Vue.js language-specific guide

- What will you learn?
- Prerequisites
- Containerize an Vue.js 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 Vue.js application image
    - Step 6: View local images
  - Run the containerized application
    - Run the application in the background
- Use containers for Vue.js 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 Vue.js
- Run vue.js tests in a container
  - Prerequisites
  - Overview

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

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

Vue.js is a progressive and flexible framework for building modern, interactive web applications. However, as applications scale, managing dependencies, environments, and deployments can become complex. Docker simplifies these challenges by providing a consistent, isolated environment for both development and production.

> **Acknowledgment**
>
> Docker extends its sincere gratitude to Kristiyan Velkov for authoring this guide. As a Docker Captain and highly skilled Front-end engineer, Kristiyan brings exceptional expertise in modern web development, Docker, and DevOps. His hands-on approach and clear, actionable guidance make this guide an essential resource for developers aiming to build, optimize, and secure Vue.js applications with Docker.

What will you learn?

In this guide, you will learn how to:

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

You'll start by containerizing an existing Vue.js 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 Vue.js 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 Vue.js getting started modules, you'll be fully prepared to containerize your own Vue.js application using the detailed examples and best practices outlined in this guide.

Containerize an Vue.js 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 Vue.js 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 Vue.js 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-vuejs-sample
```

Build the Docker image

Vue.js is a front-end framework that compiles into static assets, so the Dockerfile is customized to align with how Vue.js applications are built and efficiently served in a production environment.

> 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.

> 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
   ```

3. Pull the Nginx DHI (check the catalog for available versions):

   ```
   $ docker pull dhi.io/nginx:1.28.0-alpine3.21-dev
   ```

In the following Dockerfile, the `FROM` instructions use `dhi.io/node:24-alpine3.22-dev` and `dhi.io/nginx:1.28.0-alpine3.21-dev` as the base images.

```
# ========================================
# Stage 1: Build the Vue.js 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 Vue.js 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 /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;"]
```

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

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

```
# ========================================
# Stage 1: Build the Vue.js 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 Vue.js 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 /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;"]
```

> 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 plays a crucial role in optimizing your Docker image by specifying which files and directories should be excluded from the build context.

> 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:

```
# ------------------------------
# Dependency directories
# ------------------------------
node_modules/

# ------------------------------
# Production and build outputs
# ------------------------------
dist/
out/
build/
public/build/
```

```
# ------------------------------
# Vite, VuePress, and cache dirs
# ------------------------------
.vite/
.vitepress/
.cache/
.tmp/

# ------------------------------
# Test output and coverage
# ------------------------------
coverage/
reports/
jest/
cypress/
cypress/screenshots/
cypress/videos/

# ------------------------------
# Environment and config files
# ------------------------------
*.env*
!.env.production    # Keep production env if needed
*.local
*.log

# ------------------------------
# TypeScript artifacts
# ------------------------------
*.tsbuildinfo

# ------------------------------
# Editor and IDE config
# ------------------------------
.vscode/
.idea/
*.swp

# ------------------------------
```

```
# System files
# ------------------------------
.DS_Store
Thumbs.db

# ------------------------------
# Lockfiles (optional)
# ------------------------------
npm-debug.log*
yarn-debug.log*
yarn-error.log*
pnpm-debug.log*

# ------------------------------
# Git files
# ------------------------------
.git/
.gitignore

# ------------------------------
# Docker-related files
# ------------------------------
Dockerfile
.dockerignore
docker-compose.yml
docker-compose.override.yml
```

Step 4: Create the `nginx.conf` file

To serve your Vue.js 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:

> 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;
    charset       utf-8;

    access_log    off;
    error_log     /dev/stderr warn;

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

    gzip on;
    gzip_comp_level 6;
    gzip_proxied any;
    gzip_min_length 256;
    gzip_vary on;
    gzip_types text/plain text/css application/json application/javascript text/xml
application/xml application/xml+rss text/javascript image/svg+xml;

    server {
        listen       8080;
        server_name  localhost;

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

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

        location ~* \.(?:ico|css|js|gif|jpe?g|png|woff2?|eot|ttf|svg|map)$ {
            expires 1y;
```

```
            access_log off;
            add_header Cache-Control "public, immutable";
            add_header X-Content-Type-Options nosniff;
        }

        location /assets/ {
            expires 1y;
            add_header Cache-Control "public, immutable";
            add_header X-Content-Type-Options nosniff;
        }

        error_page 404 /index.html;
    }
}
```

Step 5: Build the Vue.js application image

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

The updated setup includes:

- The updated setup includes a clean, production-ready Nginx configuration tailored specifically for Vue.js.
- 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-vuejs-sample/
│ ├── Dockerfile
│ ├── .dockerignore
│ ├── compose.yaml
│ └── nginx.conf
```

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

> 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-vuejs-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-vuejs-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-vuejs-sample     latest           8c9c199179d4   14 seconds ago   76.2MB
```

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-vuejs-sample` image listed.

Run the containerized application

In the previous step, you created a Dockerfile for your Vue.js 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-vuejs-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 Vue.js 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-vuejs-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 Vue.js 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
37a1fa85e4b0   docker-vuejs-sample-server     "nginx -c /etc/nginx…"   About a minute ago  Up
About a minute  0.0.0.0:8080->8080/tcp   docker-vuejs-sample-server-1
```

To stop the application, run:

```
$ docker compose down
```

> For more information about Compose commands, see the Compose CLI reference.

Use containers for Vue.js development

Prerequisites

Complete Containerize Vue.js application.

Overview

In this section, you'll set up both production and development environments for your Vue.js application using Docker Compose. This approach streamlines your workflow—delivering a lightweight, static site via Nginx in production, and providing a fast, live-reloading dev server with Compose Watch for efficient local development.

You'll learn how to:

- Configure isolated environments: Set up separate containers optimized for production and development use cases.
- Live-reload in development: Use Compose Watch to automatically sync file changes, enabling real-time updates without manual intervention.
- Preview and debug with ease: Develop inside containers with a seamless preview and debug experience—no rebuilds required after every change.

Automatically update services (development mode)

Leverage Compose Watch to enable real-time file synchronization between your local machine and the containerized Vue.js development environment. This powerful feature eliminates the need to manually rebuild or restart containers, providing a fast, seamless, and efficient development workflow.

With Compose Watch, your code updates are instantly reflected inside the container—perfect for rapid testing, debugging, and live previewing changes.

Step 1: Create a development Dockerfile

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

```
# ========================================
# Stage 1: Develop the Vue.js Application
# ========================================
ARG NODE_VERSION=24.12.0-alpine

# Use a lightweight Node.js 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 package-related files first to leverage Docker's caching mechanism
COPY package.json package-lock.json* ./

# Install project dependencies
RUN --mount=type=cache,target=/root/.npm npm install

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

# Change ownership of the application directory to the node user
RUN chown -R node:node /app

# Switch to the node user
USER node

# Expose the port used by the Vite development server
EXPOSE 5173

# Use a default command, can be overridden in Docker compose.yml file
CMD [ "npm", "run", "dev", "--", "--host" ]
```

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

Step 2: Update your `compose.yaml` file

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

Here's an example configuration for an Vue.js application:

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

  vuejs-dev:
    build:
      context: .
      dockerfile: Dockerfile.dev
    ports:
      - "5173:5173"
    develop:
      watch:
        - path: ./src
          target: /app/src
          action: sync
        - path: ./package.json
          target: /app/package.json
          action: restart
        - path: ./vite.config.js
          target: /app/vite.config.js
          action: restart
```

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

> 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-vuejs-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 vuejs-dev
```

Step 5: Test Compose Watch with Vue.js

To confirm that Compose Watch is functioning correctly:

1. Open the `src/App.vue` file in your text editor.
2. Locate the following line:

   ```
   <HelloWorld msg="You did it!" />
   ```

3. Change it to:

   ```
   <HelloWorld msg="Hello from Docker Compose Watch" />
   ```

4. Save the file.
5. Open your browser at http://localhost:5173.

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 vue.js tests in a container

Prerequisites

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

Overview

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

- Run unit tests using Vitest inside a Docker container.
- Use Docker Compose to run tests in an isolated, reproducible environment.

You'll use Vitest — a blazing fast test runner designed for Vite — together with @vue/test-utils to write unit tests that validate your component logic, props, events, and reactive behavior.

This setup ensures your Vue.js components are tested in an environment that mirrors how users actually interact with your application.

Run tests during development

`docker-vuejs-sample` application includes a sample test file at location:

```
$ src/components/__tests__/HelloWorld.spec.ts
```

This test uses Vitest and Vue Test Utils to verify the behavior of the HelloWorld component.

Step 1: Update compose.yaml

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

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

  vuejs-dev:
    build:
```

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

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

The vuejs-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-vuejs-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 vuejs-test
```

This command will:

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

You should see output similar to the following:

```
Test Files: 1 passed (1)
Tests:      1 passed (1)
Start at:   16:50:55
Duration:   718ms
```

> For more information about Compose commands, see the Compose CLI reference.

Summary

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

What you accomplished:

- Created a `vuejs-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 vuejs-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 18:48:38 UTC
□ □□□ □□□ 2026-10-31 18:48:40 UTC