# Compose Build Specification

Table of contents



---

Build is an optional part of the Compose Specification. It tells Compose how to (re)build an application from source and lets you define the build process within a Compose file in a portable way. `build` can be either specified as a single string defining a context path, or as a detailed build definition.

In the former case, the whole path is used as a Docker context to execute a Docker build, looking for a canonical `Dockerfile` at the root of the directory. The path can be absolute or relative. If it is relative, it is resolved from the Compose file's parent folder. If it is absolute, the path prevents the Compose file from being portable so Compose displays a warning.

In the latter case, build arguments can be specified, including an alternate `Dockerfile` location. The path can be absolute or relative. If it is relative, it is resolved from the Compose file's parent folder. If it is absolute, the path prevents the Compose file from being portable so Compose displays a warning.

## Using `build` and `image`

When Compose is confronted with both a `build` subsection for a service and an `image` attribute, it follows the rules defined by the `pull_policy` attribute.

If `pull_policy` is missing from the service definition, Compose attempts to pull the image first and then builds from source if the image isn't found in the registry or platform cache.

## Publishing built images

Compose with `build` support offers an option to push built images to a registry. When doing so, it doesn't try to push service images without an `image` attribute. Compose warns you about the missing `image` attribute which prevents images being pushed.

## Illustrative example

The following example illustrates Compose Build Specification concepts with a concrete sample application. The sample is non-normative.

```
services:
 frontend:
   image: example/webapp
   build: ./webapp

 backend:
   image: example/database
   build:
     context: backend
```

```
    dockerfile: ../backend.Dockerfile

custom:
  build: ~/custom
```

When used to build service images from source, the Compose file creates three Docker images:

- `example/webapp`: A Docker image is built using `webapp` sub-directory, within the Compose file's parent folder, as the Docker build context. Lack of a `Dockerfile` within this folder throws an error.
- `example/database`: A Docker image is built using `backend` sub-directory within the Compose file parent folder. `backend.Dockerfile` file is used to define build steps, this file is searched relative to the context path, which means `..` resolves to the Compose file's parent folder, so `backend.Dockerfile` is a sibling file.
- A Docker image is built using the `custom` directory with the user's HOME as the Docker context. Compose displays a warning about the non-portable path used to build image.

On push, both `example/webapp` and `example/database` Docker images are pushed to the default registry. The `custom` service image is skipped as no `image` attribute is set and Compose displays a warning about this missing attribute.

## Attributes

The `build` subsection defines configuration options that are applied by Compose to build Docker images from source. `build` can be specified either as a string containing a path to the build context or as a detailed structure:

Using the string syntax, only the build context can be configured as either:

A relative path to the Compose file's parent folder. This path must be a directory and must contain a `Dockerfile`

```
services:
 webapp:
   build: ./dir
```

A git repository URL. Git URLs accept context configuration in their fragment section, separated by a colon (`:`). The first part represents the reference that Git checks out, and can be either a branch, a tag, or a remote reference. The second part represents a subdirectory inside the repository that is used as a build context.

```
services:
 webapp:
   build: https://github.com/mycompany/example.git#branch_or_tag:subdirectory
```

Alternatively `build` can be an object with fields defined as follows:

### context

`context` defines either a path to a directory containing a Dockerfile, or a URL to a git repository.

When the value supplied is a relative path, it is interpreted as relative to the location of the Compose file. Compose warns you about the absolute path used to define the build context as those prevent the Compose file from being portable.

```
build:
 context: ./dir

services:
 webapp:
   build: https://github.com/mycompany/webapp.git
```

If not set explicitly, `context` defaults to project directory (`.`).

### dockerfile

`dockerfile` sets an alternate Dockerfile. A relative path is resolved from the build context. Compose warns you about the absolute path used to define the Dockerfile as it prevents Compose files from being portable.

When set, `dockerfile_inline` attribute is not allowed and Compose rejects any Compose file having both set.

```
build:
 context: .
 dockerfile: webapp.Dockerfile
```

### dockerfile_inline

Introduced in Docker Compose version 2.17.0

`dockerfile_inline` defines the Dockerfile content as an inlined string in a Compose file. When set, the `dockerfile` attribute is not allowed and Compose rejects any Compose file having both set.

Use of YAML multi-line string syntax is recommended to define the Dockerfile content:

```
build:
 context: .
 dockerfile_inline: |
   FROM baseimage
   RUN some command
```

### args

`args` define build arguments, i.e. Dockerfile `ARG` values.

Using the following Dockerfile as an example:

```
ARG GIT_COMMIT
RUN echo "Based on commit: $GIT_COMMIT"
```

`args` can be set in the Compose file under the `build` key to define `GIT_COMMIT`. `args` can be set as a mapping or a list:

```
build:
 context: .
 args:
   GIT_COMMIT: cdc3b19
```

```
build:
 context: .
 args:
   - GIT_COMMIT=cdc3b19
```

Values can be omitted when specifying a build argument, in which case its value at build time must be obtained by user interaction, otherwise the build arg won't be set when building the Docker image.

```
args:
 - GIT_COMMIT
```

### ssh

`ssh` defines SSH authentications that the image builder should use during image build (e.g., cloning private repository).

`ssh` property syntax can be either:

- `default`: Let the builder connect to the ssh-agent.
- `ID=path`: A key/value definition of an ID and the associated path. It can be either a PEM file, or path to ssh-agent socket.

```
build:
 context: .
 ssh:
   - default   # mount the default ssh agent
```

or

```
build:
 context: .
 ssh: ["default"]   # mount the default ssh agent
```

Using a custom id `myproject` with path to a local SSH key:

```
build:
 context: .
 ssh:
   - myproject=~/.ssh/myproject.pem
```

The image builder can then rely on this to mount the SSH key during build. For illustration, BuildKit extended syntax can be used to mount the SSH key set by ID and access a secured resource:

```
RUN --mount=type=ssh,id=myproject git clone ...
```

### cache_from

`cache_from` defines a list of sources the image builder should use for cache resolution.

Cache location syntax follows the global format `[NAME|type=TYPE[,KEY=VALUE]]`. Simple `NAME` is actually a shortcut notation for `type=registry,ref=NAME`.

Compose Build implementations may support custom types, the Compose Specification defines canonical types which must be supported:

- `registry` to retrieve build cache from an OCI image set by key `ref`

```
build:
 context: .
 cache_from:
   - alpine:latest
   - type=local,src=path/to/cache
   - type=gha
```

Unsupported caches are ignored and don't prevent you from building images.

### cache_to

`cache_to` defines a list of export locations to be used to share build cache with future builds.

```
build:
 context: .
 cache_to:
  - user/app:cache
  - type=local,dest=path/to/cache
```

Cache target is defined using the same `type=TYPE[,KEY=VALUE]` syntax defined by `cache_from`.

Unsupported caches are ignored and don't prevent you from building images.

### additional_contexts

Introduced in Docker Compose version 2.17.0

`additional_contexts` defines a list of named contexts the image builder should use during image build.

`additional_contexts` can be a mapping or a list:

```
build:
 context: .
 additional_contexts:
   - resources=/path/to/resources
   - app=docker-image://my-app:latest
   - source=https://github.com/myuser/project.git
```

```
build:
 context: .
 additional_contexts:
   resources: /path/to/resources
   app: docker-image://my-app:latest
   source: https://github.com/myuser/project.git
```

When used as a list, the syntax follows the `NAME=VALUE` format, where `VALUE` is a string. Validation beyond that is the responsibility of the image builder (and is builder specific). Compose supports at least absolute and relative paths to a directory AND Git repository URLs, like context does. Other context flavours must be prefixed to avoid ambiguity with a `type://` prefix.

Compose warns you if the image builder does not support additional contexts and may list the unused contexts.

Illustrative examples of how this is used in Buildx can be found here.

### extra_hosts

`extra_hosts` adds hostname mappings at build-time. Use the same syntax as extra_hosts.

```
extra_hosts:
 - "somehost=162.242.195.82"
 - "otherhost=50.31.209.229"
 - "myhostv6=::1"
```

IPv6 addresses can be enclosed in square brackets, for example:

```
extra_hosts:
 - "myhostv6=[::1]"
```

The separator = is preferred, but `:` can also be used. Introduced in Docker Compose version 2.24.1. For example:

```
extra_hosts:
 - "somehost:162.242.195.82"
 - "myhostv6:::1"
```

Compose creates matching entry with the IP address and hostname in the container's network configuration, which means for Linux `/etc/hosts` will get extra lines:

```
162.242.195.82  somehost
50.31.209.229   otherhost
::1             myhostv6
```

### isolation

`isolation` specifies a buildâ■■s container isolation technology. Like isolation, supported values are platform specific.

### privileged

Introduced in Docker Compose version 2.15.0

`privileged` configures the service image to build with elevated privileges. Support and actual impacts are platform specific.

```
build:
 context: .
 privileged: true
```

### labels

`labels` add metadata to the resulting image. `labels` can be set either as an array or a map.

It's recommended that you use reverse-DNS notation to prevent your labels from conflicting with other software.

```
build:
 context: .
 labels:
   com.example.description: "Accounting webapp"
   com.example.department: "Finance"
   com.example.label-with-empty-value: ""
```

```
build:
 context: .
 labels:
   - "com.example.description=Accounting webapp"
   - "com.example.department=Finance"
   - "com.example.label-with-empty-value"
```

### no_cache

`no_cache` disables image builder cache and enforces a full rebuild from source for all image layers. This only applies to layers declared in the Dockerfile, referenced images COULD be retrieved from local image store whenever tag has been updated on registry (see pull).

### pull

`pull` requires the image builder to pull referenced images (`FROM` Dockerfile directive), even if those are already available in the local image store.

### network

Set the network containers connect to for the `RUN` instructions during build.

```
build:
 context: .
 network: host
```

```
build:
 context: .
 network: custom_network_1
```

Use `none` to disable networking during build:

```
build:
 context: .
 network: none
```

### shm_size

`shm_size` sets the size of the shared memory (`/dev/shm` partition on Linux) allocated for building Docker images. Specify as an integer value representing the number of bytes or as a string expressing a byte value.

```
build:
 context: .
 shm_size: '2gb'
```

```
build:
 context: .
 shm_size: 10000000
```

### target

`target` defines the stage to build as defined inside a multi-stage `Dockerfile`.

```
build:
 context: .
 target: prod
```

### secrets

`secrets` grants access to sensitive data defined by secrets on a per-service build basis. Two different syntax variants are supported: the short syntax and the long syntax.

Compose reports an error if the secret isn't defined in the `secrets` section of this Compose file.

#### Short syntax

The short syntax variant only specifies the secret name. This grants the container access to the secret and mounts it as read-only to `/run/secrets/<secret_name>` within the container. The source name and destination mountpoint are both set to the secret name.

The following example uses the short syntax to grant the build of the `frontend` service access to the `server-certificate` secret. The value of `server-certificate` is set to the contents of the file `./server.cert`.

```
services:
 frontend:
   build:
     context: .
     secrets:
       - server-certificate
secrets:
 server-certificate:
   file: ./server.cert
```

#### Long syntax

The long syntax provides more granularity in how the secret is created within the service's containers.

- `source`: The name of the secret as it exists on the platform.
- `target`: The name of the file to be mounted in `/run/secrets/` in the service's task containers. Defaults to `source` if not specified.
- `uid` and `gid`: The numeric UID or GID that owns the file within `/run/secrets/` in the service's task containers. Default value is USER running container.
- `mode`: The permissions for the file to be mounted in `/run/secrets/` in the service's task containers, in octal notation. Default value is world-readable permissions (mode `0444`). The writable bit must be ignored if set. The executable bit may be set.

The following example sets the name of the `server-certificate` secret file to `server.crt` within the container, sets the mode to `0440` (group-readable) and sets the user and group to `103`. The value of `server-certificate` secret is provided by the platform through a lookup and the secret lifecycle not directly managed by Compose.

```
services:
 frontend:
   build:
     context: .
     secrets:
       - source: server-certificate
         target: server.cert
```

```
        uid: "103"
        gid: "103"
        mode: 0440
secrets:
 server-certificate:
   external: true
```

Service builds may be granted access to multiple secrets. Long and short syntax for secrets may be used in the same Compose file. Defining a secret in the top-level `secrets` must not imply granting any service build access to it. Such grant must be explicit within service specification as secrets service element.

### tags

`tags` defines a list of tag mappings that must be associated to the build image. This list comes in addition to the `image` property defined in the service section

```
tags:
 - "myimage:mytag"
 - "registry/username/myrepos:my-other-tag"
```

### ulimits

Introduced in Docker Compose version 2.23.1

`ulimits` overrides the default ulimits for a container. It's specified either as an integer for a single limit or as mapping for soft/hard limits.

```
services:
 frontend:
   build:
     context: .
     ulimits:
       nproc: 65535
       nofile:
         soft: 20000
         hard: 40000
```

### platforms

`platforms` defines a list of target platforms.

```
build:
 context: "."
 platforms:
   - "linux/amd64"
   - "linux/arm64"
```

When the `platforms` attribute is omitted, Compose includes the service's platform in the list of the default build target platforms.

When the `platforms` attribute is defined, Compose includes the service's platform, otherwise users won't be able to run images they built.

Composes reports an error in the following cases:

When the list contains multiple platforms but the implementation is incapable of storing multi-platform images.

When the list contains an unsupported platform.

```
build:
 context: "."
 platforms:
   - "linux/amd64"
   - "unsupported/unsupported"
```

When the list is non-empty and does not contain the service's platform

```
services:
 frontend:
   platform: "linux/amd64"
   build:
     context: "."
     platforms:
       - "linux/arm64"
```