Create and build a Docker Hardened Image

- How DHI builds differ from Dockerfiles
- Explore the catalog for reference
- Try it: build a catalog image
- YAML schema reference
  - Required fields
  - Image metadata
  - Container configuration
  - Advanced fields
- Create a minimal image
- Use a Debian base with third-party repositories
- Create paths
- Add build stages
- Use OCI artifacts as package sources
- Create a dev variant
- Create a compatibility variant
- Set ports and volumes
- Set annotations
- Build and verify
  - Build the image
  - Inspect the SBOM
  - Scan for vulnerabilities
  - Compare with a non-hardened image
  - Inspect with Docker Debug
- Push to a registry
- Contribute to the catalog

Docker Hardened Images (DHI) are built from declarative YAML definition files instead of traditional Dockerfiles. A single YAML file describes exactly what goes into an image: packages, users, environment variables, entrypoint, and metadata. The DHI build system produces a signed image containing only the required packages, with a Software Bill of Materials (SBOM) and SLSA Build Level 3 provenance.

This page explains how to write a DHI definition file, build images locally, and use advanced patterns such as build stages, third-party repositories, file paths, and dev variants.

> The DHI build system pulls base images and build tools from `dhi.io`, so you must authenticate to that registry before building a definition file. Use your Docker ID credentials (the same username and password you use for Docker Hub) when signing in.
>
> Run `docker login dhi.io` to authenticate.

How DHI builds differ from Dockerfiles

A Dockerfile is a sequence of imperative instructions: `RUN`, `COPY`, `FROM`. A DHI definition file is a declarative specification. You describe the desired state of the image, and the build system figures out how to produce it.

Every DHI definition starts with a syntax directive that tells BuildKit which DHI build frontend to use. The frontend is the component that parses and processes YAML definitions instead of the default Dockerfile parser:

```
# syntax=dhi.io/build:2-alpine3.23
```

The frontend version corresponds to the base distribution:

Distribution	Syntax directive
Alpine 3.22	`# syntax=dhi.io/build:2-alpine3.22`
Alpine 3.23	`# syntax=dhi.io/build:2-alpine3.23`
Alpine 3.24	`# syntax=dhi.io/build:2-alpine3.24`
Debian 12 (Bookworm)	`# syntax=dhi.io/build:2-debian12`
Debian 13 (Trixie)	`# syntax=dhi.io/build:2-debian13`

The DHI build system reads the YAML, resolves packages from the specified repositories, assembles the filesystem, creates user accounts, sets metadata, and produces a signed OCI image.

Explore the catalog for reference

The DHI catalog repository is open source under Apache 2.0 and contains every official image definition. Studying existing definitions is the best way to learn the YAML patterns for different image types.

The catalog follows this directory structure:

```
catalog/
├── image/
│   ├── alpine-base/
│   │   ├── alpine-3.23/
│   │   │   ├── 3.23.yaml            # runtime variant
│   │   │   └── 3.23-dev.yaml        # dev variant
│   │   ├── guides.md
│   │   ├── info.yaml
│   │   ├── logo.svg
│   │   └── overview.md
│   ├── nginx/
│   │   ├── alpine-3.22/
│   │   ├── alpine-3.23/
│   │   │   ├── mainline.yaml
│   │   │   ├── mainline-dev.yaml
│   │   │   ├── stable.yaml
│   │   │   └── stable-dev.yaml
│   │   ├── debian-12/
│   │   ├── debian-13/
│   │   ├── bin/
│   │   ├── guides.md
│   │   ├── info.yaml
│   │   ├── logo.svg
│   │   └── overview.md
│   └── redis/
│       ├── debian-13/
│       │   ├── 8.0.yaml              # runtime
│       │   ├── 8.0-dev.yaml          # dev
```

```
│       │   ├── 8.0-compat.yaml       # compat runtime
│       │   └── 8.0-compat-dev.yaml   # compat dev
│       ├── guides.md
│       ├── info.yaml
│       ├── logo.svg
│       └── overview.md
├── chart/
└── package/
```

Each image organizes its variants by distribution. Images support multiple variant types:

- A `runtime` variant is minimal and typically runs as a non-root user.
- A `dev` variant adds a shell, package manager, and development tools.
- A compatibility variant adds common shell utilities such as `bash`, `coreutils`, `grep`, and `sed` for use with existing workflows. Compatibility images use the `flavor: compat` field alongside a `runtime` or `dev` variant.
- A compatibility-dev variant combines the compatibility packages with dev tools.

Some images also support additional flavors such as `sfw` (software framework) variants. Refer to the catalog for the full list of available variants for each image.

Try it: build a catalog image

Before writing your own definition, try building an existing catalog image directly from GitHub:

```
$ docker buildx build \
    https://raw.githubusercontent.com/docker-hardened-
images/catalog/refs/heads/main/image/alpine-base/alpine-3.23/3.23.yaml \
    --sbom=generator=dhi.io/scout-sbom-indexer:1 \
    --provenance=1 \
    --tag my-alpine-base:3.23 \
    --load
```

This downloads the definition file directly from GitHub and builds it locally. After the build completes, verify the image:

```
$ docker images my-alpine-base
```

To modify an image, clone the catalog and edit the YAML files locally:

```
$ git clone https://github.com/docker-hardened-images/catalog.git
$ cd catalog
```

YAML schema reference

The following sections describe the fields available in a DHI definition file.

Required fields

Every definition must include these top-level fields:

Field	Description
`name`	Human-readable name for the image.
`image`	Full registry path, such as `dhi.io/my-image`.
`variant`	Image variant type: `runtime` or `dev`.
`tags`	List of image tags.
`platforms`	Target architectures, such as `linux/amd64` and `linux/arm64`.
`contents`	Package repositories and packages to install.

Image metadata

These fields add metadata to the image:

Field	Description
`os-release`	Defines the `/etc/os-release` contents inside the image.
`annotations`	OCI image annotations such as description and license.
`dates`	Release date and end-of-life date.
`vars`	Build-time variables for templating.
`flavor`	Image flavor modifier, such as `compat` for compatibility images.

Container configuration

These fields control how the container runs:

Field	Description
`accounts`	Users, groups, and the `run-as` user.
`environment`	Environment variables.
`entrypoint`	Container entrypoint command.
`cmd`	Default command arguments.
`work-dir`	Working directory inside the container.
`volumes`	Volume mount points.
`ports`	Exposed network ports.
`paths`	Directories, files, and symlinks to create.

Advanced fields

These fields support more complex build patterns:

Field	Description
`contents.builds`	Build stages with shell pipelines.
`contents.keyring`	Signing keys for third-party package repositories.
`contents.artifacts`	Pre-built OCI artifacts to include.
`contents.mappings`	Package URL (purl) mappings for SBOM accuracy.
`contents.files`	Source files fetched from Git URLs with checksums.

Create a minimal image

Start with the simplest possible definition: an Alpine base image with a non-root user.

Create a directory for your project and add a file called `base.yaml`:

```
# syntax=dhi.io/build:2-alpine3.23

name: My Base Image
image: my-registry/my-base
variant: runtime
tags:
```

```
  - "1.0.0"
  - "1.0"
platforms:
  - linux/amd64
  - linux/arm64

contents:
  repositories:
    - https://dl-cdn.alpinelinux.org/alpine/v3.23/main
    - https://dl-cdn.alpinelinux.org/alpine/v3.23/community
  packages:
    - alpine-baselayout-data
    - busybox
    - ca-certificates-bundle

accounts:
  run-as: nonroot
  users:
    - name: nonroot
      uid: 65532
      gid: 65532
  groups:
    - name: nonroot
      gid: 65532
      members:
        - nonroot

os-release:
  name: Docker Hardened Images (Alpine)
  id: alpine
  version-id: "3.23"
  pretty-name: My Hardened Image
  home-url: https://docker.com/products/hardened-images/
  bug-report-url: https://docker.com/support/

environment:
  SSL_CERT_FILE: /etc/ssl/certs/ca-certificates.crt

annotations:
  org.opencontainers.image.description: A minimal Alpine base image
```

```
cmd:
  - /bin/sh
```

In this definition:

- `contents.repositories` uses full URLs to Alpine package mirrors.
- `contents.packages` lists exact Alpine package names.
- The `accounts` block creates a `nonroot` user (UID 65532) and sets it as the default user for the container.
- The `os-release` block defines what appears in `/etc/os-release`. Always include `bug-report-url` alongside `home-url`.
- The `annotations` block adds OCI metadata visible in registries and Docker Scout reports.

Build the image:

```
$ docker buildx build . -f base.yaml \
    --sbom=generator=dhi.io/scout-sbom-indexer:1 \
    --provenance=1 \
    --tag my-base:latest \
    --load
```

> The `tags` field in the spec file defines the image metadata (variant and version labels embedded in the image manifest). The `--tag` flag on the CLI sets the OCI image reference used to push or load the image. These serve different purposes - the spec file tags describe *what the image is*, while the CLI tag determines *where it's stored*.

Use a Debian base with third-party repositories

For applications that require Debian packages or third-party APT repositories, use the Debian syntax directive. The following example builds a Redis image from the official Redis APT repository.

Create a file called `redis.yaml`:

```
# syntax=dhi.io/build:2-debian13

name: Redis 8.0.x
```

```
image: my-registry/my-redis
variant: runtime
tags:
  - "8.0"
  - "8.0.5"
platforms:
  - linux/amd64
  - linux/arm64

contents:
  repositories:
    - deb [signed-by=/usr/share/keyrings/redis-archive-keyring.gpg]
https://packages.redis.io/deb trixie main
  keyring:
    - https://packages.redis.io/gpg
  packages:
    - '!libelogind0'
    - '!mawk'
    - '!original-awk'
    - base-files
    - libpcre2-8-0
    - libssl3t64
    - libstdc++6
    - libsystemd0
    - redis=6:8.0.5-1rl1~trixie1
    - redis-server=6:8.0.5-1rl1~trixie1
    - redis-tools=6:8.0.5-1rl1~trixie1
    - tini
  mappings:
    redis: pkg:deb/redis/redis@6:8.0.5-1rl1~trixie1?os_name=debian&os_version=13
    redis-server: pkg:deb/redis/redis-server@6:8.0.5-1rl1~trixie1?os_name=debian&os_version=13
    redis-tools: pkg:deb/redis/redis-tools@6:8.0.5-1rl1~trixie1?os_name=debian&os_version=13

accounts:
  run-as: nonroot
  users:
    - name: nonroot
      uid: 65532
      gid: 65532
  groups:
```

```
    - name: nonroot
      gid: 65532
      members:
        - nonroot

os-release:
  name: Docker Hardened Images (Debian)
  id: debian
  version-id: "13"
  version-codename: trixie
  pretty-name: Docker Hardened Images/Debian GNU/Linux 13 (trixie)
  home-url: https://docker.com/products/hardened-images/
  bug-report-url: https://docker.com/support/

work-dir: /data

environment:
  REDIS_VERSION: 8.0.5

annotations:
  org.opencontainers.image.description: A minimal Redis image
  org.opencontainers.image.licenses: AGPL-3.0-only

entrypoint:
  - /usr/bin/tini
  - --

cmd:
  - redis-server
  - /etc/redis/redis.conf
  - --include
  - /etc/redis/conf.d/*.conf
```

This example introduces several patterns:

- **Third-party repositories**: The `repositories` field uses the Debian `deb [signed-by=...] URL suite component` format for APT sources.
- **Keyring**: The `keyring` field downloads the GPG key used to verify packages from the third-party repository.
- **Package exclusions**: Prefix a package name with `!` to explicitly exclude it. This prevents unwanted dependencies from being installed. In this case, `!libelogind0`, `!mawk`,

and `!original-awk` are excluded.
- **Debian version pinning**: Use the full epoch format, `redis-server=6:8.0.5-1rl1~trixie1`, to pin exact package versions.
- **SBOM mappings**: The `mappings` field provides Package URL (purl) metadata so that Docker Scout can accurately identify the software in the SBOM.
- **Init process**: The `entrypoint` uses `tini` as a lightweight init process (PID 1) to handle signal forwarding and zombie process reaping.
- **Config includes**: The `cmd` uses `--include /etc/redis/conf.d/*.conf` so that configuration files created in the `paths` section are loaded at startup.

Create paths

Use the `paths` field to create directories, files with inline content, and symlinks inside the image. The following example extends the Redis definition with the paths required for operation:

```
paths:
  - type: directory
    path: /var/lib/redis
    uid: 65532
    gid: 65532
    mode: "0755"
  - type: directory
    path: /var/log/redis
    uid: 65532
    gid: 65532
    mode: "0755"
  - type: directory
    path: /run/redis/
    uid: 65532
    gid: 65532
    mode: "0755"
  - type: directory
    path: /data
    uid: 65532
    gid: 65532
    mode: "0755"
  - type: file
    path: /etc/redis/conf.d/docker.conf
    content: |
      daemonize no
```

```
      bind 0.0.0.0 -::1
      logfile ""
    uid: 0
    gid: 0
    mode: "0555"
  - type: symlink
    path: /usr/bin/redis-sentinel
    uid: 0
    gid: 0
    source: /usr/bin/redis-check-rdb
```

Three path types are available:

Type	Required fields	Description
`directory`	`path`, `uid`, `gid`, `mode`	Creates an empty directory.
`file`	`path`, `content`, `uid`, `gid`, `mode`	Creates a file with inline content.
`symlink`	`path`, `source`, `uid`, `gid`	Creates a symbolic link.

The `mode` field uses a string representation of the octal permission bits, such as `"0755"` for read-write-execute by owner or `"0555"` for read-execute by all. Note that the `file` type supports inline `content` using a YAML multi-line string.

Add build stages

For images that need to run shell commands during the build, such as configuring files, creating symlinks, or adjusting permissions, use the `contents.builds` field. Each build stage has its own packages, a pipeline of named steps, and output mappings.

The following example configures Nginx during the build to run on an unprivileged port and disable server tokens:

```
# syntax=dhi.io/build:2-alpine3.23

name: Nginx mainline
image: my-registry/my-nginx
variant: runtime
tags:
  - "1.29"
platforms:
```

```
  - linux/amd64
  - linux/arm64

contents:
  repositories:
    - https://dl-cdn.alpinelinux.org/alpine/v3.23/main
    - https://dl-cdn.alpinelinux.org/alpine/v3.23/community
    - http://nginx.org/packages/mainline/alpine/v3.23/main
  keyring:
    - https://nginx.org/keys/nginx_signing.rsa.pub
  packages:
    - alpine-baselayout-data
    - busybox
    - musl-utils
    - nginx=1.29.5-r1
  builds:
    - name: nginx
      contents:
        repositories:
          - https://dl-cdn.alpinelinux.org/alpine/v3.23/main
          - https://dl-cdn.alpinelinux.org/alpine/v3.23/community
          - http://nginx.org/packages/mainline/alpine/v3.23/main
        keyring:
          - https://nginx.org/keys/nginx_signing.rsa.pub
        packages:
          - alpine-baselayout-data
          - bash
          - musl-utils
          - nginx=1.29.5-r1
      pipeline:
        - name: install
          runs: |
            set -eux -o pipefail

            ln -sf /dev/stdout /var/log/nginx/access.log
            ln -sf /dev/stderr /var/log/nginx/error.log

            sed -i "s,listen       80;,listen       8080;," /etc/nginx/conf.d/default.conf
            sed -i "/user  nginx;/d" /etc/nginx/nginx.conf
            sed -i "s,pid        /run/nginx.pid;,pid        /var/run/nginx.pid;,"
```

```
/etc/nginx/nginx.conf
            sed -i '/^http {$/a\    server_tokens off;' /etc/nginx/nginx.conf

            chown -R 65532:65532 /var/cache/nginx
            chmod -R g+w /var/cache/nginx
            chown -R 65532:65532 /etc/nginx
            chmod -R g+w /etc/nginx
            chown -R 65532:65532 /run
            chown -R 65532:65532 /run/lock
            chown -R 65532:65532 /var/run
            chown -R 65532:65532 /var/log/nginx
      outputs:
        - source: /
          target: /
          uid: 0
          gid: 0
          diff: true

accounts:
  run-as: nginx
  users:
    - name: nginx
      uid: 65532
      gid: 65532
  groups:
    - name: nginx
      gid: 65532
      members:
        - nginx
    - name: www-data
      gid: 82

os-release:
  name: Docker Hardened Images (Alpine)
  id: alpine
  version-id: "3.23"
  pretty-name: Docker Hardened Images/Alpine Linux v3.23
  home-url: https://docker.com/products/hardened-images/
  bug-report-url: https://docker.com/support/
```

```
environment:
  NGINX_VERSION: 1.29.5-r1

annotations:
  org.opencontainers.image.description: A minimal Nginx image
  org.opencontainers.image.licenses: BSD-2-Clause

entrypoint:
  - nginx

cmd:
  - -g
  - daemon off;

ports:
  - 8080/tcp
```

Key patterns in this definition:

Element	Description
`contents`	Each build stage has its own `contents` section. Include packages needed only during the build, such as `bash`.
`pipeline`	Contains named steps that run shell commands. Always start scripts with `set -eux -o pipefail`.
`outputs`	Copies results from the build stage into the final image. Setting `diff: true` copies only files that changed, keeping the image minimal.
`accounts`	Nginx uses a dedicated `nginx` user (UID 65532) instead of `nonroot`. The `www-data` group (GID 82) is also created for web server compatibility.
`musl-utils`	Required in both the main and build packages for Alpine-based Nginx images.

Use OCI artifacts as package sources

Instead of installing packages from Alpine or Debian repositories, you can pull pre-built binaries from DHI package artifacts. This is how the catalog builds images like Python and Node.js — the

runtime is compiled separately and published as an OCI artifact, then referenced by digest in the image definition.

Add the `artifacts` field under `contents`:

```
contents:
  repositories:
    - https://dl-cdn.alpinelinux.org/alpine/v3.23/main
    - https://dl-cdn.alpinelinux.org/alpine/v3.23/community
  packages:
    - alpine-baselayout-data
    - bzip2
    - ca-certificates-bundle
    - expat
    - gdbm
    - libffi
    - mpdecimal
    - musl
    - ncurses
    - openssl
    - readline
    - sqlite-libs
    - tzdata
    - zlib
  artifacts:
    - name: dhi.io/pkg-python:3.13.12-
alpine3.23@sha256:052b3b915055006a27c42470eed5c65d7ee92d2c3de47ecaedcc6bbd36077b95
      includes:
        - opt/**
      uid: 0
      gid: 0
```

Field	Description
`name`	Full OCI reference with digest pin. Always use `@sha256:` for reproducibility.
`includes`	Glob patterns for files to extract from the artifact. Paths are resolved from the filesystem root; `opt/**` includes everything under the `/opt` path.
`excludes`	Glob patterns for files to skip. Useful for removing headers, docs, or unused binaries.
`uid`, `gid`	Ownership for extracted files.

Available DHI packages are in the `package/` directory of the catalog repository.

Create a dev variant

A dev variant of an image adds a shell, package manager, and development tools. This is useful for debugging and for use as a build stage in multi-stage workflows.

To create a dev variant, change the `variant` field and enable root access:

```
# syntax=dhi.io/build:2-alpine3.23

name: Alpine 3.23 Base (dev)
image: my-registry/my-base
variant: dev
tags:
  - "1.0-dev"
platforms:
  - linux/amd64
  - linux/arm64

contents:
  repositories:
    - https://dl-cdn.alpinelinux.org/alpine/v3.23/main
    - https://dl-cdn.alpinelinux.org/alpine/v3.23/community
  packages:
    - alpine-baselayout-data
    - apk-tools
    - busybox
    - ca-certificates-bundle

accounts:
  root: true
  run-as: root
  users:
    - name: nonroot
      uid: 65532
      gid: 65532
  groups:
    - name: nonroot
```

```
      gid: 65532
      members:
        - nonroot

os-release:
  name: Docker Hardened Images (Alpine)
  id: alpine
  version-id: "3.23"
  pretty-name: Docker Hardened Images/Alpine Linux v3.23
  home-url: https://docker.com/products/hardened-images/
  bug-report-url: https://docker.com/support/

environment:
  SSL_CERT_FILE: /etc/ssl/certs/ca-certificates.crt

annotations:
  org.opencontainers.image.description: A minimal Alpine base image

cmd:
  - /bin/sh
```

The key differences from a runtime variant:

- `variant: dev` instead of `variant: runtime`.
- `accounts.root: true` enables the root account.
- `run-as: root` sets root as the default user.
- `apk-tools` is added to packages, giving the image a package manager.
- The `nonroot` user is still defined so that applications can switch to an unprivileged user at runtime.

For Debian-based dev variants, add `apt` instead of `apk-tools` and include the `DEBIAN_FRONTEND: noninteractive` environment variable.

Create a compatibility variant

A compatibility variant includes common shell utilities for use with scripts and automation tools that expect a standard Linux userland. Compatibility images use the `flavor` field:

```
variant: runtime
flavor: compat
```

A compatibility variant adds packages such as `bash`, `coreutils`, `findutils`, `grep`, `hostname`, `openssl`, `procps`, and `sed` alongside the application packages. A compatibility-dev variant combines both the compatibility packages and the dev tools:

```
variant: dev
flavor: compat
```

Refer to the Redis compatibility images in the catalog for a complete example of the compatibility pattern.

Set ports and volumes

Use the `ports` field to declare which ports the container exposes. Always use unprivileged ports (higher than 1024) when the container runs as a non-root user.

```
ports:
  - 8080/tcp
```

Use the `volumes` field to declare volume mount points:

```
volumes:
  - /data
```

Set annotations

OCI annotations add machine-readable metadata to the image. Use the `annotations` field:

```
annotations:
  org.opencontainers.image.description: A minimal hardened application image
  org.opencontainers.image.licenses: Apache-2.0
```

These annotations appear in Docker Scout reports and container registry interfaces.

Build and verify

Build the image

Build a single-platform image for local testing:

```
$ docker buildx build . -f my-image.yaml \
    --sbom=generator=dhi.io/scout-sbom-indexer:1 \
    --provenance=1 \
    --tag my-image:latest \
    --load
```

Inspect the SBOM

View the generated Software Bill of Materials:

```
$ docker scout sbom my-image:latest
```

Scan for vulnerabilities

Check the image against known CVE databases:

```
$ docker scout cves my-image:latest
```

Compare with a non-hardened image

Measure the security improvement against an equivalent non-hardened image:

```
$ docker scout compare my-image:latest \
    --to <non-hardened-equivalent>:<tag> \
    --platform linux/amd64
```

Replace `<non-hardened-equivalent>` with the Docker Official Image or community image you're comparing against.

Inspect with Docker Debug

Verify the os-release and entrypoint configuration:

```
$ docker debug my-image:latest
```

The output shows the detected distribution name from your `os-release` configuration and runs an entrypoint lint check.

Push to a registry

Tag and push the image to your container registry:

```
$ docker tag my-image:latest <your-namespace>/my-image:latest
```

```
$ docker push <your-namespace>/my-image:latest
```

Replace `<your-namespace>` with your Docker Hub username or organization namespace.

Contribute to the catalog

Docker Hardened Images is an open source project. You can contribute new image definitions or improve existing ones by submitting a pull request to the catalog repository.

To contribute a new image:

1. Fork the catalog repository.
2. Create a directory under `image/` following the naming convention: `image/<image-name>/<distribution>/`.
3. Add your YAML definition files (one per variant).
4. Add an `info.yaml` with display name, description, and categories.
5. Add an `overview.md` describing the image.
6. Add a `logo.svg` for the image icon.
7. Add a `guides.md` with usage documentation.
8. Open a pull request against the `main` branch.

For more details, read the contributing guide in the catalog repository.

□□□ #2
□ □□□ □□□ 2026-10-31 19:16:52 UTC
□ □□□ □□□ 2026-10-31 19:16:54 UTC