**Base images**

All Dockerfiles start from a base image. A base is the image that your image extends. It refers to the contents of the FROM instruction in the Dockerfile.

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) debian

For most cases, you don't need to create your own base image. Docker Hub contains a vast library of Docker images that are suitable for use as a base image in your build. [Docker Official Images](https://docs.docker.com/trusted-content/official-images/) are specifically designed as a set of hardened, battle-tested images that support a wide variety of platforms, languages, and frameworks. There are also [Docker Verified Publisher](https://hub.docker.com/search?q=&image_filter=store" \t "_blank) images, created by trusted publishing partners, verified by Docker.

[Create a base image](https://docs.docker.com/build/building/base-images/" \l "create-a-base-image)

If you need to completely control the contents of your image, you can create your own base image from a Linux distribution of your choosing, or use the special FROM scratch base:

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) scratch

The scratch image is typically used to create minimal images containing only just what an application needs. See [Create a minimal base image using scratch](https://docs.docker.com/build/building/base-images/" \l "create-a-minimal-base-image-using-scratch).

To create a distribution base image, you can use a root filesystem, packaged as a tar file, and import it to Docker with docker import. The process for creating your own base image depends on the Linux distribution you want to package. See [Create a full image using tar](https://docs.docker.com/build/building/base-images/" \l "create-a-full-image-using-tar).

[Create a minimal base image using scratch](https://docs.docker.com/build/building/base-images/" \l "create-a-minimal-base-image-using-scratch)

The reserved, minimal scratch image serves as a starting point for building containers. Using the scratch image signals to the build process that you want the next command in the Dockerfile to be the first filesystem layer in your image.

While scratch appears in Docker's [repository on Docker Hub](https://hub.docker.com/_/scratch" \t "_blank), you can't pull it, run it, or tag any image with the name scratch. Instead, you can refer to it in your Dockerfile. For example, to create a minimal container using scratch:

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) scratch

[ADD](https://docs.docker.com/reference/dockerfile/" \l "add" \o "Learn more about the ADD instruction) hello /

[CMD](https://docs.docker.com/reference/dockerfile/" \l "cmd" \o "Learn more about the CMD instruction) ["/hello"]

Assuming an executable binary named hello exists at the root of the [build context](https://docs.docker.com/build/concepts/context/). You can build this Docker image using the following docker build command:

docker build --tag hello .

To run your new image, use the docker run command:

docker run --rm hello

This example image can only successfully execute as long as the hello binary doesn't have any runtime dependencies. Computer programs tend to depend on certain other programs or resources to exist in the runtime environment. For example:

* Programming language runtimes
* Dynamically linked C libraries
* CA certificates

When building a base image, or any image, this is an important aspect to consider. And this is why creating a base image using FROM scratch can be difficult, for anything other than small, simple programs. On the other hand, it's also important to include only the things you need in your image, to reduce the image size and attack surface.

[Create a full image using tar](https://docs.docker.com/build/building/base-images/" \l "create-a-full-image-using-tar)

In general, start with a working machine that is running the distribution you'd like to package as a base image, though that is not required for some tools like Debian's [Debootstrap](https://wiki.debian.org/Debootstrap" \t "_blank), which you can also use to build Ubuntu images.

For example, to create an Ubuntu base image:

$ sudo debootstrap focal focal > /dev/null

$ sudo tar -C focal -c . | docker import - focal

sha256:81ec9a55a92a5618161f68ae691d092bf14d700129093158297b3d01593f4ee3

$ docker run focal cat /etc/lsb-release

DISTRIB\_ID=Ubuntu

DISTRIB\_RELEASE=20.04

DISTRIB\_CODENAME=focal

DISTRIB\_DESCRIPTION="Ubuntu 20.04 LTS"

There are more example scripts for creating base images in [the Moby GitHub repository](https://github.com/moby/moby/blob/master/contrib" \t "_blank).

[More resources](https://docs.docker.com/build/building/base-images/" \l "more-resources)

For more information about building images and writing Dockerfiles, see:

* [Dockerfile reference](https://docs.docker.com/reference/dockerfile/)
* [Dockerfile best practices](https://docs.docker.com/build/building/best-practices/)
* [Docker Official Images](https://docs.docker.com/trusted-content/official-images/)

**Building best practices**

[Use multi-stage builds](https://docs.docker.com/build/building/best-practices/" \l "use-multi-stage-builds)

Multi-stage builds let you reduce the size of your final image, by creating a cleaner separation between the building of your image and the final output. Split your Dockerfile instructions into distinct stages to make sure that the resulting output only contains the files that's needed to run the application.

Using multiple stages can also let you build more efficiently by executing build steps in parallel.

See [Multi-stage builds](https://docs.docker.com/build/building/multi-stage/) for more information.

[Create reusable stages](https://docs.docker.com/build/building/best-practices/" \l "create-reusable-stages)

If you have multiple images with a lot in common, consider creating a reusable stage that includes the shared components, and basing your unique stages on that. Docker only needs to build the common stage once. This means that your derivative images use memory on the Docker host more efficiently and load more quickly.

It's also easier to maintain a common base stage ("Don't repeat yourself"), than it is to have multiple different stages doing similar things.

[Choose the right base image](https://docs.docker.com/build/building/best-practices/" \l "choose-the-right-base-image)

The first step towards achieving a secure image is to choose the right base image. When choosing an image, ensure it's built from a trusted source and keep it small.

* [Docker Official Images](https://hub.docker.com/search?image_filter=official" \t "_blank) are some of the most secure and dependable images on Docker Hub. Typically, Docker Official images have few or no packages containing CVEs, and are thoroughly reviewed by Docker and project maintainers.
* [Verified Publisher](https://hub.docker.com/search?image_filter=store" \t "_blank) images are high-quality images published and maintained by the organizations partnering with Docker, with Docker verifying the authenticity of the content in their repositories.
* [Docker-Sponsored Open Source](https://hub.docker.com/search?image_filter=open_source" \t "_blank) are published and maintained by open source projects sponsored by Docker through an [open source program](https://docs.docker.com/trusted-content/dsos-program/).

When you pick your base image, look out for the badges indicating that the image is part of these programs.

When building your own image from a Dockerfile, ensure you choose a minimal base image that matches your requirements. A smaller base image not only offers portability and fast downloads, but also shrinks the size of your image and minimizes the number of vulnerabilities introduced through the dependencies.

You should also consider using two types of base image: one for building and unit testing, and another (typically slimmer) image for production. In the later stages of development, your image may not require build tools such as compilers, build systems, and debugging tools. A small image with minimal dependencies can considerably lower the attack surface.

[Rebuild your images often](https://docs.docker.com/build/building/best-practices/" \l "rebuild-your-images-often)

Docker images are immutable. Building an image is taking a snapshot of that image at that moment. That includes any base images, libraries, or other software you use in your build. To keep your images up-to-date and secure, make sure to rebuild your image often, with updated dependencies.

To ensure that you're getting the latest versions of dependencies in your build, you can use the --no-cache option to avoid cache hits.

docker build --no-cache -t my-image:my-tag .

The following Dockerfile uses the 24.04 tag of the ubuntu image. Over time, that tag may resolve to a different underlying version of the ubuntu image, as the publisher rebuilds the image with new security patches and updated libraries. Using the --no-cache, you can avoid cache hits and ensure a fresh download of base images and dependencies.

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) ubuntu:24.04

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) apt-get -y update && apt-get install -y python

Also consider [pinning base image versions](https://docs.docker.com/build/building/best-practices/" \l "pin-base-image-versions).

[Exclude with .dockerignore](https://docs.docker.com/build/building/best-practices/" \l "exclude-with-dockerignore)

To exclude files not relevant to the build, without restructuring your source repository, use a .dockerignore file. This file supports exclusion patterns similar to .gitignore files.

For example, to exclude all files with the .md extension:

\*.md

For information on creating one, see [Dockerignore file](https://docs.docker.com/build/concepts/context/" \l "dockerignore-files).

[Create ephemeral containers](https://docs.docker.com/build/building/best-practices/" \l "create-ephemeral-containers)

The image defined by your Dockerfile should generate containers that are as ephemeral as possible. Ephemeral means that the container can be stopped and destroyed, then rebuilt and replaced with an absolute minimum set up and configuration.

Refer to [Processes](https://12factor.net/processes" \t "_blank) under *The Twelve-factor App* methodology to get a feel for the motivations of running containers in such a stateless fashion.

[Don't install unnecessary packages](https://docs.docker.com/build/building/best-practices/" \l "dont-install-unnecessary-packages)

Avoid installing extra or unnecessary packages just because they might be nice to have. For example, you don’t need to include a text editor in a database image.

When you avoid installing extra or unnecessary packages, your images have reduced complexity, reduced dependencies, reduced file sizes, and reduced build times.

[Decouple applications](https://docs.docker.com/build/building/best-practices/" \l "decouple-applications)

Each container should have only one concern. Decoupling applications into multiple containers makes it easier to scale horizontally and reuse containers. For instance, a web application stack might consist of three separate containers, each with its own unique image, to manage the web application, database, and an in-memory cache in a decoupled manner.

Limiting each container to one process is a good rule of thumb, but it's not a hard and fast rule. For example, not only can containers be [spawned with an init process](https://docs.docker.com/engine/containers/run/" \l "specify-an-init-process), some programs might spawn additional processes of their own accord. For instance, [Celery](https://docs.celeryproject.org/" \t "_blank) can spawn multiple worker processes, and [Apache](https://httpd.apache.org/" \t "_blank) can create one process per request.

Use your best judgment to keep containers as clean and modular as possible. If containers depend on each other, you can use [Docker container networks](https://docs.docker.com/engine/network/) to ensure that these containers can communicate.

[Sort multi-line arguments](https://docs.docker.com/build/building/best-practices/" \l "sort-multi-line-arguments)

Whenever possible, sort multi-line arguments alphanumerically to make maintenance easier. This helps to avoid duplication of packages and make the list much easier to update. This also makes PRs a lot easier to read and review. Adding a space before a backslash (\) helps as well.

Here’s an example from the [buildpack-deps image](https://github.com/docker-library/buildpack-deps" \t "_blank):

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) apt-get update && apt-get install -y \

bzr \

cvs \

git \

mercurial \

subversion \

&& rm -rf /var/lib/apt/lists/\*

[Leverage build cache](https://docs.docker.com/build/building/best-practices/" \l "leverage-build-cache)

When building an image, Docker steps through the instructions in your Dockerfile, executing each in the order specified. For each instruction, Docker checks whether it can reuse the instruction from the build cache.

Understanding how the build cache works, and how cache invalidation occurs, is critical for ensuring faster builds. For more information about the Docker build cache and how to optimize your builds, see [Docker build cache](https://docs.docker.com/build/cache/).

[Pin base image versions](https://docs.docker.com/build/building/best-practices/" \l "pin-base-image-versions)

Image tags are mutable, meaning a publisher can update a tag to point to a new image. This is useful because it lets publishers update tags to point to newer versions of an image. And as an image consumer, it means you automatically get the new version when you re-build your image.

For example, if you specify FROM alpine:3.19 in your Dockerfile, 3.19 resolves to the latest patch version for 3.19.

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) alpine:3.19

At one point in time, the 3.19 tag might point to version 3.19.1 of the image. If you rebuild the image 3 months later, the same tag might point to a different version, such as 3.19.4. This publishing workflow is best practice, and most publishers use this tagging strategy, but it isn't enforced.

The downside with this is that you're not guaranteed to get the same for every build. This could result in breaking changes, and it means you also don't have an audit trail of the exact image versions that you're using.

To fully secure your supply chain integrity, you can pin the image version to a specific digest. By pinning your images to a digest, you're guaranteed to always use the same image version, even if a publisher replaces the tag with a new image. For example, the following Dockerfile pins the Alpine image to the same tag as earlier, 3.19, but this time with a digest reference as well.

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) alpine:3.19@sha256:13b7e62e8df80264dbb747995705a986aa530415763a6c58f84a3ca8af9a5bcd

With this Dockerfile, even if the publisher updates the 3.19 tag, your builds would still use the pinned image version: 13b7e62e8df80264dbb747995705a986aa530415763a6c58f84a3ca8af9a5bcd.

While this helps you avoid unexpected changes, it's also more tedious to have to look up and include the image digest for base image versions manually each time you want to update it. And you're opting out of automated security fixes, which is likely something you want to get.

Docker Scout has a built-in **[No outdated base images](https://docs.docker.com/scout/policy/" \l "no-outdated-base-images)**[policy](https://docs.docker.com/scout/policy/" \l "no-outdated-base-images) that checks for whether the base image version you're using is in fact the latest version. This policy also checks if pinned digests in your Dockerfile correspond to the correct version. If a publisher updates an image that you've pinned, the policy evaluation returns a non-compliant status, indicating that you should update your image.

Docker Scout also supports an automated remediation workflow for keeping your base images up-to-date. When a new image digest is available, Docker Scout can automatically raise a pull request on your repository to update your Dockerfiles to use the latest version. This is better than using a tag that changes the version automatically, because you're in control and you have an audit trail of when and how the change occurred.

For more information about automatically updating your base images with Docker Scout, see [Remediation](https://docs.docker.com/scout/policy/remediation/" \l "automatic-base-image-updates)

[Build and test your images in CI](https://docs.docker.com/build/building/best-practices/" \l "build-and-test-your-images-in-ci)

When you check in a change to source control or create a pull request, use [GitHub Actions](https://docs.docker.com/build/ci/github-actions/) or another CI/CD pipeline to automatically build and tag a Docker image and test it.

[Dockerfile instructions](https://docs.docker.com/build/building/best-practices/" \l "dockerfile-instructions)

Follow these recommendations on how to properly use the [Dockerfile instructions](https://docs.docker.com/reference/dockerfile/) to create an efficient and maintainable Dockerfile.

[FROM](https://docs.docker.com/build/building/best-practices/" \l "from)

Whenever possible, use current official images as the basis for your images. Docker recommends the [Alpine image](https://hub.docker.com/_/alpine/" \t "_blank) as it is tightly controlled and small in size (currently under 6 MB), while still being a full Linux distribution.

For more information about the FROM instruction, see [Dockerfile reference for the FROM instruction](https://docs.docker.com/reference/dockerfile/" \l "from).

[LABEL](https://docs.docker.com/build/building/best-practices/" \l "label)

You can add labels to your image to help organize images by project, record licensing information, to aid in automation, or for other reasons. For each label, add a line beginning with LABEL with one or more key-value pairs. The following examples show the different acceptable formats. Explanatory comments are included inline.

Strings with spaces must be quoted or the spaces must be escaped. Inner quote characters ("), must also be escaped. For example:

# Set one or more individual labels

[LABEL](https://docs.docker.com/reference/dockerfile/" \l "label" \o "Learn more about the LABEL instruction) com.example.version="0.0.1-beta"

[LABEL](https://docs.docker.com/reference/dockerfile/" \l "label" \o "Learn more about the LABEL instruction) vendor1="ACME Incorporated"

[LABEL](https://docs.docker.com/reference/dockerfile/" \l "label" \o "Learn more about the LABEL instruction) vendor2=ZENITH\ Incorporated

[LABEL](https://docs.docker.com/reference/dockerfile/" \l "label" \o "Learn more about the LABEL instruction) com.example.release-date="2015-02-12"

[LABEL](https://docs.docker.com/reference/dockerfile/" \l "label" \o "Learn more about the LABEL instruction) com.example.version.is-production=""

An image can have more than one label. Prior to Docker 1.10, it was recommended to combine all labels into a single LABEL instruction, to prevent extra layers from being created. This is no longer necessary, but combining labels is still supported. For example:

# Set multiple labels on one line

[LABEL](https://docs.docker.com/reference/dockerfile/" \l "label" \o "Learn more about the LABEL instruction) com.example.version="0.0.1-beta" com.example.release-date="2015-02-12"

The above example can also be written as:

# Set multiple labels at once, using line-continuation characters to break long lines

[LABEL](https://docs.docker.com/reference/dockerfile/" \l "label" \o "Learn more about the LABEL instruction) vendor=ACME\ Incorporated \

com.example.is-beta= \

com.example.is-production="" \

com.example.version="0.0.1-beta" \

com.example.release-date="2015-02-12"

See [Understanding object labels](https://docs.docker.com/engine/manage-resources/labels/) for guidelines about acceptable label keys and values. For information about querying labels, refer to the items related to filtering in [Managing labels on objects](https://docs.docker.com/engine/manage-resources/labels/" \l "manage-labels-on-objects). See also [LABEL](https://docs.docker.com/reference/dockerfile/" \l "label) in the Dockerfile reference.

[RUN](https://docs.docker.com/build/building/best-practices/" \l "run)

Split long or complex RUN statements on multiple lines separated with backslashes to make your Dockerfile more readable, understandable, and maintainable.

For example, you can chain commands with the && operator, and use use escape characters to break long commands into multiple lines.

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) apt-get update && apt-get install -y \

package-bar \

package-baz \

package-foo

By default, backslash escapes a newline character, but you can change it with the [escape directive](https://docs.docker.com/reference/dockerfile/" \l "escape).

You can also use here documents to run multiple commands without chaining them with a pipeline operator:

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) <<EOF

apt-get update

apt-get install -y \

package-bar \

package-baz \

package-foo

EOF

For more information about RUN, see [Dockerfile reference for the RUN instruction](https://docs.docker.com/reference/dockerfile/" \l "run).

[apt-get](https://docs.docker.com/build/building/best-practices/" \l "apt-get)

One common use case for RUN instructions in Debian-based images is to install software using apt-get. Because apt-get installs packages, the RUN apt-get command has several counter-intuitive behaviors to look out for.

Always combine RUN apt-get update with apt-get install in the same RUN statement. For example:

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) apt-get update && apt-get install -y \

package-bar \

package-baz \

package-foo

Using apt-get update alone in a RUN statement causes caching issues and subsequent apt-get install instructions to fail. For example, this issue will occur in the following Dockerfile:

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) ubuntu:22.04

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) apt-get update

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) apt-get install -y curl

After building the image, all layers are in the Docker cache. Suppose you later modify apt-get install by adding an extra package as shown in the following Dockerfile:

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) ubuntu:22.04

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) apt-get update

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) apt-get install -y curl nginx

Docker sees the initial and modified instructions as identical and reuses the cache from previous steps. As a result the apt-get update isn't executed because the build uses the cached version. Because the apt-get update isn't run, your build can potentially get an outdated version of the curl and nginx packages.

Using RUN apt-get update && apt-get install -y ensures your Dockerfile installs the latest package versions with no further coding or manual intervention. This technique is known as cache busting. You can also achieve cache busting by specifying a package version. This is known as version pinning. For example:

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) apt-get update && apt-get install -y \

package-bar \

package-baz \

package-foo=1.3.\*

Version pinning forces the build to retrieve a particular version regardless of what’s in the cache. This technique can also reduce failures due to unanticipated changes in required packages.

Below is a well-formed RUN instruction that demonstrates all the apt-get recommendations.

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) apt-get update && apt-get install -y \

aufs-tools \

automake \

build-essential \

curl \

dpkg-sig \

libcap-dev \

libsqlite3-dev \

mercurial \

reprepro \

ruby1.9.1 \

ruby1.9.1-dev \

s3cmd=1.1.\* \

&& rm -rf /var/lib/apt/lists/\*

The s3cmd argument specifies a version 1.1.\*. If the image previously used an older version, specifying the new one causes a cache bust of apt-get update and ensures the installation of the new version. Listing packages on each line can also prevent mistakes in package duplication.

In addition, when you clean up the apt cache by removing /var/lib/apt/lists it reduces the image size, since the apt cache isn't stored in a layer. Since the RUN statement starts with apt-get update, the package cache is always refreshed prior to apt-get install.

Official Debian and Ubuntu images [automatically run apt-get clean](https://github.com/moby/moby/blob/03e2923e42446dbb830c654d0eec323a0b4ef02a/contrib/mkimage/debootstrap" \l "L82-L105" \t "_blank), so explicit invocation is not required.

[Using pipes](https://docs.docker.com/build/building/best-practices/" \l "using-pipes)

Some RUN commands depend on the ability to pipe the output of one command into another, using the pipe character (|), as in the following example:

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) wget -O - https://some.site | wc -l > /number

Docker executes these commands using the /bin/sh -c interpreter, which only evaluates the exit code of the last operation in the pipe to determine success. In the example above, this build step succeeds and produces a new image so long as the wc -l command succeeds, even if the wget command fails.

If you want the command to fail due to an error at any stage in the pipe, prepend set -o pipefail && to ensure that an unexpected error prevents the build from inadvertently succeeding. For example:

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) set -o pipefail && wget -O - https://some.site | wc -l > /number

**Note**

Not all shells support the -o pipefail option.

In cases such as the dash shell on Debian-based images, consider using the *exec* form of RUN to explicitly choose a shell that does support the pipefail option. For example:

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) ["/bin/bash", "-c", "set -o pipefail && wget -O - https://some.site | wc -l > /number"]

[CMD](https://docs.docker.com/build/building/best-practices/" \l "cmd)

The CMD instruction should be used to run the software contained in your image, along with any arguments. CMD should almost always be used in the form of CMD ["executable", "param1", "param2"]. Thus, if the image is for a service, such as Apache and Rails, you would run something like CMD ["apache2","-DFOREGROUND"]. Indeed, this form of the instruction is recommended for any service-based image.

In most other cases, CMD should be given an interactive shell, such as bash, python and perl. For example, CMD ["perl", "-de0"], CMD ["python"], or CMD ["php", "-a"]. Using this form means that when you execute something like docker run -it python, you’ll get dropped into a usable shell, ready to go. CMD should rarely be used in the manner of CMD ["param", "param"] in conjunction with [ENTRYPOINT](https://docs.docker.com/reference/dockerfile/" \l "entrypoint), unless you and your expected users are already quite familiar with how ENTRYPOINT works.

For more information about CMD, see [Dockerfile reference for the CMD instruction](https://docs.docker.com/reference/dockerfile/" \l "cmd).

[EXPOSE](https://docs.docker.com/build/building/best-practices/" \l "expose)

The EXPOSE instruction indicates the ports on which a container listens for connections. Consequently, you should use the common, traditional port for your application. For example, an image containing the Apache web server would use EXPOSE 80, while an image containing MongoDB would use EXPOSE 27017 and so on.

For external access, your users can execute docker run with a flag indicating how to map the specified port to the port of their choice. For container linking, Docker provides environment variables for the path from the recipient container back to the source (for example, MYSQL\_PORT\_3306\_TCP).

For more information about EXPOSE, see [Dockerfile reference for the EXPOSE instruction](https://docs.docker.com/reference/dockerfile/" \l "expose).

[ENV](https://docs.docker.com/build/building/best-practices/" \l "env)

To make new software easier to run, you can use ENV to update the PATH environment variable for the software your container installs. For example, ENV PATH=/usr/local/nginx/bin:$PATH ensures that CMD ["nginx"] just works.

The ENV instruction is also useful for providing the required environment variables specific to services you want to containerize, such as Postgres’s PGDATA.

Lastly, ENV can also be used to set commonly used version numbers so that version bumps are easier to maintain, as seen in the following example:

[ENV](https://docs.docker.com/reference/dockerfile/" \l "env" \o "Learn more about the ENV instruction) PG\_MAJOR=9.3

[ENV](https://docs.docker.com/reference/dockerfile/" \l "env" \o "Learn more about the ENV instruction) PG\_VERSION=9.3.4

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) curl -SL https://example.com/postgres-$PG\_VERSION.tar.xz | tar -xJC /usr/src/postgres && …

[ENV](https://docs.docker.com/reference/dockerfile/" \l "env" \o "Learn more about the ENV instruction) PATH=/usr/local/postgres-$PG\_MAJOR/bin:$PATH

Similar to having constant variables in a program, as opposed to hard-coding values, this approach lets you change a single ENV instruction to automatically bump the version of the software in your container.

Each ENV line creates a new intermediate layer, just like RUN commands. This means that even if you unset the environment variable in a future layer, it still persists in this layer and its value can be dumped. You can test this by creating a Dockerfile like the following, and then building it.

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) alpine

[ENV](https://docs.docker.com/reference/dockerfile/" \l "env" \o "Learn more about the ENV instruction) ADMIN\_USER="mark"

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) echo $ADMIN\_USER > ./mark

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) unset ADMIN\_USER

docker run --rm test sh -c 'echo $ADMIN\_USER'

mark

To prevent this, and really unset the environment variable, use a RUN command with shell commands, to set, use, and unset the variable all in a single layer. You can separate your commands with ; or &&. If you use the second method, and one of the commands fails, the docker build also fails. This is usually a good idea. Using \ as a line continuation character for Linux Dockerfiles improves readability. You could also put all of the commands into a shell script and have the RUN command just run that shell script.

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) alpine

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) export ADMIN\_USER="mark" \

&& echo $ADMIN\_USER > ./mark \

&& unset ADMIN\_USER

[CMD](https://docs.docker.com/reference/dockerfile/" \l "cmd" \o "Learn more about the CMD instruction) sh

docker run --rm test sh -c 'echo $ADMIN\_USER'

For more information about ENV, see [Dockerfile reference for the ENV instruction](https://docs.docker.com/reference/dockerfile/" \l "env).

[ADD or COPY](https://docs.docker.com/build/building/best-practices/" \l "add-or-copy)

ADD and COPY are functionally similar. COPY supports basic copying of files into the container, from the [build context](https://docs.docker.com/build/concepts/context/) or from a stage in a [multi-stage build](https://docs.docker.com/build/building/multi-stage/). ADD supports features for fetching files from remote HTTPS and Git URLs, and extracting tar files automatically when adding files from the build context.

You'll mostly want to use COPY for copying files from one stage to another in a multi-stage build. If you need to add files from the build context to the container temporarily to execute a RUN instruction, you can often substitute the COPY instruction with a bind mount instead. For example, to temporarily add a requirements.txt file for a RUN pip install instruction:

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) --mount=type=bind,source=requirements.txt,target=/tmp/requirements.txt \

pip install --requirement /tmp/requirements.txt

Bind mounts are more efficient than COPY for including files from the build context in the container. Note that bind-mounted files are only added temporarily for a single RUN instruction, and don't persist in the final image. If you need to include files from the build context in the final image, use COPY.

The ADD instruction is best for when you need to download a remote artifact as part of your build. ADD is better than manually adding files using something like wget and tar, because it ensures a more precise build cache. ADD also has built-in support for checksum validation of the remote resources, and a protocol for parsing branches, tags, and subdirectories from [Git URLs](https://docs.docker.com/reference/cli/docker/buildx/build/" \l "git-repositories).

The following example uses ADD to download a .NET installer. Combined with multi-stage builds, only the .NET runtime remains in the final stage, no intermediate files.

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) scratch AS src

[ARG](https://docs.docker.com/reference/dockerfile/" \l "arg" \o "Learn more about the ARG instruction) DOTNET\_VERSION=8.0.0-preview.6.23329.7

[ADD](https://docs.docker.com/reference/dockerfile/" \l "add" \o "Learn more about the ADD instruction) --checksum=sha256:270d731bd08040c6a3228115de1f74b91cf441c584139ff8f8f6503447cebdbb \

https://dotnetcli.azureedge.net/dotnet/Runtime/$DOTNET\_VERSION/dotnet-runtime-$DOTNET\_VERSION-linux-arm64.tar.gz /dotnet.tar.gz

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) mcr.microsoft.com/dotnet/runtime-deps:8.0.0-preview.6-bookworm-slim-arm64v8 AS installer

# Retrieve .NET Runtime

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) --mount=from=src,target=/src <<EOF

mkdir -p /dotnet

tar -oxzf /src/dotnet.tar.gz -C /dotnet

EOF

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) mcr.microsoft.com/dotnet/runtime-deps:8.0.0-preview.6-bookworm-slim-arm64v8

[COPY](https://docs.docker.com/reference/dockerfile/" \l "copy" \o "Learn more about the COPY instruction) --from=installer /dotnet /usr/share/dotnet

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) ln -s /usr/share/dotnet/dotnet /usr/bin/dotnet

For more information about ADD or COPY, see the following:

* [Dockerfile reference for the ADD instruction](https://docs.docker.com/reference/dockerfile/" \l "add)
* [Dockerfile reference for the COPY instruction](https://docs.docker.com/reference/dockerfile/" \l "copy)

[ENTRYPOINT](https://docs.docker.com/build/building/best-practices/" \l "entrypoint)

The best use for ENTRYPOINT is to set the image's main command, allowing that image to be run as though it was that command, and then use CMD as the default flags.

The following is an example of an image for the command line tool s3cmd:

[ENTRYPOINT](https://docs.docker.com/reference/dockerfile/" \l "entrypoint" \o "Learn more about the ENTRYPOINT instruction) ["s3cmd"]

[CMD](https://docs.docker.com/reference/dockerfile/" \l "cmd" \o "Learn more about the CMD instruction) ["--help"]

You can use the following command to run the image and show the command's help:

docker run s3cmd

Or, you can use the right parameters to execute a command, like in the following example:

docker run s3cmd ls s3://mybucket

This is useful because the image name can double as a reference to the binary as shown in the command above.

The ENTRYPOINT instruction can also be used in combination with a helper script, allowing it to function in a similar way to the command above, even when starting the tool may require more than one step.

For example, the [Postgres Official Image](https://hub.docker.com/_/postgres/" \t "_blank) uses the following script as its ENTRYPOINT:

#!/bin/bash

set -e

if [ "$1" = 'postgres' ]; then

chown -R postgres "$PGDATA"

if [ -z "$(ls -A "$PGDATA")" ]; then

gosu postgres initdb

fi

exec gosu postgres "$@"

fi

exec "$@"

This script uses [the exec Bash command](https://wiki.bash-hackers.org/commands/builtin/exec" \t "_blank) so that the final running application becomes the container's PID 1. This allows the application to receive any Unix signals sent to the container. For more information, see the [ENTRYPOINT reference](https://docs.docker.com/reference/dockerfile/" \l "entrypoint).

In the following example, a helper script is copied into the container and run via ENTRYPOINT on container start:

[COPY](https://docs.docker.com/reference/dockerfile/" \l "copy" \o "Learn more about the COPY instruction) ./docker-entrypoint.sh /

[ENTRYPOINT](https://docs.docker.com/reference/dockerfile/" \l "entrypoint" \o "Learn more about the ENTRYPOINT instruction) ["/docker-entrypoint.sh"]

[CMD](https://docs.docker.com/reference/dockerfile/" \l "cmd" \o "Learn more about the CMD instruction) ["postgres"]

This script lets you interact with Postgres in several ways.

It can simply start Postgres:

docker run postgres

Or, you can use it to run Postgres and pass parameters to the server:

docker run postgres postgres --help

Lastly, you can use it to start a totally different tool, such as Bash:

docker run --rm -it postgres bash

For more information about ENTRYPOINT, see [Dockerfile reference for the ENTRYPOINT instruction](https://docs.docker.com/reference/dockerfile/" \l "entrypoint).

[VOLUME](https://docs.docker.com/build/building/best-practices/" \l "volume)

You should use the VOLUME instruction to expose any database storage area, configuration storage, or files and folders created by your Docker container. You are strongly encouraged to use VOLUME for any combination of mutable or user-serviceable parts of your image.

For more information about VOLUME, see [Dockerfile reference for the VOLUME instruction](https://docs.docker.com/reference/dockerfile/" \l "volume).

[USER](https://docs.docker.com/build/building/best-practices/" \l "user)

If a service can run without privileges, use USER to change to a non-root user. Start by creating the user and group in the Dockerfile with something like the following example:

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) groupadd -r postgres && useradd --no-log-init -r -g postgres postgres

**Note**

Consider an explicit UID/GID.

Users and groups in an image are assigned a non-deterministic UID/GID in that the "next" UID/GID is assigned regardless of image rebuilds. So, if it’s critical, you should assign an explicit UID/GID.

**Note**

Due to an [unresolved bug](https://github.com/golang/go/issues/13548" \t "_blank) in the Go archive/tar package's handling of sparse files, attempting to create a user with a significantly large UID inside a Docker container can lead to disk exhaustion because /var/log/faillog in the container layer is filled with NULL (\0) characters. A workaround is to pass the --no-log-init flag to useradd. The Debian/Ubuntu adduser wrapper does not support this flag.

Avoid installing or using sudo as it has unpredictable TTY and signal-forwarding behavior that can cause problems. If you absolutely need functionality similar to sudo, such as initializing the daemon as root but running it as non-root, consider using [“gosu”](https://github.com/tianon/gosu" \t "_blank).

Lastly, to reduce layers and complexity, avoid switching USER back and forth frequently.

For more information about USER, see [Dockerfile reference for the USER instruction](https://docs.docker.com/reference/dockerfile/" \l "user).

[WORKDIR](https://docs.docker.com/build/building/best-practices/" \l "workdir)

For clarity and reliability, you should always use absolute paths for your WORKDIR. Also, you should use WORKDIR instead of proliferating instructions like RUN cd … && do-something, which are hard to read, troubleshoot, and maintain.

For more information about WORKDIR, see [Dockerfile reference for the WORKDIR instruction](https://docs.docker.com/reference/dockerfile/" \l "workdir).

[ONBUILD](https://docs.docker.com/build/building/best-practices/" \l "onbuild)

An ONBUILD command executes after the current Dockerfile build completes. ONBUILD executes in any child image derived FROM the current image. Think of the ONBUILD command as an instruction that the parent Dockerfile gives to the child Dockerfile.

A Docker build executes ONBUILD commands before any command in a child Dockerfile.

ONBUILD is useful for images that are going to be built FROM a given image. For example, you would use ONBUILD for a language stack image that builds arbitrary user software written in that language within the Dockerfile, as you can see in [Ruby’s ONBUILD variants](https://github.com/docker-library/ruby/blob/c43fef8a60cea31eb9e7d960a076d633cb62ba8d/2.4/jessie/onbuild/Dockerfile" \t "_blank).

Images built with ONBUILD should get a separate tag. For example, ruby:1.9-onbuild or ruby:2.0-onbuild.

Be careful when putting ADD or COPY in ONBUILD. The onbuild image fails catastrophically if the new build's context is missing the resource being added. Adding a separate tag, as recommended above, helps mitigate this by allowing the Dockerfile author to make a choice.

For more information about ONBUILD, see [Dockerfile reference for the ONBUILD instruction](https://docs.docker.com/reference/dockerfile/" \l "onbuild).

**Multi-stage builds**

Multi-stage builds are useful to anyone who has struggled to optimize Dockerfiles while keeping them easy to read and maintain.

[Use multi-stage builds](https://docs.docker.com/build/building/multi-stage/" \l "use-multi-stage-builds)

With multi-stage builds, you use multiple FROM statements in your Dockerfile. Each FROM instruction can use a different base, and each of them begins a new stage of the build. You can selectively copy artifacts from one stage to another, leaving behind everything you don't want in the final image.

The following Dockerfile has two separate stages: one for building a binary, and another where the binary gets copied from the first stage into the next stage.

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) golang:1.21

[WORKDIR](https://docs.docker.com/reference/dockerfile/" \l "workdir" \o "Learn more about the WORKDIR instruction) /src

[COPY](https://docs.docker.com/reference/dockerfile/" \l "copy" \o "Learn more about the COPY instruction) <<EOF ./main.go

package main

import "fmt"

func main() {

fmt.Println("hello, world")

}

EOF

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) go build -o /bin/hello ./main.go

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) scratch

[COPY](https://docs.docker.com/reference/dockerfile/" \l "copy" \o "Learn more about the COPY instruction) --from=0 /bin/hello /bin/hello

[CMD](https://docs.docker.com/reference/dockerfile/" \l "cmd" \o "Learn more about the CMD instruction) ["/bin/hello"]

You only need the single Dockerfile. No need for a separate build script. Just run docker build.

docker build -t hello .

The end result is a tiny production image with nothing but the binary inside. None of the build tools required to build the application are included in the resulting image.

How does it work? The second FROM instruction starts a new build stage with the scratch image as its base. The COPY --from=0 line copies just the built artifact from the previous stage into this new stage. The Go SDK and any intermediate artifacts are left behind, and not saved in the final image.

[Name your build stages](https://docs.docker.com/build/building/multi-stage/" \l "name-your-build-stages)

By default, the stages aren't named, and you refer to them by their integer number, starting with 0 for the first FROM instruction. However, you can name your stages, by adding an AS <NAME> to the FROM instruction. This example improves the previous one by naming the stages and using the name in the COPY instruction. This means that even if the instructions in your Dockerfile are re-ordered later, the COPY doesn't break.

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) golang:1.21 AS build

[WORKDIR](https://docs.docker.com/reference/dockerfile/" \l "workdir" \o "Learn more about the WORKDIR instruction) /src

[COPY](https://docs.docker.com/reference/dockerfile/" \l "copy" \o "Learn more about the COPY instruction) <<EOF /src/main.go

package main

import "fmt"

func main() {

fmt.Println("hello, world")

}

EOF

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) go build -o /bin/hello ./main.go

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) scratch

[COPY](https://docs.docker.com/reference/dockerfile/" \l "copy" \o "Learn more about the COPY instruction) --from=build /bin/hello /bin/hello

[CMD](https://docs.docker.com/reference/dockerfile/" \l "cmd" \o "Learn more about the CMD instruction) ["/bin/hello"]

[Stop at a specific build stage](https://docs.docker.com/build/building/multi-stage/" \l "stop-at-a-specific-build-stage)

When you build your image, you don't necessarily need to build the entire Dockerfile including every stage. You can specify a target build stage. The following command assumes you are using the previous Dockerfile but stops at the stage named build:

docker build --target build -t hello .

A few scenarios where this might be useful are:

* Debugging a specific build stage
* Using a debug stage with all debugging symbols or tools enabled, and a lean production stage
* Using a testing stage in which your app gets populated with test data, but building for production using a different stage which uses real data

[Use an external image as a stage](https://docs.docker.com/build/building/multi-stage/" \l "use-an-external-image-as-a-stage)

When using multi-stage builds, you aren't limited to copying from stages you created earlier in your Dockerfile. You can use the COPY --from instruction to copy from a separate image, either using the local image name, a tag available locally or on a Docker registry, or a tag ID. The Docker client pulls the image if necessary and copies the artifact from there. The syntax is:

[COPY](https://docs.docker.com/reference/dockerfile/" \l "copy" \o "Learn more about the COPY instruction) --from=nginx:latest /etc/nginx/nginx.conf /nginx.conf

[Use a previous stage as a new stage](https://docs.docker.com/build/building/multi-stage/" \l "use-a-previous-stage-as-a-new-stage)

You can pick up where a previous stage left off by referring to it when using the FROM directive. For example:

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) alpine:latest AS builder

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) apk --no-cache add build-base

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) builder AS build1

[COPY](https://docs.docker.com/reference/dockerfile/" \l "copy" \o "Learn more about the COPY instruction) source1.cpp source.cpp

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) g++ -o /binary source.cpp

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) builder AS build2

[COPY](https://docs.docker.com/reference/dockerfile/" \l "copy" \o "Learn more about the COPY instruction) source2.cpp source.cpp

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) g++ -o /binary source.cpp

[Differences between legacy builder and BuildKit](https://docs.docker.com/build/building/multi-stage/" \l "differences-between-legacy-builder-and-buildkit)

The legacy Docker Engine builder processes all stages of a Dockerfile leading up to the selected --target. It will build a stage even if the selected target doesn't depend on that stage.

[BuildKit](https://docs.docker.com/build/buildkit/) only builds the stages that the target stage depends on.

For example, given the following Dockerfile:

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) ubuntu AS base

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) echo "base"

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) base AS stage1

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) echo "stage1"

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) base AS stage2

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) echo "stage2"

With [BuildKit enabled](https://docs.docker.com/build/buildkit/" \l "getting-started), building the stage2 target in this Dockerfile means only base and stage2 are processed. There is no dependency on stage1, so it's skipped.

DOCKER\_BUILDKIT=1 docker build --no-cache -f Dockerfile --target stage2 .

[+] Building 0.4s (7/7) FINISHED

=> [internal] load build definition from Dockerfile 0.0s

=> => transferring dockerfile: 36B 0.0s

=> [internal] load .dockerignore 0.0s

=> => transferring context: 2B 0.0s

=> [internal] load metadata for docker.io/library/ubuntu:latest 0.0s

=> CACHED [base 1/2] FROM docker.io/library/ubuntu 0.0s

=> [base 2/2] RUN echo "base" 0.1s

=> [stage2 1/1] RUN echo "stage2" 0.2s

=> exporting to image 0.0s

=> => exporting layers 0.0s

=> => writing image sha256:f55003b607cef37614f607f0728e6fd4d113a4bf7ef12210da338c716f2cfd15 0.0s

On the other hand, building the same target without BuildKit results in all stages being processed:

DOCKER\_BUILDKIT=0 docker build --no-cache -f Dockerfile --target stage2 .

Sending build context to Docker daemon 219.1kB

Step 1/6 : FROM ubuntu AS base

---> a7870fd478f4

Step 2/6 : RUN echo "base"

---> Running in e850d0e42eca

base

Removing intermediate container e850d0e42eca

---> d9f69f23cac8

Step 3/6 : FROM base AS stage1

---> d9f69f23cac8

Step 4/6 : RUN echo "stage1"

---> Running in 758ba6c1a9a3

stage1

Removing intermediate container 758ba6c1a9a3

---> 396baa55b8c3

Step 5/6 : FROM base AS stage2

---> d9f69f23cac8

Step 6/6 : RUN echo "stage2"

---> Running in bbc025b93175

stage2

Removing intermediate container bbc025b93175

---> 09fc3770a9c4

Successfully built 09fc3770a9c4

The legacy builder processes stage1, even if stage2 doesn't depend on it.

**Build context**

The docker build and docker buildx build commands build Docker images from a [Dockerfile](https://docs.docker.com/reference/dockerfile/) and a context.

[What is a build context?](https://docs.docker.com/build/concepts/context/" \l "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](https://docs.docker.com/build/concepts/context/" \l "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](https://docs.docker.com/build/concepts/context/" \l "local-context)
* [Git repositories](https://docs.docker.com/build/concepts/context/" \l "git-repositories)
* [Remote tarballs](https://docs.docker.com/build/concepts/context/" \l "remote-tarballs)

[Text file contexts](https://docs.docker.com/build/concepts/context/" \l "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](https://docs.docker.com/build/concepts/context/" \l "empty-context).

[Local context](https://docs.docker.com/build/concepts/context/" \l "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](https://docs.docker.com/build/concepts/context/" \l "local-tarballs).

[Local directories](https://docs.docker.com/build/concepts/context/" \l "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](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) node:latest

[WORKDIR](https://docs.docker.com/reference/dockerfile/" \l "workdir" \o "Learn more about the WORKDIR instruction) /src

[COPY](https://docs.docker.com/reference/dockerfile/" \l "copy" \o "Learn more about the COPY instruction) package.json package-lock.json .

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) npm ci

[COPY](https://docs.docker.com/reference/dockerfile/" \l "copy" \o "Learn more about the COPY instruction) index.ts src .

docker build .

[Local context with Dockerfile from stdin](https://docs.docker.com/build/concepts/context/" \l "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](https://docs.docker.com/build/concepts/context/" \l "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](https://docs.docker.com/build/concepts/context/" \l "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](https://docs.docker.com/build/concepts/context/" \l "git-repositories).
* For tarballs, the builder downloads and extracts the contents of the tarball. See [Tarballs](https://docs.docker.com/build/concepts/context/" \l "remote-tarballs).

If the remote tarball is a text file, the builder receives no [filesystem context](https://docs.docker.com/build/concepts/context/" \l "filesystem-contexts), and instead assumes that the remote file is a Dockerfile. See [Empty build context](https://docs.docker.com/build/concepts/context/" \l "empty-context).

[Git repositories](https://docs.docker.com/build/concepts/context/" \l "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](https://docs.docker.com/build/concepts/context/" \l "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

[Keep .git directory](https://docs.docker.com/build/concepts/context/" \l "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](https://docs.docker.com/reference/dockerfile/" \l "buildkit-built-in-build-args). This can be useful to if you want to retrieve Git information during your build:

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) alpine

[WORKDIR](https://docs.docker.com/reference/dockerfile/" \l "workdir" \o "Learn more about the WORKDIR instruction) /src

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) --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](https://docs.docker.com/build/concepts/context/" \l "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](https://docs.docker.com/reference/cli/docker/buildx/build/" \l "ssh).

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](https://docs.docker.com/reference/cli/docker/buildx/build/" \l "secret).

GIT\_AUTH\_TOKEN=<token> docker buildx build \

--secret id=GIT\_AUTH\_TOKEN \

https://github.com/user/private.git

**Note**

Don't use --build-arg for secrets.

[Remote context with Dockerfile from stdin](https://docs.docker.com/build/concepts/context/" \l "remote-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- *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](https://github.com/docker-library/hello-world" \t "_blank) 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](https://docs.docker.com/build/concepts/context/" \l "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](https://docs.docker.com/build/concepts/context/" \l "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](https://docs.docker.com/build/concepts/context/" \l "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.

Unix pipe PowerShell Heredocs Remote file

docker build - < Dockerfile

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](https://docs.docker.com/build/concepts/context/" \l "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](https://docs.docker.com/build/concepts/context/" \l "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](https://docs.docker.com/build/concepts/context/" \l "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](https://github.com/moby/patternmatcher/tree/main/ignorefile" \t "_blank) on GitHub, which contains the source code.

[Matching](https://docs.docker.com/build/concepts/context/" \l "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](https://golang.org/pkg/path/filepath" \l "Match" \t "_blank) rules. A preprocessing step uses Go's [filepath.Clean function](https://golang.org/pkg/path/filepath/" \l "Clean" \t "_blank) to trim whitespace and remove . and ... Lines that are blank after preprocessing are ignored.

**Note**

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](https://docs.docker.com/build/concepts/context/" \l "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.

**Dockerfile overview**

[Dockerfile](https://docs.docker.com/build/concepts/dockerfile/" \l "dockerfile)

It all starts with a Dockerfile.

Docker builds images by reading the instructions from a Dockerfile. A Dockerfile is a text file containing instructions for building your source code. The Dockerfile instruction syntax is defined by the specification reference in the [Dockerfile reference](https://docs.docker.com/reference/dockerfile/).

Here are the most common types of instructions:

| **Instruction** | **Description** |
| --- | --- |
| [FROM <image>](https://docs.docker.com/reference/dockerfile/" \l "from) | Defines a base for your image. |
| [RUN <command>](https://docs.docker.com/reference/dockerfile/" \l "run) | Executes any commands in a new layer on top of the current image and commits the result. RUN also has a shell form for running commands. |
| [WORKDIR <directory>](https://docs.docker.com/reference/dockerfile/" \l "workdir) | Sets the working directory for any RUN, CMD, ENTRYPOINT, COPY, and ADD instructions that follow it in the Dockerfile. |
| [COPY <src> <dest>](https://docs.docker.com/reference/dockerfile/" \l "copy) | Copies new files or directories from <src> and adds them to the filesystem of the container at the path <dest>. |
| [CMD <command>](https://docs.docker.com/reference/dockerfile/" \l "cmd) | Lets you define the default program that is run once you start the container based on this image. Each Dockerfile only has one CMD, and only the last CMD instance is respected when multiple exist. |

Dockerfiles are crucial inputs for image builds and can facilitate automated, multi-layer image builds based on your unique configurations. Dockerfiles can start simple and grow with your needs to support more complex scenarios.

[Filename](https://docs.docker.com/build/concepts/dockerfile/" \l "filename)

The default filename to use for a Dockerfile is Dockerfile, without a file extension. Using the default name allows you to run the docker build command without having to specify additional command flags.

Some projects may need distinct Dockerfiles for specific purposes. A common convention is to name these <something>.Dockerfile. You can specify the Dockerfile filename using the --file flag for the docker build command. Refer to the [docker build CLI reference](https://docs.docker.com/reference/cli/docker/buildx/build/" \l "file) to learn about the --file flag.

**Note**

We recommend using the default (Dockerfile) for your project's primary Dockerfile.

[Docker images](https://docs.docker.com/build/concepts/dockerfile/" \l "docker-images)

Docker images consist of layers. Each layer is the result of a build instruction in the Dockerfile. Layers are stacked sequentially, and each one is a delta representing the changes applied to the previous layer.

[Example](https://docs.docker.com/build/concepts/dockerfile/" \l "example)

Here's what a typical workflow for building applications with Docker looks like.

The following example code shows a small "Hello World" application written in Python, using the Flask framework.

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route("/")

def hello():

return "Hello World!"

In order to ship and deploy this application without Docker Build, you would need to make sure that:

* The required runtime dependencies are installed on the server
* The Python code gets uploaded to the server's filesystem
* The server starts your application, using the necessary parameters

The following Dockerfile creates a container image, which has all the dependencies installed and that automatically starts your application.

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) ubuntu:22.04

# install app dependencies

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) apt-get update && apt-get install -y python3 python3-pip

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) pip install flask==3.0.\*

# install app

[COPY](https://docs.docker.com/reference/dockerfile/" \l "copy" \o "Learn more about the COPY instruction) hello.py /

# final configuration

[ENV](https://docs.docker.com/reference/dockerfile/" \l "env" \o "Learn more about the ENV instruction) FLASK\_APP=hello

[EXPOSE](https://docs.docker.com/reference/dockerfile/" \l "expose" \o "Learn more about the EXPOSE instruction) 8000

[CMD](https://docs.docker.com/reference/dockerfile/" \l "cmd" \o "Learn more about the CMD instruction) ["flask", "run", "--host", "0.0.0.0", "--port", "8000"]

Here's a breakdown of what this Dockerfile does:

* [Dockerfile syntax](https://docs.docker.com/build/concepts/dockerfile/" \l "dockerfile-syntax)
* [Base image](https://docs.docker.com/build/concepts/dockerfile/" \l "base-image)
* [Environment setup](https://docs.docker.com/build/concepts/dockerfile/" \l "environment-setup)
* [Comments](https://docs.docker.com/build/concepts/dockerfile/" \l "comments)
* [Installing dependencies](https://docs.docker.com/build/concepts/dockerfile/" \l "installing-dependencies)
* [Copying files](https://docs.docker.com/build/concepts/dockerfile/" \l "copying-files)
* [Setting environment variables](https://docs.docker.com/build/concepts/dockerfile/" \l "setting-environment-variables)
* [Exposed ports](https://docs.docker.com/build/concepts/dockerfile/" \l "exposed-ports)
* [Starting the application](https://docs.docker.com/build/concepts/dockerfile/" \l "starting-the-application)

[Dockerfile syntax](https://docs.docker.com/build/concepts/dockerfile/" \l "dockerfile-syntax)

The first line to add to a Dockerfile is a [# syntax parser directive](https://docs.docker.com/reference/dockerfile/" \l "syntax). While optional, this directive instructs the Docker builder what syntax to use when parsing the Dockerfile, and allows older Docker versions with [BuildKit enabled](https://docs.docker.com/build/buildkit/" \l "getting-started) to use a specific [Dockerfile frontend](https://docs.docker.com/build/buildkit/frontend/) before starting the build. [Parser directives](https://docs.docker.com/reference/dockerfile/" \l "parser-directives) must appear before any other comment, whitespace, or Dockerfile instruction in your Dockerfile, and should be the first line in Dockerfiles.

# syntax=docker/dockerfile:1

**Tip**

We recommend using docker/dockerfile:1, which always points to the latest release of the version 1 syntax. BuildKit automatically checks for updates of the syntax before building, making sure you are using the most current version.

[Base image](https://docs.docker.com/build/concepts/dockerfile/" \l "base-image)

The line following the syntax directive defines what base image to use:

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) ubuntu:22.04

The [FROM instruction](https://docs.docker.com/reference/dockerfile/" \l "from) sets your base image to the 22.04 release of Ubuntu. All instructions that follow are executed in this base image: an Ubuntu environment. The notation ubuntu:22.04, follows the name:tag standard for naming Docker images. When you build images, you use this notation to name your images. There are many public images you can leverage in your projects, by importing them into your build steps using the Dockerfile FROM instruction.

[Docker Hub](https://hub.docker.com/search?image_filter=official&q=&type=image" \t "_blank) contains a large set of official images that you can use for this purpose.

[Environment setup](https://docs.docker.com/build/concepts/dockerfile/" \l "environment-setup)

The following line executes a build command inside the base image.

# install app dependencies

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) apt-get update && apt-get install -y python3 python3-pip

This [RUN instruction](https://docs.docker.com/reference/dockerfile/" \l "run) executes a shell in Ubuntu that updates the APT package index and installs Python tools in the container.

[Comments](https://docs.docker.com/build/concepts/dockerfile/" \l "comments)

Note the # install app dependencies line. This is a comment. Comments in Dockerfiles begin with the # symbol. As your Dockerfile evolves, comments can be instrumental to document how your Dockerfile works for any future readers and editors of the file, including your future self!

**Note**

You might've noticed that comments are denoted using the same symbol as the [syntax directive](https://docs.docker.com/build/concepts/dockerfile/" \l "dockerfile-syntax) on the first line of the file. The symbol is only interpreted as a directive if the pattern matches a directive and appears at the beginning of the Dockerfile. Otherwise, it's treated as a comment.

[Installing dependencies](https://docs.docker.com/build/concepts/dockerfile/" \l "installing-dependencies)

The second RUN instruction installs the flask dependency required by the Python application.

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) pip install flask==3.0.\*

A prerequisite for this instruction is that pip is installed into the build container. The first RUN command installs pip, which ensures that we can use the command to install the flask web framework.

[Copying files](https://docs.docker.com/build/concepts/dockerfile/" \l "copying-files)

The next instruction uses the [COPY instruction](https://docs.docker.com/reference/dockerfile/" \l "copy) to copy the hello.py file from the local build context into the root directory of our image.

[COPY](https://docs.docker.com/reference/dockerfile/" \l "copy" \o "Learn more about the COPY instruction) hello.py /

A [build context](https://docs.docker.com/build/concepts/context/) is the set of files that you can access in Dockerfile instructions such as COPY and ADD.

After the COPY instruction, the hello.py file is added to the filesystem of the build container.

[Setting environment variables](https://docs.docker.com/build/concepts/dockerfile/" \l "setting-environment-variables)

If your application uses environment variables, you can set environment variables in your Docker build using the [ENV instruction](https://docs.docker.com/reference/dockerfile/" \l "env).

[ENV](https://docs.docker.com/reference/dockerfile/" \l "env" \o "Learn more about the ENV instruction) FLASK\_APP=hello

This sets a Linux environment variable we'll need later. Flask, the framework used in this example, uses this variable to start the application. Without this, flask wouldn't know where to find our application to be able to run it.

[Exposed ports](https://docs.docker.com/build/concepts/dockerfile/" \l "exposed-ports)

The [EXPOSE instruction](https://docs.docker.com/reference/dockerfile/" \l "expose) marks that our final image has a service listening on port 8000.

[EXPOSE](https://docs.docker.com/reference/dockerfile/" \l "expose" \o "Learn more about the EXPOSE instruction) 8000

This instruction isn't required, but it is a good practice and helps tools and team members understand what this application is doing.

[Starting the application](https://docs.docker.com/build/concepts/dockerfile/" \l "starting-the-application)

Finally, [CMD instruction](https://docs.docker.com/reference/dockerfile/" \l "cmd) sets the command that is run when the user starts a container based on this image.

[CMD](https://docs.docker.com/reference/dockerfile/" \l "cmd" \o "Learn more about the CMD instruction) ["flask", "run", "--host", "0.0.0.0", "--port", "8000"]

This command starts the flask development server listening on all addresses on port 8000. The example here uses the "exec form" version of CMD. It's also possible to use the "shell form":

[CMD](https://docs.docker.com/reference/dockerfile/" \l "cmd" \o "Learn more about the CMD instruction) flask run --host 0.0.0.0 --port 8000

There are subtle differences between these two versions, for example in how they trap signals like SIGTERM and SIGKILL. For more information about these differences, see [Shell and exec form](https://docs.docker.com/reference/dockerfile/" \l "shell-and-exec-form)

[Building](https://docs.docker.com/build/concepts/dockerfile/" \l "building)

To build a container image using the Dockerfile example from the [previous section](https://docs.docker.com/build/concepts/dockerfile/" \l "example), you use the docker build command:

docker build -t test:latest .

The -t test:latest option specifies the name and tag of the image.

The single dot (.) at the end of the command sets the [build context](https://docs.docker.com/build/concepts/context/) to the current directory. This means that the build expects to find the Dockerfile and the hello.py file in the directory where the command is invoked. If those files aren't there, the build fails.

After the image has been built, you can run the application as a container with docker run, specifying the image name:

docker run -p 127.0.0.1:8000:8000 test:latest

This publishes the container's port 8000 to http://localhost:8000 on the Docker host.

**Docker Build Overview**

Docker Build implements a client-server architecture, where:

* Client: Buildx is the client and the user interface for running and managing builds.
* Server: BuildKit is the server, or builder, that handles the build execution.

When you invoke a build, the Buildx client sends a build request to the BuildKit backend. BuildKit resolves the build instructions and executes the build steps. The build output is either sent back to the client or uploaded to a registry, such as Docker Hub.

Buildx and BuildKit are both installed with Docker Desktop and Docker Engine out-of-the-box. When you invoke the docker build command, you're using Buildx to run a build using the default BuildKit bundled with Docker.

[Buildx](https://docs.docker.com/build/concepts/overview/" \l "buildx)

Buildx is the CLI tool that you use to run builds. The docker build command is a wrapper around Buildx. When you invoke docker build, Buildx interprets the build options and sends a build request to the BuildKit backend.

The Buildx client can do more than just run builds. You can also use Buildx to create and manage BuildKit backends, referred to as builders. It also supports features for managing images in registries, and for running multiple builds concurrently.

Docker Buildx is installed by default with Docker Desktop. You can also build the CLI plugin from source, or grab a binary from the GitHub repository and install it manually. See [Buildx README](https://github.com/docker/buildx" \l "manual-download" \t "_blank) on GitHub for more information.

[BuildKit](https://docs.docker.com/build/concepts/overview/" \l "buildkit)

BuildKit is the daemon process that executes the build workloads.

A build execution starts with the invocation of a docker build command. Buildx interprets your build command and sends a build request to the BuildKit backend. The build request includes:

* The Dockerfile
* Build arguments
* Export options
* Caching options

BuildKit resolves the build instructions and executes the build steps. While BuildKit is executing the build, Buildx monitors the build status and prints the progress to the terminal.

If the build requires resources from the client, such as local files or build secrets, BuildKit requests the resources that it needs from Buildx.

This is one way in which BuildKit is more efficient compared to the legacy builder used in earlier versions of Docker. BuildKit only requests the resources that the build needs when they're needed. The legacy builder, in comparison, always takes a copy of the local filesystem.

Examples of resources that BuildKit can request from Buildx include:

* Local filesystem build contexts
* Build secrets
* SSH sockets
* Registry authentication tokens

For more information about BuildKit, see [BuildKit](https://docs.docker.com/build/buildkit/).

**BuildKit**

[Overview](https://docs.docker.com/build/buildkit/" \l "overview)

[BuildKit](https://github.com/moby/buildkit" \t "_blank) is an improved backend to replace the legacy builder. BuildKit is the default builder for users on Docker Desktop, and Docker Engine as of version 23.0.

BuildKit provides new functionality and improves your builds' performance. It also introduces support for handling more complex scenarios:

* Detect and skip executing unused build stages
* Parallelize building independent build stages
* Incrementally transfer only the changed files in your [build context](https://docs.docker.com/build/concepts/context/) between builds
* Detect and skip transferring unused files in your [build context](https://docs.docker.com/build/concepts/context/)
* Use [Dockerfile frontend](https://docs.docker.com/build/buildkit/frontend/) implementations with many new features
* Avoid side effects with rest of the API (intermediate images and containers)
* Prioritize your build cache for automatic pruning

Apart from many new features, the main areas BuildKit improves on the current experience are performance, storage management, and extensibility. From the performance side, a significant update is a new fully concurrent build graph solver. It can run build steps in parallel when possible and optimize out commands that don't have an impact on the final result. We have also optimized the access to the local source files. By tracking only the updates made to these files between repeated build invocations, there is no need to wait for local files to be read or uploaded before the work can begin.

[LLB](https://docs.docker.com/build/buildkit/" \l "llb)

At the core of BuildKit is a [Low-Level Build (LLB)](https://github.com/moby/buildkit" \l "exploring-llb" \t "_blank) definition format. LLB is an intermediate binary format that allows developers to extend BuildKit. LLB defines a content-addressable dependency graph that can be used to put together very complex build definitions. It also supports features not exposed in Dockerfiles, like direct data mounting and nested invocation.

Everything about execution and caching of your builds is defined in LLB. The caching model is entirely rewritten compared to the legacy builder. Rather than using heuristics to compare images, LLB directly tracks the checksums of build graphs and content mounted to specific operations. This makes it much faster, more precise, and portable. The build cache can even be exported to a registry, where it can be pulled on-demand by subsequent invocations on any host.

LLB can be generated directly using a [golang client package](https://pkg.go.dev/github.com/moby/buildkit/client/llb" \t "_blank) that allows defining the relationships between your build operations using Go language primitives. This gives you full power to run anything you can imagine, but will probably not be how most people will define their builds. Instead, most users would use a frontend component, or LLB nested invocation, to run a prepared set of build steps.

[Frontend](https://docs.docker.com/build/buildkit/" \l "frontend)

A frontend is a component that takes a human-readable build format and converts it to LLB so BuildKit can execute it. Frontends can be distributed as images, and the user can target a specific version of a frontend that is guaranteed to work for the features used by their definition.

For example, to build a [Dockerfile](https://docs.docker.com/reference/dockerfile/) with BuildKit, you would [use an external Dockerfile frontend](https://docs.docker.com/build/buildkit/frontend/).

[Getting started](https://docs.docker.com/build/buildkit/" \l "getting-started)

BuildKit is the default builder for users on Docker Desktop and Docker Engine v23.0 and later.

If you have installed Docker Desktop, you don't need to enable BuildKit. If you are running a version of Docker Engine version earlier than 23.0, you can enable BuildKit either by setting an environment variable, or by making BuildKit the default setting in the daemon configuration.

To set the BuildKit environment variable when running the docker build command, run:

DOCKER\_BUILDKIT=1 docker build .

**Note**

Buildx always uses BuildKit.

To use Docker BuildKit by default, edit the Docker daemon configuration in /etc/docker/daemon.json as follows, and restart the daemon.

{

"features": {

"buildkit": true

}

}

If the /etc/docker/daemon.json file doesn't exist, create new file called daemon.json and then add the following to the file. And restart the Docker daemon.

[BuildKit on Windows](https://docs.docker.com/build/buildkit/" \l "buildkit-on-windows)

**Warning**

BuildKit only fully supports building Linux containers. Windows container support is experimental, and is tracked in [moby/buildkit#616](https://github.com/moby/buildkit/issues/616" \t "_blank).

BuildKit has experimental support for Windows containers (WCOW) as of version 0.13. This section walks you through the steps for trying it out. We appreciate any feedback you submit by [opening an issue here](https://github.com/moby/buildkit/issues/new" \t "_blank), especially buildkitd.exe.

[Known limitations](https://docs.docker.com/build/buildkit/" \l "known-limitations)

* BuildKit on Windows currently only supports the containerd worker. Support for non-OCI workers is tracked in [moby/buildkit#4836](https://github.com/moby/buildkit/issues/4836" \t "_blank).

[Prerequisites](https://docs.docker.com/build/buildkit/" \l "prerequisites)

* Architecture: amd64, arm64 (binaries available but not officially tested yet).
* Supported OS: Windows Server 2019, Windows Server 2022, Windows 11.
* Base images: ServerCore:ltsc2019, ServerCore:ltsc2022, NanoServer:ltsc2022. See the [compatibility map here](https://learn.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/version-compatibility?tabs=windows-server-2019%2Cwindows-11" \l "windows-server-host-os-compatibility" \t "_blank).
* Docker Desktop version 4.29 or later

[Steps](https://docs.docker.com/build/buildkit/" \l "steps)

**Note**

The following commands require administrator (elevated) privileges in a PowerShell terminal.

1. Enable the **Hyper-V** and **Containers** Windows features.
2. Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V, Containers -All

If you see RestartNeeded as True, restart your machine and re-open a PowerShell terminal as an administrator. Otherwise, continue with the next step.

1. Switch to Windows containers in Docker Desktop.

Select the Docker icon in the taskbar, and then **Switch to Windows containers...**.

1. Install containerd version 1.7.7 or later following the setup instructions [here](https://github.com/containerd/containerd/blob/main/docs/getting-started.md" \l "installing-containerd-on-windows" \t "_blank).
2. Download and extract the latest BuildKit release.
3. $version = "v0.13.1" # specify the release version, v0.13+
4. $arch = "amd64" # arm64 binary available too
5. curl.exe -LO https://github.com/moby/buildkit/releases/download/$version/buildkit-$version.windows-$arch.tar.gz
6. # there could be another `.\bin` directory from containerd instructions
7. # you can move those
8. mv bin bin2
9. tar.exe xvf .\buildkit-$version.windows-$arch.tar.gz
10. ## x bin/
11. ## x bin/buildctl.exe

## x bin/buildkitd.exe

1. Install BuildKit binaries on PATH.
2. # after the binaries are extracted in the bin directory
3. # move them to an appropriate path in your $Env:PATH directories or:
4. Copy-Item -Path ".\bin" -Destination "$Env:ProgramFiles\buildkit" -Recurse -Force
5. # add `buildkitd.exe` and `buildctl.exe` binaries in the $Env:PATH
6. $Path = [Environment]::GetEnvironmentVariable("PATH", "Machine") + `
7. [IO.Path]::PathSeparator + "$Env:ProgramFiles\buildkit"
8. [Environment]::SetEnvironmentVariable( "Path", $Path, "Machine")
9. $Env:Path = [System.Environment]::GetEnvironmentVariable("Path","Machine") + ";" + `

[System.Environment]::GetEnvironmentVariable("Path","User")

1. Start the BuildKit daemon.
2. buildkitd.exe
3. In another terminal with administrator privileges, create a remote builder that uses the local BuildKit daemon.

**Note**

This requires Docker Desktop version 4.29 or later.

docker buildx create --name buildkit-exp --use --driver=remote npipe:////./pipe/buildkitd

buildkit-exp

1. Verify the builder connection by running docker buildx inspect.
2. docker buildx inspect

The output should indicate that the builder platform is Windows, and that the endpoint of the builder is a named pipe.

Name: buildkit-exp

Driver: remote

Last Activity: 2024-04-15 17:51:58 +0000 UTC

Nodes:

Name: buildkit-exp0

Endpoint: npipe:////./pipe/buildkitd

Status: running

BuildKit version: v0.13.1

Platforms: windows/amd64

...

1. Create a Dockerfile and build a hello-buildkit image.
2. mkdir sample\_dockerfile
3. cd sample\_dockerfile
4. Set-Content Dockerfile @"
5. FROM mcr.microsoft.com/windows/nanoserver:ltsc2022
6. USER ContainerAdministrator
7. COPY hello.txt C:/
8. RUN echo "Goodbye!" >> hello.txt
9. CMD ["cmd", "/C", "type C:\\hello.txt"]
10. "@
11. Set-Content hello.txt @"
12. Hello from BuildKit!
13. This message shows that your installation appears to be working correctly.
14. "@
15. Build and push the image to a registry.
16. docker buildx build --push -t <username>/hello-buildkit .
17. After pushing to the registry, run the image with docker run.

docker run <username>/hello-buildkit

**Docker build cache**

When you build the same Docker image multiple times, knowing how to optimize the build cache is a great tool for making sure the builds run fast.

[How the build cache works](https://docs.docker.com/build/cache/" \l "how-the-build-cache-works)

Understanding Docker's build cache helps you write better Dockerfiles that result in faster builds.

The following example shows a small Dockerfile for a program written in C.

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) ubuntu:latest

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) apt-get update && apt-get install -y build-essentials

[COPY](https://docs.docker.com/reference/dockerfile/" \l "copy" \o "Learn more about the COPY instruction) main.c Makefile /src/

[WORKDIR](https://docs.docker.com/reference/dockerfile/" \l "workdir" \o "Learn more about the WORKDIR instruction) /src/

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) make build

Each instruction in this Dockerfile translates to a layer in your final image. You can think of image layers as a stack, with each layer adding more content on top of the layers that came before it:

Whenever a layer changes, that layer will need to be re-built. For example, suppose you make a change to your program in the main.c file. After this change, the COPY command will have to run again in order for those changes to appear in the image. In other words, Docker will invalidate the cache for this layer.

If a layer changes, all other layers that come after it are also affected. When the layer with the COPY command gets invalidated, all layers that follow will need to run again, too:

And that's the Docker build cache in a nutshell. Once a layer changes, then all downstream layers need to be rebuilt as well. Even if they wouldn't build anything differently, they still need to re-run.

For more details about how cache invalidation works, see [Cache invalidation](https://docs.docker.com/build/cache/invalidation/).

[Optimizing how you use the build cache](https://docs.docker.com/build/cache/" \l "optimizing-how-you-use-the-build-cache)

Now that you understand how the cache works, you can begin to use the cache to your advantage. While the cache will automatically work on any docker build that you run, you can often refactor your Dockerfile to get even better performance. These optimizations can save precious seconds (or even minutes) off of your builds.

[Order your layers](https://docs.docker.com/build/cache/" \l "order-your-layers)

Putting the commands in your Dockerfile into a logical order is a great place to start. Because a change causes a rebuild for steps that follow, try to make expensive steps appear near the beginning of the Dockerfile. Steps that change often should appear near the end of the Dockerfile, to avoid triggering rebuilds of layers that haven't changed.

Consider the following example. A Dockerfile snippet that runs a JavaScript build from the source files in the current directory:

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) node

[WORKDIR](https://docs.docker.com/reference/dockerfile/" \l "workdir" \o "Learn more about the WORKDIR instruction) /app

[COPY](https://docs.docker.com/reference/dockerfile/" \l "copy" \o "Learn more about the COPY instruction) . . # Copy over all files in the current directory

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) npm install # Install dependencies

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) npm build # Run build

This Dockerfile is rather inefficient. Updating any file causes a reinstall of all dependencies every time you build the Docker image even if the dependencies didn't change since last time!

Instead, the COPY command can be split in two. First, copy over the package management files (in this case, package.json and yarn.lock). Then, install the dependencies. Finally, copy over the project source code, which is subject to frequent change.

# syntax=docker/dockerfile:1

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) node

[WORKDIR](https://docs.docker.com/reference/dockerfile/" \l "workdir" \o "Learn more about the WORKDIR instruction) /app

[COPY](https://docs.docker.com/reference/dockerfile/" \l "copy" \o "Learn more about the COPY instruction) package.json yarn.lock . # Copy package management files

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) npm install # Install dependencies

[COPY](https://docs.docker.com/reference/dockerfile/" \l "copy" \o "Learn more about the COPY instruction) . . # Copy over project files

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) npm build # Run build

By installing dependencies in earlier layers of the Dockerfile, there is no need to rebuild those layers when a project file has changed.

[Keep layers small](https://docs.docker.com/build/cache/" \l "keep-layers-small)

One of the best things you can do to speed up image building is to just put less stuff into your build. Fewer parts means the cache stay smaller, but also that there should be fewer things that could be out-of-date and need rebuilding.

To get started, here are a few tips and tricks:

[Don't include unnecessary files](https://docs.docker.com/build/cache/" \l "dont-include-unnecessary-files)

Be considerate of what files you add to the image.

Running a command like COPY . /src will copy your entire [build context](https://docs.docker.com/build/concepts/context/) into the image. If you've got logs, package manager artifacts, or even previous build results in your current directory, those will also be copied over. This could make your image larger than it needs to be, especially as those files are usually not useful.

Avoid adding unnecessary files to your builds by explicitly stating the files or directories you intend to copy over. For example, you might only want to add a Makefile and your src directory to the image filesystem. In that case, consider adding this to your Dockerfile:

[COPY](https://docs.docker.com/reference/dockerfile/" \l "copy" \o "Learn more about the COPY instruction) ./src ./Makefile /src

As opposed to this:

[COPY](https://docs.docker.com/reference/dockerfile/" \l "copy" \o "Learn more about the COPY instruction) . /src

You can also create a [.dockerignore file](https://docs.docker.com/build/concepts/context/" \l "dockerignore-files), and use that to specify which files and directories to exclude from the build context.

[Use your package manager wisely](https://docs.docker.com/build/cache/" \l "use-your-package-manager-wisely)

Most Docker image builds involve using a package manager to help install software into the image. Debian has apt, Alpine has apk, Python has pip, NodeJS has npm, and so on.

When installing packages, be considerate. Make sure to only install the packages that you need. If you're not going to use them, don't install them. Remember that this might be a different list for your local development environment and your production environment. You can use multi-stage builds to split these up efficiently.

[Use the dedicated RUN cache](https://docs.docker.com/build/cache/" \l "use-the-dedicated-run-cache)

The RUN command supports a specialized cache, which you can use when you need a more fine-grained cache between runs. For example, when installing packages, you don't always need to fetch all of your packages from the internet each time. You only need the ones that have changed.

To solve this problem, you can use RUN --mount type=cache. For example, for your Debian-based image you might use the following:

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) \

--mount=type=cache,target=/var/cache/apt \

apt-get update && apt-get install -y git

Using the explicit cache with the --mount flag keeps the contents of the target directory preserved between builds. When this layer needs to be rebuilt, then it'll use the apt cache in /var/cache/apt.

[Minimize the number of layers](https://docs.docker.com/build/cache/" \l "minimize-the-number-of-layers)

Keeping your layers small is a good first step, and the logical next step is to reduce the number of layers that you have. Fewer layers mean that you have less to rebuild, when something in your Dockerfile changes, so your build will complete faster.

The following sections outline some tips you can use to keep the number of layers to a minimum.

[Use an appropriate base image](https://docs.docker.com/build/cache/" \l "use-an-appropriate-base-image)

Docker provides over 170 pre-built [official images](https://hub.docker.com/search?q=&image_filter=official" \t "_blank) for almost every common development scenario. For example, if you're building a Java web server, use a dedicated image such as [eclipse-temurin](https://hub.docker.com/_/eclipse-temurin/" \t "_blank). Even when there's not an official image for what you might want, Docker provides images from [verified publishers](https://hub.docker.com/search?q=&image_filter=store" \t "_blank) and [open source partners](https://hub.docker.com/search?q=&image_filter=open_source" \t "_blank) that can help you on your way. The Docker community often produces third-party images to use as well.

Using official images saves you time and ensures you stay up to date and secure by default.

[Use multi-stage builds](https://docs.docker.com/build/cache/" \l "use-multi-stage-builds)

[Multi-stage builds](https://docs.docker.com/build/building/multi-stage/) let you split up your Dockerfile into multiple distinct stages. Each stage completes a step in the build process, and you can bridge the different stages to create your final image at the end. The Docker builder will work out dependencies between the stages and run them using the most efficient strategy. This even allows you to run multiple builds concurrently.

Multi-stage builds use two or more FROM commands. The following example illustrates building a simple web server that serves HTML from your docs directory in Git:

# syntax=docker/dockerfile:1

# stage 1

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) alpine as git

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) apk add git

# stage 2

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) git as fetch

[WORKDIR](https://docs.docker.com/reference/dockerfile/" \l "workdir" \o "Learn more about the WORKDIR instruction) /repo

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) git clone https://github.com/your/repository.git .

# stage 3

[FROM](https://docs.docker.com/reference/dockerfile/" \l "from" \o "Learn more about the FROM instruction) nginx as site

[COPY](https://docs.docker.com/reference/dockerfile/" \l "copy" \o "Learn more about the COPY instruction) --from=fetch /repo/docs/ /usr/share/nginx/html

This build has 3 stages: git, fetch and site. In this example, git is the base for the fetch stage. It uses the COPY --from flag to copy the data from the docs/ directory into the Nginx server directory.

Each stage has only a few instructions, and when possible, Docker will run these stages in parallel. Only the instructions in the site stage will end up as layers in the final image. The entire git history doesn't get embedded into the final result, which helps keep the image small and secure.

[Combine commands together wherever possible](https://docs.docker.com/build/cache/" \l "combine-commands-together-wherever-possible)

Most Dockerfile commands, and RUN commands in particular, can often be joined together. For example, instead of using RUN like this:

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) echo "the first command"

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) echo "the second command"

It's possible to run both of these commands inside a single RUN, which means that they will share the same cache! This is achievable using the && shell operator to run one command after another:

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) echo "the first command" && echo "the second command"

# or to split to multiple lines

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) echo "the first command" && \

echo "the second command"

Another shell feature that allows you to simplify and concatenate commands in a neat way are [heredocs](https://en.wikipedia.org/wiki/Here_document" \t "_blank). It enables you to create multi-line scripts with good readability:

[RUN](https://docs.docker.com/reference/dockerfile/" \l "run" \o "Learn more about the RUN instruction) <<EOF

set -e

echo "the first command"

echo "the second command"

EOF

(Note the set -e command to exit immediately after any command fails, instead of continuing.)

[Other resources](https://docs.docker.com/build/cache/" \l "other-resources)

For more information on using cache to do efficient builds, see:

* [Cache invalidation](https://docs.docker.com/build/cache/invalidation/)
* [Garbage collection](https://docs.docker.com/build/cache/garbage-collection/)
* [Cache storage backends](https://docs.docker.com/build/cache/backends/)