Angular language-specific guide

- What will you learn?
- Prerequisites
- Containerize an Angular Application
  - Prerequisites
  - Overview
  - Get the sample application
  - Build the Docker image
    - Step 1: Create the Dockerfile
    - Step 2: Create the compose.yaml file
    - Step 3: Create the .dockerignore file
    - Step 4: Create the nginx.conf file
    - Step 5: Build the Angular application image
    - Step 6: View local images
  - Run the containerized application
    - Run the application in the background
- Use containers for Angular development
  - Prerequisites
  - Overview
  - Automatically update services (development mode)
  - Step 1: Create a development Dockerfile
    - Step 2: Update your compose.yaml file
    - Step 4: Start Compose Watch
    - Step 5: Test Compose Watch with Angular
- Run Angular tests in a container
  - Prerequisites
  - Overview

- Run tests during development
  - Step 1: Update compose.yaml
  - Step 2: Run the tests
- Summary
- Related resources

The Angular language-specific guide shows you how to containerize an Angular application using Docker, following best practices for creating efficient, production-ready containers.

Angular is a robust and widely adopted framework for building dynamic, enterprise-grade web applications. However, managing dependencies, environments, and deployments can become complex as applications scale. Docker streamlines these challenges by offering a consistent, isolated environment for development and production.

> **Acknowledgment**
>
> Docker extends its sincere gratitude to Kristiyan Velkov for authoring this guide. As a Docker Captain and experienced Front-end engineer, his expertise in Docker, DevOps, and modern web development has made this resource essential for the community, helping developers navigate and optimize their Docker workflows.

What will you learn?

In this guide, you will learn how to:

- Containerize and run an Angular application using Docker.
- Set up a local development environment for Angular inside a container.
- Run tests for your Angular application within a Docker container.

You'll start by containerizing an existing Angular application and work your way up to production-level deployments.

Prerequisites

Before you begin, ensure you have a working knowledge of:

- Basic understanding of TypeScript and JavaScript.
- Familiarity with Node.js and npm for managing dependencies and running scripts.
- Familiarity with Angular fundamentals.
- Understanding of core Docker concepts such as images, containers, and Dockerfiles. If you're new to Docker, start with the Docker basics guide.

Once you've completed the Angular getting started modules, you'll be fully prepared to containerize your own Angular application using the detailed examples and best practices outlined in this guide.

Containerize an Angular Application

Prerequisites

Before you begin, make sure the following tools are installed and available on your system:

- You have installed the latest version of Docker Desktop.
- You have a git client. The examples in this section use a command-line based git client, but you can use any client.

> **New to Docker?**
>
> Start with the Docker basics guide to get familiar with key concepts like images, containers, and Dockerfiles.

Overview

This guide walks you through the complete process of containerizing an Angular application with Docker. You'll learn how to create a production-ready Docker image using best practices that improve performance, security, scalability, and deployment efficiency.

By the end of this guide, you will:

- Containerize an Angular application using Docker.
- Create and optimize a Dockerfile for production builds.
- Use multi-stage builds to minimize image size.
- Serve the application efficiently with a custom Nginx configuration.
- Build secure and maintainable Docker images by following best practices.

Get the sample application

Clone the sample application to use with this guide. Open a terminal, navigate to the directory where you want to work, and run the following command to clone the git repository:

```
$ git clone https://github.com/kristiyan-velkov/docker-angular-sample
```

Build the Docker image

Angular is a front-end framework that compiles into static assets, so the Dockerfile uses a multi-stage build: one stage compiles the app with Node.js, and a second minimal stage serves the static output with Nginx.

> [!TIP]
>
> Gordon, Docker's AI assistant, can generate Docker assets for your project. Ask Gordon to create a Dockerfile, Compose file, and `.dockerignore` tailored to your application.

Step 1: Create the Dockerfile

Before creating a Dockerfile, you need to choose a base image. You can either use the Node.js Official Image or a Docker Hardened Image (DHI) from the Hardened Image catalog.

Choosing DHI offers the advantage of a production-ready image that is lightweight and secure. For more information, see Docker Hardened Images.

> [!IMPORTANT] This guide uses a stable Node.js LTS image tag that is considered secure when the guide is written. Because new releases and security patches are published regularly, the tag shown here may no longer be the safest option

> when you follow the guide. Always review the latest available image tags and select a secure, up-to-date version before building or deploying your application.
>
> Official Node.js Docker Images: https://hub.docker.com/_/node

{{< tabs >}} {{< tab name="Using Docker Hardened Images" >}} Docker Hardened Images (DHIs) are available for Node.js in the Docker Hardened Images catalog. Docker Hardened Images are freely available to everyone with no subscription required. You can pull and use them like any other Docker image after signing in to the DHI registry. For more information, see the DHI quickstart guide.

1. Sign in to the DHI registry:

   ```
   $ docker login dhi.io
   ```

2. Pull the Node.js DHI (check the catalog for available versions):

   ```
   $ docker pull dhi.io/node:24-alpine3.22-dev
   ```

In the following Dockerfile, the `FROM` instruction uses `dhi.io/node:24-alpine3.22-dev` as the base image.

```
# ========================================
# Stage 1: Build the Angular Application
# ========================================

# Use a lightweight DHI Node.js image for building
FROM dhi.io/node:24-alpine3.22-dev AS builder

# Set the working directory inside the container
WORKDIR /app

# Copy package-related files first to leverage Docker's caching mechanism
COPY package.json package-lock.json* ./

# Install project dependencies using npm ci (ensures a clean, reproducible install)
RUN --mount=type=cache,target=/root/.npm npm ci

# Copy the rest of the application source code into the container
COPY . .
```

```
# Build the Angular application
RUN npm run build

# ========================================
# Stage 2: Prepare Nginx to Serve Static Files
# ========================================

FROM dhi.io/nginx:1.28.0-alpine3.21-dev AS runner

# Copy custom Nginx config
COPY nginx.conf /etc/nginx/nginx.conf

# Copy the static build output from the build stage to Nginx's default HTML serving directory
COPY --chown=nginx:nginx --from=builder /app/dist/*/browser /usr/share/nginx/html

# Use a non-root user for security best practices
USER nginx

# Expose port 8080 to allow HTTP traffic
# Note: The default Nginx container now listens on port 8080 instead of 80
EXPOSE 8080

# Start Nginx directly with custom config
ENTRYPOINT ["nginx", "-c", "/etc/nginx/nginx.conf"]
CMD ["-g", "daemon off;"]
```

{{< /tab >}} {{< tab name="Using the Docker Official Image" >}}

Create a file named `Dockerfile` with the following contents:

```
# ========================================
# Stage 1: Build the Angular Application
# ========================================
ARG NODE_VERSION=24.12.0-alpine
ARG NGINX_VERSION=alpine3.22

# Use a lightweight Node.js image for building (customizable via ARG)
FROM node:${NODE_VERSION} AS builder

# Set the working directory inside the container
```

```
WORKDIR /app

# Copy package-related files first to leverage Docker's caching mechanism
COPY package.json *package-lock.json* ./

# Install project dependencies using npm ci (ensures a clean, reproducible install)
RUN --mount=type=cache,target=/root/.npm npm ci

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

# Build the Angular application
RUN npm run build

# ========================================
# Stage 2: Prepare Nginx to Serve Static Files
# ========================================

FROM nginxinc/nginx-unprivileged:${NGINX_VERSION} AS runner

# Copy custom Nginx config
COPY nginx.conf /etc/nginx/nginx.conf

# Copy the static build output from the build stage to Nginx's default HTML serving directory
COPY --chown=nginx:nginx --from=builder /app/dist/*/browser /usr/share/nginx/html

# Use a built-in non-root user for security best practices
USER nginx

# Expose port 8080 to allow HTTP traffic
# Note: The default Nginx container now listens on port 8080 instead of 80
EXPOSE 8080

# Start Nginx directly with custom config
ENTRYPOINT ["nginx", "-c", "/etc/nginx/nginx.conf"]
CMD ["-g", "daemon off;"]
```

> [!NOTE] We are using nginx-unprivileged instead of the standard Nginx image to follow security best practices. Running as a non-root user in the final image:
>
> - Reduces the attack surface
> - Aligns with Docker's recommendations for container hardening
> - Helps comply with stricter security policies in production environments

{{< /tab >}} {{< /tabs >}}

Step 2: Create the compose.yaml file

Create a file named `compose.yaml` with the following contents:

```
services:
  server:
    build:
      context: .
    ports:
      - 8080:8080
```

Step 3: Create the .dockerignore file

The `.dockerignore` file tells Docker which files and folders to exclude when building the image.

> [!NOTE] This helps:
>
> - Reduce image size
> - Speed up the build process
> - Prevent sensitive or unnecessary files (like `.env`, `.git`, or `node_modules`) from being added to the final image.
>
> To learn more, visit the .dockerignore reference.

Create a file named `.dockerignore` with the following contents:

```
# ================================
# Node and build output
# ================================
node_modules
dist
```

```
out-tsc
.angular
.cache
.tmp

# ================================
# Testing & Coverage
# ================================
coverage
jest
cypress
cypress/screenshots
cypress/videos
reports
playwright-report
.vite
.vitepress

# ================================
# Environment & log files
# ================================
*.env*
!*.env.production
*.log
*.tsbuildinfo

# ================================
# IDE & OS-specific files
# ================================
.vscode
.idea
.DS_Store
Thumbs.db
*.swp

# ================================
# Version control & CI files
# ================================
.git
.gitignore
```

```
# ================================
# Docker & local orchestration
# ================================
Dockerfile
Dockerfile.*
.dockerignore
docker-compose.yml
docker-compose*.yml

# ================================
# Miscellaneous
# ================================
*.bak
*.old
*.tmp
```

Step 4: Create the `nginx.conf` file

To serve your Angular application efficiently inside the container, you'll configure Nginx with a custom setup. This configuration is optimized for performance, browser caching, gzip compression, and support for client-side routing.

Create a file named `nginx.conf` in the root of your project directory, and add the following content:

> [!NOTE] To learn more about configuring Nginx, see the official Nginx documentation.

```
worker_processes auto;

pid /tmp/nginx.pid;

events {
    worker_connections 1024;
}

http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;
```

```
client_body_temp_path /tmp/client_temp;
proxy_temp_path       /tmp/proxy_temp_path;
fastcgi_temp_path     /tmp/fastcgi_temp;
uwsgi_temp_path       /tmp/uwsgi_temp;
scgi_temp_path        /tmp/scgi_temp;

# Logging
access_log off;
error_log  /dev/stderr warn;

# Performance
sendfile        on;
tcp_nopush      on;
tcp_nodelay     on;
keepalive_timeout  65;
keepalive_requests 1000;

# Compression
gzip on;
gzip_vary on;
gzip_proxied any;
gzip_min_length 256;
gzip_comp_level 6;
gzip_types
    text/plain
    text/css
    text/xml
    text/javascript
    application/javascript
    application/x-javascript
    application/json
    application/xml
    application/xml+rss
    font/ttf
    font/otf
    image/svg+xml;

server {
    listen       8080;
```

```
        server_name  localhost;

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

        # Angular Routing
        location / {
            try_files $uri $uri/ /index.html;
        }

        # Static Assets Caching
        location ~* \.(?:ico|css|js|gif|jpe?g|png|woff2?|eot|ttf|svg|map)$ {
            expires 1y;
            access_log off;
            add_header Cache-Control "public, immutable";
        }

        # Optional: Explicit asset route
        location /assets/ {
            expires 1y;
            add_header Cache-Control "public, immutable";
        }
    }
}
```

Step 5: Build the Angular application image

With your custom configuration in place, you're now ready to build the Docker image for your Angular application.

The updated setup includes:

- The updated setup includes a clean, production-ready Nginx configuration tailored specifically for Angular.
- Efficient multi-stage Docker build, ensuring a small and secure final image.

After completing the previous steps, your project directory should now contain the following files:

```
├── docker-angular-sample/
│ ├── Dockerfile
│ ├── .dockerignore
│ ├── compose.yaml
```

```
| └── nginx.conf
```

Now that your Dockerfile is configured, you can build the Docker image for your Angular application.

> [!NOTE] The `docker build` command packages your application into an image using the instructions in the Dockerfile. It includes all necessary files from the current directory (called the build context).

Run the following command from the root of your project:

```
$ docker build --tag docker-angular-sample .
```

What this command does:

- Uses the Dockerfile in the current directory (.)
- Packages the application and its dependencies into a Docker image
- Tags the image as docker-angular-sample so you can reference it later

Step 6: View local images

After building your Docker image, you can check which images are available on your local machine using either the Docker CLI or Docker Desktop. Since you're already working in the terminal, let's use the Docker CLI.

To list all locally available Docker images, run the following command:

```
$ docker images
```

Example Output:

```
REPOSITORY              TAG              IMAGE ID       CREATED          SIZE
docker-angular-sample   latest           34e66bdb9d40   14 seconds ago   76.4MB
```

This output provides key details about your images:

- **Repository** – The name assigned to the image.
- **Tag** – A version label that helps identify different builds (e.g., latest).
- **Image ID** – A unique identifier for the image.
- **Created** – The timestamp indicating when the image was built.
- **Size** – The total disk space used by the image.

If the build was successful, you should see `docker-angular-sample` image listed.

Run the containerized application

In the previous step, you created a Dockerfile for your Angular application and built a Docker image using the docker build command. Now it’s time to run that image in a container and verify that your application works as expected.

Inside the `docker-angular-sample` directory, run the following command in a terminal.

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

Open a browser and view the application at http://localhost:8080. You should see a simple Angular web application.

Press `ctrl+c` in the terminal to stop your application.

Run the application in the background

You can run the application detached from the terminal by adding the `-d` option. Inside the `docker-angular-sample` directory, run the following command in a terminal.

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

Open a browser and view the application at http://localhost:8080. You should see your Angular application running in the browser.

To confirm that the container is running, use `docker ps` command:

```
$ docker ps
```

This will list all active containers along with their ports, names, and status. Look for a container exposing port 8080.

Example Output:

```
CONTAINER ID   IMAGE                              COMMAND                  CREATED
STATUS             PORTS                     NAMES
eb13026806d1   docker-angular-sample-server   "nginx -c /etc/nginx…"   About a minute ago  Up
About a minute  0.0.0.0:8080->8080/tcp   docker-angular-sample-server-1
```

To stop the application, run:

```
$ docker compose down
```

> [!NOTE] For more information about Compose commands, see the Compose CLI reference.

Use containers for Angular development

Prerequisites

Complete Containerize Angular application.

Overview

In this section, you'll learn how to set up both production and development environments for your containerized Angular application using Docker Compose. This setup allows you to serve a static production build via Nginx and to develop efficiently inside containers using a live-reloading dev server with Compose Watch.

You'll learn how to:

- Configure separate containers for production and development
- Enable automatic file syncing using Compose Watch in development
- Debug and live-preview your changes in real-time without manual rebuilds

---

Automatically update services (development mode)

Use Compose Watch to automatically sync source file changes into your containerized development environment. This provides a seamless, efficient development experience without

restarting or rebuilding containers manually.

Step 1: Create a development Dockerfile

Create a file named `Dockerfile.dev` in your project root with the following content:

```
# ========================================
# Stage 1: Development - Angular Application
# ========================================

# Define the Node.js version to use (Alpine for a small footprint)
ARG NODE_VERSION=24.12.0-alpine

# Set the base image for development
FROM node:${NODE_VERSION} AS dev

# Set environment variable to indicate development mode
ENV NODE_ENV=development

# Set the working directory inside the container
WORKDIR /app

# Copy only the dependency files first to optimize Docker caching
COPY package.json package-lock.json* ./

# Install dependencies using npm with caching to speed up subsequent builds
RUN --mount=type=cache,target=/root/.npm npm install

# Copy all application source files into the container
COPY . .

# Expose the port Angular uses for the dev server (default is 4200)
EXPOSE 4200

# Start the Angular dev server and bind it to all network interfaces
CMD ["npm", "start", "--", "--host=0.0.0.0"]
```

This file sets up a lightweight development environment for your Angular application using the dev server.

Step 2: Update your `compose.yaml` file

Open your `compose.yaml` file and define two services: one for production (`angular-prod`) and one for development (`angular-dev`).

Here’s an example configuration for an Angular application:

```
services:
  angular-prod:
    build:
      context: .
      dockerfile: Dockerfile
    image: docker-angular-sample
    ports:
      - "8080:8080"

  angular-dev:
    build:
      context: .
      dockerfile: Dockerfile.dev
    ports:
      - "4200:4200"
    develop:
      watch:
        - action: sync
          path: .
          target: /app
```

- The `angular-prod` service builds and serves your static production app using Nginx.
- The `angular-dev` service runs your Angular development server with live reload and hot module replacement.
- `watch` triggers file sync with Compose Watch.

> [!NOTE] For more details, see the official guide: Use Compose Watch.

After completing the previous steps, your project directory should now contain the following files:

```
├── docker-angular-sample/
│ ├── Dockerfile
│ ├── Dockerfile.dev
```

```
| ├── .dockerignore
| ├── compose.yaml
| └── nginx.conf
```

Step 4: Start Compose Watch

Run the following command from the project root to start the container in watch mode

```
$ docker compose watch angular-dev
```

Step 5: Test Compose Watch with Angular

To verify that Compose Watch is working correctly:

1. Open the `src/app/app.component.html` file in your text editor.
2. Locate the following line:

   ```
   <h1>Docker Angular Sample Application</h1>
   ```

3. Change it to:

   ```
   <h1>Hello from Docker Compose Watch</h1>
   ```

4. Save the file.
5. Open your browser at http://localhost:4200.

You should see the updated text appear instantly, without needing to rebuild the container manually. This confirms that file watching and automatic synchronization are working as expected.

Run Angular tests in a container

Prerequisites

Complete all the previous sections of this guide, starting with Containerize Angular application.

Overview

Testing is a critical part of the development process. In this section, you'll learn how to:

- Run Jasmine unit tests using the Angular CLI inside a Docker container.
- Use Docker Compose to isolate your test environment.
- Ensure consistency between local and container-based testing.

The `docker-angular-sample` project comes pre-configured with Jasmine, so you can get started quickly without extra setup.

Run tests during development

The `docker-angular-sample` application includes a sample test file at the following location:

```
$ src/app/app.component.spec.ts
```

This test uses Jasmine to validate the AppComponent logic.

Step 1: Update compose.yaml

Add a new service named `angular-test` to your `compose.yaml` file. This service allows you to run your test suite in an isolated, containerized environment.

```
services:
  angular-dev:
    build:
      context: .
      dockerfile: Dockerfile.dev
    ports:
      - "5173:5173"
    develop:
      watch:
        - action: sync
          path: .
          target: /app

  angular-prod:
    build:
      context: .
      dockerfile: Dockerfile
    image: docker-angular-sample
    ports:
      - "8080:8080"
```

```
  angular-test:
    build:
      context: .
      dockerfile: Dockerfile.dev
    command: ["npm", "run", "test"]
```

The angular-test service reuses the same `Dockerfile.dev` used for development and overrides the default command to run tests with `npm run test`. This setup ensures a consistent test environment that matches your local development configuration.

After completing the previous steps, your project directory should contain the following files:

```
├── docker-angular-sample/
│ ├── Dockerfile
│ ├── Dockerfile.dev
│ ├── .dockerignore
│ ├── compose.yaml
│ └── nginx.conf
```

Step 2: Run the tests

To execute your test suite inside the container, run the following command from your project root:

```
$ docker compose run --rm angular-test
```

This command will:

- Start the `angular-test` service defined in your `compose.yaml` file.
- Execute the `npm run test` script using the same environment as development.
- Automatically removes the container after tests complete, using the `docker compose run --rm` command.

You should see output similar to the following:

```
Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        1.529 s
```

> [!NOTE] For more information about Compose commands, see the Compose CLI reference.

Summary

In this section, you learned how to run unit tests for your Angular application inside a Docker container using Jasmine and Docker Compose.

What you accomplished:

- Created a `angular-test` service in `compose.yaml` to isolate test execution.
- Reused the development `Dockerfile.dev` to ensure consistency between dev and test environments.
- Ran tests inside the container using `docker compose run --rm angular-test`.
- Ensured reliable, repeatable testing across environments without depending on your local machine setup.

Related resources

Explore official references and best practices to sharpen your Docker testing workflow:

- Dockerfile reference – Understand all Dockerfile instructions and syntax.
- Best practices for writing Dockerfiles – Write efficient, maintainable, and secure Dockerfiles.
- Compose file reference – Learn the full syntax and options available for configuring services in `compose.yaml`.
- `docker compose run` CLI reference – Run one-off commands in a service container.

□□□ #2
□ □□□ □□□ 2026-10-31 18:44:27 UTC
□ □□□ □□□ 2026-10-31 18:44:29 UTC