Containerize a Next.js application

- What will you learn?
- Prerequisites
- Containerize a Next.js Application
  - Prerequisites
  - Overview
  - Get the sample application
  - Build the Docker image
    - Step 1: Configure Next.js and create the Dockerfile
      - 1.1 Next.js with standalone output
      - 1.2 Next.js with export output
    - Step 2: Create the compose.yaml file
    - Step 3: Create the .dockerignore file
    - Step 4: Build the Next.js application image
    - Step 5: View local images
  - Run the containerized application
    - Run the application in the background
- Use containers for Next.js development
  - Prerequisites
  - Overview
  - Automatically update services (development mode)
  - Step 1: Create a development Dockerfile
    - Step 2: Update your compose.yaml file
    - Step 3: Configure Next.js for Docker development
    - Step 4: Start Compose Watch
    - Step 5: Test Compose Watch with Next.js
- Run Next.js tests in a container

- Prerequisites
- Overview
- Run tests during development
  - Step 1: Install Vitest and React Testing Library (custom projects)
  - Step 2: Configure Vitest (custom projects)
  - Step 3: Update compose.yaml
  - Step 4: Run the tests
  - Step 5: Run lint in the container
- Summary
- Related resources

This guide shows you how to containerize a Next.js application using Docker, following best practices for creating efficient, production-ready containers.

Next.js is a React framework that enables server-side rendering, static site generation, and full-stack capabilities. Docker provides a consistent containerized environment from development to production.

> **Acknowledgment**
>
> Docker extends its sincere gratitude to Kristiyan Velkov for authoring this guide and contributing the official Next.js Docker examples to the Vercel Next.js repository, including the standalone and export output examples. As a Docker Captain and experienced engineer, his expertise in Docker, DevOps, and modern web development has made this resource invaluable 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 a Next.js application using Docker.
- Set up a local development environment for Next.js inside a container.
- Run tests for your Next.js application within a Docker container.

To begin, you'll start by containerizing an existing Next.js application.

Prerequisites

Before you begin, make sure you're familiar with the following:

- Basic understanding of JavaScript or TypeScript.
- Basic knowledge of Node.js and npm for managing dependencies and running scripts.
- Familiarity with React and Next.js fundamentals.
- Understanding of 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 Next.js getting started modules, you'll be ready to containerize your own Next.js application using the examples and instructions provided in this guide.

Containerize a Next.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.

> [!NOTE] 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 containerizing a Next.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 a Next.js application using Docker.
- Create and optimize a Dockerfile for production builds.
- Use multi-stage builds to minimize image size.
- Leverage Next.js standalone or export output for efficient containerization.
- Follow best practices for building secure and maintainable Docker images.

Get the sample application

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

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

Build the Docker image

Next.js has specific requirements for production deployments. This guide shows two approaches: `standalone` output (Node.js server) and `export` output (static files 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: Configure Next.js and create the Dockerfile

Before creating a Dockerfile, choose a base image: the Node.js Official Image or a Docker Hardened Image (DHI) from the Hardened Image catalog. Choosing DHI gives you a production-ready, lightweight, and secure image. For more information, see Docker Hardened Images.

> [!IMPORTANT] This guide uses stable Node.js LTS image tags that are considered secure when the guide is written. Because new releases and security patches are published regularly, always review the official Node.js Docker images and select a secure, up-to-date version before building or deploying.

1.1 Next.js with standalone output

Standalone output (`output: "standalone"`) makes Next.js build a self-contained output that includes only the files and dependencies needed to run the application. A single `node server.js` can serve the app, which is ideal for Docker and supports server-side rendering, API routes, and incremental static regeneration. For details, see the Next.js output configuration documentation (including the "standalone" option).

The container runs the Next.js server with Node.js on port 3000.

Configure Next.js — Open or create `next.config.ts` in your project root:

```
import type { NextConfig } from "next";

const nextConfig: NextConfig = {
  output: "standalone",
};

export default nextConfig;
```

Choose either a Docker Hardened Image or the Docker Official Image, then create a `Dockerfile` using the content from the selected tab below.

{{< tabs >}} {{< tab name="Using Docker Hardened Images" >}}

Docker Hardened Images (DHIs) are available for Node.js in the Docker Hardened Images catalog. 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. Create a file named `Dockerfile` with the following contents. The `FROM` instructions use `dhi.io/node:24-alpine3.22-dev`. Check the Docker Hardened Images catalog for the latest versions and update the image tags as needed for security and compatibility.

   ```
   # ================================================
   # Stage 1: Dependencies Installation Stage
   # ================================================
   ```

```
# IMPORTANT: Docker Hardened Image (DHI) Version Maintenance
# This Dockerfile uses dhi.io/node. Regularly validate and update to the latest DHI
versions in the catalog for security and compatibility.

FROM dhi.io/node:24-alpine3.22-dev AS dependencies

# Set working directory
WORKDIR /app

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

# Install project dependencies with frozen lockfile for reproducible builds
RUN --mount=type=cache,target=/root/.npm \
    --mount=type=cache,target=/usr/local/share/.cache/yarn \
    --mount=type=cache,target=/root/.local/share/pnpm/store \
  if [ -f package-lock.json ]; then \
    npm ci --no-audit --no-fund; \
  elif [ -f yarn.lock ]; then \
    corepack enable yarn && yarn install --frozen-lockfile --production=false; \
  elif [ -f pnpm-lock.yaml ]; then \
    corepack enable pnpm && pnpm install --frozen-lockfile; \
  else \
    echo "No lockfile found." && exit 1; \
  fi

# ============================================
# Stage 2: Build Next.js application in standalone mode
# ============================================

FROM dhi.io/node:24-alpine3.22-dev AS builder

# Set working directory
WORKDIR /app

# Copy project dependencies from dependencies stage
COPY --from=dependencies /app/node_modules ./node_modules

# Copy application source code
```

```
COPY . .

ENV NODE_ENV=production

# Next.js collects completely anonymous telemetry data about general usage.
# Learn more here: https://nextjs.org/telemetry
# Uncomment the following line in case you want to disable telemetry during the
build.
# ENV NEXT_TELEMETRY_DISABLED=1

# Build Next.js application
# If you want to speed up Docker rebuilds, you can cache the build artifacts
# by adding: --mount=type=cache,target=/app/.next/cache
# This caches the .next/cache directory across builds, but it also prevents
# .next/cache/fetch-cache from being included in the final image, meaning
# cached fetch responses from the build won't be available at runtime.
RUN if [ -f package-lock.json ]; then \
    npm run build; \
  elif [ -f yarn.lock ]; then \
    corepack enable yarn && yarn build; \
  elif [ -f pnpm-lock.yaml ]; then \
    corepack enable pnpm && pnpm build; \
  else \
    echo "No lockfile found." && exit 1; \
  fi

# ==========================================
# Stage 3: Run Next.js application
# ==========================================

FROM dhi.io/node:24-alpine3.22-dev AS runner

# Set working directory
WORKDIR /app

# Set production environment variables
ENV NODE_ENV=production
ENV PORT=3000
ENV HOSTNAME="0.0.0.0"
```

```
# Next.js collects completely anonymous telemetry data about general usage.
# Learn more here: https://nextjs.org/telemetry
# Uncomment the following line in case you want to disable telemetry during the run
time.
# ENV NEXT_TELEMETRY_DISABLED=1

# Copy production assets
COPY --from=builder --chown=node:node /app/public ./public

# Set the correct permission for prerender cache
RUN mkdir .next
RUN chown node:node .next

# Automatically leverage output traces to reduce image size
# https://nextjs.org/docs/advanced-features/output-file-tracing
COPY --from=builder --chown=node:node /app/.next/standalone ./
COPY --from=builder --chown=node:node /app/.next/static ./.next/static

# If you want to persist the fetch cache generated during the build so that
# cached responses are available immediately on startup, uncomment this line:
# COPY --from=builder --chown=node:node /app/.next/cache ./.next/cache

# Switch to non-root user for security best practices
USER node

# Expose port 3000 to allow HTTP traffic
EXPOSE 3000

# Start Next.js standalone server
CMD ["node", "server.js"]
```

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

Create a file named `Dockerfile` with the following contents (uses `node`):

```
# ==========================================
# Stage 1: Dependencies Installation Stage
# ==========================================
```

```
ARG NODE_VERSION=24.14.0-slim

FROM node:${NODE_VERSION} AS dependencies

# Set working directory
WORKDIR /app

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

# Install project dependencies with frozen lockfile for reproducible builds
RUN --mount=type=cache,target=/root/.npm \
    --mount=type=cache,target=/usr/local/share/.cache/yarn \
    --mount=type=cache,target=/root/.local/share/pnpm/store \
  if [ -f package-lock.json ]; then \
    npm ci --no-audit --no-fund; \
  elif [ -f yarn.lock ]; then \
    corepack enable yarn && yarn install --frozen-lockfile --production=false; \
  elif [ -f pnpm-lock.yaml ]; then \
    corepack enable pnpm && pnpm install --frozen-lockfile; \
  else \
    echo "No lockfile found." && exit 1; \
  fi

# ==========================================
# Stage 2: Build Next.js application in standalone mode
# ==========================================

FROM node:${NODE_VERSION} AS builder

# Set working directory
WORKDIR /app

# Copy project dependencies from dependencies stage
COPY --from=dependencies /app/node_modules ./node_modules

# Copy application source code
COPY . .

ENV NODE_ENV=production
```

```
# Next.js collects completely anonymous telemetry data about general usage.
# Learn more here: https://nextjs.org/telemetry
# Uncomment the following line in case you want to disable telemetry during the build.
# ENV NEXT_TELEMETRY_DISABLED=1

# Build Next.js application
# If you want to speed up Docker rebuilds, you can cache the build artifacts
# by adding: --mount=type=cache,target=/app/.next/cache
# This caches the .next/cache directory across builds, but it also prevents
# .next/cache/fetch-cache from being included in the final image, meaning
# cached fetch responses from the build won't be available at runtime.
RUN if [ -f package-lock.json ]; then \
    npm run build; \
  elif [ -f yarn.lock ]; then \
    corepack enable yarn && yarn build; \
  elif [ -f pnpm-lock.yaml ]; then \
    corepack enable pnpm && pnpm build; \
  else \
    echo "No lockfile found." && exit 1; \
  fi

# ==========================================
# Stage 3: Run Next.js application
# ==========================================

FROM node:${NODE_VERSION} AS runner

# Set working directory
WORKDIR /app

# Set production environment variables
ENV NODE_ENV=production
ENV PORT=3000
ENV HOSTNAME="0.0.0.0"

# Next.js collects completely anonymous telemetry data about general usage.
# Learn more here: https://nextjs.org/telemetry
# Uncomment the following line in case you want to disable telemetry during the run time.
# ENV NEXT_TELEMETRY_DISABLED=1
```

```
# Copy production assets
COPY --from=builder --chown=node:node /app/public ./public

# Set the correct permission for prerender cache
RUN mkdir .next
RUN chown node:node .next

# Automatically leverage output traces to reduce image size
# https://nextjs.org/docs/advanced-features/output-file-tracing
COPY --from=builder --chown=node:node /app/.next/standalone ./
COPY --from=builder --chown=node:node /app/.next/static ./.next/static

# If you want to persist the fetch cache generated during the build so that
# cached responses are available immediately on startup, uncomment this line:
# COPY --from=builder --chown=node:node /app/.next/cache ./.next/cache

# Switch to non-root user for security best practices
USER node

# Expose port 3000 to allow HTTP traffic
EXPOSE 3000

# Start Next.js standalone server
CMD ["node", "server.js"]
```

> [!NOTE] This Dockerfile uses three stages: `dependencies`, `builder`, and `runner`. The final image runs `node server.js` and listens on port 3000.

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

1.2 Next.js with export output

Output export (`output: "export"`) makes Next.js build a fully static site at build time. It generates HTML, CSS, and JavaScript into an `out` directory that can be served by any static host or CDN—no Node.js server at runtime. Use this when you don't need server-side rendering or API routes. For details, see the Next.js output configuration documentation.

Configure Next.js — Open `next.config.ts` in your project root and add the following code:

```
import type { NextConfig } from "next";

const nextConfig: NextConfig = {
  output: "export",
  trailingSlash: true,
  images: {
    unoptimized: true,
  },
};

export default nextConfig;
```

Choose either a Docker Hardened Image or the Docker Official Image, then create a `Dockerfile` using the content from the selected tab below.

{{< tabs >}} {{< tab name="Using Docker Hardened Images" >}}

Docker Hardened Images (DHIs) are available for Node.js and Nginx in the Docker Hardened Images catalog. 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
   ```

4. Create a file named `Dockerfile` with the following contents. The `FROM` instructions use Docker Hardened Images: `dhi.io/node:24-alpine3.22-dev` and `dhi.io/nginx:1.28.0-alpine3.21-dev`. Check the Docker Hardened Images catalog for the latest versions and update the image tags as needed for security and compatibility.

   ```
   # ==========================================
   # Stage 1: Dependencies Installation Stage
   # ==========================================

   # IMPORTANT: Docker Hardened Image (DHI) Version Maintenance
   # This Dockerfile uses dhi.io/node and dhi.io/nginx. Regularly validate and update to
   ```

```
the latest DHI versions in the catalog for security and compatibility.

FROM dhi.io/node:24-alpine3.22-dev AS dependencies

# Set the working directory
WORKDIR /app

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

# Install project dependencies with frozen lockfile for reproducible builds
RUN --mount=type=cache,target=/root/.npm \
    --mount=type=cache,target=/usr/local/share/.cache/yarn \
    --mount=type=cache,target=/root/.local/share/pnpm/store \
  if [ -f package-lock.json ]; then \
    npm ci --no-audit --no-fund; \
  elif [ -f yarn.lock ]; then \
    corepack enable yarn && yarn install --frozen-lockfile --production=false; \
  elif [ -f pnpm-lock.yaml ]; then \
    corepack enable pnpm && pnpm install --frozen-lockfile; \
  else \
    echo "No lockfile found." && exit 1; \
  fi

# ==========================================
# Stage 2: Build Next.js Application
# ==========================================

FROM dhi.io/node:24-alpine3.22-dev AS builder

# Set the working directory
WORKDIR /app

# Copy project dependencies from dependencies stage
COPY --from=dependencies /app/node_modules ./node_modules

# Copy application source code
COPY . .
```

```
ENV NODE_ENV=production

# Next.js collects completely anonymous telemetry data about general usage.
# Learn more here: https://nextjs.org/telemetry
# Uncomment the following line in case you want to disable telemetry during the
build.
# ENV NEXT_TELEMETRY_DISABLED=1

# Build Next.js application
RUN --mount=type=cache,target=/app/.next/cache \
  if [ -f package-lock.json ]; then \
    npm run build; \
  elif [ -f yarn.lock ]; then \
    corepack enable yarn && yarn build; \
  elif [ -f pnpm-lock.yaml ]; then \
    corepack enable pnpm && pnpm build; \
  else \
    echo "No lockfile found." && exit 1; \
  fi

# ========================================
# Stage 3: Serve Static Files with Nginx
# ========================================

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

# Set the working directory
WORKDIR /app

# Next.js collects completely anonymous telemetry data about general usage.
# Learn more here: https://nextjs.org/telemetry
# Uncomment the following line in case you want to disable telemetry during the run
time.
# ENV NEXT_TELEMETRY_DISABLED=1

# 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/out /usr/share/nginx/html

# Non-root user for security best practices
USER nginx

# Expose port 8080 to allow HTTP traffic
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 (uses `node` and `nginxinc/nginx-unprivileged`):

```
# ============================================
# Stage 1: Dependencies Installation Stage
# ============================================

ARG NODE_VERSION=24.14.0-slim
ARG NGINXINC_IMAGE_TAG=alpine3.22

FROM node:${NODE_VERSION} AS dependencies

# Set the working directory
WORKDIR /app

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

# Install project dependencies with frozen lockfile for reproducible builds
RUN --mount=type=cache,target=/root/.npm \
    --mount=type=cache,target=/usr/local/share/.cache/yarn \
    --mount=type=cache,target=/root/.local/share/pnpm/store \
  if [ -f package-lock.json ]; then \
    npm ci --no-audit --no-fund; \
  elif [ -f yarn.lock ]; then \
```

```
    corepack enable yarn && yarn install --frozen-lockfile --production=false; \
  elif [ -f pnpm-lock.yaml ]; then \
    corepack enable pnpm && pnpm install --frozen-lockfile; \
  else \
    echo "No lockfile found." && exit 1; \
  fi

# ==========================================
# Stage 2: Build Next.js Application
# ==========================================

FROM node:${NODE_VERSION} AS builder

# Set the working directory
WORKDIR /app

# Copy project dependencies from dependencies stage
COPY --from=dependencies /app/node_modules ./node_modules

# Copy application source code
COPY . .

ENV NODE_ENV=production

# Next.js collects completely anonymous telemetry data about general usage.
# Learn more here: https://nextjs.org/telemetry
# Uncomment the following line in case you want to disable telemetry during the build.
# ENV NEXT_TELEMETRY_DISABLED=1

# Build Next.js application
RUN --mount=type=cache,target=/app/.next/cache \
  if [ -f package-lock.json ]; then \
    npm run build; \
  elif [ -f yarn.lock ]; then \
    corepack enable yarn && yarn build; \
  elif [ -f pnpm-lock.yaml ]; then \
    corepack enable pnpm && pnpm build; \
  else \
    echo "No lockfile found." && exit 1; \
  fi
```

```
# ========================================
# Stage 3: Serve Static Files with Nginx
# ========================================

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

# Set the working directory
WORKDIR /app

# Next.js collects completely anonymous telemetry data about general usage.
# Learn more here: https://nextjs.org/telemetry
# Uncomment the following line in case you want to disable telemetry during the run time.
# ENV NEXT_TELEMETRY_DISABLED=1

# 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 --from=builder /app/out /usr/share/nginx/html

# Non-root user for security best practices
USER nginx

# Expose port 8080 to allow HTTP traffic
EXPOSE 8080

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

> [!NOTE] This guide uses nginx-unprivileged instead of the standard Nginx image to run as a non-root user, following security best practices.

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

1. Create `nginx.conf` (required for export output only) — Create a file named `nginx.conf` in the root of your project:

```
# Minimal Nginx config for static Next.js app
worker_processes 1;

# Store PID in /tmp (always writable)
pid /tmp/nginx.pid;

events {
    worker_connections 1024;
}

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

    # Disable logging to avoid permission issues
    access_log off;
    error_log  /dev/stderr;

    # Optimize static file serving
    sendfile        on;
    tcp_nopush      on;
    tcp_nodelay     on;
    keepalive_timeout  65;

    # Gzip compression
    gzip on;
    gzip_types text/plain text/css application/json application/javascript text/xml
application/xml application/xml+rss text/javascript;
    gzip_min_length 256;

    server {
        listen       8080;
        server_name  localhost;

        # Serve static files
        root /usr/share/nginx/html;
        index index.html;
```

```
        # Handle Next.js static export routing
        # See: https://nextjs.org/docs/app/guides/static-exports#deploying
        location / {
            try_files $uri $uri.html $uri/ =404;
        }

        # This is necessary when `trailingSlash: false` (default).
        # You can omit this when `trailingSlash: true` in next.config.
        # Handles nested routes like /blog/post -> /blog/post.html
        location ~ ^/(.+)/$ {
            rewrite ^/(.+)/$ /$1.html break;
        }

        # Serve Next.js static assets
        location ~ ^/_next/ {
            try_files $uri =404;
            expires 1y;
            add_header Cache-Control "public, immutable";
        }

        # Optional 404 handling
        error_page 404 /404.html;
        location = /404.html {
            internal;
        }
    }
}
```

> [!NOTE] Export uses port 8080. For more details, see the Next.js output configuration and Nginx documentation.

Step 2: Create the compose.yaml file

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

```
services:
  server:
    build:
```

```
      context: .
    ports:
      - 3000:3000
```

> [!NOTE] If using export output (Nginx), change the port mapping to `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:

```
# Dependencies (installed inside the image, never copy from host)
node_modules/
.pnp/
.pnp.js
.pnpm-store/

# Next.js build output (generated during the image build)
.next/
out/
dist/
build/
.vercel/

# Testing (not needed in the production image)
coverage/
.nyc_output/
__tests__/
```

```
__mocks__/
jest/
cypress/
playwright-report/
test-results/
.vitest/

# Environment files (avoid leaking secrets into the build context)
.env
.env*
.env.local
.env.development.local
.env.test.local
.env.production.local

# Debug and log files
npm-debug.log*
yarn-debug.log*
yarn-error.log*
pnpm-debug.log*
lerna-debug.log*
*.log

# IDE and editor files
.vscode/
.idea/
.cursor/
.cursorrules
.copilot/
*.swp
*.swo
*~

# Git
.git/
.gitignore
.gitattributes

# Docker files (reduce build context; not needed inside the image)
Dockerfile*
```

```
.dockerignore
docker-compose*.yml
compose*.yaml

# Documentation (not needed in the image)
*.md
docs/

# CI/CD (not needed in the image)
.github/
.gitlab-ci.yml
.travis.yml
.circleci/
Jenkinsfile

# TypeScript and build metadata
*.tsbuildinfo

# Cache and temporary directories
.cache/
.parcel-cache/
.eslintcache
.stylelintcache
.turbo/
.tmp/
.temp/

# Sensitive or dev-only config (optional; omit if your build needs these)
.pem
.editorconfig
.prettierrc*
.eslintrc*
.stylelintrc*
.babelrc*
*.iml

# OS-specific files
.DS_Store
._*
.Spotlight-V100
```

```
.Trashes
ehthumbs.db
Thumbs.db
Desktop.ini
```

Step 4: Build the Next.js application image

With your custom configuration in place, you're now ready to build the Docker image. Use the Dockerfile you created in Step 1 (standalone or export).

The setup includes:

- Multi-stage builds for optimized image size
- Standalone: Node.js server on port 3000; Export: Nginx serving static files on port 8080
- Non-root user for enhanced security
- Proper file permissions and ownership

After completing the previous steps, your project directory should contain at least the following files (export also requires `nginx.conf`):

```
├── docker-nextjs-sample/
│   ├── Dockerfile
│   ├── .dockerignore
│   ├── compose.yaml
│   └── next.config.ts
```

Now that your Dockerfile is configured, you can build the Docker image for your Next.js 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 nextjs-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 nextjs-sample so you can reference it later

Step 5: 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
nextjs-sample           latest           8c5fc80f098e   14 seconds ago   130MB
```

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 `nextjs-sample` image listed.

Run the containerized application

In the previous step, you created a Dockerfile for your Next.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.

Run the following command in a terminal. Use the port that matches your setup: standalone uses port 3000, export uses port 8080.

```
$ docker run -p 3000:3000 nextjs-sample
```

For export output, use port 8080 instead:

```
$ docker run -p 8080:8080 nextjs-sample
```

Open a browser and view the application: http://localhost:3000 for standalone or http://localhost:8080 for export. You should see your Next.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 and `--name` to give the container a name so you can stop it later:

```
$ docker run -d -p 3000:3000 --name nextjs-app nextjs-sample
```

For export output, use port 8080:

```
$ docker run -d -p 8080:8080 --name nextjs-app nextjs-sample
```

Open a browser and view the application: http://localhost:3000 for standalone or http://localhost:8080 for export. You should see your web application.

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

```
$ docker ps
```

This will list all active containers along with their ports, names, and status. Look for a container exposing port 3000 (standalone) or 8080 (export).

Example Output:

```
CONTAINER ID   IMAGE           COMMAND                  CREATED
STATUS             PORTS                    NAMES
f49b74736a9d   nextjs-sample   "node server.js"         About a minute ago   Up About a
minute   0.0.0.0:3000->3000/tcp nextjs-app
```

To stop the application, run:

```
$ docker stop nextjs-app
```

> [!NOTE] For more information about running containers, see the `docker run` CLI reference and the `docker stop` CLI reference.

Use containers for Next.js development

Prerequisites

Complete Containerize Next.js application.

Overview

In this section, you'll learn how to set up both production and development environments for your containerized Next.js application using Docker Compose. This setup allows you to run a production build using the standalone server and to develop efficiently inside containers using Next.js's built-in hot reloading 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 automatically syncs file changes without needing to restart or rebuild containers manually.

Step 1: Create a development Dockerfile

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

```
# ==========================================
# Development Dockerfile for Next.js
# ==========================================
ARG NODE_VERSION=24.14.0-slim

FROM node:${NODE_VERSION} AS dev

WORKDIR /app

COPY package.json yarn.lock* package-lock.json* pnpm-lock.yaml* .npmrc* ./

RUN --mount=type=cache,target=/root/.npm \
    --mount=type=cache,target=/usr/local/share/.cache/yarn \
    --mount=type=cache,target=/root/.local/share/pnpm/store \
  if [ -f package-lock.json ]; then \
    npm ci --no-audit --no-fund; \
  elif [ -f yarn.lock ]; then \
    corepack enable yarn && yarn install --frozen-lockfile --production=false; \
  elif [ -f pnpm-lock.yaml ]; then \
    corepack enable pnpm && pnpm install --frozen-lockfile; \
  else \
    echo "No lockfile found." && exit 1; \
  fi

COPY . .

ENV WATCHPACK_POLLING=true
ENV HOSTNAME="0.0.0.0"

RUN chown -R node:node /app
USER node

EXPOSE 3000

CMD ["sh", "-c", "if [ -f package-lock.json ]; then npm run dev; elif [ -f yarn.lock ]; then
yarn dev; elif [ -f pnpm-lock.yaml ]; then pnpm dev; else npm run dev; fi"]
```

This file sets up a development environment for your Next.js app with hot module replacement and supports npm, yarn, and pnpm.

Step 2: Update your `compose.yaml` file

Open your `compose.yaml` file and define two services: one for production (`nextjs-prod-standalone`) and one for development (`nextjs-dev`). This matches the sample project structure.

Here's an example configuration for a Next.js application:

```
services:
  nextjs-prod-standalone:
    build:
      context: .
      dockerfile: Dockerfile
    image: nextjs-sample:prod
    container_name: nextjs-sample-prod
    ports:
      - "3000:3000"

  nextjs-dev:
    build:
      context: .
      dockerfile: Dockerfile.dev
    image: nextjs-sample:dev
    container_name: nextjs-sample-dev
    ports:
      - "3000:3000"
    environment:
      - WATCHPACK_POLLING=true
    develop:
      watch:
        - action: sync
          path: .
          target: /app
          ignore:
            - node_modules/
            - .next/
        - action: rebuild
          path: package.json
```

- The `nextjs-prod-standalone` service builds and runs your production Next.js app using the standalone output.

- The `nextjs-dev` service runs your Next.js development server with hot module replacement.
- `watch` triggers file sync with Compose Watch.
- `WATCHPACK_POLLING=true` ensures file changes are detected properly inside Docker.
- The `rebuild` action for `package.json` ensures dependencies are reinstalled when the file changes.

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

Step 3: Configure Next.js for Docker development

Next.js works well inside Docker containers out of the box, but there are a few configurations that can improve the development experience.

The `next.config.ts` file you created during containerization already includes the `output: "standalone"` option for production. For development, Next.js automatically uses its built-in development server with hot reloading enabled.

> [!NOTE] The Next.js development server automatically:
>
> - Enables Hot Module Replacement (HMR) for instant updates
> - Watches for file changes and recompiles automatically
> - Provides detailed error messages in the browser
>
> The `WATCHPACK_POLLING=true` environment variable in the compose file ensures file watching works correctly inside Docker containers.

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

```
├── docker-nextjs-sample/
│ ├── Dockerfile
│ ├── Dockerfile.dev
│ ├── .dockerignore
│ ├── compose.yaml
│ └── next.config.ts
```

Step 4: Start Compose Watch

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

```
$ docker compose watch nextjs-dev
```

Step 5: Test Compose Watch with Next.js

To verify that Compose Watch is working correctly:

1. Open the `app/page.tsx` file in your text editor (or `src/app/page.tsx` if your project uses a `src` directory).
2. Locate the main content area and find a text element to modify.
3. Make a visible change, for example, update a heading:

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

4. Save the file.
5. Open your browser at http://localhost:3000.

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 Next.js tests in a container

Prerequisites

Complete all the previous sections of this guide, starting with Containerize Next.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 (or Jest) inside a Docker container.
- Run lint (e.g. ESLint) inside a Docker container.
- Use Docker Compose to run tests and lint in an isolated, reproducible environment.

The sample project uses Vitest with Testing Library for component testing. You can use the same setup or follow the alternative Jest configuration later.

Run tests during development

The sample project already includes lint (ESLint) and sample tests (Vitest, `app/page.test.tsx`) in place. If you're using the sample app, you can skip to **Step 3: Update compose.yaml** and run tests or lint with the commands below. If you're using your own project, follow the install and configuration steps to add the packages and scripts.

The sample includes a test file at:

```
app/page.test.tsx
```

This file uses Vitest and React Testing Library to verify the behavior of page components.

Step 1: Install Vitest and React Testing Library (custom projects)

If you're using a custom project and haven't already added the necessary testing tools, install them by running:

```
$ npm install --save-dev vitest @vitejs/plugin-react @testing-library/react @testing-library/dom jsdom
```

Then, update the scripts section of your `package.json` file to include:

```
"scripts": {
  "test": "vitest",
  "test:run": "vitest run"
}
```

For lint, add a `lint` script (and optionally `lint:fix`). For example, with ESLint:

```
"scripts": {
  "test": "vitest",
  "test:run": "vitest run",
  "lint": "eslint .",
  "lint:fix": "eslint . --fix"
}
```

The sample project uses `eslint` and `eslint-config-next` for Next.js. Install them in a custom project with:

```
$ npm install --save-dev eslint eslint-config-next @eslint/eslintrc
```

Create an ESLint config file (e.g. `eslint.config.cjs`) in your project root with Next.js rules and global ignores:

```
const { defineConfig, globalIgnores } = require("eslint/config");
const { FlatCompat } = require("@eslint/eslintrc");

const compat = new FlatCompat({ baseDirectory: __dirname });

module.exports = defineConfig([
  ...compat.extends(
    "eslint-config-next/core-web-vitals",
    "eslint-config-next/typescript",
  ),
  globalIgnores([
    ".next/**",
    "out/**",
    "build/**",
    "next-env.d.ts",
    "node_modules/**",
    "eslint.config.cjs",
  ]),
]);
```

Step 2: Configure Vitest (custom projects)

If you're using a custom project, create a `vitest.config.ts` file in your project root (matching the sample project):

```
import { defineConfig } from "vitest/config";
import react from "@vitejs/plugin-react";

export default defineConfig({
  plugins: [react()],
  test: {
    environment: "jsdom",
    setupFiles: "./vitest.setup.ts",
    globals: true,
  },
});
```

Create a `vitest.setup.ts` file in your project root:

```
import "@testing-library/jest-dom/vitest";
```

> [!NOTE] Vitest works well with Next.js and provides fast execution and ESM support. For more details, see the Next.js testing documentation and Vitest docs .

Step 3: Update compose.yaml

Add `nextjs-test` and `nextjs-lint` services to your `compose.yaml` file. In the sample project these services use the `tools` profile so they don't start with a normal `docker compose up`. Both reuse `Dockerfile.dev` and run the test or lint command:

```
services:
  nextjs-prod-standalone:
    build:
      context: .
      dockerfile: Dockerfile
    image: nextjs-sample:prod
    container_name: nextjs-sample-prod
    ports:
      - "3000:3000"

  nextjs-dev:
    build:
      context: .
      dockerfile: Dockerfile.dev
    image: nextjs-sample:dev
    container_name: nextjs-sample-dev
    ports:
      - "3000:3000"
    environment:
      - WATCHPACK_POLLING=true
    develop:
      watch:
        - action: sync
          path: .
          target: /app
```

```
          ignore:
            - node_modules/
            - .next/
        - action: rebuild
          path: package.json

  nextjs-test:
    build:
      context: .
      dockerfile: Dockerfile.dev
    image: nextjs-sample:dev
    container_name: nextjs-sample-test
    command:
      [
        "sh",
        "-c",
        "if [ -f package-lock.json ]; then npm run test:run 2>/dev/null || npm run test -- --
run; elif [ -f yarn.lock ]; then yarn test:run 2>/dev/null || yarn test --run; elif [ -f pnpm-
lock.yaml ]; then pnpm run test:run; else npm run test -- --run; fi",
      ]
    profiles:
      - tools

  nextjs-lint:
    build:
      context: .
      dockerfile: Dockerfile.dev
    image: nextjs-sample:dev
    container_name: nextjs-sample-lint
    command:
      [
        "sh",
        "-c",
        "if [ -f package-lock.json ]; then npm run lint; elif [ -f yarn.lock ]; then yarn
lint; elif [ -f pnpm-lock.yaml ]; then pnpm lint; else npm run lint; fi",
      ]
    profiles:
      - tools
```

The `nextjs-test` and `nextjs-lint` services reuse the same `Dockerfile.dev` used for development and override the default command to run tests or lint. The `profiles: [tools]` means these services only run when you use the `--profile tools` option.

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

```
├── docker-nextjs-sample/
│ ├── Dockerfile
│ ├── Dockerfile.dev
│ ├── .dockerignore
│ ├── compose.yaml
│ ├── vitest.config.ts
│ ├── vitest.setup.ts
│ └── next.config.ts
```

Step 4: Run the tests

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

```
$ docker compose --profile tools run --rm nextjs-test
```

This command will:

- Start the `nextjs-test` service (because of `--profile tools`).
- Run your test script (`test:run` or `test -- --run`) in the same environment as development.
- Remove the container after the tests complete (`docker compose run --rm`).

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

Step 5: Run lint in the container

To run your linter (e.g. ESLint) inside the container, use the `nextjs-lint` service with the same `tools` profile:

```
$ docker compose --profile tools run --rm nextjs-lint
```

This command will:

- Start the `nextjs-lint` service (because of `--profile tools`).

- Run your lint script (`npm run lint`, `yarn lint`, or `pnpm lint` depending on your lockfile) in the same environment as development.
- Remove the container after lint completes.

Ensure your `package.json` includes a `lint` script. The sample project already has `"lint": "eslint ."` and `"lint:fix": "eslint . --fix"`; for a custom project, add the same and install `eslint` and `eslint-config-next` if needed.

Summary

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

What you accomplished:

- Installed and configured Vitest and React Testing Library for testing Next.js components.
- Created `nextjs-test` and `nextjs-lint` services in `compose.yaml` (with `tools` profile) to isolate test and lint execution.
- Reused the development `Dockerfile.dev` to ensure consistency between dev, test, and lint environments.
- Ran tests inside the container using `docker compose --profile tools run --rm nextjs-test`.
- Ran lint inside the container using `docker compose --profile tools run --rm nextjs-lint`.
- Ensured reliable, repeatable testing and linting across environments without relying on 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.
- Next.js Testing Documentation – Official Next.js testing guide.

□□□ #2
□ □□□ □□□ 2026-10-31 18:47:00 UTC
□ □□□ □□□ 2026-10-31 18:47:02 UTC