate how services communicate securely over an internal network, laying the groundwork for more advanced configurations and production-ready deployments. You'll be able to bring your entire application stack up and tear it down with simple, consistent commands.

## Project Overview: A Multi-Service Web Application

In this chapter, we are building a foundational multi-service application stack. The goal is to deploy a simple web application that connects to a database, simulating a common real-world scenario. This setup will serve as the base for implementing further production-ready practices in subsequent chapters.

The core components we will integrate are:

- **Web Service:** A lightweight Python Flask application that exposes an HTTP endpoint. Its primary function in this chapter is to demonstrate connectivity to the database.

- **Database Service:** A PostgreSQL database instance responsible for persistent data storage.

These services will be defined in a `docker-compose.yml` file, which orchestrates their deployment, networking, and initial configuration.

---

## Tech Stack: Docker Compose for Orchestration

Our primary tool for this chapter is Docker Compose.

- **Docker Engine:** (Version unknown as of 2026-05-22). We assume you have a working Docker Engine installation (on Linux, macOS, or Windows with WSL2).
- **Docker Compose:** Adheres to the Compose Specification (as of 2026-05-22). This specification defines the `docker-compose.yml` file format. Modern Docker Compose installations automatically follow this specification, meaning you no longer need to specify a `version` field within your `docker-compose.yml` file. This ensures forward compatibility and access to the latest features.
- **Python 3.11:** For the web application.
- **Flask 2.3.3:** The web framework.
- **PostgreSQL 16:** The database server.

---

## Build Plan: Orchestrating Your Services

To achieve our goal of a multi-service application, we will follow these steps:

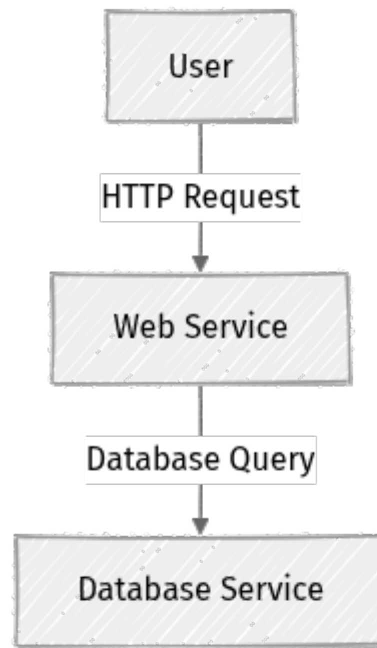
1. **Review existing application files:** Ensure our Flask app and its `Dockerfile` are ready.
2. **Define `docker-compose.yml`:** Create the central configuration file for our services.
3. **Configure `web` service:** Detail how our Flask app will be built and exposed.
4. **Configure `db` service:** Set up the PostgreSQL database container.
5. **Define custom network:** Establish an isolated network for inter-service communication.
6. **Add data volume definition:** Prepare for persistent database storage (though detailed volume management is next chapter).

7. **Deploy and verify:** Bring up the stack and confirm connectivity.

---

## Architecture: Multi-Service Interaction

Our application architecture is straightforward for this milestone, focusing on a client-server pattern with a database backend.




**Explanation:**

- A `User` sends an HTTP request to the `Web Service` (our Flask application).
- The `Web Service` attempts to connect to and query the `Database Service` (PostgreSQL).
- All communication between `Web_Service` and `Database_Service` happens over an internal Docker network, ensuring isolation.

### Project Directory Structure

To keep our project organized, we'll maintain the following structure. This assumes you've already set up the `web/Dockerfile` and `web/app.py` from previous chapters.

```
your-project/
├── docker-compose.yml       # Our orchestration file
├── web/
│   ├── Dockerfile           # Dockerfile for the web service
│   ├── app.py               # Python Flask application
│   └── requirements.txt     # Python dependencies
└── .env                     # (Will be added in a later chapter for secrets)
```

## Step-by-Step Implementation: Defining `docker-compose.yml`

Let's create our `docker-compose.yml` file and define our services.

### 1. Prepare Web Application Files

Ensure your `web` directory contains the necessary files. If you're starting fresh, create these files:

**`your-project/web/requirements.txt`**:

```
Flask==2.3.3
psycopg2-binary==2.9.9
```

- **Decision:** We use `psycopg2-binary` for simpler installation without requiring local PostgreSQL development headers, which is often preferred in containerized environments for faster builds.

**`your-project/web/app.py`**:

```python
# your-project/web/app.py
import os
from flask import Flask
import psycopg2
from psycopg2 import OperationalError

app = Flask(__name__)

@app.route('/')
def hello():
    db_url = os.environ.get("DATABASE_URL")
    if not db_url:
        return "Error: DATABASE_URL environment variable not set.", 500

    try:
        # Attempt to connect to the database
        conn = psycopg2.connect(db_url)
        cur = conn.cursor()
        cur.execute("SELECT 1") # A simple query to check connection
        cur.close()
        conn.close()
        return "Hello from Flask! Connected to database successfully!"
    except OperationalError as e:
        # Specific error for database connection issues
        return f"Hello from Flask! Failed to connect to database: {e}", 500
    except Exception as e:
        # Catch any other unexpected errors
        return f"Hello from Flask! An unexpected error occurred: {e}", 500
```

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8000)
```

- **Explanation:** This Flask app attempts to connect to a PostgreSQL database using the `DATABASE_URL` environment variable. It then performs a simple query to verify the connection. Error handling is included to provide clearer messages if the database connection fails.

**`your-project/web/Dockerfile`**:

```
# your-project/web/Dockerfile

# Stage 1: Build dependencies
FROM python:3.11-slim-bullseye AS builder

WORKDIR /app

# Install build dependencies required for psycopg2-binary
# These are only needed for the build stage, not the final runtime image
RUN apt-get update && apt-get install -y --no-install-recommends \
    build-essential \
    libpq-dev \
    gcc \
    && rm -rf /var/lib/apt/lists/*

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

# Stage 2: Create final image
FROM python:3.11-slim-bullseye

WORKDIR /app

# Copy only runtime dependencies from builder stage
COPY --from=builder /usr/local/lib/python3.11/site-packages /usr/local/lib/
python3.11/site-packages
# Copy Flask executable if it's installed to a custom bin path (depends on pip
env)
# For simplicity and robustness, ensure Flask is in site-packages and invoked
via python -m flask
# If flask command is not found, you might need to adjust this or rely on
python -m flask
# For standard installations, this line might not be strictly necessary as
Flask is found via PATH in site-packages
# COPY --from=builder /usr/local/bin/flask /usr/local/bin/flask

COPY app.py .

# Expose the port the app runs on
EXPOSE 8000

# Set environment variables for the application
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0

# Command to run the application
# Use python -m flask run for better compatibility with different Flask
```

```
installations
CMD ["python", "-m", "flask", "run", "--port=8000"]
```

- **Decision:** This `Dockerfile` uses a multi-stage build. The `builder` stage includes `build-essential` and `libpq-dev` to compile `psycopg2-binary`. The final stage, `python:3.11-slim-bullseye`, is much smaller as it only copies the compiled Python packages and the application code, discarding unnecessary build tools. This is a crucial production practice for smaller, more secure images.
- **Modern Flask execution:** Changed `CMD ["flask", "run", ...]` to `CMD ["python", "-m", "flask", "run", ...]` for better reliability in diverse Python environments and container setups.

### 2. Create the `docker-compose.yml` File

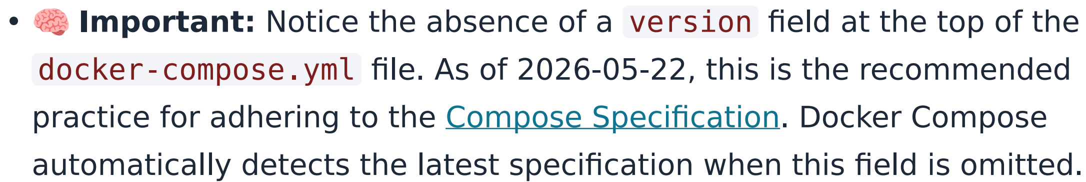
In the root of your `your-project` directory, create a new file named `docker-compose.yml`.

```
# your-project/docker-compose.yml
services:
  web:
    build: ./web
    ports:
      - "8000:8000"
    depends_on:
      - db
    environment:
      DATABASE_URL: postgresql://user:password@db:5432/mydatabase
    networks:
      - app_network

  db:
    image: postgres:16-alpine # Using a specific, lightweight version
    environment:
      POSTGRES_DB: mydatabase
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    volumes:
      - db_data:/var/lib/postgresql/data # Persistent data volume
    networks:
      - app_network

networks:
  app_network:
    driver: bridge # Default driver, explicitly stated for clarity
```

```
volumes:
  db_data: # Define the named volume for database persistence
```



- 🧠 **Important:** Notice the absence of a `version` field at the top of the `docker-compose.yml` file. As of 2026-05-22, this is the recommended practice for adhering to the [Compose Specification](). Docker Compose automatically detects the latest specification when this field is omitted.

## 3. Understanding Each Section of `docker-compose.yml`

Let's break down what each part of this configuration does.

### `services` Block

This top-level key defines the individual containers that form your application stack. Each key under `services` (e.g., `web`, `db`) represents a service. Docker Compose will manage these as separate containers.

### `web` Service Configuration

```
web:
  build: ./web
  ports:
    - "8000:8000"
  depends_on:
    - db
  environment:
    DATABASE_URL: postgresql://user:password@db:5432/mydatabase
  networks:
    - app_network
```

- **`build: ./web`**: This instruction tells Docker Compose to build the image for the `web` service using the `Dockerfile` located in the `./web` directory, relative to `docker-compose.yml`.
- **`ports: - "8000:8000"`**: This maps port `8000` on your host machine to port `8000` inside the `web` container. This allows you to access your web application from your host machine's browser at `<http://localhost:8000`>.

- **`depends_on: - db`**: This specifies that the `web` service has a dependency on the `db` service. Docker Compose will start the `db` container before the `web` container.
  - ⚠️ **What can go wrong:** While `depends_on` ensures startup order, it **does not wait for the dependent service to be fully ready** (e.g., the database accepting connections). This is a common source of "connection refused" errors during startup. We will implement robust health checks in a future chapter to address this.
- **`environment:`**: This section sets environment variables inside the `web` container.
  - `DATABASE_URL: postgresql://user:password@db:5432/mydatabase`: This URL configures the Flask application to connect to our PostgreSQL database. The hostname `db` is resolved by Docker Compose to the `db` service's container IP within the `app_network`. The port `5432` is the default for PostgreSQL.
- **`networks: - app_network`**: This assigns the `web` service to our custom `app_network`. Services on the same network can communicate with each other using their service names as hostnames.

#### db Service Configuration

```
db:
  image: postgres:16-alpine # Using a specific, lightweight version
  environment:
    POSTGRES_DB: mydatabase
    POSTGRES_USER: user
    POSTGRES_PASSWORD: password
  volumes:
    - db_data:/var/lib/postgresql/data # Persistent data volume
  networks:
    - app_network
```

- **`image: postgres:16-alpine`**: This specifies that the `db` service should use the official `postgres` Docker image, specifically version `16` based on the lightweight `alpine` Linux distribution. Using specific, lightweight base images is a good production practice to reduce image size and attack surface.

- `environment:` : These variables are used by the PostgreSQL image to initialize the database when the container starts for the first time.
  - `POSTGRES_DB`, `POSTGRES_USER`, `POSTGRES_PASSWORD` : These define the default database name, username, and password.
  - ⚠️ **What can go wrong:** Hardcoding sensitive information like passwords directly in `docker-compose.yml` is a **significant security risk in production**. For demonstration purposes, we are doing it now, but we will cover proper secrets management using `.env` files and Docker secrets in upcoming chapters.
- `volumes: - db_data:/var/lib/postgresql/data` : This line mounts a named volume called `db_data` to the `/var/lib/postgresql/data` directory inside the `db` container. This is crucial for **data persistence**. Without it, all your database data would be lost every time the `db` container is removed. We'll explore volumes in detail in the next chapter.
- `networks: - app_network` : Assigns the `db` service to the `app_network`, allowing it to communicate with the `web` service.

#### `networks` Block

This top-level key defines custom networks for your services.

```
networks:
  app_network:
    driver: bridge
```

- `app_network:` : This is the name of our custom network.
- `driver: bridge` : Specifies the network driver. `bridge` is the default and most common driver for single-host Docker setups. It creates a private, isolated internal network that services can join, providing secure communication channels.

#### `volumes` Block

This top-level key defines named volumes for data persistence.

```
volumes:
  db_data:
```

- `db_data:` : This line defines a named volume called `db_data`. Docker manages the actual storage location on the host system. Named volumes are the preferred way to persist data in Docker as they are managed by Docker, easier to back up, and don't tie your data to specific host paths.

## Testing & Verification

Now that our `docker-compose.yml` file and application code are ready, let's deploy our stack and verify its functionality.

1. **Navigate to your project root:** Open your terminal and ensure you are in the `your-project` directory, where `docker-compose.yml` is located.

```
cd your-project
```

1. **Start the services:** Use `docker compose up` to build (if necessary) and start all services defined in your `docker-compose.yml` file. The `-d` flag runs them in detached mode, meaning the containers will run in the background, freeing up your terminal.

```
docker compose up -d
```

```
- **Explanation:** This command orchestrates the entire startup process:
    - It first builds the `web` service image (if it doesn't exist or if its
`Dockerfile` or context has changed).
    - It then pulls the `postgres:16-alpine` image (if not already local).
    - It creates the `app_network` and the `db_data` volume.
    - Finally, it starts both the `db` and `web` containers in the defined
order.
```

1. **Verify service status:** Check that your containers are running as expected.

```
docker compose ps
```

```
You should see output similar to this, indicating both services are `running`:
```

```
    NAME                 COMMAND                  SERVICE
STATUS              PORTS
    your-project-web-1   "python -m flask run…"   web
running             0.0.0.0:8000->8000/tcp, :::8000->8000/tcp
    your-project-db-1    "docker-entrypoint.s…"   db
running             5432/tcp
```

```
- **Verification:** Confirm that both the `web` and `db` services show a
`running` status.
```

1. **Inspect logs:** Review the logs of your services to ensure they started without errors and are behaving as expected.

```
docker compose logs web
docker compose logs db
```

```
- **Verification:** You should see Flask startup messages for the `web`
service and PostgreSQL initialization/startup messages for the `db` service.
Look for any `ERROR` or `FAIL` messages.
```

1. **Access the web application:** Open your web browser and navigate to `<http://localhost:8000>`.
   - **Expected Output:** You should see the message: `Hello from Flask! Connected to database successfully!`
   - **Verification:** If you see this message, your Flask application successfully started, connected to the PostgreSQL database, and performed a simple query. This confirms inter-service communication and basic database functionality. If you receive an error, it indicates a problem with the database connection from the Flask app or the database itself.
2. **Stop and remove services:** When you're finished experimenting, you can stop and remove all services, networks, and optionally volumes defined in your `docker-compose.yml` with:

```
docker compose down
```

```
- **Explanation:** This command gracefully stops the running containers,
removes them, and also removes the custom network `app_network`. By default,
`docker compose down` *does not* remove named volumes (like `db_data`), which
is a safety measure to prevent accidental data loss.
- **🔥 Pro tip:** To remove volumes as well (e.g., for a clean slate during
development), you would explicitly use `docker compose down --volumes`. Be
cautious with this command in production-like environments, as it will delete
your persistent data.
```

## Production Considerations

While Docker Compose is excellent for development and testing, some aspects require careful thought for production deployments:

- **Network Isolation:** Docker Compose automatically creates a dedicated, isolated network for your services. This is a fundamental security practice, as it prevents your database from being directly accessible from the host network or the internet unless explicitly exposed via `ports` (which we deliberately did not do for `db`).
- **Environment Variables and Secrets:** We currently hardcode database credentials in `docker-compose.yml`. This is **unacceptable for production environments**. Sensitive information should be managed using:
    - `.env` files (for local development, handled in the next chapter).
    - Docker Secrets (for Docker Swarm, more secure than `.env` files).
    - Dedicated secrets management systems (e.g., HashiCorp Vault, AWS Secrets Manager, Azure Key Vault) for robust, large-scale deployments.
- **Resource Limits:** For production, it's critical to define CPU and memory limits for your containers using the `deploy.resources.limits` and `deploy.resources.reservations` keys in your `docker-compose.yml` (part of the Compose Specification). This prevents a single misbehaving service from consuming all host resources and impacting other services. We will cover this in a later chapter.
- **Logging:** While `docker compose logs` is useful for development, production systems require centralized logging. You would configure your services to send logs to a dedicated logging solution (e.g., ELK stack, Splunk, cloud logging services) for aggregation, analysis, and alerting.
- **Health Checks:** As noted, `depends_on` only guarantees startup order, not readiness. In production, implementing `healthcheck` configurations for your services is vital to ensure dependent services only connect when a service is truly ready to handle requests. This topic will be covered in a dedicated chapter.

## Common Issues & Solutions

1. **"Service 'db' failed to build" or "Error response from daemon: pull access denied"**:
    - **Issue:** Docker couldn't pull the `postgres:16-alpine` image, or there's a typo in the image name.
    - **Solution:** Double-check the `image` name and tag in your `docker-compose.yml`. Ensure your machine has an active internet connection and that Docker Hub is accessible. A simple `docker pull postgres:16-alpine` can help diagnose connectivity issues.
2. **Web app shows "Failed to connect to database"**:
    - **Issue:** The `web` service started before the `db` service was fully initialized and ready to accept connections, or the `DATABASE_URL` is incorrect.
    - **Solution:**
        - **Verify `DATABASE_URL`:** Double-check that the `DATABASE_URL` in `docker-compose.yml` for the `web` service correctly references the `db` service name, user, password, and database defined for PostgreSQL.
        - **Database Readiness:** This is the most common cause. While `depends_on` helps, it's not a readiness check. For now, try restarting just the `web` service after giving the `db` service a moment to fully start: `docker compose restart web`. In a future chapter, we will implement `healthcheck` configurations to properly handle service readiness.
        - **Check `db` logs:** Use `docker compose logs db` to ensure the PostgreSQL container started without errors and is listening on its port.

3. **Port conflict: "Bind for 0.0.0.0:8000 failed: port is already allocated"**:
    - **Issue:** Another process on your host machine is already using port `8000`. This could be a previously running container, another application, or even a system service.
    - **Solution:**
        - **Identify and stop the conflicting process:** On Linux/macOS, you can use `sudo lsof -i :8000` to find which process is using the port. Then, stop that process.
        - **Change host port mapping:** Modify the `ports` mapping in your `docker-compose.yml` for the `web` service to use a different host port (e.g., `"8001:8000"`). This maps host port `8001` to container port `8000`.

---

## Summary & Next Step

You have successfully orchestrated a multi-service application using Docker Compose! This chapter was a significant step in moving beyond single containers to managing a complete application stack.

What you've accomplished:

- **Defined a `docker-compose.yml` file:** This central configuration file now defines your `web` service (a Python Flask application) and a `db` service (PostgreSQL).
- **Configured internal networking:** Services communicate securely over a dedicated Docker network, enhancing isolation.
- **Mapped ports:** Your web application is accessible from your host machine.
- **Understood key concepts:** You now grasp how `services`, `networks`, `volumes`, `build`, `ports`, `depends_on`, and `environment` variables work together in Docker Compose.

You can now bring your entire application stack up and down with simple, consistent commands, greatly simplifying the management of your development environment.

In the next chapter, we will delve deeper into **managing persistent data with Docker volumes**. We'll explore named volumes in more detail, understand their lifecycle, and ensure that your critical database data is never lost, even when containers are recreated, updated, or moved. This is a crucial aspect of building production-ready applications with Docker.

---

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

---

## References


- Docker Documentation: [https://docs.docker.com/](https://docs.docker.com/)
- Compose Specification Versioning: [https://github.com/jamesatdocker/docker-docs/blob/main/compose/compose-file/compose-versioning.md](https://github.com/jamesatdocker/docker-docs/blob/main/compose/compose-file/compose-versioning.md)
- PostgreSQL Docker Official Image: [https://hub.docker.com/_/postgres](https://hub.docker.com/_/postgres)
- Flask Documentation: [https://flask.palletsprojects.com/](https://flask.palletsprojects.com/)
- Psycopg2 Documentation: [https://www.psycopg.org/docs/](https://www.psycopg.org/docs/)

CHAPTER 05

# Integrating a Database Service (PostgreSQL)

Modern applications demand robust data storage. In this chapter, we'll integrate a PostgreSQL database into our Docker Compose stack, transforming our simple web application into a dynamic system capable of storing and retrieving information persistently. By the end, you'll have a fully containerized, multi-service application with a reliable database backend, a cornerstone for any production system.

## Project Overview: Adding Persistent Data

Our overall project aims to build a production-ready multi-service application using Docker Compose. Until now, our web application has been stateless. This chapter introduces a stateful component: a PostgreSQL database. This allows our application to manage user accounts, store content, or maintain any dynamic state required for its functionality. We will focus on ensuring the database's data persists across container restarts and updates, a critical aspect for production environments.

## Tech Stack: PostgreSQL and Docker Volumes

- **PostgreSQL:** We've chosen PostgreSQL for its reputation as a powerful, open-source object-relational database system. It offers strong data integrity, advanced features, and high reliability, making it suitable for demanding applications. We'll use the official `postgres` Docker image, specifically `postgres:16-alpine` (checked 2026-05-22, PostgreSQL 16 is the latest stable release), which provides a minimal, secure base.
- **Docker Volumes:** Data persistence for databases in containers is achieved through Docker volumes. Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. They are completely managed by Docker, separate from the container's writable layer, and designed for efficient I/O.
- **Docker Networks:** We'll leverage Docker's internal networking to allow our web application to communicate securely with the database service without exposing the database port directly to the host machine or public internet.

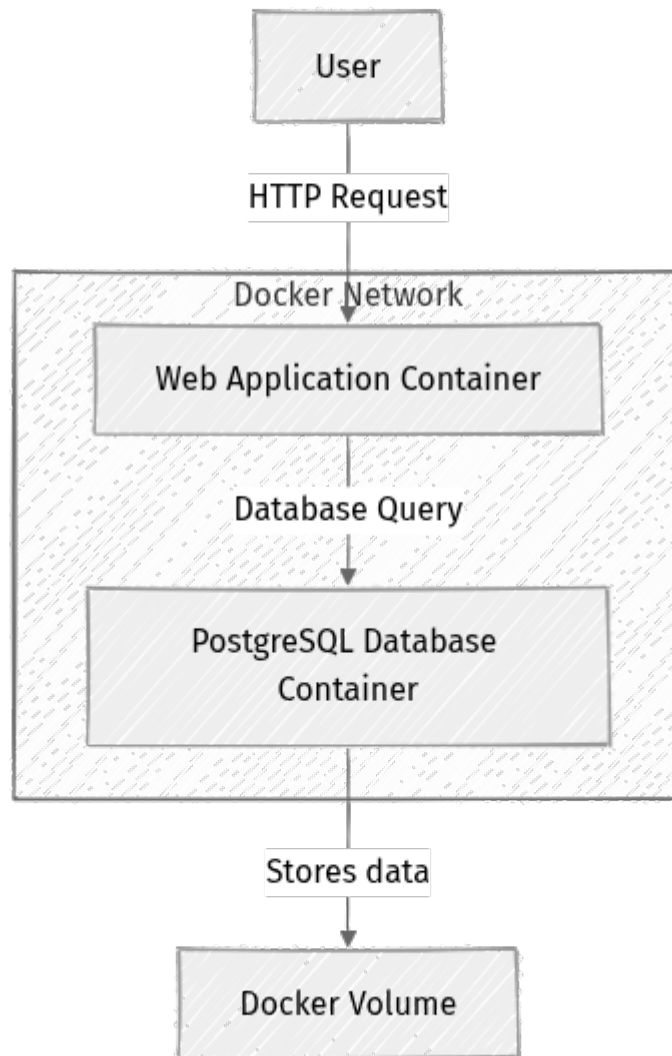
### Build Plan: Integrating the Database

To achieve a persistent and functional database integration, we'll follow these key steps:

1. **Define the PostgreSQL Service:** Add a new `db` service to our `docker-compose.yml`, specifying its image, environment variables, and network.
2. **Configure Data Persistence:** Attach a named Docker volume to the database service to ensure data is not lost.
3. **Establish Internal Networking:** Ensure both the web and database services are on the same custom Docker network for secure communication.
4. **Update Web Application Requirements:** Modify the web application's `Dockerfile` to include the necessary database client library (e.g., `psycopg2-binary` for Python).
5. **Connect from the Web Application:** Adjust the web application code to use environment variables to connect to the database.

### Architecture: Web App with Persistent Database

Our updated architecture introduces a dedicated database container, isolated but networked with our web application. All database files will reside on a Docker volume.




Key architectural decisions for this milestone:

- **Container Isolation:** Each service (web, database) runs in its own isolated container, promoting modularity and easier management.
- **Official Images:** Using the `postgres:16-alpine` official image ensures security, regular updates, and adherence to best practices. The Alpine variant significantly reduces image size, leading to faster pulls and a smaller attack surface.
- **Named Volumes:** Docker named volumes (`db_data`) provide a robust and Docker-managed way to persist database files. This decouples data from the container lifecycle.
- **Internal Network Communication:** The `app_network` ensures that `web` and `db` can communicate using service names as hostnames (e.g., `db` for the database container) within a private, isolated network. This enhances security by preventing direct external access to the database.

- **Environment-based Configuration:** Database credentials and connection strings are passed via environment variables, a standard practice for containerized applications.

### Step-by-Step Implementation

Let's modify our `docker-compose.yml` and web application to integrate PostgreSQL.

#### 1. Defining the PostgreSQL Service

Open your `docker-compose.yml` file and add a new service named `db`. We'll also update the `web` service to include a placeholder for the database connection string and declare its dependency on `db`.

```
# Path: docker-compose.yml
services:
  web:
    build: .
    ports:
      - "8000:8000"
    environment:
      # Placeholder for database connection string, will be updated
      DATABASE_URL: "postgresql://user:password@db:5432/mydatabase"
    depends_on:
      - db # Ensures 'db' service starts before 'web'
    networks:
      - app_network

  db:
    image: postgres:16-alpine # Use specific version and Alpine for smaller
size
    environment:
      POSTGRES_DB: mydatabase
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password # Temporary: will be secured later
    volumes:
      - db_data:/var/lib/postgresql/data # Mount a named volume for
persistence
    networks:
      - app_network

volumes:
  db_data: # Define the named volume for PostgreSQL data

networks:
  app_network: # Define the custom bridge network
    driver: bridge
```

**Explanation of additions and changes:**

- **`db:` service definition:**
    - `image: postgres:16-alpine`: Specifies the official PostgreSQL Docker image. The `alpine` tag indicates a minimal, lightweight base image, which is a production best practice for reduced attack surface and faster image pulls.
    - `environment:`: These variables are used by the `postgres` image to initialize the database:
        - `POSTGRES_DB`: The name of the database that will be created on startup.
        - `POSTGRES_USER`: The username for accessing the database.
        - `POSTGRES_PASSWORD`: The password for the database user.
        - ⚠️ **What can go wrong:** Directly hardcoding `POSTGRES_PASSWORD` in `docker-compose.yml` is **not recommended for production**. This is a temporary setup for functionality. We will implement Docker secrets for secure credential management in a future chapter.
    - `volumes: - db_data:/var/lib/postgresql/data`: This line is crucial for data persistence.
        - `db_data`: Refers to a named volume defined at the root of `docker-compose.yml`. Docker manages this volume, storing its data on the host system.
        - `/var/lib/postgresql/data`: This is the default directory inside the PostgreSQL container where it stores its database files. By mounting `db_data` to this path, all database changes are written to the persistent volume, ensuring data survives container recreation.
    - `networks: - app_network`: Connects the `db` service to our custom `app_network`. This allows `web` to resolve `db` by its service name.

- **`web:` service updates:**
    - `environment: DATABASE_URL: "postgresql://user:password@db:5432/mydatabase"`: A placeholder connection string. The `db` hostname will resolve to the PostgreSQL container's IP address within `app_network`.
    - `depends_on: - db`: This tells Docker Compose to start the `db` service before the `web` service.
        - 🧠 **Important:** `depends_on` only guarantees the start order of containers, not that the `db` service is fully ready to accept connections. We'll address robust service readiness checks in a later chapter.
    - `networks: - app_network`: Ensures the `web` service can communicate with `db`.
- **Root-level `volumes:` and `networks:` definitions:**
    - These top-level keys explicitly define the named volume (`db_data`) and the custom bridge network (`app_network`). This is the recommended way per the Compose Specification (checked 2026-05-22).
    - ⚡ **Quick Note:** The Compose Specification recommends not specifying a `version` field in `docker-compose.yml` files, as it's now implicitly handled by the specification itself.

### 2. Update Web Application Dockerfile (Python Example)

For your web application to connect to PostgreSQL, it needs a database driver. If you're using Python, this means installing `psycopg2-binary`.

```
# Path: Dockerfile (example for a Python application)
# ... previous stages (e.g., build stage for dependencies) ...

# Stage 2: Runtime image
FROM python:3.11-slim-bookworm AS runtime

# Set environment variables for non-root user and Python specifics
ENV PYTHONUNBUFFERED 1
ENV PYTHONDONTWRITEBYTECODE 1

WORKDIR /app

# Copy only necessary files from the build stage or directly
COPY --from=builder /app/requirements.txt /app/requirements.txt
COPY --from=builder /app/your_app.py /app/your_app.py
COPY --from=builder /app/config.py /app/config.py
# ... any other application files ...
```

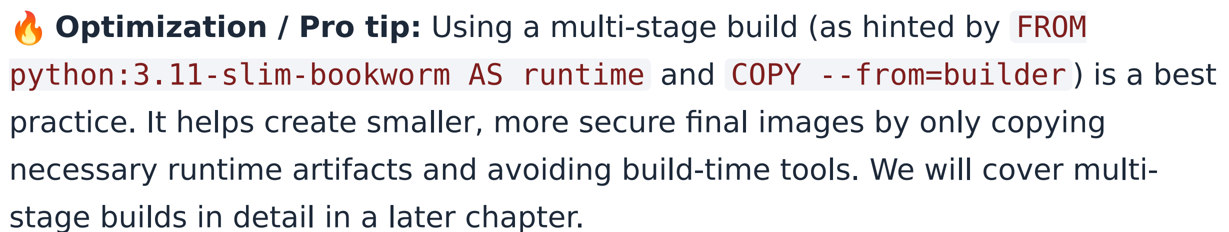
```
# Install runtime dependencies, including the PostgreSQL client
RUN pip install --no-cache-dir -r requirements.txt \
    # Add psycopg2-binary for PostgreSQL connectivity
    && pip install --no-cache-dir psycopg2-binary

# Create a non-root user for security best practices
RUN addgroup --system appgroup && adduser --system --ingroup appgroup appuser
USER appuser

EXPOSE 8000
CMD ["python", "your_app.py"]
```

Ensure your `requirements.txt` includes `psycopg2-binary`:

```
# Path: requirements.txt
# ... other dependencies ...
psycopg2-binary==2.9.9 # Checked 2026-05-22, latest stable version.
```



🔥 **Optimization / Pro tip:** Using a multi-stage build (as hinted by `FROM python:3.11-slim-bookworm AS runtime` and `COPY --from=builder`) is a best practice. It helps create smaller, more secure final images by only copying necessary runtime artifacts and avoiding build-time tools. We will cover multi-stage builds in detail in a later chapter.

If you are using a Node.js application, you would install the `pg` package via `npm install pg` in your `Dockerfile` and add it to `package.json`. The underlying principle remains: install the appropriate database connector for your language/framework.

### 3. Connecting from the Web Application

Your web application needs to read the `DATABASE_URL` environment variable and use it to establish a connection. Here's a minimal Python example:

```python
# Path: your_app.py (example snippet)
import os
import psycopg2 # Make sure this is installed via Dockerfile/requirements.txt

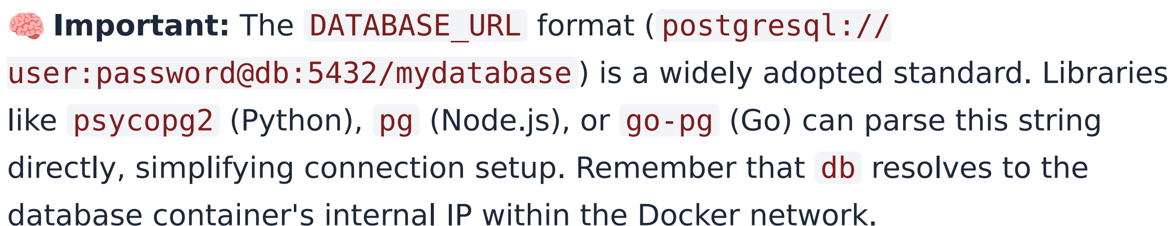
# ... (your existing web app code) ...

def get_db_connection():
    """Establishes a connection to the PostgreSQL database."""
    try:
        # psycopg2 can parse the standard DATABASE_URL format directly
        conn = psycopg2.connect(os.getenv("DATABASE_URL"))
        print("Database connection established successfully.")
        return conn
    except Exception as e:
        print(f"⚠️ Database connection failed: {e}")
        # In a real production application, you'd log this error
        # with a proper logging framework and potentially implement
        # retry logic or graceful degradation.
```

```
        return None

# Example usage (e.g., in an application startup routine or API endpoint)
if __name__ == "__main__":
    conn = get_db_connection()
    if conn:
        try:
            with conn.cursor() as cur:
                cur.execute("SELECT 1") # A simple query to test connectivity
                result = cur.fetchone()
                print(f"Database test query result: {result}")
            conn.close()
        except Exception as e:
            print(f"Error during database query: {e}")
    else:
        print("Could not connect to database, skipping test query.")

# ... (rest of your web application logic) ...
```



🧠 **Important:** The `DATABASE_URL` format (`postgresql://user:password@db:5432/mydatabase`) is a widely adopted standard. Libraries like `psycopg2` (Python), `pg` (Node.js), or `go-pg` (Go) can parse this string directly, simplifying connection setup. Remember that `db` resolves to the database container's internal IP within the Docker network.

### Testing & Verification

With the `docker-compose.yml` and application code updated, let's verify that PostgreSQL is running correctly and our web application can connect to it.

1. **Build and Run Services:** Navigate to your project root (where `docker-compose.yml` is located) and execute the following command:

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

```
- `--build`: Crucial here, as it rebuilds your `web` service image to include
the `psycopg2-binary` dependency from your `Dockerfile` changes.
- `-d`: Runs the services in detached mode, allowing them to run in the
background.
```

1. **Check Service Status:** Confirm that both your `web` and `db` containers are running:

```
docker compose ps
```

```
You should see output indicating both services are `running` and healthy (if
health checks were already configured, which we'll do later):
```

```
    NAME                IMAGE               COMMAND
SERVICE             STATUS              PORTS
    myproject-db-1      postgres:16-alpine  "docker-entrypoint.s…"
db                  running             5432/tcp
    myproject-web-1     myproject-web       "python your_app.py"
web                 running             0.0.0.0:8000->8000/tcp
```

1. **Inspect Database Logs:** Review the PostgreSQL container logs to ensure it started without errors and is ready to accept connections:

```
docker compose logs db
```

```
Look for a line similar to `database system is ready to accept connections`
near the end of the startup sequence.
```

1. **Connect to PostgreSQL from Host (for advanced verification):** You can use `docker compose exec` to run the `psql` client directly inside the database container. This is excellent for verifying connectivity and inspecting the database state.

```
docker compose exec db psql -U user -d mydatabase
```

```
- `docker compose exec db`: Executes a command within the `db` service
container.
- `psql -U user -d mydatabase`: Invokes the PostgreSQL command-line client,
connecting as `user` to `mydatabase`.

If successful, you'll be dropped into a `psql` prompt:
```

```
    psql (16.2)
    Type "help" for help.

    mydatabase=# \dt
    No relations found. # Expected, as we haven't created any tables yet.
    mydatabase=# \q
```

```
Type `\q` to exit `psql`. This confirms the database container is healthy and
accessible.
```

1. **Verify Web App Connectivity:** If you included the connection test snippet in your `your_app.py`, check the logs of your web service:

```
docker compose logs web
```

```
You should see `Database connection established successfully.` and `Database
test query result: (1,)` (or similar output from your test query) indicating
that the web application successfully connected to PostgreSQL. If you see
connection errors, proceed to the "Common Issues" section.
```

### Production Considerations

Integrating a database introduces critical production concerns beyond just getting it to run.

- **Security of Credentials:** Hardcoding `POSTGRES_PASSWORD` is a significant security risk. For production, leverage Docker secrets or a dedicated secret management solution (e.g., HashiCorp Vault, AWS Secrets Manager) to inject credentials securely at runtime. We will cover Docker secrets in the next chapter.
- **Data Backups and Recovery:** Docker volumes ensure data persistence, but they are not a backup solution. A comprehensive backup strategy is essential. This might involve:
  - Regularly snapshotting the `db_data` volume.
  - Using `pg_dump` to create logical backups that are stored externally.
  - Implementing point-in-time recovery with continuous archiving of WAL (Write-Ahead Log) files.
- **Resource Limits:** Databases can be resource-intensive. In production, always define CPU and memory limits for your database container to prevent it from exhausting host resources, which could destabilize other services. You can add a `deploy` section to your `db` service:

```
# Path: docker-compose.yml (snippet)
db:
  image: postgres:16-alpine
  environment:
    POSTGRES_DB: mydatabase
    POSTGRES_USER: user
    POSTGRES_PASSWORD: password
```

```
    volumes:
      - db_data:/var/lib/postgresql/data
    networks:
      - app_network
    deploy: # Define resource constraints for production
      resources:
        limits:
          cpus: '1.0' # Max 1 CPU core
          memory: 2GB # Max 2GB RAM
        reservations:
          cpus: '0.5' # Reserve 0.5 CPU core
          memory: 1GB # Reserve 1GB RAM
```

```
`cpus` are specified as a fraction of a CPU core (e.g., `0.5` is half a core).
`memory` can use suffixes like `B`, `K`, `M`, `G`. These values should be
tuned based on your application's actual load and database size.
```

- **Database Migrations:** As your application evolves, its database schema will change. Integrate a database migration tool (e.g., Alembic for Python, Flyway for Java, Knex.js for Node.js) into your deployment pipeline. This ensures schema changes are applied consistently and safely across environments.
- **Monitoring and Alerting:** Set up comprehensive monitoring for your database (CPU usage, memory consumption, disk I/O, active connections, query performance, error rates). Implement alerts for critical thresholds or anomalies to ensure operational visibility and proactive issue resolution.
- **Connection Pooling:** For high-traffic applications, consider using a connection pooler (like PgBouncer) to manage database connections efficiently, reducing the overhead on the PostgreSQL server.

### Common Issues & Solutions

1. **`web` service fails to connect to `db` with "Connection refused" or "Host not found":**

    - **Issue:** The web application started before PostgreSQL was fully initialized and ready to accept connections, or there's a networking/hostname issue.
    - **Solution:** While `depends_on` ensures startup order, it doesn't guarantee readiness.
        - **Immediate Fix:** Restart the `web` service after `db` is confirmed running: `docker compose restart web`. Or restart the whole stack: `docker compose down && docker compose up --build -d`.
        - **Long-term Fix:** Implement robust health checks (covered in a later chapter) to ensure the `web` service waits until `db` is truly healthy.
    - **Check:**
        - Verify the `DATABASE_URL` in your `web` service's `environment` section uses `db` as the hostname and the correct port (`5432`).
        - Ensure both `web` and `db` services are on the same `app_network`.

2. **Database data is lost after `docker compose down`:**

    - **Issue:** The named volume `db_data` was removed. This typically happens if `docker compose down --volumes` was used, or the volume was never correctly configured.
    - **Solution:** Ensure `volumes: - db_data:/var/lib/postgresql/data` is correctly configured for the `db` service, and `db_data:` is defined at the root level of `docker-compose.yml`. To preserve data, always use `docker compose down` without the `--volumes` flag.
    - **Check:** Run `docker volume ls` to verify that your `myproject_db_data` volume exists and is managed by Docker.

3. **`psql` command not found inside container during `docker compose exec`:**
    - **Issue:** While unlikely with the official `postgres` image, if you were using a different, highly minimal image, the `psql` client might not be installed.
    - **Solution:** The `postgres` official image does include `psql`. Double-check the `docker compose exec db psql -U user -d mydatabase` command for typos. If you were truly using a custom minimal image, you'd need to add `postgresql-client` to its `Dockerfile` explicitly.

### Summary & Next Step

You have successfully integrated a PostgreSQL database service into your Docker Compose application stack, laying the foundation for a truly dynamic application. You've mastered:

- Defining a database service using the official `postgres:16-alpine` Docker image.
- Ensuring critical data persistence with Docker named volumes.
- Configuring the database via environment variables.
- Enabling secure, internal communication between services using a custom Docker network.
- Verifying the database's operational status and connectivity from your web application.

This milestone significantly enhances our application's capabilities, making it capable of storing and managing persistent state. The database is now ready to support your application's data needs reliably.

In the next chapter, we will address the crucial aspect of security by moving beyond hardcoded values to manage sensitive information using environment variables and Docker secrets, making your application more robust and production-ready.

---

## References


- Docker Documentation: [https://docs.docker.com/](https://docs.docker.com/)
- Compose Specification: [https://docs.docker.com/compose/compose-file/](https://docs.docker.com/compose/compose-file/)
- PostgreSQL Docker Official Image: [https://hub.docker.com/_/postgres](https://hub.docker.com/_/postgres)
- PostgreSQL Official Website: [https://www.postgresql.org/](https://www.postgresql.org/)
- `psycopg2-binary` PyPI: [https://pypi.org/project/psycopg2-binary/](https://pypi.org/project/psycopg2-binary/)


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

CHAPTER 06

# Establishing Secure Inter-Service Networking

In a multi-service application, the way your components communicate is as critical as what they do. This chapter focuses on establishing secure and isolated networking for our Docker Compose stack. We'll move beyond Docker's default networking to create a dedicated network for our services, enhancing both security and clarity.

By the end of this milestone, our web application and database will communicate over a private, isolated network managed by Docker Compose. This ensures that only authorized services within our stack can reach each other, laying a robust foundation for a production-ready deployment.

---

## Project Overview

Our overarching project goal is to build a production-ready multi-service web application stack using Docker and Docker Compose. This involves containerizing a simple web application, integrating it with a database, and applying best practices for deployment, security, and maintainability.

In this chapter, we specifically address the crucial aspect of inter-service communication. We are moving from implicit, default networking to an explicitly defined, isolated network to secure the communication pathways between our `web` and `db` services. This is a fundamental security and architectural pattern for any real-world containerized application.

---

## Tech Stack

For this chapter, we continue to leverage:

- **Docker Engine**: The core containerization platform. (Version unknown, checked 2026-05-22. We recommend using the latest stable release available for your OS).
- **Docker Compose**: Used to define and run our multi-container Docker application. We adhere to the **Compose Specification**, which recommends omitting the explicit `version` field in `docker-compose.yml` files (checked 2026-05-22).

- **YAML**: The language for defining our `docker-compose.yml` file.

---

## Milestones for Secure Networking

To achieve our goal of isolated inter-service communication, we will follow these steps:

1. **Understand Docker Networking Defaults**: Briefly review how Docker Compose handles networking by default.
2. **Define a Custom Bridge Network**: Add a `networks` section to `docker-compose.yml` to create a dedicated network.
3. **Attach Services to the Network**: Explicitly connect our `web` and `db` services to this new custom network.
4. **Update Service Discovery**: Configure the `web` service to use the database service name for internal resolution.
5. **Verify Network Configuration**: Confirm that the network is correctly created and services are attached and communicating.

---

## Planning & Design: Isolated Service Communication

When you run multiple Docker containers that need to communicate, Docker provides robust networking capabilities. By default, Docker Compose places all services in a single, default network. While functional for simple setups, this default network can become less manageable as your application grows, and it doesn't explicitly segment traffic.

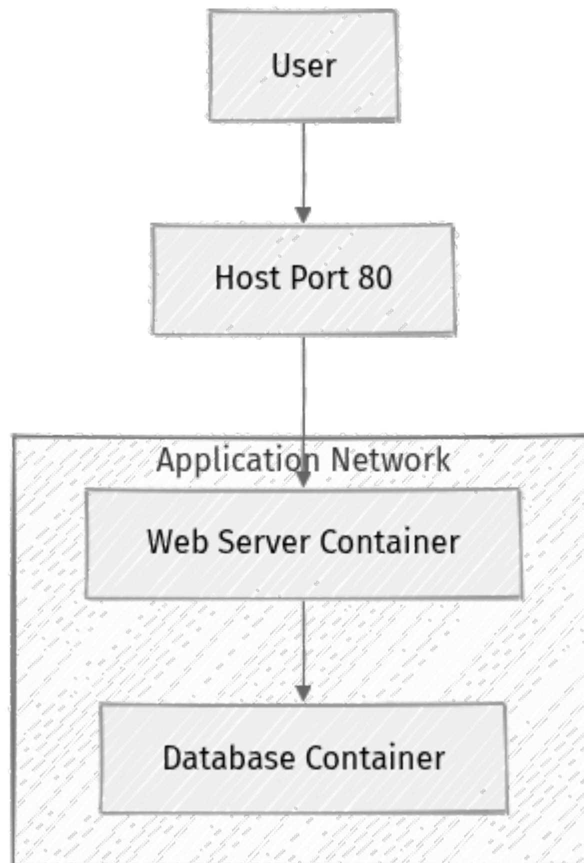
For production systems, it's a best practice to define custom networks. This allows you to:

- **Isolate Services**: Only services explicitly attached to a network can communicate over it. This prevents unintended communication paths and reduces the attack surface.
- **Improve Name Resolution**: Docker's internal DNS allows services to resolve each other by their service names (e.g., `web` can reach `db` by simply using `db` as the hostname). This is more reliable than relying on dynamic IP addresses.
- **Enhance Security**: By default, custom bridge networks are isolated from the host's network unless specific ports are published. This minimizes external exposure.

Our goal is to define a custom bridge network within our `docker-compose.yml` file and explicitly attach our `web` and `db` services to it.

### Network Architecture Overview

Our application's communication flow will now look like this, with a dedicated, isolated network layer:




In this refined setup:

- `User` access still comes through a mapped host port (e.g., `80`) to the `WebServer`. This is the only entry point from outside the Docker host.
- The `WebServer` and `Database` containers are now explicitly connected to `App_Network`.
- Communication between `WebServer` and `Database` happens entirely within this isolated `App_Network`. This traffic never leaves the Docker host's internal network interface, enhancing security.

---

## Step-by-Step Implementation

We will modify our existing `docker-compose.yml` file to define a custom network and attach our services.

### 1. Define the Custom Network

Open your `docker-compose.yml` file (from the previous chapter). We will add a new top-level `networks` section.

Add the following to your `docker-compose.yml` file, typically at the end of the file, alongside the `services` and `volumes` (if any) sections:

```
# docker-compose.yml
# ... (existing services and volumes sections)

networks:
  app_network:
    driver: bridge
```

#### Explanation:

- `networks:`: This is a top-level key in `docker-compose.yml` where you declare custom networks for your application.
- `app_network:`: This is the name we've chosen for our custom network. It should be descriptive and unique within your Compose file.
- `driver: bridge`: Specifies that Docker should create a standard bridge network. This is the most common type for single-host multi-container applications. It provides network isolation and internal DNS resolution, allowing containers on the same bridge network to communicate by their service names.

### 2. Attach Services to the Network

Now that we've defined `app_network`, we need to instruct our `web` and `db` services to connect to it. We do this by adding a `networks` key under each service definition.

Modify your `docker-compose.yml` to include these changes within the `web` and `db` service definitions:

```
# docker-compose.yml
services:
  web:
    build: .
    ports:
      - "80:80"
    environment:
      # ... (existing environment variables)
      DATABASE_HOST: db # Crucial: Use the service name for internal
resolution
    networks:
      - app_network # Attach the web service to our custom network
```

```
  db:
    image: postgres:16-alpine # Using a specific, lightweight version as of
2026-05-22
    environment:
      POSTGRES_DB: mydatabase
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    networks:
      - app_network # Attach the db service to our custom network
    # ... (existing volumes if any)

networks:
  app_network:
    driver: bridge
```

**Explanation:**

- `networks:` : Under each service, this key lists the networks the service should connect to. A service can connect to multiple networks if needed, but for our current setup, one is sufficient.
- `- app_network`: This line explicitly attaches both the `web` and `db` services to our `app_network`. This is a list item, so remember the hyphen.
- `DATABASE_HOST: db`: This is a critical change. Because both services are now on `app_network`, Docker's internal DNS will automatically resolve the service name `db` to the correct IP address of the `db` container within that network. This makes our application configuration robust and independent of dynamically assigned IP addresses.

## 3. Review the Complete `docker-compose.yml`

Your complete `docker-compose.yml` should now reflect these changes, looking similar to this:

```
# docker-compose.yml
# As of 2026-05-22, the Compose Specification recommends omitting the
'version' field.
# This allows Compose to use the latest specification.
# Reference: https://github.com/jamesatdocker/docker-docs/blob/main/compose/
compose-file/compose-versioning.md

services:
  web:
    build: .
    ports:
      - "80:80"
    environment:
      APP_ENV: production
      DATABASE_HOST: db # Use the service name for internal resolution
      DATABASE_PORT: 5432 # Default PostgreSQL port
    networks:
      - app_network
```

```
  db:
    image: postgres:16-alpine # Using a specific, lightweight version as of 
2026-05-22
    environment:
      POSTGRES_DB: mydatabase
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    networks:
      - app_network
    # volumes:
    #   - db_data:/var/lib/postgresql/data # Uncomment if you set up volumes 
previously

# volumes:
#   db_data: # Uncomment if you set up volumes previously

networks:
  app_network:
    driver: bridge
```

⚡ **Quick Note:** The Docker Engine version is unknown as of 2026-05-22. For Docker Compose (Compose Specification), the best practice is to omit the `version` field from `docker-compose.yml` files. This ensures you're using the latest specification without needing to update a version number manually.

---

## Testing & Verification

After modifying the `docker-compose.yml`, it's vital to verify that the network is created correctly and services are communicating as expected.

### 1. Rebuild and Start Services

First, stop any running containers from previous steps to ensure a clean slate, then bring up the new stack with the updated network configuration.

```
docker compose down --volumes --remove-orphans
docker compose up -d
```

- `docker compose down --volumes --remove-orphans`: This command stops and removes containers, networks, and optionally volumes. The `--volumes` flag is important if you made changes that affect volumes (though not strictly necessary for just network changes), and `--remove-orphans` removes services that are no longer defined in the Compose file.
- `docker compose up -d`: This command starts the services in detached mode (`-d`), creating the new `app_network` and attaching the specified services.

### 2. Verify Network Creation

Check if the `app_network` has been created by Docker.

```
docker network ls
```

You should see an entry similar to `yourprojectname_app_network` in the output. Docker Compose prefixes network names with the project directory name by default (e.g., if your project folder is `my-app`, the network might be `my-app_app_network`).

### 3. Inspect the Network

To see precisely which containers are attached to the network, use `docker network inspect`. Replace `yourprojectname_app_network` with the actual network name identified in the previous step.

```
docker network inspect yourprojectname_app_network
```

In the output, under the `Containers` section, you should see entries for both your `web` and `db` services, along with their assigned IP addresses within that network. This confirms they are correctly connected.

### 4. Test Inter-Service Communication

You can test if the `web` service can resolve and communicate with the `db` service using the `ping` command.

```
docker compose exec web ping db
```

You should see successful `ping` responses from the `db` container. This confirms that Docker's internal DNS resolution is working correctly within `app_network`, and the `web` service can reach `db` by its service name.

If your `web` application connects to the database during startup, you should also check its logs for connection success:

```
docker compose logs web
```

Look for successful database connection messages and crucially, no errors related to `DATABASE_HOST` or connection refused.

### 5. Access the Application

Finally, ensure your web application is still accessible and fully functional via your browser at `<http://localhost>`. If your application has a database interaction (e.g., retrieving data or displaying a list from the database), verify that this functionality works correctly.

---

## Production Considerations

- **Network Segmentation**: For larger, more complex applications with many services, consider creating multiple custom networks. For instance, a `frontend_network` for web servers and load balancers, and a `backend_network` for application servers and databases. This further limits the blast radius if one segment is compromised and improves overall network security. `🧠 Important:` Granular network segmentation is a key security practice, especially in microservices architectures.
- **Ingress/Egress Control**: By default, custom bridge networks are very secure. The `ports` mapping in `docker-compose.yml` is the only way to expose a service to the host or external network. Be extremely mindful of which ports you expose and why. Never expose database ports (like `5432` for PostgreSQL) directly to the host or public network unless absolutely necessary and secured with strict firewall rules and authentication.
- **Host Firewall Rules**: While Docker manages internal container networking, ensure your host's firewall (e.g., `ufw` on Linux, `Windows Defender Firewall`) allows traffic to the host ports mapped by Docker Compose (e.g., port `80` for the web service).
- **Service Discovery**: Docker's internal DNS is excellent for service discovery within a Compose stack. Always use service names (like `db`) for inter-service communication within the `docker-compose.yml` file and your application code. Avoid hardcoding IP addresses, as they are ephemeral and can change.

## Common Issues & Solutions

1. **Services cannot communicate (e.g., `web` can't connect to `db`)**:
    - **Cause**: The most common reasons are that services are not on the same network, or the `DATABASE_HOST` (or similar environment variable) in the consuming service is incorrect.
    - **Solution**:
        - Double-check `docker-compose.yml` to ensure both services explicitly list `app_network` under their `networks` key with correct indentation.
        - Verify that environment variables in the consuming service (e.g., `web`) use the service name (e.g., `db`) and not `localhost`, `127.0.0.1`, or an IP address.
        - Use `docker network inspect yourprojectname_app_network` to confirm both containers are listed.
        - Use `docker compose exec [service_name] ping [target_service_name]` to test basic connectivity and DNS resolution.
2. **`docker network ls` does not show the network after `up`**:
    - **Cause**: `docker compose up` was not run after modifying the `docker-compose.yml`, or there's a YAML syntax error in the `networks` section preventing Compose from parsing it correctly.
    - **Solution**:
        - Ensure you ran `docker compose down` followed by `docker compose up -d`.
        - Carefully review your `docker-compose.yml` for YAML syntax errors (e.g., incorrect indentation, missing colons). Even a single space can cause issues.

3. **Application fails to start with network-related errors (e.g., "connection refused")**:
    - **Cause**: While defining networks helps with connectivity, it doesn't solve startup order issues. A service might try to connect to a dependency (like a database) before the dependency is fully ready and listening for connections.
    - **Solution**: For now, check container logs for specific "connection refused" messages. We will address robust service startup order and readiness checks using Docker Compose health checks in a future chapter. For immediate debugging, a simple `sleep` command in your `web` service's entrypoint or a retry logic in your application code can temporarily mitigate this.

---

## Summary & Next Step

You've successfully established a secure and isolated network for your multi-service Docker Compose application. Your `web` and `db` services now communicate over a private bridge network, using Docker's internal DNS for seamless service discovery. This is a critical step towards a robust and production-ready architecture, significantly enhancing both security and maintainability by isolating internal traffic.

Next, we'll dive into managing persistent data with Docker volumes, ensuring that our database's data survives container restarts, upgrades, and migrations, a non-negotiable requirement for any stateful production application.

---

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

## References


- Docker Documentation - Networking overview: https://docs.docker.com/network/
- Docker Compose - Define networks: https://docs.docker.com/compose/compose-file/06-networks/
- Compose Specification Versioning: https://github.com/jamesatdocker/docker-docs/blob/main/compose/compose-file/compose-versioning.md
- PostgreSQL Docker Official Image: https://hub.docker.com/_/postgres
- Docker CLI `network inspect` command: https://docs.docker.com/engine/reference/commandline/network_inspect/
- Docker CLI `compose exec` command: https://docs.docker.com/engine/reference/commandline/compose_exec/ +++

CHAPTER 07

# Handling Configuration and Secrets Securely

Managing application configuration and sensitive data is a critical aspect of building production-ready applications. Hardcoding API keys, database credentials, or other environment-specific settings directly into your code or Dockerfiles is a significant security risk and a maintenance nightmare. In this chapter, we'll learn how to separate configuration from code and handle sensitive information (secrets) securely within our Docker Compose stack.

By the end of this milestone, your multi-service application will properly load non-sensitive configuration from `.env` files and securely consume sensitive secrets using Docker's built-in secrets management. This significantly improves the security posture and maintainability of your deployment.

---

## Project Overview

This chapter focuses on a critical aspect of application hardening: secure configuration management. We're enhancing our existing multi-service web application (Flask + PostgreSQL) to handle its settings and sensitive data in a production-minded way.

- **Finished Artifact:** A Docker Compose deployed web application and database, where all environment-specific settings are externalized. Non-sensitive settings are loaded via `.env` files, and sensitive credentials are provided through Docker Secrets.
- **Target User:** Any developer or operations engineer working with Dockerized applications who needs to deploy them securely and maintainably across different environments.
- **Success Criteria:** The application successfully starts, connects to the database, and serves requests. All environment variables are correctly populated, and sensitive data is demonstrably consumed from Docker Secrets files, not exposed directly in environment variables or application code.

## Tech Stack

For this chapter, we'll continue working with our established stack, focusing on how its components interact with configuration and secrets:

- **Docker Engine:** The underlying runtime for containers. (Version: Latest stable as of 2026-05-22, specific version unknown but assumed current).
- **Docker Compose:** Used to define and run our multi-container application. (Version: Adheres to the Compose Specification, recommended approach without an explicit `version` field, as of 2026-05-22).
- **Python 3.11:** The runtime for our Flask web application.
- **Flask:** The web framework for our application service.
- **PostgreSQL 16-alpine:** The database service. (Version: `16-alpine`, latest stable as of 2026-05-22).

## Milestones and Build Plan

This chapter is structured to incrementally build our secure configuration setup:

1. **Externalize Non-Sensitive Configuration:** Move general application settings into a `.env` file and configure Docker Compose to inject these as environment variables.
2. **Introduce Docker Secrets:** Create secure files for sensitive data (database password, application secret key) and define them as Docker Secrets in `docker-compose.yml`.
3. **Update Application Code:** Modify the Flask application to read non-sensitive settings from environment variables and sensitive secrets from the special files mounted by Docker Secrets.

## Architecture: Configuration and Secrets Flow

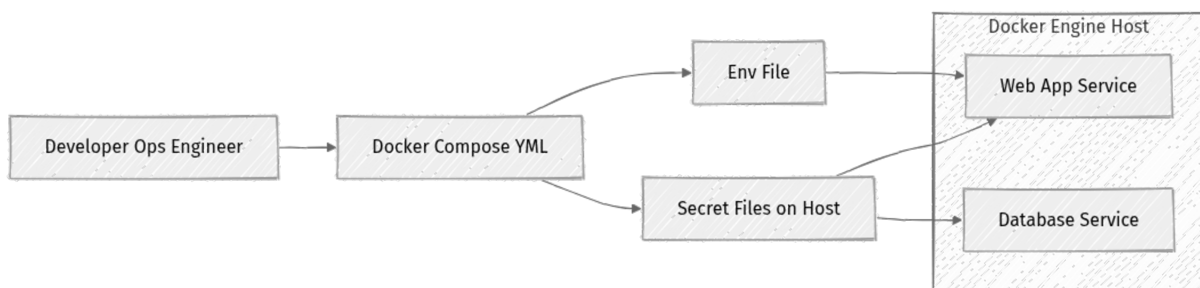
The core principle is the 12-Factor App methodology, which advocates for strictly separating configuration from code. Configuration should be injected into the application environment at runtime, rather than being part of the build artifact. This allows the same application image to be deployed across different environments (development, staging, production) with different configurations.

We'll address two distinct types of configuration:

1. **Non-Sensitive Configuration:** Settings like port numbers, API endpoints, or feature flags that don't pose a security risk if exposed. These are ideal for `.env` files, which are simple and widely understood for local development.
2. **Sensitive Configuration (Secrets):** Database passwords, API keys, encryption keys, or other credentials that must be protected. For these, we will use Docker Secrets, which provide a more robust and secure mechanism for passing sensitive data to containers.

Our `web` application service will be updated to read non-sensitive configuration from environment variables and sensitive secrets (like its own secret key and the database password) from designated files within the container's filesystem. The `db` service will also consume its root password as a secret file.

Here's a high-level view of how configuration and secrets will flow into our services:




- **`EnvFile` (non-sensitive):** Provides general configuration to services via environment variables.
- **`SecretFiles` (sensitive):** Provides sensitive data to services by mounting them as files within the container's `/run/secrets` directory.
- **`DockerCompose`:** Orchestrates how these configuration sources are injected into `WebService` and `DbService`.

---

## Step-by-Step Implementation

Let's assume we have a simple Flask web application (from previous chapters) that needs a database connection string, a secret key, and the database password to connect.

### 1. Externalizing Non-Sensitive Configuration with `.env`

First, we'll create an `.env` file at the root of our project, alongside `docker-compose.yml`. This file will hold non-sensitive environment variables. Notice that the `DATABASE_URL` here no longer contains the password, as that will be handled as a secret.

**Action:** Create or update `./.env` in your project root:

`./.env````text APP_PORT=5000 FLASK_ENV=development DATABASE_HOST=db DATABASE_PORT=5432 DATABASE_NAME=mydatabase DATABASE_USER=user

```
- `APP_PORT`: Defines the port our Flask app will listen on.
- `FLASK_ENV`: Sets the Flask environment mode.
- `DATABASE_HOST`, `DATABASE_PORT`, `DATABASE_NAME`, `DATABASE_USER`: These
provide the non-sensitive connection details for our PostgreSQL database.

Next, modify your `docker-compose.yml` to instruct Docker Compose to load
these variables into the `web` service and utilize them for the `db` service.

**Action:** Update your `docker-compose.yml`:

`./docker-compose.yml````yaml
# docker-compose.yml
# This file adheres to the Compose Specification.
# For more information, see: https://docs.docker.com/compose/compose-file/
services:
  web:
    build: .
    ports:
      - "${APP_PORT}:${APP_PORT}" # Use APP_PORT from .env
    env_file: # Load environment variables from ./.env
      - ./.env
    # We will add secrets here shortly
    # ... other configurations for web service ...
  db:
    image: postgres:16-alpine # Latest stable as of 2026-05-22
    environment:
      POSTGRES_DB: ${DATABASE_NAME} # From .env
      POSTGRES_USER: ${DATABASE_USER} # From .env
      # POSTGRES_PASSWORD is now provided via a secret
    volumes:
      - db_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${DATABASE_USER} -d $
{DATABASE_NAME}"]
      interval: 10s
      timeout: 5s
      retries: 5
    # We will add secrets here shortly
    # ... other configurations for db service ...
```

```
volumes:
  db_data:
```

- `ports: - "${APP_PORT}:${APP_PORT}"`: This line now uses variable interpolation. Docker Compose will substitute `${APP_PORT}` with the value found in `/.env` (which is `5000`). This makes the port easily configurable without modifying the `docker-compose.yml` file itself.
- `env_file: - ./.env`: This directive is crucial. It tells Docker Compose to read all key-value pairs from the specified `.env` file and expose them as environment variables to the `web` service's containers. This is a common and convenient way to manage non-sensitive configuration for local development.
- `environment` variables in the `db` service: `${DATABASE_NAME}` and `${DATABASE_USER}` are also populated from the `.env` file. This demonstrates how a single `.env` file can centralize configuration for multiple services, improving consistency.

## 2. Managing Sensitive Secrets with Docker Secrets

Now, let's secure the database password for both the `db` service and the `web` service, as well as the `web` application's own secret key. Docker Secrets offer a more secure way to handle sensitive data than environment variables, especially in multi-service deployments.

First, create files containing your sensitive secrets. It's best practice to keep these files out of version control (e.g., add them to `.gitignore`). For this example, we'll create simple text files.

**Action:** Create new files in your project root:

`./db_password.txt` (for the PostgreSQL database itself)```text
my_secure_db_password_123

```
`./web_db_password.txt` (for the web application to connect to the
database)```text
my_secure_db_password_123
```

`./app_secret_key.txt` (for the Flask web application's secret key)```text
this_is_a_very_secret_key_for_flask_app

```
- **`db_password.txt`**: This file contains the password that the PostgreSQL
database will use for its `postgres` superuser.
- **`web_db_password.txt`**: This file contains the password that our `web`
application will use to connect to the PostgreSQL database. While it's the
same value as `db_password.txt` in this example, in a real application, you
```

````
might use different credentials for different services for a "least privilege"
approach.
- **`app_secret_key.txt`**: This is a secret key specific to our Flask
application, used for session management and security features.

**🧠 Important:** In a real production scenario, these files would contain
strong, randomly generated passwords/keys and would be managed by a dedicated
secret management system or manually placed on the host, *never* committed to
version control. Add these files to your `.gitignore` immediately to prevent
accidental exposure.

Next, we'll modify `docker-compose.yml` to define these as secrets at the top
level and then make them available to the appropriate services.

**Action:** Update your `docker-compose.yml` with the `secrets` blocks:

`./docker-compose.yml````yaml
# docker-compose.yml
# This file adheres to the Compose Specification.
services:
  web:
    build: .
    ports:
      - "${APP_PORT}:${APP_PORT}"
    env_file:
      - ./.env
    secrets: # Declare that the web service consumes secrets
      - app_secret_key
      - web_db_password # Web app's password for the DB
    # ...
  db:
    image: postgres:16-alpine
    environment:
      POSTGRES_DB: ${DATABASE_NAME}
      POSTGRES_USER: ${DATABASE_USER}
      # POSTGRES_PASSWORD is removed. We use POSTGRES_PASSWORD_FILE instead.
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password # Tell Postgres to read
from the secret file
    volumes:
      - db_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${DATABASE_USER} -d $
{DATABASE_NAME}"]
      interval: 10s
      timeout: 5s
      retries: 5
    secrets: # Declare that the db service consumes a secret
      - db_password # This refers to the secret defined in the top-level
'secrets' section
    # ...

volumes:
  db_data:

secrets: # Top-level declaration of secrets
  db_password: # The name of the secret for the DB's root password
    file: ./db_password.txt # Path to the file containing the secret's value
  web_db_password: # The name of the secret for the web app's DB connection
    file: ./web_db_password.txt
````

```
  app_secret_key: # Another secret for the web app
    file: ./app_secret_key.txt
```

- **Top-level `secrets` block:** This new block defines all secrets available to our Compose application.
  - `db_password:`: This is a named secret. `file: ./db_password.txt` tells Docker Compose to read the content of `db_password.txt` from the host and securely make it available as a secret.
  - `web_db_password:`: Similarly, this secret is sourced from `web_db_password.txt`.
  - `app_secret_key:`: This secret is sourced from `app_secret_key.txt`.
- **Service-level `secrets` block:**
  - For the `db` service: `secrets: - db_password` makes the `db_password` secret available to the `db` container. Docker Compose mounts this secret as a read-only file at `/run/secrets/db_password` inside the `db` container.
  - `POSTGRES_PASSWORD_FILE: /run/secrets/db_password`: This PostgreSQL-specific environment variable instructs the `postgres` image to read the database password from the specified file path, rather than from a `POSTGRES_PASSWORD` environment variable. This is a more secure way to provide passwords to PostgreSQL containers.
  - For the `web` service: `secrets: - app_secret_key` and `secrets: - web_db_password` make these secrets available. They will be mounted as files at `/run/secrets/app_secret_key` and `/run/secrets/web_db_password` respectively inside the `web` container.

### 3. Updating the Application Code

Now, let's update our `web` service's `Dockerfile` and `app.py` to consume these secrets and environment variables.

**Action:** Ensure your `Dockerfile` for the `web` service is up-to-date.

`./Dockerfile````dockerfile

# Use a minimal base image for the web application

FROM python:3.11-slim-bookworm AS builder

# Set environment variables for the builder stage

ENV PYTHONUNBUFFERED 1

# Install build dependencies

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

# Final stage for the production image

FROM python:3.11-slim-bookworm

# Set environment variables for the runtime stage

ENV PYTHONUNBUFFERED 1 ENV FLASK_APP=app.py

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

EXPOSE 5000

# Run the application

CMD ["flask", "run", "--host=0.0.0.0", "--port", "5000"]

```
- This `Dockerfile` uses a multi-stage build, first installing dependencies in
a `builder` stage and then copying only the necessary runtime components to a
smaller final image. This keeps our image size down and reduces the attack
surface.

**Action:** Ensure your `requirements.txt` includes the PostgreSQL adapter.

`./requirements.txt````text
Flask
psycopg2-binary # For PostgreSQL connection
```

**Action:** Update your `app.py` to read from environment variables and secret files.

`./app.py````python import os from flask import Flask, jsonify import psycopg2

app = Flask(**name**)

# --- Load non-sensitive configuration from environment variables ---

# These are populated by Docker Compose from the ./.env file

APP_PORT = os.getenv('APP_PORT', '5000') # Default to 5000 if not set FLASK_ENV = os.getenv('FLASK_ENV', 'production')

# Database connection details (user, host, port, name) from .env

DB_HOST = os.getenv('DATABASE_HOST') DB_PORT = os.getenv('DATABASE_PORT') DB_NAME = os.getenv('DATABASE_NAME') DB_USER = os.getenv('DATABASE_USER')

# --- Load sensitive secrets from Docker Secret files ---

# Flask application secret key

APP_SECRET_KEY_PATH = "/run/secrets/app_secret_key" APP_SECRET_KEY = None if os.path.exists(APP_SECRET_KEY_PATH): with open(APP_SECRET_KEY_PATH, 'r') as secret_file: APP_SECRET_KEY = secret_file.read().strip() else: print(f"Warning: App secret key file not found at {APP_SECRET_KEY_PATH}")

if APP_SECRET_KEY: app.secret_key = APP_SECRET_KEY else: # ⚠️ What can go wrong: This fallback should ONLY be for development/testing. # In production, a missing secret should halt startup or trigger an alert. app.secret_key = 'super-insecure-fallback-key' # Fallback for development, NOT for production

# Database password for the web application

WEB_DB_PASSWORD_PATH = "/run/secrets/web_db_password" WEB_DB_PASSWORD = None if os.path.exists(WEB_DB_PASSWORD_PATH): with open(WEB_DB_PASSWORD_PATH, 'r') as secret_file: WEB_DB_PASSWORD = secret_file.read().strip() else: print(f"Warning: Web DB password file not found at {WEB_DB_PASSWORD_PATH}")

# Construct the full DATABASE_URL using secrets and environment variables

DATABASE_URL = None if DB_USER and WEB_DB_PASSWORD and DB_HOST and DB_PORT and DB_NAME: DATABASE_URL = f"postgresql://{DB_USER}:{WEB_DB_PASSWORD}@{DB_HOST}:{DB_PORT}/{DB_NAME}" else: print("Warning: Missing database connection details (user, password, host, port, or name).") # In a real app, you might raise an error here to prevent startup without critical config.

print(f"Flask Environment: {FLASK_ENV}") print(f"Database Host: {DB_HOST}, Port: {DB_PORT}, Name: {DB_NAME}, User: {DB_USER}")

# ⚡ Quick Note: Do NOT log secrets in production!

# print(f"App Secret Key: {APP_SECRET_KEY}")

# print(f"Web DB Password: {WEB_DB_PASSWORD}")

# Database connection function

def get_db_connection(): if not DATABASE_URL: print("Error: DATABASE_URL not constructed.") return None try: conn = psycopg2.connect(DATABASE_URL) return conn except Exception as e: print(f"Database connection error: {e}") return None

@app.route('/') def hello(): return jsonify(message="Hello from the Dockerized Flask App!", environment=FLASK_ENV)

@app.route('/db-test') def db_test(): conn = get_db_connection() if conn: cursor = conn.cursor() cursor.execute('SELECT 1;') result = cursor.fetchone() conn.close() return jsonify(message="Database connection successful!", result=result) else: return jsonify(message="Database connection failed!", error="Could not connect to database."), 500

if **name** == '**main**': app.run(host='0.0.0.0', port=int(APP_PORT))

```
- `os.getenv('APP_PORT', '5000')`: Reads `APP_PORT` from the container's
environment variables. Docker Compose populates this from our `.env` file. The
second argument (`'5000'`) provides a default if the variable isn't set,
useful for local testing or when the `.env` file is optional.
- `DB_HOST`, `DB_PORT`, `DB_NAME`, `DB_USER`: These are all read from
environment variables, which are populated from the `.env` file.
- `APP_SECRET_KEY_PATH` and `WEB_DB_PASSWORD_PATH`: Define the expected paths
for the secret files. Docker Secrets are mounted as files inside the container
at `/run/secrets/<secret_name>`.
- The `if os.path.exists(...)` blocks: This is the standard and recommended
way for an application to read Docker secrets. The application opens and reads
the file content, which is the secret value.
- `DATABASE_URL = f"postgresql://{DB_USER}:{WEB_DB_PASSWORD}@{DB_HOST}:
{DB_PORT}/{DB_NAME}"`: The full database connection string is now dynamically
constructed within the application. It combines non-sensitive environment
variables (user, host, port, name) with the sensitive password read from a
secret file.
- **Production Awareness:** The `print` statements are helpful for debugging,
but you should avoid logging sensitive data (like `APP_SECRET_KEY` or
```

```
`WEB_DB_PASSWORD`) in production environments. The warning about missing
database details is a good practice for robust applications.

## Testing & Verification

Now, let's build and run our services to verify that configuration and secrets
are being handled correctly.

1.  **Build and Run the Services:**
    Navigate to your project root in the terminal.

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

```
`docker compose build` ensures our Docker image for the `web` service is
updated with the latest `app.py` changes. `docker compose up -d` starts our
services in detached mode.
```

1. **Verify Environment Variables (Non-Sensitive Config):** We can inspect the environment variables of the `web` container.

```
    docker compose exec web env | grep APP_PORT
    docker compose exec web env | grep FLASK_ENV
    docker compose exec web env | grep DATABASE_HOST
    docker compose exec web env | grep DATABASE_USER
```

```
You should see output similar to:
```

```
    APP_PORT=5000
    FLASK_ENV=development
    DATABASE_HOST=db
    DATABASE_USER=user
```

```
This confirms that the non-sensitive variables from `.env` are correctly
loaded into the `web` service's environment.
```

1. **Verify Secrets (Sensitive Config):**
    - **For the `web` service:** Check if the `app_secret_key` and `web_db_password` files exist within the container and verify their content.

```
        docker compose exec web ls -l /run/secrets/
        docker compose exec web cat /run/secrets/app_secret_key
```
```

```
docker compose exec web cat /run/secrets/web_db_password
```

```
    You should see both secret files listed (e.g., `app_secret_key`,
`web_db_password`) and their content printed (e.g.,
`this_is_a_very_secret_key_for_flask_app`).
- **For the `db` service:** Confirm that `POSTGRES_PASSWORD` is *not* in the
environment variables, but `POSTGRES_PASSWORD_FILE` is set.
```

```
docker compose exec db env | grep POSTGRES_PASSWORD
docker compose exec db env | grep POSTGRES_PASSWORD_FILE
```

```
    You should see no output for `POSTGRES_PASSWORD` (or only `_FILE`
variants), but `POSTGRES_PASSWORD_FILE=/run/secrets/db_password` should be
present.
    Also, verify the `db_password` secret file itself:
```

```
docker compose exec db ls -l /run/secrets/
docker compose exec db cat /run/secrets/db_password
```

```
    You should see `db_password` listed and its content printed.

**⚡ Real-world insight:** By using `POSTGRES_PASSWORD_FILE` and Docker
Secrets, the actual password string never appears in the container's process
environment, which can be easily inspected (e.g., via `docker inspect`).
Similarly, for the `web` app, reading secrets from `/run/secrets/` files is
more secure than exposing them as environment variables. Docker mounts these
secret files with restricted permissions (mode `0444`), making them read-only
for the container's user.
```

1. **Test Application Endpoints:** Open your browser or use `curl` to access the application:

   - `<http://localhost:5000/` > Expected output: `{"environment":"development","message":"Hello from the Dockerized Flask App!"}`
   - `<http://localhost:5000/db-test` > Expected output: `{"message":"Database connection successful!","result":[1]}` This confirms that the application successfully connected to the database using the credentials provided via secrets and environment variables.

2. **Clean up:**

```
docker compose down
```

```
This command stops and removes the containers, networks, and volumes created
by `docker compose up`.
```

---

## Production Considerations

Securing configuration and secrets is paramount in production. Here are key considerations:

- **`.env` vs. Docker Secrets in Production:**
    - **`.env` files:** Ideal for local development and non-sensitive configuration. **Do not use `.env` files for sensitive data in production environments, especially if they are committed to version control.** They lack the security mechanisms of dedicated secret managers.
    - **Docker Secrets:** A significant step up for sensitive data in production for Docker Compose and Swarm deployments. They are mounted as files with restricted permissions inside the container, not exposed as easily inspectable environment variables. Docker handles their secure storage and transmission within the Docker daemon.
    - **External Secret Managers:** For highly sensitive, critical applications, consider integrating with dedicated secret management services like HashiCorp Vault, AWS Secrets Manager, Azure Key Vault, or Google Secret Manager. These services offer advanced features like secret rotation, auditing, and fine-grained access control, which are beyond the scope of basic Docker Secrets. Docker Secrets are a good intermediary step for smaller deployments or when a full-fledged secret manager is overkill.
- **Secret Rotation:** Regularly rotate your secrets (e.g., database passwords, API keys). Docker Secrets don't automate rotation, so you'd need an external process or script to update the secret files on the host and then redeploy your services (e.g., `docker compose up -d`).

- **Permissions on Host Files:** Ensure your secret files (like `db_password.txt`) on the host machine have strict permissions, preventing unauthorized access. They should typically be readable only by the user running Docker. For example, `chmod 600 db_password.txt` makes the file readable and writable only by the owner.
- **Environment Variables & Security:** While `.env` files inject environment variables, and some images allow consuming secrets as environment variables (e.g., `POSTGRES_PASSWORD`), be aware that environment variables can be inspected more easily (e.g., `docker inspect <container_id>` or `docker exec <container_id> env`). Always prefer file-based secret consumption (like `/run/secrets/`) when the application or image supports it, as it offers a higher level of isolation.
- **Security Auditing with `docker-bench-security`:** For a comprehensive security posture, regularly audit your Docker host and container configurations. Tools like `docker-bench-security` (from the official Docker team, available at [https://github.com/docker/docker-bench-security](https://github.com/docker/docker-bench-security)) can scan your setup against common best practices to identify potential vulnerabilities in your Docker daemon, images, and running containers. This tool helps ensure your Docker environment itself adheres to security best practices.

## Common Issues & Solutions

1. **Secret File Not Found in Container:**
    - **Issue:** The application reports that `/run/secrets/<secret_name>` does not exist, or the `cat` command shows an empty file or "No such file or directory".
    - **Solution:** Double-check your `docker-compose.yml`.
        - Ensure the top-level `secrets:` block is correctly defined and that the `file:` path (e.g., `./db_password.txt`) is correct relative to `docker-compose.yml`.
        - Verify that the service's `secrets:` block (e.g., `secrets: - db_password`) correctly references the named secret defined at the top level.
        - Make sure the actual secret file (e.g., `db_password.txt`) exists on your host machine in the specified path.
2. **Application Not Reading Environment Variables:**
    - **Issue:** `os.getenv()` returns `None` or a default value, even though the variable is defined in `.env`.
    - **Solution:**
        - Confirm `env_file: - ./.env` is present and correctly indented under the service in `docker-compose.yml`.
        - Ensure the `.env` file is in the correct location (usually the project root alongside `docker-compose.yml`).
        - Check for typos in the variable name in both `.env` and your application code (e.g., `APP_PORT` vs. `app_port`). Environment variable names are case-sensitive.
        - Remember to restart your services (`docker compose up -d`) after changing `docker-compose.yml` or `.env` files for changes to take effect. If you've previously built the image, `docker compose up --build` will ensure the latest image is used.

3. **Permissions on Secret Files:**
    - **Issue:** You get "permission denied" when the container tries to read /run/secrets/<secret_name>, or Docker Compose fails to load the secret.
    - **Solution:** While Docker mounts secrets with `0444` permissions inside the container, ensuring the host file (e.g., `db_password.txt`) has appropriate permissions can prevent issues. Typically, `chmod 600 db_password.txt` is a good practice, making it readable only by the file owner. If Docker's user mapping is complex, or if the Docker daemon itself doesn't have access to the secret file on the host, this can cause problems.

---

## Summary & Next Step

You've successfully implemented secure configuration and secret management for your Docker Compose application. You now understand the distinction between non-sensitive configuration and sensitive secrets, and how to leverage `.env` files for local development and Docker Secrets for production-grade security. This greatly enhances the security and maintainability of your application, paving the way for more robust deployments. Remember to also utilize tools like `docker-bench-security` to audit your overall Docker environment for security best practices.

In the next chapter, we'll dive into implementing health checks for our services. Health checks are crucial for ensuring that our containers are not just running, but are actually healthy and ready to serve requests, which is vital for application reliability and automated recovery.

---

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

---

## References


- Docker Documentation: Compose file reference
- Docker Documentation: Secrets
- The Twelve-Factor App: III. Config
- PostgreSQL Docker Image: Environment Variables
- docker-bench-security GitHub Repository

CHAPTER 08

# Implementing Health Checks for Service Robustness

## Introduction: Building Resilient Services with Health Checks

In any production environment, applications are subject to transient failures, unresponsiveness, or unexpected crashes. Simply confirming a container is "running" isn't sufficient; we need to know if the application inside that container is truly healthy, responsive, and ready to serve traffic. This chapter focuses on implementing **health checks** for your Docker Compose services, a cornerstone practice for building robust, self-healing, and reliable applications.

By the conclusion of this chapter, you will have configured sophisticated health checks for both your web application and database services. This setup enables Docker Compose to automatically detect unhealthy containers and respond appropriately—such as restarting them or delaying the startup of dependent services—thereby significantly enhancing your application's operational resilience and stability.

## Project Overview: Securing Application Uptime

Our overarching project aims to build a production-ready, multi-service web application stack using Docker and Docker Compose. Each chapter incrementally adds crucial best practices. This particular chapter tackles service reliability by integrating health checks.

The goal is to ensure that our `web` application and `db` services accurately report their operational status. This isn't just about knowing if a process is alive; it's about verifying that the application can actually perform its intended function, including connecting to its dependencies. Achieving this improves:

- **Reliability**: Services automatically recover from transient issues.
- **Availability**: Unhealthy services are identified and isolated, preventing them from accepting traffic.
- **Deployment Stability**: Dependent services only start when their prerequisites are genuinely ready.

## Core Concepts: Liveness, Readiness, and Self-Healing

Health checks are fundamental for ensuring the reliability and availability of containerized applications. They provide the necessary intelligence to container orchestrators like Docker Compose, allowing them to make informed decisions about service state.

### Why Health Checks?

Without explicit health checks, Docker Compose (or any orchestrator) only monitors if a container's main process is running. This is a weak signal. An application process might be running, but could be:

- **Stuck**: In a deadlock or infinite loop.
- **Unresponsive**: Overloaded or out of memory.
- **Disconnected**: Unable to reach its database or other critical dependencies.
- **Not yet ready**: Still initializing during startup.

Health checks bridge this gap by executing custom commands or HTTP requests inside the container, providing a true assessment of application health.

### Liveness vs. Readiness Checks

While Docker's `healthcheck` directive combines aspects of both, it's important to understand the conceptual difference, especially when moving to orchestrators like Kubernetes.

- **Liveness Checks**: These determine if a container is still capable of performing its core function. If a liveness check repeatedly fails, it signals that the container is "dead" or irrevocably stuck. The typical response is to restart the container, hoping to restore it to a healthy state. This ensures the application doesn't remain in a broken state indefinitely.
  - `⚠️ What can go wrong:` If a liveness check is too aggressive or fails for transient reasons, it can lead to a "restart loop" where the service constantly restarts, never truly stabilizing.

- **Readiness Checks**: These ascertain if a container is ready to accept incoming traffic. This is crucial during startup, after a restart, or during scaling events. A service might be alive but not yet ready (e.g., still loading data, warming up caches, or connecting to a database). Readiness checks prevent traffic from being routed to services that are not yet fully initialized, avoiding client-side errors.
    - `⚡ Real-world insight:` In production, load balancers often use readiness checks to determine which instances can receive new requests.

Our Docker Compose `healthcheck` configuration will serve both purposes: determining if a service is alive and, through `depends_on: service_healthy`, if it's ready for its dependents.

---

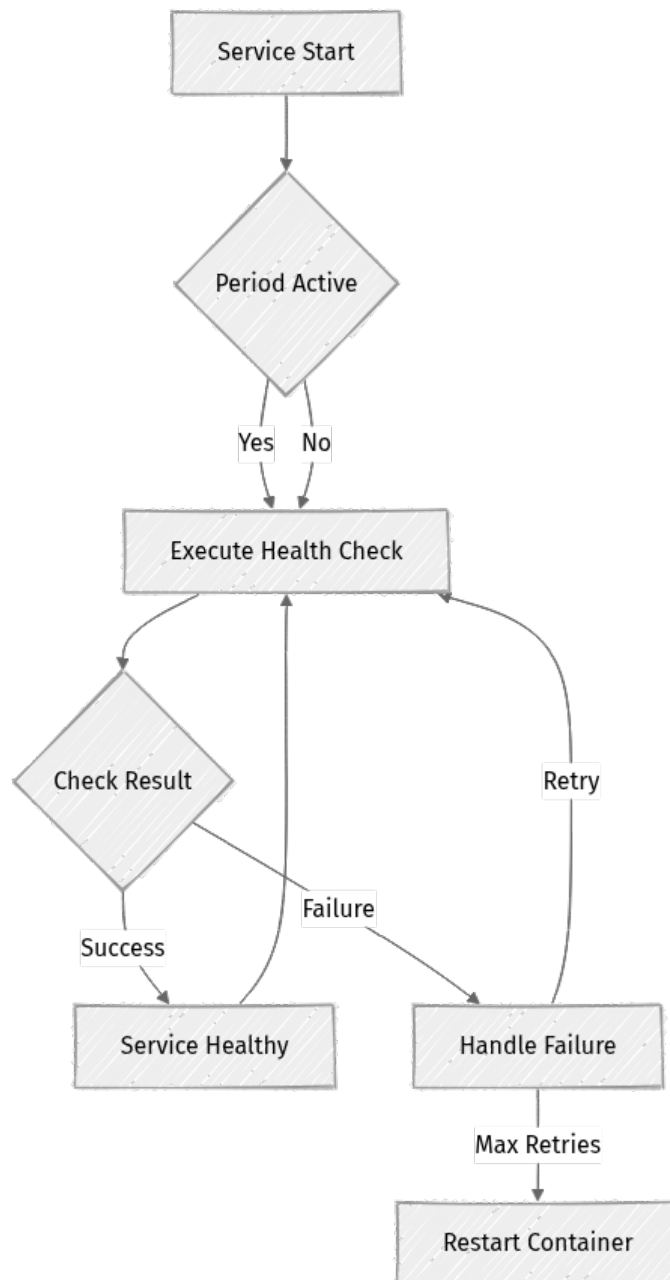
## Architectural Design: Integrating Health Checks into Our Stack

Our application stack consists of a `web` service (Flask application) and a `db` service (PostgreSQL). We will embed health check configurations directly into their respective service definitions within `docker-compose.yml`.

The `web` service's health check will perform an HTTP request to an internal `/health` endpoint, which in turn will verify its critical dependency: the `db` service. The `db` service will use `pg_isready`, a PostgreSQL utility, to confirm its availability. The `web` service will explicitly wait for the `db` service to be `healthy` before starting.

### Health Check Operational Flow

The following diagram illustrates the lifecycle of a service with integrated health checks.




**Explanation:** When a service starts, Docker Compose initiates a `start_period`. During this time, health checks run, but failures don't count towards the `retries` limit. Once a check passes, the service is marked `healthy`. If it later fails consecutively and exceeds the `retries` limit, Docker Compose will restart the container. This self-healing mechanism is vital for maintaining service uptime.

## Build Plan: Implementing Health Checks

To integrate health checks effectively, we'll follow these steps:

1. **Enhance Web Application with a Health Endpoint**: Add a `/health` endpoint to our Flask application that not only verifies the application process but also attempts a connection to the database.
2. **Update Web Dockerfile for Health Check Tools**: Ensure the `web` service's Docker image includes `curl` for HTTP checks and necessary libraries for database connectivity within the health check.
3. **Configure Health Checks in Docker Compose**: Add the `healthcheck` directive to both `web` and `db` services in `docker-compose.yml`, specifying commands, intervals, timeouts, and retry logic. We will also use `depends_on: service_healthy` for robust service orchestration.

## Step-by-Step Implementation

We will modify our existing files to incorporate these health check mechanisms.

### 1. Enhance Web Application with a Health Endpoint (app/main.py)

A robust health endpoint should do more than just return a 200 OK. It should confirm that critical internal components and external dependencies are operational. Let's update our Flask application to include a database connection test in its `/health` endpoint.

Create or modify `app/main.py` in your web application directory:

```
# app/main.py
from flask import Flask
import os
import psycopg2 # Required for PostgreSQL connection
import logging

app = Flask(__name__)
# Configure basic logging
logging.basicConfig(level=logging.INFO)

@app.route('/')
def hello_world():
    return 'Hello, Docker Compose! This is our web app.'

@app.route('/health')
def health_check():
    """
    Performs a health check, including a database connection test.
    Returns 200 OK if healthy, 500 Internal Server Error otherwise.
```

```
    """
    try:
        # Attempt to connect to the database using environment variables
        # 🧠 Important: Use connection pooling in a real application to avoid
        # opening/closing connections on every health check request.
        conn = psycopg2.connect(
            host=os.getenv('DB_HOST', 'db'),
            database=os.getenv('DB_NAME', 'mydatabase'),
            user=os.getenv('DB_USER', 'user'),
            password=os.getenv('DB_PASSWORD', 'password')
        )
        conn.close() # Close connection immediately after testing
        app.logger.info("Health check: DB connection successful.")
        return 'OK', 200
    except Exception as e:
        app.logger.error(f"Health check failed: Database connection error -
{e}")
        return 'DB Connection Failed', 500

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8080)
```

**Explanation:**

- The new `/health` route is designed to return `OK` (HTTP 200) if both the Flask application is running and it can successfully establish a connection to the PostgreSQL database.
- If the database connection fails, it returns `DB Connection Failed` (HTTP 500). This provides a more accurate and comprehensive assessment of the web application's operational readiness.
- We use `os.getenv` to fetch database credentials, reinforcing the practice of externalizing configuration.

## 2. Update Web Dockerfile for Health Check Tools (`web/Dockerfile`)

For our health check to function correctly, the `web` container needs `curl` to make HTTP requests to its own `/health` endpoint. Additionally, `psycopg2` (the PostgreSQL adapter for Python) requires certain system libraries to compile correctly.

Modify `web/Dockerfile`:

```
# web/Dockerfile
# Use a minimal Python image for production (Python 3.10-slim-buster as of
2026-05-22)
FROM python:3.10-slim-buster

# Set environment variables for Python and Flask
ENV PYTHONUNBUFFERED 1
ENV FLASK_APP main.py
```

```
# Install system dependencies and Python packages
WORKDIR /app
COPY requirements.txt .
RUN apt-get update && apt-get install -y --no-install-recommends \
    curl \
    build-essential \
    libpq-dev \
    && rm -rf /var/lib/apt/lists/*
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY app/ .

# Expose the port the app runs on (for documentation, not security)
EXPOSE 8080

# Command to run the application
CMD ["flask", "run", "--host", "0.0.0.0", "--port", "8080"]
```

**Explanation:**

- `curl` is added to the `apt-get install` command. This command-line tool will be used by our health check to query the `/health` endpoint.
- `build-essential` and `libpq-dev` are critical for the `psycopg2` Python package to compile and link correctly with PostgreSQL client libraries during the `pip install` step. `libpq-dev` provides the necessary header files and static libraries for PostgreSQL client development.

### 3. Configure Health Checks in Docker Compose (`docker-compose.yml`)

Now, let's add the `healthcheck` directives to both the `web` and `db` services in your `docker-compose.yml` file. As of 2026-05-22, the Compose Specification is the current standard, and explicitly specifying a `version` field in `docker-compose.yml` is no longer recommended.

Modify `docker-compose.yml`:

```
# docker-compose.yml
# This file adheres to the Compose Specification (as of 2026-05-22).
# Explicitly specifying 'version' is no longer recommended.
# See: https://github.com/jamesatdocker/docker-docs/blob/main/compose/compose-file/compose-versioning.md

services:
  web:
    build: ./web
    ports:
      - "80:8080"
    environment:
      - DB_HOST=db
      - DB_NAME=mydatabase
      - DB_USER=user
```

```
      - DB_PASSWORD=password
    depends_on:
      db:
        condition: service_healthy # ⚡ Pro tip: Wait for 'db' to be truly
healthy before 'web' starts
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 20s # Give the web app time to start and connect to DB

  db:
    image: postgres:15-alpine # Using a specific, stable version (PostgreSQL
15 as of 2026-05-22)
    environment:
      POSTGRES_DB: mydatabase
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    volumes:
      - db_data:/var/lib/postgresql/data # Persistent data for the database
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U user -d mydatabase"]
      interval: 10s
      timeout: 5s
      retries: 5
      start_period: 10s # Give PostgreSQL time to initialize

volumes:
  db_data: # Define the named volume for the database
```

**Explanation of `healthcheck` parameters:**

- **`test`**: The command executed to determine health.
  - `["CMD", "curl", "-f", "http://localhost:8080/health"]`: For the `web` service, this instructs Docker to run `curl -f <http://localhost:8080/health>`. The `-f` (fail) flag causes `curl` to exit with a non-zero status code if the HTTP response indicates an error (e.g., 4xx or 5xx status codes). If `curl` exits non-zero, the check fails.
  - `["CMD-SHELL", "pg_isready -U user -d mydatabase"]`: For the `db` service, `pg_isready` is a PostgreSQL utility checking connection status. `CMD-SHELL` executes the command within a shell (e.g., `/bin/sh -c "..."`), which is often preferred for commands with complex arguments or environment setup.
- **`interval`**: Specifies how often the health check command is run (e.g., `30s`). This directly impacts how quickly Docker detects a change in service health.

- **`timeout`**: The maximum duration to wait for the health check command to complete. If it exceeds this, the check is considered failed (e.g., `10s`). A timeout prevents a hung health check from blocking status updates.
- **`retries`**: The number of consecutive failures allowed before the container is marked as `unhealthy` and potentially restarted (e.g., `3` for `web`, `5` for `db`). This prevents flapping due to transient issues.
- **`start_period`**: An initial period during which health check failures do not count towards the `retries` limit. This is vital for services that take time to start up (e.g., `20s` for `web`, `10s` for `db`). If a health check passes during this period, the service is marked healthy. Failures after this period trigger the retry mechanism. Without `start_period`, a slow-starting service could enter a restart loop.

**`depends_on` with `condition: service_healthy`:** Notice the `depends_on` for the `web` service now includes `condition: service_healthy`. This is a crucial production-grade feature: it instructs Docker Compose to wait until the `db` service reports itself as `healthy` (as determined by its health check) before initiating the `web` service. This prevents the web application from attempting to connect to an unready database, significantly reducing startup errors and improving overall system stability.

---

## Verification: Observing Service Health

With health checks configured, let's build our services and observe their behavior.

1. **Rebuild and Start Services:** Ensure you are in the directory containing your `docker-compose.yml`. Then, rebuild your images to include the `curl` utility and the updated application code, and start the services:

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

```
The `-d` flag runs the containers in detached mode.
```

1. **Monitor Service Health Status:** You can observe the health status of your services using `docker compose ps`:

```
docker compose ps
```

```
You should see output similar to this (exact names and ports might vary):
```

```
    NAME                    COMMAND                        SERVICE
STATUS              PORTS
    myproject-db-1      "docker-entrypoint.s…"    db                    running
(healthy)   5432/tcp
    myproject-web-1     "flask run --host 0.…"    web                   running
(healthy)   0.0.0.0:80->8080/tcp
```

```
Initially, services might display `(starting)` or `(unhealthy)` statuses
before transitioning to `(healthy)`. The `start_period` and `interval`
settings directly influence the duration of this transition. For example, the
`db` service will start first, become healthy, and *then* the `web` service
will begin its `start_period`.
```

1. **Inspect Detailed Health Check Logs:** For a granular view of a container's health checks, use `docker inspect`:

```
    docker inspect myproject-web-1 | grep Health -A 5
```

```
Replace `myproject-web-1` with the actual name of your web service container
(obtainable from `docker compose ps`). You'll see structured output detailing:
```

```
            "Health": {
                "Status": "healthy",
                "FailingStreak": 0,
                "Log": [
                    {
                        "Start": "2026-05-22T10:00:00.123456789Z",
                        "End": "2026-05-22T10:00:00.567890123Z",
                        "ExitCode": 0,
                        "Output": "  % Total    % Received % Xferd  Average
Speed   Time    Time     Time  Current\n
Dload  Upload   Total   Spent    Left  Speed\n100    19  100    19    0     0
19000      0 --:--:00 --:--:00 --:--:00 19000\nOK"
                    }
                ]
            },
```

```
This output shows the current `Status`, `FailingStreak`, and a log of recent
health check command executions, including their `ExitCode` and `Output`. An
`ExitCode` of `0` indicates successful execution.
```

1. **Simulate a Service Failure:** To witness health checks in action, let's simulate a failure for the web application. We can do this by temporarily stopping the Flask process inside the container.

   First, identify the container ID for your `web` service:

```
docker ps | grep web
```

```
Then, execute a command inside that container to terminate the Flask process:
```

```
docker exec <web_container_id> pkill -f "flask run"
```

```
Immediately after, run `docker compose ps` again:
```

```
docker compose ps
```

```
You should observe the `web` service quickly transitioning to `(unhealthy)`.
After a few retries (as defined by `retries` in `docker-compose.yml`), Docker
Compose will automatically restart the container, bringing it back to
`(starting)` and eventually `(healthy)`. This demonstrates the powerful self-
healing capability provided by well-configured health checks.

Once you're done with the demonstration, gracefully bring down the services:
```

```
docker compose down
```

## Production Best Practices for Health Checks

Implementing health checks is a significant stride towards production readiness, but understanding their nuances is key:

- **Granularity of Checks**: A simple HTTP 200 OK might be insufficient. For critical services, health checks should probe deeper into application logic, verify database connectivity, or confirm access to essential external APIs. Our updated web app health check, which includes a database connection test, is a good example of this.
- **Resource Overhead**: Health checks run periodically. Very frequent or resource-intensive checks can consume CPU and network resources, particularly across a large number of containers. Carefully balance `interval` and `timeout` settings to avoid unnecessary load.
- **Startup vs. Liveness**: The `start_period` is crucial for services with prolonged initialization times. Without it, a service might be prematurely marked unhealthy and restarted before it's even had a chance to fully boot, leading to a restart loop.
- **Dependencies**: Employing `condition: service_healthy` within `depends_on` is a fundamental best practice. It guarantees that services only commence operation once their critical dependencies are genuinely ready, effectively preventing cascading startup failures across your stack.
- **Orchestration Integration**: In larger-scale deployments utilizing orchestrators like Kubernetes, these health check concepts directly translate to `livenessProbe` and `readinessProbe` configurations, which are foundational for achieving high availability and enabling seamless rolling updates. The principles you learn here are directly transferable.

## Troubleshooting Common Health Check Issues

1. **Health Check Command Fails Due to Missing Tools:**
    - **Issue**: The specified `test` command (e.g., `curl`, `pg_isready`) is not installed within the container image.
    - **Solution**: Add the necessary package installation to your `Dockerfile` using the appropriate package manager (e.g., `apt-get install`, `apk add`, `yum install`) for your chosen base image. We addressed this by adding `curl` to our `web/Dockerfile`.
2. **Service Never Becomes Healthy / Stuck in `(starting)`:**
    - **Issue**: The `start_period` might be too short, or the application itself requires more time to initialize than anticipated. Alternatively, the health check might be failing due to a legitimate underlying problem (e.g., incorrect port, invalid database credentials, application crash).
    - **Solution**:
        - Increase the `start_period` to allow the service ample time to boot.
        - Inspect container logs (`docker compose logs <service_name>`) for any errors occurring during startup or directly from the health check command's execution.
        - Manually execute the health check command inside the running container (`docker exec -it <container_id> <health_check_command>`) to debug its output and exit code directly.
3. **Health Check Command Returns Success But Application is Unresponsive:**
    - **Issue**: The health check is too simplistic (e.g., merely checking if a port is open) and doesn't accurately reflect the application's true operational status (e.g., the database connection has dropped internally, an internal message queue is full, or a critical background process has failed).
    - **Solution**: Design more comprehensive health checks. For a web application, this might involve querying a specific endpoint that, in turn, attempts to connect to the database or an important external service. For a database, ensure the check verifies not just network accessibility but also the ability to process simple queries. Our updated web app health check, which includes a database connection test, is a significant step in this direction.

## Summary and Next Steps

You have successfully implemented robust health checks for your Docker Compose services! You now understand how to define `healthcheck` directives, the significance of parameters like `interval`, `timeout`, `retries`, and `start_period`, and how to leverage `depends_on: service_healthy` for reliable service startup. These practices significantly enhance the resilience and self-healing capabilities of your application stack, moving it closer to production readiness.

Your services are now better equipped to handle transient failures and accurately report their operational status, laying a stronger foundation for production deployments. In the next chapter, we will build on this foundation by exploring how to optimize our Docker images using multi-stage builds, further improving deployment efficiency and security.

---

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

## References


- Docker Documentation: [https://docs.docker.com/](https://docs.docker.com/)
- Compose Specification Versioning: [https://github.com/jamesatdocker/docker-docs/blob/main/compose/compose-file/compose-versioning.md](https://github.com/jamesatdocker/docker-docs/blob/main/compose/compose-file/compose-versioning.md)
- PostgreSQL `pg_isready` documentation: [https://www.postgresql.org/docs/current/app-pgisready.html](https://www.postgresql.org/docs/current/app-pgisready.html)
- `psycopg2` Official Documentation: [https://www.psycopg.org/docs/](https://www.psycopg.org/docs/)

CHAPTER 09

# Optimizing Docker Images with Multi-Stage Builds

In modern production environments, Docker image size has a direct impact on deployment speed, resource consumption, and security posture. Large images lead to slower pulls, increased storage costs, and a broader attack surface due to unnecessary tools and dependencies. This chapter tackles that problem head-on by introducing multi-stage Docker builds.

We'll refactor a typical application Dockerfile to leverage multi-stage builds, dramatically reducing its final size. By the end of this milestone, you will have a significantly smaller, more efficient, and more secure Docker image for your web application, ready for robust production deployment.

---

## Project Overview: Leaner Containers for Production

This chapter focuses on a critical aspect of containerization: optimizing Docker image size. We will take a conventional Node.js web application Dockerfile and transform it into a multi-stage build. This transformation will demonstrate how to separate build-time dependencies from runtime essentials, resulting in a production image that is minimal, fast to deploy, and more secure.

Our goal is to achieve a noticeably smaller final image without compromising application functionality, a key requirement for any production deployment.

---

## Tech Stack & Setup

For this chapter, our primary focus is on the `Dockerfile` itself and how Docker Engine processes it.

- **Docker Engine**: We assume you have Docker Engine installed and running. As of 2026-05-22, the latest stable version of Docker Engine is actively maintained. (Specific version information for Docker Engine is dynamic and best checked against official releases at the time of installation, but the core Dockerfile syntax remains stable.)
- **Node.js**: Our example application is a simple Node.js Express server. We'll use `node:20-alpine` as our base image, which refers to Node.js version 20 (an LTS release) on an Alpine Linux base, known for its small footprint.

- **Text Editor**: Any code editor (e.g., VS Code) will suffice.
- **Command Line**: Basic familiarity with `docker build` and `docker run` commands.

---

## The Challenge of Image Bloat

When containerizing an application, especially one that requires compilation or extensive build steps (like a Node.js application with frontend assets, a Go binary, or a Java JAR), the build process often pulls in many development dependencies, compilers, and tools. If these are all included in the final image, it becomes unnecessarily large.

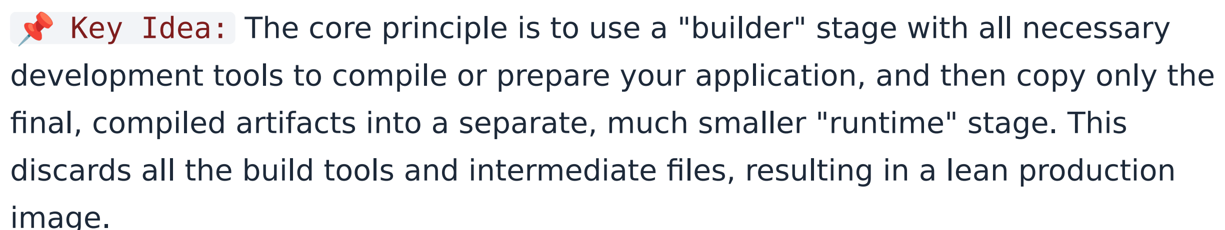
For example, a Node.js application might need `npm` or `yarn` and various build tools to compile TypeScript or bundle frontend assets. These tools are critical during the build phase but are entirely superfluous at runtime. Shipping them in your production image adds bloat, increasing:

- **Deployment Times**: Larger images take longer to pull from registries to deployment targets.
- **Storage Costs**: More disk space is consumed on registries and host machines.
- **Attack Surface**: Every additional file, library, or tool in an image introduces potential vulnerabilities that need to be patched and monitored.

---

## Core Concept: Multi-Stage Builds Explained

Multi-stage builds are a powerful Dockerfile feature designed to create smaller, more secure images. They achieve this by allowing you to define multiple `FROM` instructions within a single Dockerfile. Each `FROM` instruction starts a new build stage, and critically, you can selectively copy artifacts from previous stages into a later stage.



`📌 Key Idea:` The core principle is to use a "builder" stage with all necessary development tools to compile or prepare your application, and then copy only the final, compiled artifacts into a separate, much smaller "runtime" stage. This discards all the build tools and intermediate files, resulting in a lean production image.

### How it Solves the Problem

Consider a Node.js application:

1. **Builder Stage**: Uses a comprehensive Node.js image (e.g., `node:20-alpine`) to install all `devDependencies` and `dependencies`, compile TypeScript, or bundle frontend assets.
2. **Runtime Stage**: Uses a minimal Node.js image (e.g., `node:20-alpine`) and only copies the compiled application code and strictly necessary production dependencies (installed with `--only=production`) from the builder stage.

This separation ensures that your final image contains only what's absolutely required for the application to run, not what's needed to build it.

---

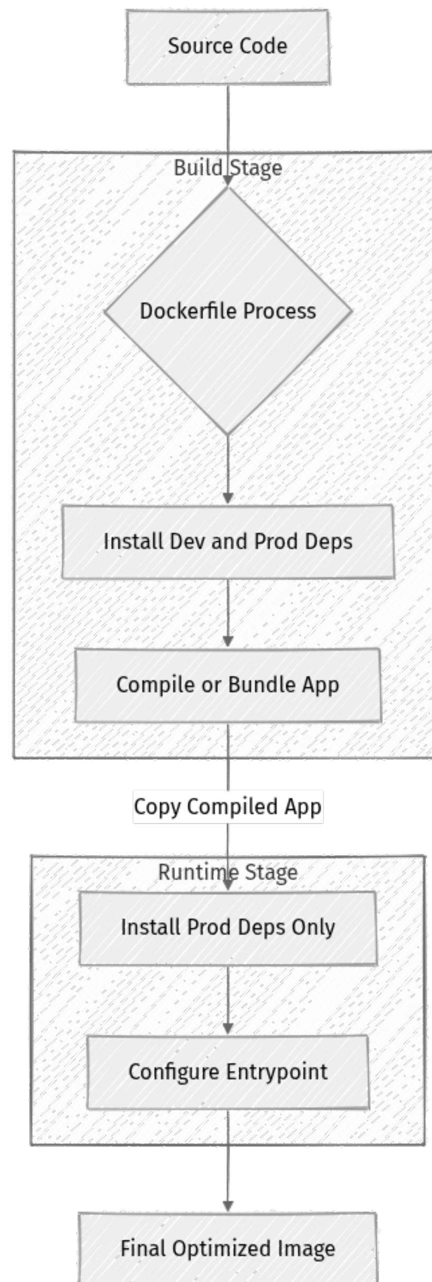
## Build Plan: Optimizing Our Dockerfile

Our strategy for optimizing the Docker image for our Node.js web application involves these distinct steps:

1. **Baseline Setup**: Create a simple Node.js application and an initial, unoptimized Dockerfile to establish a baseline image size.
2. **Identify Build vs. Runtime Needs**: Determine which dependencies and files are essential for building the application and which are only needed for its execution.
3. **Implement Builder Stage**: Modify the Dockerfile to include a "builder" stage that handles all dependency installation and any compilation steps.
4. **Implement Runtime Stage**: Add a "runtime" stage that starts from a clean, minimal base image and selectively copies only the necessary artifacts (compiled code, production dependencies) from the "builder" stage.
5. **Verify Optimization**: Build the new multi-stage image and compare its size to the baseline. Confirm the application still functions correctly.

### Architecture Overview

The multi-stage build process can be visualized as a pipeline where intermediate artifacts are passed between stages, but not the entire environment.




---

## Step-by-Step Implementation

Let's begin by setting up our sample Node.js application and then refactoring its Dockerfile.

### 1. Create a Sample Node.js Application

If you haven't already, create a directory called `my-web-app` and navigate into it.

```
mkdir my-web-app
cd my-web-app
```

Inside `my-web-app`, create `package.json`:

```
// my-web-app/package.json
{
  "name": "my-web-app",
  "version": "1.0.0",
  "description": "A simple Node.js web app",
  "main": "src/server.js",
  "scripts": {
    "start": "node src/server.js"
  },
  "dependencies": {
    "express": "^4.19.2"
  }
}
```

Next, create the `src` directory and the `server.js` file:

```
mkdir src
```

Inside `my-web-app/src/server.js`:

```
// my-web-app/src/server.js
const express = require('express');
const app = express();
const port = process.env.PORT || 3000;

app.get('/', (req, res) => {
  res.send('Hello from the optimized Docker container!');
});

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

Install the local dependencies:

```
npm install
```

## 2. Initial, Unoptimized Dockerfile

Now, create a basic, single-stage Dockerfile that doesn't use multi-stage builds. This will serve as our baseline.

Create `my-web-app/Dockerfile.unoptimized`:

```
# my-web-app/Dockerfile.unoptimized
FROM node:20-alpine

# Set the working directory inside the container
```

```
WORKDIR /app

# Copy package.json and package-lock.json to install dependencies
# This layer is cached if these files don't change
COPY package*.json ./

# Install all dependencies (dev and prod)
RUN npm install

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

# Expose the port the application listens on
EXPOSE 3000

# Define the command to run the application
CMD ["npm", "start"]
```

Build this image and tag it `my-web-app:unoptimized`:

```
docker build -t my-web-app:unoptimized -f Dockerfile.unoptimized .
```

### 3. Refactor with Multi-Stage Build

Now, let's create our optimized Dockerfile using multi-stage builds. We'll define two stages: `builder` and the final runtime stage.

Create `my-web-app/Dockerfile`:

```
# my-web-app/Dockerfile (Optimized with Multi-Stage Build)

# --- Stage 1: Builder ---
# This stage installs all dependencies (dev + prod) and prepares the
application.
FROM node:20-alpine AS builder

WORKDIR /app

# Copy package.json and package-lock.json first to leverage Docker layer
caching.
# If these files don't change, this layer won't be rebuilt.
COPY package*.json ./

# Install all dependencies, including development ones if they were present.
# 'npm ci' is preferred for CI/CD as it uses package-lock.json for exact
versions,
# ensuring reproducible builds.
RUN npm ci

# Copy the rest of the application source code into the builder stage.
COPY . .

# If you had a build step (e.g., TypeScript compilation, Webpack bundling),
# it would typically go here:
# RUN npm run build
# For our simple Express app, there's no separate 'build' script beyond 'npm
```

```
start',
# so we'll just copy the raw source in the next stage.

# --- Stage 2: Production Runtime ---
# This stage uses a minimal base image and copies only the necessary
# production artifacts from the builder stage.
FROM node:20-alpine

WORKDIR /app

# Copy only the package.json and package-lock.json from the builder stage.
# This allows us to install only production dependencies in this final stage.
COPY --from=builder /app/package*.json ./

# Install only production dependencies.
# This command is crucial for keeping the final image lean.
RUN npm ci --only=production

# Copy the application source code (or compiled output) from the builder
stage.
# If 'npm run build' produced a 'dist' folder, you'd copy that:
# COPY --from=builder /app/dist ./dist
# For our simple app, we copy the source directly.
COPY --from=builder /app/src ./src

EXPOSE 3000

# Define the command to run the application in production
CMD ["npm", "start"]
```

**Why these changes are production-minded:**

- **`FROM node:20-alpine AS builder`**: By naming the first stage `builder`, we clearly define its purpose and make it referenceable.
- **`RUN npm ci` (in builder)**: `npm ci` ensures a clean installation of exact dependency versions as specified in `package-lock.json`, which is vital for reproducible builds.
- **`FROM node:20-alpine` (second instance)**: This starts a completely new, clean image build. None of the layers from the `builder` stage, except what we explicitly `COPY --from`, are included.
- **`COPY --from=builder /app/package*.json ./`**: This is the key to multi-stage builds. It copies specific files from the `builder` stage's filesystem (`/app/package*.json`) into the current stage's working directory (`./`). We copy these to correctly install production dependencies.
- **`RUN npm ci --only=production`**: This command is critical. It installs only the dependencies listed under `dependencies` in `package.json`, excluding `devDependencies`. This dramatically reduces the `node_modules` size.

- `COPY --from=builder /app/src ./src`: We copy only the application's source code (or compiled output if a build step was involved) from the `builder` stage. This prevents the final image from including unnecessary files from the build context.

---

## Testing & Verification

Now, let's build the optimized image and compare its size to our baseline.

1. **Build the optimized image:**

```
docker build -t my-web-app:optimized .
```

```
This command will use the `Dockerfile` we just created.
```

1. **Compare image sizes:**

```
docker images my-web-app
```

```
You should see two images listed: `my-web-app:unoptimized` and `my-web-
app:optimized`.
You will observe that the `optimized` version is smaller. For a very simple
app like this, the difference might be minor (e.g., a few MB), but for real-
world applications with many development dependencies or complex build steps
(like bundling frontend assets or compiling a Go binary), the savings can be
hundreds of megabytes or even gigabytes.

⚡ **Real-world insight:** A Go application compiled in a `golang:latest`
builder stage and then copied to a `scratch` or `alpine` runtime stage can
shrink from hundreds of MBs to just a few MBs for the final binary.
```

1. **Run the optimized container:**

   Verify that the application still functions correctly with the optimized image.

```
docker run -p 3000:3000 my-web-app:optimized
```

```
Open your web browser and navigate to `<http://localhost:3000`>. You should
see the message "Hello from the optimized Docker container!". This confirms
that all necessary files were successfully copied into the final stage.
```

1. **Inspect container layers (optional but recommended):**

   To understand the layers that make up your image and confirm the absence of build-time artifacts, use `docker history`:

```
docker history my-web-app:optimized
```

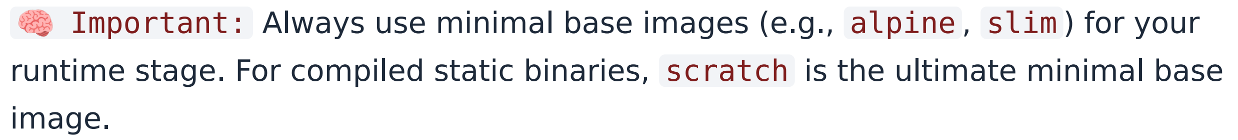
```
Compare this output to `docker history my-web-app:unoptimized`. You'll notice
that the `optimized` image's history is cleaner, reflecting only the steps
taken in the final runtime stage, without the full `npm install` history of
all dependencies.
```

---

## Production Considerations

Multi-stage builds are a fundamental best practice for production Docker images due to their critical benefits:

- **Enhanced Security**: By strictly including only runtime necessities, the attack surface is significantly reduced. Fewer installed packages, libraries, and tools mean fewer potential vulnerabilities for attackers to exploit.
- **Faster Deployment and Scaling**: Smaller images transfer more quickly across networks, leading to faster build times, quicker deployments, and more responsive scaling operations in orchestrated environments like Kubernetes.
- **Reduced Resource Consumption**: Smaller images consume less disk space on host machines and registries, which can lead to cost savings, especially at scale. They may also have a smaller memory footprint due to fewer loaded libraries.
- **Improved Cache Utilization**: Docker layers are cached. By structuring your Dockerfile with multi-stage builds, you can optimize layer caching more effectively. For instance, changes to your application code won't necessarily invalidate the `npm ci --only=production` layer in the runtime stage, speeding up rebuilds.
- **Clearer Separation of Concerns**: The Dockerfile explicitly separates the "how to build" from the "how to run," making the build process more transparent and easier to maintain.



🧠 `Important:` Always use minimal base images (e.g., `alpine`, `slim`) for your runtime stage. For compiled static binaries, `scratch` is the ultimate minimal base image.

## Common Issues & Solutions

1. **Application Fails to Start Due to Missing Files:**
    - **Issue:** The container starts, but the application immediately exits or logs "file not found" errors. This indicates that a critical file or directory required at runtime was not copied from the `builder` stage to the final `runtime` stage.
    - **Debugging:** Carefully examine your `COPY --from=builder` commands in the final stage. Common culprits include missing `config` directories, static assets, or even the main application entry point script. Check the application logs (`docker logs <container_id>`) for specific file path errors.
    - **Solution:** Ensure every file or folder necessary for the application's runtime is explicitly copied. For example, if your application has a `public` folder for static files, ensure `COPY --from=builder /app/public ./public` is present.
2. **Image Size Still Larger Than Expected:**
    - **Issue:** Despite using multi-stage builds, the final image size isn't as small as anticipated.
    - **Debugging:**
        - **Base Image Choice:** Are you using a minimal base image for your runtime stage (e.g., `node:20-alpine` vs. `node:20`)?
        - **Over-copying:** Are you copying too much from the `builder` stage? A common mistake is `COPY --from=builder /app .` instead of being selective (e.g., `COPY --from=builder /app/dist ./dist`, `COPY --from=builder /app/node_modules ./node_modules`).
    - **Solution:**
        - Always default to `alpine` or `slim` variants for your runtime base image.
        - Be as granular as possible with `COPY --from` commands, only including truly essential runtime artifacts.

3. **Dependency Installation Issues in Runtime Stage (`--only=production`):**
    - **Issue:** `npm ci --only=production` (or equivalent for other package managers) fails in the final stage, often due to missing system libraries required by a production dependency with native bindings.
    - **Debugging:** This typically happens when a production dependency requires compilation (e.g., `node-sass`, `sqlite3`) and the minimal `alpine` base image lacks necessary build tools like `gcc`, `g++`, or `make`.
    - **Solution:**
        - **Pre-build in Builder:** The best approach is to ensure any native dependencies are fully built and linked in the `builder` stage, and then copy only the resulting compiled modules (e.g., the `node_modules` directory) to the runtime stage.
        - **Temporary Install**: If pre-building is complex, you might temporarily install build tools in the runtime stage before `npm ci --only=production`, then immediately remove them (`apk del ...`) in the same `RUN` command to avoid increasing final image size.
        - **Less Minimal Base Image**: As a last resort, if native dependencies are unavoidable and complex, consider a slightly less minimal base image (e.g., `node:20-slim`) that might include more common system libraries.

---

## Summary & Next Step

You've successfully implemented multi-stage builds, a cornerstone of production-ready Docker image optimization. Your application's Docker image is now significantly smaller, more efficient, and inherently more secure by shedding unnecessary build-time components. This practice is crucial for maintainable and performant containerized applications.

We've verified the size reduction and confirmed the application still functions correctly within its optimized container. This lean image is now better prepared for deployment to any environment.

Next, we'll shift our focus to securing sensitive information. In the upcoming chapter, we'll explore how to handle environment variables and secrets effectively, ensuring that credentials and confidential data are managed securely within your Dockerized application stack.

---

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

---

## References


- Docker Documentation: [https://docs.docker.com/](https://docs.docker.com/)
- Docker Multi-stage builds: [https://docs.docker.com/build/building/multi-stage/](https://docs.docker.com/build/building/multi-stage/)
- Node.js Docker Official Image: [https://hub.docker.com/_/node](https://hub.docker.com/_/node)
- npm ci documentation: [https://docs.npmjs.com/cli/commands/npm-ci](https://docs.npmjs.com/cli/commands/npm-ci)

CHAPTER 10

# Securing Containers with Non-Root Users and Resource Limits

Running applications in production demands not just functionality but also robust security and stable performance. A common oversight in container deployments is operating services with excessive privileges or without proper resource constraints. This can turn a minor vulnerability into a critical system compromise or a simple traffic spike into a cascading outage.

In this chapter, we'll implement two fundamental production best practices for Docker containers: running services as non-root users and defining explicit CPU and memory limits. These measures significantly reduce your application's attack surface and ensure predictable resource consumption, making your multi-service stack more resilient.

By the end of this milestone, you will have updated your application's `Dockerfile` and `docker-compose.yml` to incorporate these hardening techniques. We'll verify these changes by inspecting container user IDs and observing resource usage, confirming that your services are operating under the principle of least privilege and with controlled stability.

---

## Project Overview

Our overarching goal is to build a production-ready, multi-service web application stack using Docker and Docker Compose. Each chapter focuses on a critical aspect of this journey, from initial containerization to advanced deployment and security patterns. This chapter specifically addresses critical security and stability concerns by modifying existing service configurations.

---

## Tech Stack

For this chapter, we will primarily interact with:

- **Docker Engine:** The core platform for running containers. (Latest stable release: unknown as of 2026-05-22)
- **Docker Compose:** Used to define and run multi-container Docker applications. We adhere to the Compose Specification. (Recommended practice is to omit the `version` field in `docker-compose.yml` files for compatibility with the latest Compose Specification, as of 2026-05-22).

- **Dockerfile:** Defines the steps to build our application's container images.
- **Python/Flask Application:** (Our example web service) The application code running inside the container.
- **PostgreSQL Database:** (Our example database service) A widely used relational database.

---

## Planning & Design for Container Hardening

The core principles guiding our work in this chapter are **least privilege** for security and **resource isolation** for stability.

### Non-Root User Strategy: Reducing the Attack Surface

By default, processes inside a Docker container run as the `root` user. This is a significant security risk. If an attacker exploits a vulnerability in a root-run service, they could gain root access within the container, potentially escalating privileges to the Docker host if certain dangerous configurations are present. Running as a non-root user isolates the application, ensuring that even if compromised, the attacker's capabilities are severely limited.

Our approach involves:

1. **Dedicated User and Group Creation:** Within the `Dockerfile`, we'll create a new system user and group specifically for our application. System users are ideal for services as they lack login shells and home directories, further reducing exposure.
2. **Explicit File Permissions:** We'll ensure that any directories or files the application needs to write to (e.g., logs, temporary files) are explicitly owned by or writable by this new non-root user. Application code itself typically only needs read access.
3. **User Context Switch:** The `USER` instruction in the `Dockerfile` will switch the user context for all subsequent commands and the container's runtime to this non-root user.

### Resource Limits Strategy: Ensuring Stability and Predictability

Uncontrolled resource consumption is a common culprit for instability in multi-service environments. A single misbehaving container can consume all available CPU or memory, starving other services or even the host system. This leads to performance degradation, unresponsiveness, or outright crashes. Docker Compose provides mechanisms to define explicit resource limits, enforcing resource isolation.

Our approach involves:

1. **CPU Limits (`cpus`):** We will allocate a specific fraction or multiple of CPU cores to prevent a service from monopolizing processing power. This ensures other services and the host have sufficient CPU cycles.
2. **Memory Limits (`memory`):** We will set a maximum amount of RAM a container can consume. This prevents memory leaks or high-load spikes from causing out-of-memory (OOM) errors that affect the entire system.

These limits are critical for fair resource distribution, preventing cascading failures, and enabling better capacity planning within your multi-service stack.

---

## Milestones and Build Plan

We will tackle the container hardening in a structured manner:

1. **Modify Web Service Dockerfile:** Add instructions to create a non-root user (`appuser`) and switch to it.
2. **Rebuild Web Service Image:** Build the new image with the non-root user configuration.
3. **Verify Non-Root User:** Start the web service and confirm it's running as `appuser`.
4. **Modify Docker Compose Configuration:** Add CPU and memory resource limits to both the `web` and `db` services in `docker-compose.yml`.
5. **Restart Services:** Bring up the entire stack with the new resource limits applied.
6. **Verify Resource Limits:** Use `docker stats` and `docker inspect` to confirm limits are active.

---

## Step-by-Step Implementation

We'll apply these hardening techniques to our example web application (a Python Flask app) and its PostgreSQL database service.

### 1. Implementing Non-Root Users in the Web Service

First, let's modify the `Dockerfile` for our `web` service. We'll assume your `Dockerfile` is located at `web/Dockerfile`.

Here's the initial (simplified) `Dockerfile` for context:

```
# web/Dockerfile - BEFORE changes
FROM python:3.10-slim-bullseye

WORKDIR /app

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

COPY . .

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

Now, let's add the necessary instructions to create and switch to a non-root user.

```
# web/Dockerfile - AFTER changes
FROM python:3.10-slim-bullseye

# 📌 Key Idea: Define user/group IDs as build arguments for flexibility.
# This allows overriding them at build time if specific host UID/GID mappings
are needed.
ARG UID=1000
ARG GID=1000

# Create a non-root system user and group.
# --system creates an account without a login shell or home directory, ideal
for services.
# --gid and --uid ensure consistent IDs, useful for volume permissions if
mapped to host.
RUN groupadd --system --gid ${GID} appgroup && \
    useradd --system --uid ${UID} --gid ${GID} --no-create-home --shell /sbin/
nologin appuser

WORKDIR /app

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

COPY . .

# 🧠 Important: Ensure necessary directories are owned by the non-root user.
# If files were copied *before* `USER appuser` and `appuser` needs write
access to them,
# you must explicitly change ownership. For read-only application code, this
is often not needed.
# Example if /app/logs needed to be writable:
# RUN mkdir -p /app/logs && chown -R appuser:appgroup /app/logs

# Switch to the non-root user for all subsequent instructions and the
container's runtime.
# This is the critical step for enforcing least privilege.
USER appuser

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

**Explanation of changes:**

- `ARG UID=1000 GID=1000`: We define build arguments for the user and group IDs. Using `1000` is a common default for the first non-root user on many Linux systems, but you can choose others.
- `RUN groupadd ... && useradd ...`: This command creates a new system group `appgroup` and a new system user `appuser`.
  - `--system`: Designates them as system accounts, typically for services, without login capabilities.
  - `--no-create-home`: Prevents creating a home directory for `appuser`.
  - `--shell /sbin/nologin`: Ensures the user cannot log in interactively.
- `USER appuser`: This is the most crucial instruction. It changes the user context for all subsequent `RUN`, `CMD`, and `ENTRYPOINT` instructions to `appuser`. This means your application process will run with the privileges of `appuser`, not `root`.
- **Permissions Note:** The commented `chown` line is a common necessity. If your application needs to write to any directories that were `COPY`ed into the image before the `USER appuser` instruction, those directories will be owned by `root`. You'd need to explicitly change their ownership to `appuser:appgroup` before switching the user, otherwise, your application might hit "Permission denied" errors. For a typical web application that only reads its code and writes to a specific `/var/log` or `/tmp` directory, this might not be needed for `/app` itself.

## 2. Rebuilding and Verifying the Web Service

After modifying the `Dockerfile`, rebuild the `web` service image:

```
docker-compose build web
```

Now, let's start only the `web` service and verify the user context.

```
docker-compose up -d web
```

To verify the user inside the running container:

```
# Replace <web_service_container_id> with the actual ID from 'docker ps'
docker exec -it <web_service_container_id> whoami
# Expected output: appuser

docker exec -it <web_service_container_id> id
```

```
# Expected output similar to: uid=1000(appuser) gid=1000(appgroup)
groups=1000(appgroup)
```

If you see `root` or a different user, double-check your `Dockerfile` for the `USER appuser` instruction and ensure it's placed correctly.

### 3. Implementing Resource Limits in Docker Compose

Next, we'll add resource limits to our `docker-compose.yml` file. These are defined under the `deploy.resources.limits` section for each service. Remember, we are using the latest Compose Specification, so no `version` field is needed.

Locate your `docker-compose.yml` file in the project root.

Here's a simplified version of your `docker-compose.yml` before changes:

```
# docker-compose.yml - BEFORE changes (simplified)
services:
  web:
    build: ./web
    ports:
      - "8000:8000"
    environment:
      DATABASE_URL: postgres://user:password@db:5432/mydatabase
    networks:
      - app_network

  db:
    image: postgres:13
    environment:
      POSTGRES_DB: mydatabase
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    volumes:
      - db_data:/var/lib/postgresql/data
    networks:
      - app_network

networks:
  app_network:
    driver: bridge

volumes:
  db_data:
```

Now, let's add resource limits for both the `web` and `db` services.

```
# docker-compose.yml - AFTER changes
services:
  web:
    build: ./web
    ports:
      - "8000:8000"
```

```
    environment:
      DATABASE_URL: postgres://user:password@db:5432/mydatabase
    networks:
      - app_network
    # 🔥 Optimization / Pro tip: Add resource limits for the web service
    deploy:
      resources:
        limits:
          cpus: '0.5' # Limit to 50% of a CPU core (e.g., half a core)
          memory: 256M # Limit to 256 MB of RAM
        # Optional: Set reservations for guaranteed resources.
        # Docker will try to ensure these resources are available even under
contention.
        # reservations:
        #   cpus: '0.25' # Guarantee 25% of a CPU core
        #   memory: 128M # Guarantee 128 MB of RAM

  db:
    image: postgres:13
    environment:
      POSTGRES_DB: mydatabase
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    volumes:
      - db_data:/var/lib/postgresql/data
    networks:
      - app_network
    # 🔥 Optimization / Pro tip: Add resource limits for the database service
    deploy:
      resources:
        limits:
          cpus: '1.0' # Limit to 1 full CPU core
          memory: 1G # Limit to 1 GB of RAM
        # Optional: Set reservations for guaranteed resources
        # reservations:
        #   cpus: '0.5'
        #   memory: 512M

networks:
  app_network:
    driver: bridge

volumes:
  db_data:
```

**Explanation of changes:**

- `deploy`: This top-level key under a service allows you to configure deployment-related parameters, including resources.
- `resources.limits.cpus`: Specifies the maximum CPU share a container can utilize.
  - `'0.5'` means the container can use up to 50% of one CPU core.
  - `'1.0'` means one full CPU core. Values can be fractions or integers.

- `resources.limits.memory`: Specifies the maximum amount of RAM the container can consume.
    - `256M` means 256 megabytes.
    - `1G` means 1 gigabyte.
- `resources.reservations` (Optional): While `limits` are hard caps that prevent a container from exceeding resources, `reservations` define the guaranteed minimum amount of resources a container will receive. This is particularly useful in environments with resource contention, ensuring your critical services always have a baseline. We've included them as comments for awareness.

### 4. Applying and Verifying Resource Limits

Now, restart your services to apply the new resource limits. The `--build` flag ensures our updated `Dockerfile` for the `web` service is used, and all services are recreated with the new `deploy` configurations.

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

#### Verification of Resource Limits

To check if the resource limits are applied and observe real-time usage, use the `docker stats` command. This provides a live stream of resource usage for your running containers.

```
docker stats
```

You should see output similar to this, with your configured limits visible under the `MEM USAGE / LIMIT` column:

```
CONTAINER ID   NAME                 CPU %     MEM USAGE / LIMIT     MEM %
NET I/O     BLOCK I/O   PIDS
abcdef123456   yourproject-web-1    0.12%     12.34MiB / 256MiB     4.82%
1.23kB / 0B   0B / 0B     7
fedcba654321   yourproject-db-1     0.34%     80.56MiB / 1GiB       7.87%
2.34kB / 0B   0B / 0B     15
```

Observe the `MEM USAGE / LIMIT` column to confirm your configured memory limits (e.g., `256MiB` for `web`, `1GiB` for `db`). The `CPU %` will fluctuate but will be constrained by the `cpus` limit if the container attempts to exceed it.

You can also inspect a specific service's configuration at a lower level:

```
# Replace <web_service_container_id> with the actual ID from 'docker ps'
docker inspect <web_service_container_id> | grep -E "CpuPeriod|CpuQuota|
Memory"
```

This command will display the low-level Docker Engine configurations that reflect the Compose limits. For example, `CpuPeriod` and `CpuQuota` relate to CPU limits, and `Memory` relates to the memory limit. These values are typically in microseconds and bytes, respectively.

---

## Production Considerations

### Enhanced Security Posture

Running containers as non-root users is a cornerstone of modern container security. It adheres to the principle of **least privilege**, drastically reducing the "blast radius" if a containerized application is compromised. An attacker gaining control of a non-root process has significantly fewer privileges to exploit the host or other containers compared to one running as `root`.

### Stable Resource Management

Resource limits are not merely for preventing runaway processes; they are fundamental for robust capacity planning and maintaining the stability of your entire system. By setting appropriate limits:

- **Prevent Resource Exhaustion:** A single misbehaving application or an unexpected surge in load won't consume all host resources, preventing cascading failures across other critical services.
- **Improve Predictability:** Your services will operate within defined boundaries, leading to more consistent performance and easier troubleshooting.
- **Facilitate Scaling:** Understanding and quantifying the resource requirements of individual services is essential for making informed decisions when scaling your application horizontally or vertically.

### Monitoring and Alerting

While `docker stats` is excellent for immediate verification, production environments demand dedicated monitoring solutions (e.g., Prometheus with Grafana, Datadog, New Relic). These tools allow you to track container resource usage (CPU, memory, I/O) over time, set up alerts for threshold breaches, identify bottlenecks, and fine-tune your limits proactively.

### Fine-tuning Limits and Reservations

The initial resource limits you set are often estimates. It's crucial to continuously monitor your applications under various load conditions (normal usage, peak traffic, stress tests) to fine-tune these limits.

- **Too Strict Limits:** Can lead to performance degradation, application slowdowns, or `OOMKilled` errors even under normal load.
- **Too Generous Limits:** Defeat the purpose of resource isolation and can mask underlying performance issues. Finding the right balance requires iterative testing and observation. Consider using `reservations` for critical services to guarantee a minimum level of performance.

## Common Issues & Solutions

1. **Permission Denied Errors (Non-Root User):**
   - **Issue:** After switching to a non-root user, your application fails to start or crashes during operation with "Permission denied" errors. This often occurs when trying to write to a log file, access configuration, or create temporary files.
   - **Cause:** The new non-root user lacks write permissions to the necessary directories or files.
   - **Solution:** Identify the specific directories or files requiring write access. In your `Dockerfile`, before the `USER appuser` instruction, add `RUN chown -R appuser:appgroup /path/to/writable/directory`. For example, if your app writes logs to `/app/logs`, ensure `/app/logs` is created and owned by `appuser` (`RUN mkdir -p /app/logs && chown -R appuser:appgroup /app/logs`).
   - ⚡ **Real-world insight:** Be precise with `chown`. Granting write access to only what's absolutely necessary further enhances security.
2. **Container Crashes with `OOMKilled` (Resource Limits):**
   - **Issue:** Your container unexpectedly crashes, and `docker logs` or `docker inspect` (under `State.OOMKilled`) indicates an `OOMKilled` status.
   - **Cause:** The memory limit set in `docker-compose.yml` is too low for your application's actual memory requirements, especially under load or during specific operations (e.g., data processing).
   - **Solution:** Increase the `memory` limit for that service in `docker-compose.yml`. Use `docker stats` or detailed monitoring to observe the typical and peak memory usage of your application, then set the limit comfortably above the observed peak to provide a buffer.

3. **Application Slowdowns or Freezing (Resource Limits):**
    - **Issue:** Your application becomes unresponsive or very slow, but doesn't crash. `docker stats` might show its `CPU %` frequently hitting close to its `cpus` limit.
    - **Cause:** The `cpus` limit is too restrictive, and your application genuinely requires more processing power, especially during peak load or computationally intensive tasks.
    - **Solution:** Increase the `cpus` limit for that service in `docker-compose.yml`. Again, continuous monitoring of application performance and CPU usage under realistic load is essential to find the right balance between resource isolation and performance.

---

## Summary & Next Step

You've successfully implemented two critical production best practices: running containers as non-root users and setting explicit resource limits. By configuring non-root users, you've significantly reduced the attack surface and potential impact of a container compromise, adhering to the principle of least privilege. Simultaneously, by setting resource limits, you've established the foundation for a more stable and predictable production environment, preventing any single service from monopolizing system resources.

Your application stack is now more secure and resilient against common operational pitfalls. Next, we will enhance our application's external accessibility, security, and performance by integrating a reverse proxy with Nginx, further preparing it for real-world deployment.

## References


- Docker Documentation: [https://docs.docker.com/](https://docs.docker.com/)
- Docker Compose file reference: [https://docs.docker.com/compose/compose-file/](https://docs.docker.com/compose/compose-file/)
- Compose Specification (recommended to avoid `version` field): [https://docs.docker.com/compose/compose-file/07-general-params/#version](https://docs.docker.com/compose/compose-file/07-general-params/#version)
- Best practices for writing Dockerfiles: [https://docs.docker.com/develop/develop-images/dockerfile_best-practices/](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/)
- Docker `useradd` command: [https://docs.docker.com/engine/reference/builder/#useradd](https://docs.docker.com/engine/reference/builder/#useradd)


---

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

CHAPTER 11

# Auditing Docker Host and Containers with docker-bench-security

Securing your containerized applications isn't just about writing secure code; it's also about ensuring the underlying Docker host and its runtime environment are configured securely. In this chapter, we'll shift our focus to proactive security by auditing our Docker setup using `docker-bench-security`. This tool helps validate your Docker installation against the best practices outlined in the CIS Docker Benchmark.

By the end of this chapter, you'll be able to run a comprehensive security audit on your Docker environment, understand its findings, and begin to implement the necessary remediations. This is a critical step in hardening your production deployments and maintaining a strong security posture.

---

## Project Overview: Hardening the Docker Environment

Our goal in this chapter is to proactively identify and address potential security weaknesses in our Docker environment. We will use an industry-recognized tool to scan our Docker host and running containers, comparing their configurations against established security benchmarks. This isn't about finding bugs in our application code, but rather misconfigurations in the platform that hosts it.

The outcome of this chapter is a detailed security report for your Docker setup, highlighting areas for improvement. You'll gain practical experience in interpreting these reports and understanding the implications of various security recommendations, ultimately making your multi-service application stack more resilient to attacks.

## Tech Stack: `docker-bench-security` and CIS Docker Benchmark

The core of our security auditing process relies on two key components:

- **`docker-bench-security`**: An open-source script developed by Docker that automates security checks. It's designed to be run on a Docker host and inspects the daemon, host configuration, and running containers. It outputs results categorized as `PASS`, `INFO`, or `WARN`.
- **CIS Docker Benchmark**: A comprehensive set of guidelines from the Center for Internet Security (CIS) that provides prescriptive recommendations for securely configuring Docker. `docker-bench-security` uses these guidelines as its reference.

This combination allows us to leverage expert-defined best practices without manually checking hundreds of configuration items.

---

## Milestones: Auditing Workflow

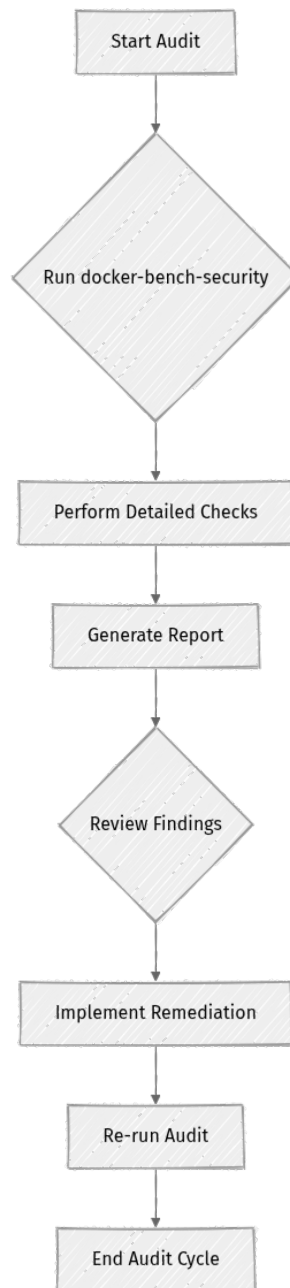
To effectively audit our Docker environment, we'll follow a structured workflow:

1. **Prepare the Environment**: Ensure Docker is running and we have access to the necessary tools.
2. **Run the Audit Tool**: Execute `docker-bench-security` as a privileged container to perform comprehensive checks.
3. **Analyze the Report**: Interpret the `PASS`, `INFO`, and `WARN` messages generated by the tool.
4. **Prioritize Remediations**: Identify critical security findings and understand their impact.
5. **Plan for Continuous Auditing**: Discuss how to integrate these checks into a production workflow.

---

## Architecture: Security Audit Process Flow

The auditing process involves `docker-bench-security` interacting with various components of your Docker host and daemon to gather configuration information.




The `docker-bench-security` container requires specific host privileges and volume mounts to access the necessary files and sockets for a thorough audit. This allows it to inspect the Docker daemon's configuration, `/etc` files for host-level settings, and `/var/lib` for Docker's internal data.

## Step-by-Step Implementation: Running docker-bench-security

`docker-bench-security` is a shell script, and the recommended way to run it is as a privileged container. This ensures it has the necessary access to inspect your Docker daemon and host configuration while being isolated from your host system.

**Prerequisites:**

- A Docker Engine installation (we're using the setup from previous chapters).
- Internet access to pull the `aquasec/docker-bench` image.
- Basic command-line proficiency.

### Method 1: Running as a Container (Recommended)

Running `docker-bench-security` as a container is generally preferred as it isolates the tool's dependencies and provides a consistent execution environment.

First, ensure your Docker daemon is running and you have no critical workloads that would be impacted by a momentary resource spike from the audit.

Open your terminal and execute the following command:

```
docker run --net host --pid host --userns host --cap-add audit_control \
    -e DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_TRUST \
    -v /var/lib:/var/lib \
    -v /var/run/docker.sock:/var/run/docker.sock \
    -v /etc:/etc --label docker_bench_security \
    --read-only --tmpfs /tmp \
    aquasec/docker-bench:latest
```

Let's break down these flags:

- `docker run`: Executes a Docker container.
- `--net host`: Allows the container to share the host's network namespace, necessary for some network-related checks.
- `--pid host`: Allows the container to share the host's PID namespace, enabling checks on host processes.
- `--userns host`: Necessary for user namespace related checks, especially important for host user/group mappings.

- `--cap-add audit_control`: Grants the container the `audit_control` capability, allowing it to interact with the kernel's audit subsystem to perform deeper security checks.
- `-e DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_TRUST`: Passes the `DOCKER_CONTENT_TRUST` environment variable from your host to the container. If you have Docker Content Trust enabled, this ensures the check is performed correctly.
- `-v /var/lib:/var/lib`: Mounts the `/var/lib` directory from the host. This contains Docker's data directory, including image layers, volumes, and container metadata.
- `-v /var/run/docker.sock:/var/run/docker.sock`: Mounts the Docker daemon's Unix socket. This is crucial for the tool to communicate with the Docker daemon and perform checks on running containers and daemon configuration.
- `-v /etc:/etc`: Mounts the `/etc` directory, allowing the tool to check host configuration files relevant to Docker and general system security.
- `--label docker_bench_security`: Adds a label to the container for easy identification if you need to inspect it later.
- `--read-only`: Makes the container's root filesystem read-only, enhancing the security of the audit tool itself.
- `--tmpfs /tmp`: Mounts `/tmp` as a `tmpfs` (in-memory filesystem), preventing any persistent writes to disk by the audit tool.
- `aquasec/docker-bench:latest`: Specifies the Docker image to use. The `aquasec` organization maintains a regularly updated image. As of 2026-05-22, `latest` refers to the most recent stable release, ensuring you use an up-to-date benchmark.

### Method 2: Running from a Cloned Repository (Alternative)

If you prefer to run the script directly, you can clone the repository. This might be useful for development, or if you need to inspect or modify the script itself.

1. **Clone the repository:**

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

1. **Run the script:**

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

```
You need `sudo` because the script performs checks that require root
privileges to access system files and Docker daemon information.

**Note:** When running directly, ensure your environment has all necessary
dependencies (e.g., `auditctl` for some audit checks). The containerized
approach (Method 1) handles these dependencies for you by including them in
the `aquasec/docker-bench` image.
```

---

## Testing & Verification: Analyzing the Audit Report

Once the command executes, you'll see a stream of output in your terminal. This output is the audit report, categorizing findings as `INFO`, `WARN`, or `PASS`.

Let's look at an example snippet of what you might see:

```
[INFO] 1  - Host Configuration
[INFO] 1.1 - Linux Host Configuration
[INFO] 1.1.1  - Ensure a separate partition for containers has been created
(Not Scored)
[INFO]      * Recommendation: Create a separate partition for /var/lib/docker
and /var/log/docker.
[INFO] 1.1.2  - Ensure the host's kernel is updated (Not Scored)
[INFO]      * Recommendation: Ensure the host's kernel is updated to the
latest version.
...
[WARN] 2  - Docker Daemon Configuration
[WARN] 2.1  - Docker daemon configuration
[WARN] 2.1.1  - Ensure the docker.service file ownership is set to root:root
(Scored)
[WARN]      * Expected value: root:root
[WARN]      * Actual value: user:user (example if misconfigured)
[INFO]      * Remediation: systemctl daemon-reload && systemctl restart docker
...
[PASS] 4  - Container Images, Build, and Registry
[PASS] 4.1  - Container Image and Build File
[PASS] 4.1.1  - Ensure a `HEALTHCHECK` instruction has been added to the
container image (Scored)
...
[INFO] Overall results:
[INFO] 18 checks PASS
[INFO] 5 checks WARN
[INFO] 2 checks INFO
```

### Interpreting the Output:

- `[PASS]`: The check passed, meaning your configuration adheres to the benchmark recommendation.

- `[INFO]`: These are informational messages, often about checks that are "Not Scored" in the benchmark, or recommendations that are good practices but not critical failures. You should still review them as they often point to general hardening opportunities.
- `[WARN]`: These are critical findings. They indicate a deviation from best practices that could pose a significant security risk. Each `WARN` typically includes a `Remediation` suggestion. These are your top priority for action.

### Verification Steps:

1. **Review the Entire Report**: Carefully scroll through the entire output. Don't just look at the summary.
2. **Focus on `[WARN]` Messages**: Identify each specific warning, noting the benchmark item number (e.g., `2.1.1`) and the exact recommendation.
3. **Understand the Remediation**: For each `[WARN]`, read the suggested `Remediation` and understand why it's recommended. For example, a warning about `docker.service` file ownership (`2.1.1`) indicates that non-root users might be able to tamper with Docker's core configuration.
4. **Prioritize Remediations**: Not all `WARN` messages have the same severity in every context. Use your judgment based on your application's risk profile and compliance requirements. An exposed Docker socket (`2.3`) is generally a higher priority than minor file permission issues on less critical files.
5. **Consider `Not Scored` Items**: Even if they don't generate `WARN`s, `INFO` messages for `Not Scored` items (like `1.1.1 Ensure a separate partition for containers`) are important best practices to consider for production environments.

By systematically reviewing these findings, you gain a clear picture of your Docker environment's current security posture and a concrete action plan for improvement.

## Production Considerations: Continuous Security Auditing

Running `docker-bench-security` once is a good start, but continuous security is paramount in production. Security posture can degrade over time due to configuration drift, new software installations, or emerging vulnerabilities.

- **Automate Audits in CI/CD**: Integrate `docker-bench-security` into your CI/CD pipeline. This can be a nightly job or triggered before significant deployments. If the audit fails (e.g., introduces new `WARN`s above a threshold), it can block the deployment, enforcing a secure baseline.
- **Scheduled Scans on Production Hosts**: Schedule `docker-bench-security` to run periodically on your production hosts (e.g., weekly or monthly) via a cron job or an orchestration tool. Send reports to a central logging or alerting system. This helps detect configuration drift or new vulnerabilities that emerge.
- **Baselining and Trend Analysis**: Establish a "clean" audit report as your security baseline. Any new `WARN`s or significant changes in subsequent scans indicate a deviation in your security posture that needs immediate investigation. Track these trends over time.
- **Alerting on Critical Findings**: Configure alerts for critical `WARN` findings. For example, if a check related to Docker daemon permissions changes from `PASS` to `WARN`, an alert should notify the security or operations team immediately via email, Slack, or PagerDuty.
- **Documentation and Risk Acceptance**: Document your security posture, all remediations, and any accepted risks. If a `WARN` cannot be fully remediated due to specific operational requirements (e.g., a necessary custom kernel module), document the justification, any compensating controls, and get appropriate sign-off.
- **Keep the Benchmark Current**: The CIS Docker Benchmark is updated periodically to reflect new threats and best practices. Ensure you are using a recent version of `docker-bench-security` and review new recommendations as they emerge.

## Common Issues & Solutions

1. **Permission Denied Errors when Running Directly:**
    - **Issue:** When running the script directly (Method 2), you might encounter permission denied errors for certain checks, especially those accessing `/proc` or specific Docker daemon files.
    - **Solution:** Ensure you are running the script with `sudo` privileges. The containerized approach (Method 1) typically handles this by mounting the necessary host paths and capabilities, abstracting away the direct `sudo` requirement for the script itself.
2. **Overwhelming Output / Difficulty in Parsing:**
    - **Issue:** The default output can be very verbose, making it hard to pinpoint critical issues quickly, especially in automated environments.
    - **Solution:** The `docker-bench-security` script supports an output file. You can redirect the output to a file for easier parsing and review:

```
        docker run --net host ... aquasec/docker-bench:latest -o /opt/
audit_report.txt
```

```
    Then, you can use `docker cp` to copy the report out of the container:
```

```
        docker cp <container_id_or_name>:/opt/audit_report.txt .
```

```
    Alternatively, you can use `grep` to filter for `WARN` messages directly
in the terminal:
```

```
docker run --net host ... aquasec/docker-bench:latest | grep WARN
```

1. **"False Positives" or Irrelevant Warnings:**
    - **Issue:** Some `WARN` messages might not be applicable to your specific setup or might be considered acceptable risks given your unique operational requirements. For example, if you intentionally expose the Docker API to a trusted internal network with strong firewall rules, a `WARN` about the exposed socket might be deemed an acceptable risk.
    - **Solution:** Understand why a check is warning. Don't blindly disable checks without understanding the security implications. If a warning is genuinely not applicable or the risk is accepted, document the justification, compensating controls, and get appropriate sign-off from security stakeholders. This process is known as risk acceptance.

---

## Summary & Next Step

In this chapter, we've taken a crucial step towards securing our Docker deployments by learning how to audit the Docker host and containers using `docker-bench-security`. We explored how to run the tool, interpret its output based on the CIS Docker Benchmark, and discussed how to integrate these audits into a production workflow for continuous security.

By proactively addressing the `WARN` messages and considering the `INFO` recommendations, you significantly improve the security posture of your container environment. Remember, security is an ongoing process, not a one-time fix. Regular audits and prompt remediation are essential for maintaining a robust, production-ready system.

You now have a solid foundation for deploying a multi-service application with Docker Compose, complete with image optimization, persistent data, environment management, health checks, and a practical understanding of security auditing.

Next, we'll focus on the final steps of preparing for deployment, including configuring resource limits and exploring more advanced deployment strategies to ensure our application runs efficiently and reliably at scale.

## References


- Docker Documentation: [https://docs.docker.com/](https://docs.docker.com/)
- CIS Docker Benchmark: httpswww.cisecurity.org/benchmark/docker
- `docker-bench-security` GitHub Repository: [https://github.com/docker/docker-bench-security](https://github.com/docker/docker-bench-security)
- Compose Specification Versioning: [https://github.com/jamesatdocker/docker-docs/blob/main/compose/compose-file/compose-versioning.md](https://github.com/jamesatdocker/docker-docs/blob/main/compose/compose-file/compose-versioning.md)


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

CHAPTER 12

# Finalizing the Production Stack and Deployment Considerations

## Finalizing the Production Stack and Deployment Considerations

Welcome to the final chapter of our Docker Compose journey! So far, we've built a multi-service application, managed data, handled secrets, and implemented health checks. These are crucial steps, but moving from a development setup to a production-ready system requires a deeper look into operational hardening.

In this chapter, we will refine our Docker Compose stack to meet production standards. This involves configuring resource limits, enhancing logging, and performing security audits. By the end, you'll have a more robust and observable application stack, ready for real-world deployment considerations. We'll also discuss the boundaries of Docker Compose and where dedicated orchestration tools become necessary.

## Project Overview: Hardening for Production

Our goal in this chapter is to transform our functional Docker Compose application into a more resilient, secure, and observable stack suitable for a single-host production environment. This involves applying a series of best practices that address common operational challenges.

By the end of this chapter, your Docker Compose application will feature:

- **Resource Governance:** CPU and memory limits applied to prevent resource exhaustion.
- **Intelligent Logging:** Configured log rotation to manage disk space and prepare for centralized logging.
- **Enhanced Security:** A read-only root filesystem for containers and an audit of your Docker environment using `docker-bench-security`.
- **Deployment Awareness:** An understanding of backup strategies, monitoring, and CI/CD integration for Docker Compose.

This final refinement will equip you with a robust foundation for deploying containerized applications, even if you eventually transition to more complex orchestration platforms.

---

## Milestones: Production Hardening Checklist

To achieve our production-ready state, we'll work through the following milestones:

1. **Configure Resource Limits:** Add CPU and memory limits to our services in `docker-compose.yml`.
2. **Implement Log Rotation:** Set up `json-file` logging driver options for local log file management.
3. **Enable Read-Only Filesystems:** Mark container filesystems as read-only for enhanced security.
4. **Audit Docker Security:** Use `docker-bench-security` to check the Docker host and containers against security benchmarks.
5. **Review Deployment Strategy:** Discuss considerations for deploying, updating, and backing up Docker Compose applications.

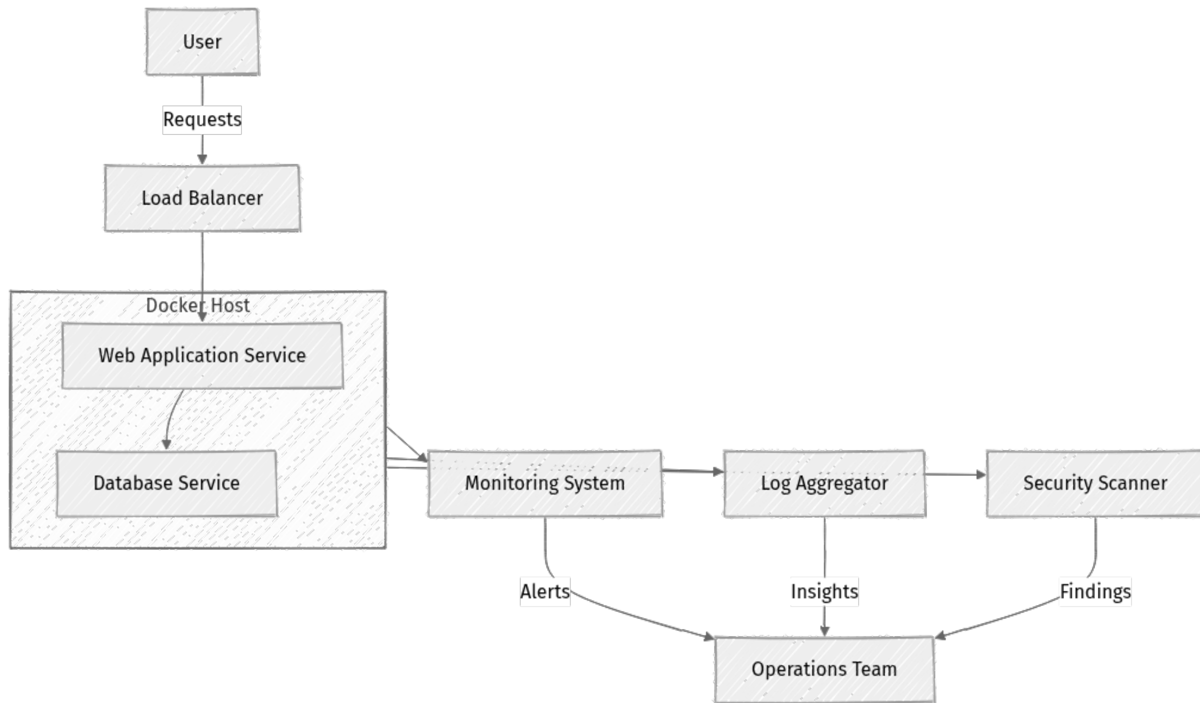
---

## Architecture: Enhanced Production Stack

Transitioning an application to production means anticipating and mitigating failures, ensuring security, and gaining visibility into its operations. Our goal is to make our Docker Compose stack more resilient, secure, and easier to monitor.

While Docker Compose is excellent for defining and running multi-container applications on a single host, it's not a full-fledged orchestrator like Kubernetes or Docker Swarm. Therefore, our "deployment considerations" will focus on preparing the stack for a single-host production environment or as a stepping stone to a larger orchestration system.

Here's a high-level view of our enhanced production stack, showing how our local Docker Compose setup fits into a broader operational context:




This diagram illustrates how our Docker Compose stack (Web Application and Database services) resides on a Docker Host. External components like a Load Balancer, Monitoring System, and Log Aggregator interact with it, while a Security Scanner directly inspects the host and containers. This broader view emphasizes the surrounding systems needed for a truly production-grade setup.

---

## Step-by-Step Implementation

Let's apply these production-grade configurations to our `docker-compose.yml` file and the Docker environment. Remember that the current best practice for `docker-compose.yml` is to omit the explicit `version` field, as Docker Compose now adheres to the Compose Specification.

### 1. Setting Resource Limits

Uncontrolled resource consumption can lead to a "noisy neighbor" problem or even bring down the entire host. Docker Compose allows us to set CPU and memory limits for each service. This helps prevent a single runaway container from consuming all host resources.

Open your `docker-compose.yml` file (or `docker-compose.production.yml` if you created a separate file for production) and add a `deploy` section with `resources.limits` for your services.

**File:** `docker-compose.yml`

```yaml
# docker-compose.yml

services:
  webapp:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "80:80"
    environment:
      # ... existing environment variables
    healthcheck:
      test: ["CMD-SHELL", "curl -f http://localhost/health || exit 1"]
      interval: 30s
      timeout: 10s
      retries: 3
    deploy: # This section applies resource limits
      resources:
        limits:
          cpus: '0.5' # Limit to 50% of one CPU core
          memory: 256M # Limit to 256 megabytes of RAM
    # ... other webapp configurations

  database:
    image: postgres:16-alpine # Using 16-alpine, latest as of 2026-05-22
    environment:
      POSTGRES_DB: ${DB_NAME}
      POSTGRES_USER: ${DB_USER}
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    volumes:
      - db_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${DB_USER} -d ${DB_NAME}"]
      interval: 10s
      timeout: 5s
      retries: 5
    deploy: # This section applies resource limits
      resources:
        limits:
          cpus: '0.25' # Limit to 25% of one CPU core
          memory: 512M # Limit to 512 megabytes of RAM
    # ... other database configurations

volumes:
  db_data:
```

**Explanation of Decisions:**

- `deploy` **Key:** While primarily used by Docker Swarm, Docker Compose respects the `deploy.resources.limits` configuration for standalone deployments.
- `cpus`**:** This limits the CPU cycles a container can consume. `'0.5'` means the container can use up to 50% of a single CPU core. This prevents a busy container from starving other processes on the host.

- **`memory`:** This sets a hard limit on the RAM available to the container. `256M` for the web app and `512M` for the database are initial estimates.
- **Tradeoff:** Setting limits too low can cause your application to perform poorly or crash due to resource starvation (e.g., Out Of Memory errors). It's crucial to start with reasonable estimates and refine them based on actual application profiling under various load conditions.

## 2. Configuring Logging Drivers for Rotation

By default, Docker uses the `json-file` logging driver, which can lead to large log files filling up disk space over time. For production, you typically want to configure log rotation to manage file sizes or send logs to an external aggregator.

We'll configure the `json-file` driver to rotate logs locally. This is a good intermediate step to prevent disk overflow before integrating with a full centralized logging solution (like ELK stack, Splunk, or cloud-native logging services).

**File:** `docker-compose.yml`

```yaml
# docker-compose.yml

services:
  webapp:
    # ... existing webapp configurations
    logging: # Configure logging for the webapp service
      driver: "json-file"
      options:
        max-size: "10m" # Max size of the log file before rotation
        max-file: "5"   # Max number of log files to keep
    deploy:
      # ... existing deploy configurations

  database:
    # ... existing database configurations
    logging: # Configure logging for the database service
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "5"
    deploy:
      # ... existing deploy configurations

volumes:
  db_data:
```

**Explanation of Decisions:**

- **`logging.driver`:** Specifies which logging driver Docker should use. `json-file` is the default, but we're explicitly configuring its options here. Docker supports various drivers, including `syslog`, `journald`, `gelf` (for Graylog), `awslogs`, and others for integration with external systems.
- **`max-size`:** When a log file reaches this size (e.g., `10m` for 10 megabytes), Docker rotates it, creating a new log file.
- **`max-file`:** This defines the maximum number of rotated log files to keep (e.g., `5`). Once this limit is reached, the oldest log file is deleted to make space for a new one.
- **Real-world insight:** While local log rotation helps prevent disk exhaustion, for true production observability, you'd centralize logs. This allows for easier searching, analysis, and correlation across multiple services and hosts.

### 3. Implementing a Read-Only Root Filesystem

For increased security, you can make a container's root filesystem read-only. This is a powerful security control that prevents malicious actors or buggy processes from writing to arbitrary locations within the container's filesystem (except for explicitly mounted volumes).

This requires careful planning: your application must be designed to write only to designated volumes, not to its own image layers or temporary internal paths.

**File:** `docker-compose.yml`

```yaml
# docker-compose.yml

services:
  webapp:
    # ... existing webapp configurations
    read_only: true # Make the container's root filesystem read-only
    logging:
      # ... existing logging configurations
    deploy:
      # ... existing deploy configurations

  database:
    # ... existing database configurations
    read_only: true # Make the container's root filesystem read-only
    logging:
      # ... existing logging configurations
    deploy:
      # ... existing deploy configurations

volumes:
  db_data:
```

**Explanation of Decisions:**

- **`read_only: true`:** This setting ensures that the container cannot write to any part of its filesystem that isn't explicitly mounted as a volume. This significantly reduces the attack surface and helps achieve immutability.
- **Important:** Test your application thoroughly with `read_only: true`. Many applications write temporary files, cache data, or logs to their internal filesystem by default. If your application attempts this, it will fail with a "Read-only file system" error. Ensure all necessary write operations are directed to Docker volumes. For our simple web app, it should work fine if all data is stored in the database. The database service already uses a volume for its data (`db_data:/var/lib/postgresql/data`), so its persistent writes will continue to function.

### 4. Auditing Docker Security with `docker-bench-security`

`docker-bench-security` is a script that checks your Docker host and containers against best practices defined in the CIS Docker Benchmark. It's a critical tool for identifying potential security vulnerabilities in your Docker setup.

First, ensure you have `git` installed to clone the repository. Then, clone and run the tool.

```
# Clone the docker-bench-security repository
git clone https://github.com/docker/docker-bench-security.git

# Navigate into the directory
cd docker-bench-security

# Run the script. This script requires root privileges to perform
comprehensive checks.
# It will analyze your Docker daemon, host configuration, and running
containers.
sudo sh docker-bench-security.sh
```

**Explanation of the Tool:**

- **CIS Docker Benchmark:** This is a security benchmark provided by the Center for Internet Security (CIS) that offers a prescriptive guide for establishing a secure configuration posture for Docker.
- **`docker-bench-security.sh`:** This script automates checks against this benchmark.

- **Output Interpretation:** The script will output `INFO`, `WARN`, and `PASS` messages.
    - `PASS`: The check passed, indicating a secure configuration.
    - `INFO`: Informational message, often a recommendation that isn't a direct security flaw but could be improved.
    - `WARN`: A potential security vulnerability or a configuration that doesn't follow best practices. **Always pay close attention to these and prioritize addressing them.**
- **Real-world insight:** Regularly running `docker-bench-security` (e.g., as part of a CI/CD pipeline or scheduled job) on your Docker hosts is a crucial security practice. Addressing warnings promptly is key to maintaining a strong security posture.

---

## Testing & Verification

After applying these production hardening changes, it's essential to verify that everything works as expected and that our new configurations are active.

1. **Restart your Docker Compose stack:** First, bring down any existing services to ensure the new `docker-compose.yml` changes are applied. Then, bring them up in detached mode.

```
docker compose down
docker compose up -d
```

1. **Verify Resource Limits:** Use `docker stats` to see the real-time resource usage of your containers. You should observe that they stay within the limits you set, or at least don't exceed them if under load.

```
docker stats
```

```
Look for the `MEM USAGE / LIMIT` and `CPU %` columns. The `LIMIT` column for
memory should reflect your configuration (e.g., `256MiB` for the webapp). For
```

```
CPU, you'll see a percentage, and if it consistently hits 50% for the webapp,
it's hitting its limit.
```

1. **Verify Logging Configuration:** Inspect the log files directly on your Docker host. Docker stores `json-file` logs in `/var/lib/docker/containers/<container_id>/<container_id>-json.log`.

```
    # Find your webapp container ID
    docker ps | grep webapp

    # Navigate to the logs directory (replace <your_webapp_container_id> with
your actual ID)
    # Example path: /var/lib/docker/containers/
a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7r8s9t0/
    sudo ls -lh /var/lib/docker/containers/<your_webapp_container_id>/
```

```
You should see multiple `*.log` files (e.g., `*-json.log`, `*-json.log.1`,
etc.) if enough logs have been generated to trigger rotation. Their sizes
should be capped around `10M`.
```

1. **Verify Read-Only Filesystem:** Try to write a file inside one of your containers. This should fail if `read_only: true` is effective.

```
    # Get into your webapp container
    docker exec -it <your_webapp_container_id> sh

    # Try to create a file in the root filesystem
    touch /test.txt
```

```
You should receive a "Read-only file system" error, confirming the security
measure is active.
```

```
    # Exit the container
    exit
```

1. **Review `docker-bench-security` Output:** Re-run the `docker-bench-security.sh` script and carefully review its output. Specifically, look for any new `WARN` messages related to your container configurations. Addressing these warnings will further harden your setup.

## Production Considerations

While our Docker Compose stack is significantly hardened, deploying to production involves more than just container configuration. These aspects are critical for reliable and maintainable systems.

### Deployment Strategy

For single-host deployments, `docker compose up -d` is often sufficient for initial deployment. However, for updates, consider the following:

- **Downtime:** A simple `docker compose down` followed by `docker compose up -d` will cause downtime as old containers are stopped before new ones are started.
- **Zero-Downtime Updates:** Achieving true zero-downtime deployments with Docker Compose on a single host typically requires manual orchestration (e.g., running new containers on different ports, then switching a reverse proxy to the new containers) or moving to a full orchestrator. Platforms like Docker Swarm or Kubernetes offer built-in rolling updates and service discovery for seamless transitions.
- **Automated Deployment:** Integrate `docker compose up -d` into a CI/CD pipeline for consistent and repeatable deployments to your target host(s).

### Backup and Recovery

Data loss is catastrophic. A robust backup and recovery strategy is non-negotiable.

- **Volume Backups:** Critical for persistent data stored in Docker volumes (like our `db_data` volume). You can back up volumes by running a temporary container that mounts the volume and copies its contents to a host path or cloud storage.

```
    # Example: Backup db_data volume to a local 'backups' directory
    # Ensure the database container is running so the volume is active.
    # Replace 'my_app_database-1' with your actual database container name.
    docker run --rm --volumes-from my_app_database-1 -v $(pwd)/backups:/
backup alpine tar cvf /backup/db_backup_$(date +%F).tar /var/lib/postgresql/
data
```

```
This command creates a tar archive of your database data directory within a
`backups` folder in your current directory.
```

- **Configuration Backups:** Always keep your `docker-compose.yml`, `.env` files, Dockerfiles, and any other configuration files under version control (e.g., Git). This ensures you can rebuild your environment from scratch.

### Monitoring and Alerting

Visibility into your application's health and performance is vital.

- **Host-level Monitoring:** Track key metrics of your Docker host itself: CPU utilization, memory usage, disk I/O, network throughput. Tools like `node_exporter` with Prometheus and Grafana are common for this.
- **Container-level Monitoring:** Collect specific metrics for individual containers, such as application request rates, error rates, latency, and resource consumption. Docker provides `docker stats`, but for production, solutions like cAdvisor, Prometheus, and Grafana offer more depth.
- **Log Aggregation:** Sending all container logs to a centralized system (e.g., ELK Stack, Grafana Loki, Splunk, cloud logging services like AWS CloudWatch or Google Cloud Logging) is crucial. This enables easier searching, analysis, and correlation of events across multiple services.
- **Alerting:** Configure alerts based on critical metrics (e.g., high CPU usage, low disk space, increased application error rates, health check failures) to notify your operations team proactively.

### CI/CD Integration

Integrating your Docker Compose stack into a Continuous Integration/Continuous Deployment (CI/CD) pipeline automates the process of building, testing, and deploying your application.

1. **Build Phase:** The CI pipeline automatically builds Docker images for your application components whenever code changes are pushed.
2. **Test Phase:** Automated tests (unit, integration, end-to-end) are run against the newly built images to ensure code quality and functionality.
3. **Deployment Phase:** Upon successful testing, the CD pipeline uses `docker compose up -d` (or orchestrator commands) to deploy the new images to your staging or production environment. This ensures consistency and reduces human error.

## Common Issues & Solutions

Even with careful planning, production deployments can encounter issues. Here are a few common problems related to the configurations we've implemented:

1. **Container Crashes Due to Resource Limits:**
    - **Issue:** Your container starts, then unexpectedly exits, often with an "Out Of Memory" (OOM) error, or experiences severe performance degradation due to CPU throttling.
    - **Debugging:** Check `docker logs <container_name>` for OOM errors. Use `docker stats` to monitor resource usage and see if containers are hitting their `LIMIT` frequently.
    - **Solution:** Increase `cpus` or `memory` limits in `deploy.resources.limits`. The best way to determine appropriate values is through application profiling and load testing under realistic conditions.
2. **"Read-Only File System" Errors in Production:**
    - **Issue:** After enabling `read_only: true`, your application fails to start or crashes because it attempts to write temporary files, cache data, or logs to its internal filesystem (which is now read-only).
    - **Debugging:** The error message `Read-only file system` will appear in `docker logs <container_name>`. You'll need to identify the specific path the application is trying to write to.
    - **Solution:** Identify where the application needs to write. Mount specific paths as volumes (e.g., `- /tmp:/tmp` for temporary files, or `- ./logs:/app/logs` for application logs) or configure the application to write to an existing volume. For logs, ensure they are directed to `stdout`/`stderr` so Docker's logging driver can capture them.

3. **Ignoring `docker-bench-security` Warnings:**
    - **Issue:** The security script outputs `WARN` messages (e.g., "Host is not configured to limit memory usage," "Container is running as root") but these are ignored, leaving potential vulnerabilities.
    - **Debugging:** Re-run `sudo sh docker-bench-security.sh` and carefully read each `WARN` message.
    - **Solution:** Treat `WARN` messages from `docker-bench-security` as actionable items. Research each warning, understand its implication for your specific environment, and implement the recommended fix. Regular security audits and continuous improvement are crucial for maintaining a strong security posture.

---

## Summary & Next Step

Congratulations! You've successfully built and hardened a multi-service application stack using Docker and Docker Compose, incorporating many production best practices.

In this chapter, we:

- **Configured resource limits** for CPU and memory to prevent resource exhaustion and improve stability.
- **Implemented local log rotation** for better log file management, preventing disk space issues.
- **Enhanced container security** by making filesystems read-only, reducing the attack surface.
- **Learned how to use `docker-bench-security`** to audit our Docker environment against best practices.
- **Discussed broader production considerations** like deployment strategies, backups, monitoring, and CI/CD integration.

You now have a solid foundation for deploying containerized applications. While Docker Compose is excellent for single-host deployments and development, scaling beyond a single machine or requiring advanced orchestration features (like automatic load balancing, self-healing, rolling updates without manual intervention, or complex service discovery) typically necessitates moving to platforms like Docker Swarm or Kubernetes.

Your next step might be to explore these more advanced orchestration platforms to deploy your hardened Docker images at scale. Alternatively, you could delve deeper into specific areas like setting up a centralized logging solution (e.g., Loki with Grafana), implementing advanced monitoring with Prometheus, or integrating your stack into a full-fledged CI/CD pipeline using tools like GitHub Actions, GitLab CI, or Jenkins.

---

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

---

## References


- Docker Documentation: [https://docs.docker.com/](https://docs.docker.com/)
- Compose Specification: [https://docs.docker.com/compose/compose-file/](https://docs.docker.com/compose/compose-file/)
- `docker-bench-security` GitHub Repository: [https://github.com/docker/docker-bench-security](https://github.com/docker/docker-bench-security)
- PostgreSQL Docker Image: [https://hub.docker.com/_/postgres](https://hub.docker.com/_/postgres)
- CIS Docker Benchmark: [https://www.cisecurity.org/benchmark/docker](https://www.cisecurity.org/benchmark/docker)

CHAPTER 13

# Building a Production-Ready Docker Compose Stack

Deploying modern applications effectively requires more than just running code; it demands a robust, secure, and maintainable infrastructure. This guide will walk you through building a multi-service web application stack using Docker and Docker Compose, applying production-minded practices every step of the way.

## Why Build a Production-Ready Docker Stack?

Production readiness isn't just about functionality; it's about reliability, security, maintainability, and efficiency. In today's cloud-native landscape, containerization with Docker has become a cornerstone for achieving these goals. However, simply containerizing an application isn't enough. You need to understand how to:

- **Secure your containers:** Prevent common vulnerabilities and minimize attack surfaces.
- **Ensure data persistence:** Protect your valuable data across container restarts and updates.
- **Build resilient services:** Implement health checks to automatically manage service availability.
- **Optimize image sizes:** Reduce deployment times and improve security.
- **Manage configurations and secrets:** Handle sensitive information safely.

This project goes beyond basic Docker tutorials, focusing on the practical decisions and configurations that differentiate a development setup from a production-grade deployment. By the end, you'll have a solid foundation for deploying your own applications with confidence.

## Core Technologies and Their Current State

This guide leverages the latest stable approaches for Docker and Docker Compose.

- **Docker Engine:** The core containerization platform. We will use a recent stable release of Docker Engine that supports the Compose Specification. For the latest version details, refer to the official Docker documentation.
    - Docker Documentation

- **Docker Compose:** A tool for defining and running multi-container Docker applications. We will adhere to the **Compose Specification**, which is the recommended way to define `docker-compose.yml` files. This means we will **not** include an explicit `version` field in our Compose files, as it is no longer required and is discouraged for new projects following the specification.
    - Compose Specification Versioning

### What You'll Need to Get Started

To follow along with this guide, ensure you have the following installed and configured:

- **Operating System:** Linux, macOS, or Windows (with WSL2 enabled for Docker Desktop).
- **Docker Engine:** Installed and running on your system.
- **Text Editor:** Such as VS Code, Sublime Text, or your preferred IDE.
- **Command-Line Proficiency:** Basic familiarity with navigating directories, running commands, and inspecting output.

### Project Architecture at a Glance

We will build a simple, yet representative, multi-service application stack. This typically includes:

- **A Web Application:** A stateless service (e.g., a basic Python Flask app or Node.js Express app) that serves content and interacts with a database.
- **A Database Service:** PostgreSQL will be used for persistent data storage.
- **Internal Networking:** Services will communicate securely over a private Docker network.

Key architectural decisions we'll implement include:

- **Stateless Application Services:** Ensuring your web application can be scaled horizontally without losing state.
- **Persistent Data with Volumes:** Using Docker volumes to safeguard database data.
- **Robust Health Checks:** Configuring services to report their operational status, enabling Docker Compose to manage their lifecycle intelligently.
- **Optimized Images:** Employing multi-stage builds to create smaller, more secure container images.

### Learning Path: Your Journey to Production Docker

This guide is structured into 13 practical milestones, each building upon the last to construct a complete, production-aware Docker Compose stack.

#### Project Setup and Docker Engine Installation

Set up the development environment, install Docker Engine, and verify its functionality.

#### Containerizing a Simple Web Application

Create a basic web application and write its initial Dockerfile, focusing on minimal dependencies.

#### Building and Running Your First Container Image

Build the Docker image for the web application and run it as a standalone container, verifying its basic operation.

#### Orchestrating Services with Docker Compose

Introduce Docker Compose, explain the Compose Specification (without the 'version' field), and define the web application service.

#### Integrating a Database Service (PostgreSQL)

Add a PostgreSQL database service to the Docker Compose setup and configure the web application to connect to it.

#### Establishing Secure Inter-Service Networking

Configure custom Docker networks for secure and isolated communication between application services.

#### Managing Persistent Data with Docker Volumes

Implement Docker volumes to ensure data persistence for the database service across container lifecycles.

#### Handling Configuration and Secrets Securely

Manage application configuration using environment variables and explore options for handling sensitive data like secrets.

#### Implementing Health Checks for Service Robustness

Add health checks to services in Docker Compose to ensure they are ready and remain healthy.

#### Optimizing Docker Images with Multi-Stage Builds

Refactor Dockerfiles to use multi-stage builds, reducing image size and attack surface.

### Securing Containers with Non-Root Users and Resource Limits

Implement security best practices by running containers as non-root users and defining resource limits (CPU, memory).

### Auditing Docker Host and Containers with docker-bench-security

Use `docker-bench-security` to audit the Docker host and container configurations for common security vulnerabilities.

### Finalizing the Production Stack and Deployment Considerations

Review the complete production-ready stack, discuss logging, monitoring, and next steps for deployment to production environments.

---

## References


- Docker Documentation
- Compose Specification Versioning
- docker-bench-security GitHub Repository
- PostgreSQL Official Docker Image
- Docker Volumes Documentation
- Docker Networks Documentation


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