Build context

- What is a build context?
  - Filesystem contexts
  - Text file contexts
- Local context
  - Local directories
  - Local context with Dockerfile from stdin
  - Local tarballs
- Remote context
  - Git repositories
    - URL fragments
    - URL queries
    - Keep .git directory
    - Private repositories
  - Remote context with Dockerfile from stdin
  - Remote tarballs
- Empty context
  - How to build without a context
- .dockerignore files
  - Filename and location
  - Syntax
    - Matching
    - Negating matches
- Named contexts
  - Using named contexts in a Dockerfile
  - Use cases for named contexts
    - Example: combine local and remote sources
    - Example: dynamic builds with custom dependencies
    - Example: pin or override images

- Named contexts with Bake
  - Linking targets with named contexts
- Further reading

The `docker build` and `docker buildx build` commands build Docker images from a Dockerfile and a context.

What is a build context?

The build context is the set of files that your build can access. The positional argument that you pass to the build command specifies the context that you want to use for the build:

```
$ docker build [OPTIONS] PATH | URL | -
                         ^^^^^^^^^^^^^^
```

You can pass any of the following inputs as the context for a build:

- The relative or absolute path to a local directory
- A remote URL of a Git repository, tarball, or plain-text file
- A plain-text file or tarball piped to the `docker build` command through standard input

Filesystem contexts

When your build context is a local directory, a remote Git repository, or a tar file, then that becomes the set of files that the builder can access during the build. Build instructions such as `COPY` and `ADD` can refer to any of the files and directories in the context.

A filesystem build context is processed recursively:

- When you specify a local directory or a tarball, all subdirectories are included
- When you specify a remote Git repository, the repository and all submodules are included

For more information about the different types of filesystem contexts that you can use with your builds, see:

- Local files
- Git repositories
- Remote tarballs

Text file contexts

When your build context is a plain-text file, the builder interprets the file as a Dockerfile. With this approach, the build doesn't use a filesystem context.

For more information, see empty build context.

Local context

To use a local build context, you can specify a relative or absolute filepath to the `docker build` command. The following example shows a build command that uses the current directory (`.`) as a build context:

```
$ docker build .
...
#16 [internal] load build context
#16 sha256:23ca2f94460dcbaf5b3c3edbaaa933281a4e0ea3d92fe295193e4df44dc68f85
#16 transferring context: 13.16MB 2.2s done
...
```

This makes files and directories in the current working directory available to the builder. The builder loads the files it needs from the build context when needed.

You can also use local tarballs as build context, by piping the tarball contents to the `docker build` command. See Tarballs.

Local directories

Consider the following directory structure:

```
.
├── index.ts
├── src/
├── Dockerfile
├── package.json
└── package-lock.json
```

Dockerfile instructions can reference and include these files in the build if you pass this directory as a context.

```
# syntax=docker/dockerfile:1
FROM node:latest
WORKDIR /src
COPY package.json package-lock.json .
RUN npm ci
COPY index.ts src .
```

```
$ docker build .
```

Local context with Dockerfile from stdin

Use the following syntax to build an image using files on your local filesystem, while using a Dockerfile from stdin.

```
$ docker build -f- <PATH>
```

The syntax uses the -f (or --file) option to specify the Dockerfile to use, and it uses a hyphen (-) as filename to instruct Docker to read the Dockerfile from stdin.

The following example uses the current directory (.) as the build context, and builds an image using a Dockerfile passed through stdin using a here-document.

```
# create a directory to work in
mkdir example
cd example

# create an example file
touch somefile.txt

# build an image using the current directory as context
# and a Dockerfile passed through stdin
docker build -t myimage:latest -f- . <<EOF
FROM busybox
COPY somefile.txt ./
RUN cat /somefile.txt
EOF
```

Local tarballs

When you pipe a tarball to the build command, the build uses the contents of the tarball as a filesystem context.

For example, given the following project directory:

```
.
├── Dockerfile
├── Makefile
├── README.md
├── main.c
├── scripts
├── src
└── test.Dockerfile
```

You can create a tarball of the directory and pipe it to the build for use as a context:

```
$ tar czf foo.tar.gz *
$ docker build - < foo.tar.gz
```

The build resolves the Dockerfile from the tarball context. You can use the `--file` flag to specify the name and location of the Dockerfile relative to the root of the tarball. The following command builds using `test.Dockerfile` in the tarball:

```
$ docker build --file test.Dockerfile - < foo.tar.gz
```

Remote context

You can specify the address of a remote Git repository, tarball, or plain-text file as your build context.

- For Git repositories, the builder automatically clones the repository. See Git repositories.
- For tarballs, the builder downloads and extracts the contents of the tarball. See Tarballs.

If the remote tarball is a text file, the builder receives no filesystem context, and instead assumes that the remote file is a Dockerfile. See Empty build context.

Git repositories

When you pass a URL pointing to the location of a Git repository as an argument to `docker build`, the builder uses the repository as the build context.

The builder performs a shallow clone of the repository, downloading only the HEAD commit, not the entire history.

The builder recursively clones the repository and any submodules it contains.

```
$ docker build https://github.com/user/myrepo.git
```

By default, the builder clones the latest commit on the default branch of the repository that you specify.

URL fragments

You can append URL fragments to the Git repository address to make the builder clone a specific branch, tag, and subdirectory of a repository.

The format of the URL fragment is `#ref:dir`, where:

- `ref` is the name of the branch, tag, or commit hash
- `dir` is a subdirectory inside the repository

For example, the following command uses the `container` branch, and the `docker` subdirectory in that branch, as the build context:

```
$ docker build https://github.com/user/myrepo.git#container:docker
```

The following table represents all the valid suffixes with their build contexts:

Build Syntax Suffix	Commit Used	Build Context Used
`myrepo.git`	`refs/heads/<default branch>`	`/`
`myrepo.git#mytag`	`refs/tags/mytag`	`/`
`myrepo.git#mybranch`	`refs/heads/mybranch`	`/`
`myrepo.git#pull/42/head`	`refs/pull/42/head`	`/`
`myrepo.git#:myfolder`	`refs/heads/<default branch>`	`/myfolder`
`myrepo.git#master:myfolder`	`refs/heads/master`	`/myfolder`
`myrepo.git#mytag:myfolder`	`refs/tags/mytag`	`/myfolder`
`myrepo.git#mybranch:myfolder`	`refs/heads/mybranch`	`/myfolder`

When you use a commit hash as the `ref` in the URL fragment, use the full, 40-character string SHA-1 hash of the commit. A short hash, for example a hash truncated to 7 characters, is not supported.

```
# ✅ The following works:
docker build github.com/docker/buildx#d4f088e689b41353d74f1a0bfcd6d7c0b213aed2
# ❌ The following doesn't work because the commit hash is truncated:
docker build github.com/docker/buildx#d4f088e
```

URL queries

{{< summary-bar feature_name="Build URL Queries" >}}

URL queries are more structured and recommended over URL fragments:

```
$ docker buildx build 'https://github.com/user/myrepo.git?branch=container&subdir=docker'
```

Build syntax suffix	Commit used	Build context used
`myrepo.git`	`refs/heads/<default branch>`	`/`
`myrepo.git?tag=mytag`	`refs/tags/mytag`	`/`
`myrepo.git?branch=mybranch`	`refs/heads/mybranch`	`/`
`myrepo.git?ref=pull/42/head`	`refs/pull/42/head`	`/`
`myrepo.git?subdir=myfolder`	`refs/heads/<default branch>`	`/myfolder`
`myrepo.git?branch=master&subdir=myfolder`	`refs/heads/master`	`/myfolder`
`myrepo.git?tag=mytag&subdir=myfolder`	`refs/tags/mytag`	`/myfolder`
`myrepo.git?branch=mybranch&subdir=myfolder`	`refs/heads/mybranch`	`/myfolder`

A commit hash can be specified as a `checksum` (alias `commit`) query, along with `tag`, `branch`, or `ref` queries to verify that the reference resolves to the expected commit:

```
$ docker buildx build 'https://github.com/moby/buildkit.git?tag=v0.21.1&checksum=66735c67'
```

If it doesn't match, the build fails:

```
$ docker buildx build 'https://github.com/user/myrepo.git?tag=v0.1.0&commit=deadbeef'
...
#3 [internal] load git source https://github.com/user/myrepo.git?tag=v0.1.0-
rc1&commit=deadbeef
#3 0.484 bb41e835b6c3523c7c45b248cf4b45e7f862bc42       refs/tags/v0.1.0
#3 ERROR: expected checksum to match deadbeef, got bb41e835b6c3523c7c45b248cf4b45e7f862bc42
```

> Short commit hash is supported with `checksum` (alias `commit`) query but for `ref`, only the full hash of the commit is supported.

If your BuildKit daemon or Dockerfile frontend doesn't support URL queries, set `BUILDX_SEND_GIT_QUERY_AS_INPUT=1` to let Buildx resolve them. This changes the context's provenance representation; see the compatibility setting.

Keep `.git` directory

By default, BuildKit doesn't keep the `.git` directory when using Git contexts. You can configure BuildKit to keep the directory by setting the `BUILDKIT_CONTEXT_KEEP_GIT_DIR` build argument. This can be useful if you want to retrieve Git information during your build:

```
# syntax=docker/dockerfile:1
FROM alpine
WORKDIR /src
RUN --mount=target=. \
  make REVISION=$(git rev-parse HEAD) build
```

```
$ docker build \
  --build-arg BUILDKIT_CONTEXT_KEEP_GIT_DIR=1
  https://github.com/user/myrepo.git#main
```

Private repositories

When you specify a Git context that's also a private repository, the builder needs you to provide the necessary authentication credentials. You can use either SSH or token-based authentication.

Buildx automatically detects and uses SSH credentials if the Git context you specify is an SSH or Git address. By default, this uses `$SSH_AUTH_SOCK`. You can configure the SSH credentials to use with the `--ssh` flag.

```
$ docker buildx build --ssh default git@github.com:user/private.git
```

If you want to use token-based authentication instead, you can pass the token using the `--secret` flag.

```
$ GIT_AUTH_TOKEN=<token> docker buildx build \
  --secret id=GIT_AUTH_TOKEN \
  https://github.com/user/private.git
```

> Don't use `--build-arg` for secrets.

Remote context with Dockerfile from stdin

Use the following syntax to build an image using files from a remote URL, while using a Dockerfile from stdin.

```
$ docker build -f- <URL>
```

The syntax uses the -f (or --file) option to specify the Dockerfile to use, and it uses a hyphen (-) as filename to instruct Docker to read the Dockerfile from stdin.

This can be useful in situations where you want to build an image from a repository that doesn't contain a Dockerfile. Or if you want to build with a custom Dockerfile, without maintaining your own fork of the repository.

The following example builds an image using a Dockerfile from stdin, and adds the `hello.c` file from the hello-world repository on GitHub.

```
docker build -t myimage:latest -f- https://github.com/docker-library/hello-world.git <<EOF
FROM busybox
COPY hello.c ./
EOF
```

Remote tarballs

If you pass the URL to a remote tarball, the URL itself is sent to the builder.

```
$ docker build http://server/context.tar.gz
#1 [internal] load remote build context
#1 DONE 0.2s

#2 copy /context /
#2 DONE 0.1s
...
```

The download operation will be performed on the host where the BuildKit daemon is running. Note that if you're using a remote Docker context or a remote builder, that's not necessarily the same machine as where you issue the build command. BuildKit fetches the `context.tar.gz` and uses it as

the build context. Tarball contexts must be tar archives conforming to the standard `tar` Unix format and can be compressed with any one of the `xz`, `bzip2`, `gzip` or `identity` (no compression) formats.

Empty context

When you use a text file as the build context, the builder interprets the file as a Dockerfile. Using a text file as context means that the build has no filesystem context.

You can build with an empty build context when your Dockerfile doesn't depend on any local files.

How to build without a context

You can pass the text file using a standard input stream, or by pointing at the URL of a remote text file.

{{< tabs >}} {{< tab name="Unix pipe" >}}

```
$ docker build - < Dockerfile
```

{{< /tab >}} {{< tab name="PowerShell" >}}

```
Get-Content Dockerfile | docker build -
```

{{< /tab >}} {{< tab name="Heredocs" >}}

```
docker build -t myimage:latest - <<EOF
FROM busybox
RUN echo "hello world"
EOF
```

{{< /tab >}} {{< tab name="Remote file" >}}

```
$ docker build https://raw.githubusercontent.com/dvdksn/clockbox/main/Dockerfile
```

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

When you build without a filesystem context, Dockerfile instructions such as `COPY` can't refer to local files:

```
$ ls
main.c
$ docker build -<<< $'FROM scratch\nCOPY main.c .'
[+] Building 0.0s (4/4) FINISHED
 => [internal] load build definition from Dockerfile       0.0s
 => => transferring dockerfile: 64B                        0.0s
 => [internal] load .dockerignore                          0.0s
 => => transferring context: 2B                            0.0s
 => [internal] load build context                          0.0s
 => => transferring context: 2B                            0.0s
 => ERROR [1/1] COPY main.c .                              0.0s
------
 > [1/1] COPY main.c .:
------
Dockerfile:2
--------------------
   1 |     FROM scratch
   2 | >>> COPY main.c .
   3 |
--------------------
ERROR: failed to solve: failed to compute cache key: failed to calculate checksum of ref
7ab2bb61-0c28-432e-abf5-a4c3440bc6b6::4lgfpdf54n5uqxnv9v6ymg7ih: "/main.c": not found
```

.dockerignore files

You can use a `.dockerignore` file to exclude files or directories from the build context.

```
# .dockerignore
node_modules
bar
```

This helps avoid sending unwanted files and directories to the builder, improving build speed, especially when using a remote builder.

Filename and location

When you run a build command, the build client looks for a file named `.dockerignore` in the root directory of the context. If this file exists, the files and directories that match patterns in the files

are removed from the build context before it's sent to the builder.

If you use multiple Dockerfiles, you can use different ignore-files for each Dockerfile. You do so using a special naming convention for the ignore-files. Place your ignore-file in the same directory as the Dockerfile, and prefix the ignore-file with the name of the Dockerfile, as shown in the following example.

```
.
├── index.ts
├── src/
├── docker
│   ├── build.Dockerfile
│   ├── build.Dockerfile.dockerignore
│   ├── lint.Dockerfile
│   ├── lint.Dockerfile.dockerignore
│   ├── test.Dockerfile
│   └── test.Dockerfile.dockerignore
├── package.json
└── package-lock.json
```

A Dockerfile-specific ignore-file takes precedence over the `.dockerignore` file at the root of the build context if both exist.

Syntax

The `.dockerignore` file is a newline-separated list of patterns similar to the file globs of Unix shells. Leading and trailing slashes in ignore patterns are disregarded. The following patterns all exclude a file or directory named `bar` in the subdirectory `foo` under the root of the build context:

- `/foo/bar/`
- `/foo/bar`
- `foo/bar/`
- `foo/bar`

If a line in `.dockerignore` file starts with `#` in column 1, then this line is considered as a comment and is ignored before interpreted by the CLI.

```
#/this/is/a/comment
```

If you're interested in learning the precise details of the `.dockerignore` pattern matching logic, check out the moby/patternmatcher repository on GitHub, which contains the source code.

Matching

The following code snippet shows an example `.dockerignore` file.

```
# comment
*/temp*
*/*/temp*
temp?
```

This file causes the following build behavior:

Rule	Behavior
`# comment`	Ignored.
`*/temp*`	Exclude files and directories whose names start with `temp` in any immediate subdirectory of the root. For example, the plain file `/somedir/temporary.txt` is excluded, as is the directory `/somedir/temp`.
`*/*/temp*`	Exclude files and directories starting with `temp` from any subdirectory that is two levels below the root. For example, `/somedir/subdir/temporary.txt` is excluded.
`temp?`	Exclude files and directories in the root directory whose names are a one-character extension of `temp`. For example, `/tempa` and `/tempb` are excluded.

Matching is done using Go's `filepath.Match` function rules. A preprocessing step uses Go's `filepath.Clean` function to trim whitespace and remove `.` and `..`. Lines that are blank after preprocessing are ignored.

> For historical reasons, the pattern `.` is ignored.

Beyond Go's `filepath.Match` rules, Docker also supports a special wildcard string `**` that matches any number of directories (including zero). For example, `**/*.go` excludes all files that end with `.go` found anywhere in the build context.

You can use the `.dockerignore` file to exclude the `Dockerfile` and `.dockerignore` files. These files are still sent to the builder as they're needed for running the build. But you can't copy the files into the image using `ADD`, `COPY`, or bind mounts.

Negating matches

You can prepend lines with a `!` (exclamation mark) to make exceptions to exclusions. The following is an example `.dockerignore` file that uses this mechanism:

```
*.md
!README.md
```

All markdown files right under the context directory *except* `README.md` are excluded from the context. Note that markdown files under subdirectories are still included.

The placement of `!` exception rules influences the behavior: the last line of the `.dockerignore` that matches a particular file determines whether it's included or excluded. Consider the following example:

```
*.md
!README*.md
README-secret.md
```

No markdown files are included in the context except README files other than `README-secret.md`.

Now consider this example:

```
*.md
README-secret.md
!README*.md
```

All of the README files are included. The middle line has no effect because `!README*.md` matches `README-secret.md` and comes last.

Named contexts

In addition to the default build context (the positional argument to the `docker build` command), you can also pass additional named contexts to builds.

Named contexts are specified using the `--build-context` flag, followed by a name-value pair. This lets you include files and directories from multiple sources during the build, while keeping them logically separated.

```
$ docker build --build-context docs=./docs .
```

In this example:

- The named `docs` context points to the `./docs` directory.
- The default context (`.`) points to the current working directory.

Using named contexts in a Dockerfile

Dockerfile instructions can reference named contexts as if they are stages in a multi-stage build.

For example, the following Dockerfile:

1. Uses a `COPY` instruction to copy files from the default context into the current build stage.
2. Bind mounts the files in a named context to process the files as part of the build.

```
# syntax=docker/dockerfile:1
FROM buildbase
WORKDIR /app

# Copy all files from the default context into /app/src in the build container
COPY . /app/src
RUN make bin

# Mount the files from the named "docs" context to build the documentation
RUN --mount=from=docs,target=/app/docs \
    make manpages
```

Use cases for named contexts

Using named contexts allows for greater flexibility and efficiency when building Docker images. Here are some scenarios where using named contexts can be useful:

Example: combine local and remote sources

You can define separate named contexts for different types of sources. For example, consider a project where the application source code is local, but the deployment scripts are stored in a Git repository:

```
$ docker build --build-context scripts=https://github.com/user/deployment-scripts.git .
```

In the Dockerfile, you can use these contexts independently:

```
# syntax=docker/dockerfile:1
FROM alpine:latest

# Copy application code from the main context
```

```
COPY . /opt/app

# Run deployment scripts using the remote "scripts" context
RUN --mount=from=scripts,target=/scripts /scripts/main.sh
```

Example: dynamic builds with custom dependencies

In some scenarios, you might need to dynamically inject configuration files or dependencies into the build from external sources. Named contexts make this straightforward by allowing you to mount different configurations without modifying the default build context.

```
$ docker build --build-context config=./configs/prod .
```

Example Dockerfile:

```
# syntax=docker/dockerfile:1
FROM nginx:alpine

# Use the "config" context for environment-specific configurations
COPY --from=config nginx.conf /etc/nginx/nginx.conf
```

Example: pin or override images

You can refer to named contexts in a Dockerfile the same way you can refer to an image. That means you can change an image reference in your Dockerfile by overriding it with a named context. For example, given the following Dockerfile:

```
FROM alpine:{{% param example_alpine_version %}}
```

If you want to force image reference to resolve to a different version, without changing the Dockerfile, you can pass a context with the same name to the build. For example:

```
docker buildx build --build-context alpine:{{% param example_alpine_version %}}=docker-
image://alpine:edge .
```

The `docker-image://` prefix marks the context as an image reference. The reference can be a local image or an image in your registry.

Named contexts with Bake

Bake is a tool built into `docker build` that lets you manage your build configuration with a configuration file. Bake fully supports named contexts.

To define named contexts in a Bake file:

```
target "app" {
  contexts = {
    docs = "./docs"
  }
}
```

This is equivalent to the following CLI invocation:

```
$ docker build --build-context docs=./docs .
```

Linking targets with named contexts

In addition to making complex builds more manageable, Bake also provides additional features on top of what you can do with `docker build` on the CLI. You can use named contexts to create build pipelines, where one target depends on and builds on top of another. For example, consider a Docker build setup where you have two Dockerfiles:

- `base.Dockerfile`: for building a base image
- `app.Dockerfile`: for building an application image

The `app.Dockerfile` uses the image produced by `base.Dockerfile` as it's base image:

```
FROM mybaseimage
```

Normally, you would have to build the base image first, and then either load it to Docker Engine's local image store or push it to a registry. With Bake, you can reference other targets directly, creating a dependency between the `app` target and the `base` target.

```
target "base" {
  dockerfile = "base.Dockerfile"
}

target "app" {
  dockerfile = "app.Dockerfile"
  contexts = {
    # the target: prefix indicates that 'base' is a Bake target
    mybaseimage = "target:base"
```

```
  }
}
```

With this configuration, references to `mybaseimage` in `app.Dockerfile` use the results from building the `base` target. Building the `app` target will also trigger a rebuild of `mybaseimage`, if necessary:

```
$ docker buildx bake app
```

Further reading

For more information about working with named contexts, see:

- `--build-context` CLI reference
- Using Bake with additional contexts

□□□ #2
□ □□□ □□□ 2026-10-31 19:05:52 UTC
□ □□□ □□□ 2026-10-31 19:05:54 UTC