### 

### What is Docker Image ?

* A **Docker image** is an immutable (unchangeable) file that contains the source code, libraries, dependencies, tools, and other files needed for an application to run.
* Due to their **read-only** quality, these images are sometimes referred to as snapshots. They represent an application and its virtual environment at a specific point in time. This consistency is one of the great features of Docker. It allows developers to test and experiment software in stable, uniform conditions.
* Since images are, in a way, just **templates**, you cannot start or run them. What you can do is use that template as a base to build a container.  A container is, ultimately, just a running image. Once you create a container, it adds a writable layer on top of the immutable image, meaning you can now modify it.
* An image is a n read only template for creating application containers.
* An image is a bunch of independent layers that are very loosely connected by manifest file/config file. The manifest file which describes the image like its ID and tags and also it includes the list of layers get stacked and how to stak them. Now these layers are unware of the bigger image , independent layer, they have no clue that they are going to be stacked up.

Anatomy of a Docker Image

A Docker image is made up of a collection of files that bundle together all the essentials – such as installations, application code, and dependencies – required to configure a fully operational container environment. You can create a Docker image by using one of two methods:

* **Interactive**: By running a container from an existing Docker image, manually changing that container environment through a series of live steps, and saving the resulting state as a new image.
* **Dockerfile**: By constructing a plain-text file, known as a **Dockerfile**, which provides the specifications for creating a Docker image.

### Image vs Container

|  |  |
| --- | --- |
| **Docker Images** | **Docker Containers** |
| It is Blueprint of the Container. | It is instance of the Image. |
| Image is a logical entity. | Container is a real world entity. |
| Image is created only once. | Containers are created any number of times using image. |
| Images are immutable. | Containers changes only if old image is deleted and new is used to build the container. |
| Images does not require computing resource to work. | Containers requires computing resources to run as they run as Docker Virtual Machine. |
| To make a docker image, you have to write script in Dockerfile. | To make container from image, you have to run “docker build .” command |
| Docker Images are used to package up applications and pre-configured server environments. | Containers use server information and file system provided by image in order to operate. |
| Images can be shared on Docker Hub. | It makes no sense in sharing a running entity, always docker images are shared. |
| There is no such running state of Docker Image. | Containers uses RAM when created and in running state |

### Layering in Docker

* Each of the files that make up a Docker image is known as a **layer**. These layers form a series of **intermediate images**, built one on top of the other in stages, where each layer is dependent on the layer immediately below it. The **hierarchy** of your layers is key to efficient lifecycle management of your Docker images. Thus, you should organize layers that change most often as high up the stack as possible. This is because, when you make changes to a layer in your image, Docker not only rebuilds that particular layer, but all layers built from it. Therefore, a change to a layer at the top of a stack involves the least amount of computational work to rebuild the entire image.
* A Docker image is built up from a series of layers. Each layer represents an instruction in the image’s Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

This Dockerfile contains four commands, each of which creates a layer. The FROM statement starts out by creating a layer from the ubuntu:18.04 image. The COPY command adds some files from your Docker client’s current directory. The RUN command builds your application using the make command. Finally, the last layer specifies what command to run within the container.

*# syntax=docker/dockerfile:1*

*FROM ubuntu:18.04*

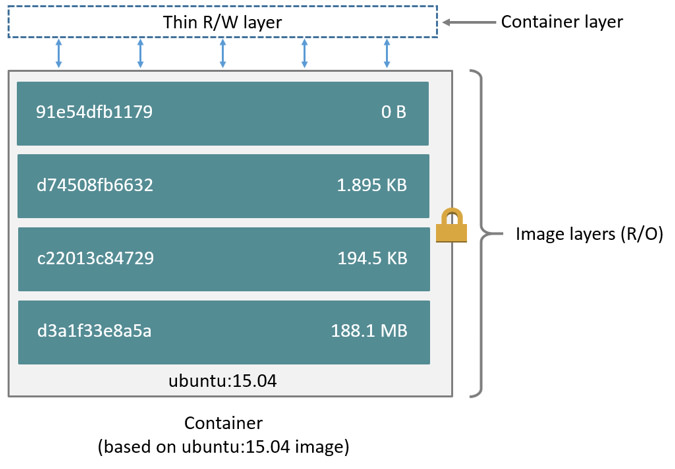
*COPY . /app*

*RUN make /app*

*CMD python /app/app.py*

Each layer is only a set of differences from the layer before it. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the “container layer”.

All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on the Ubuntu 15.04 image.

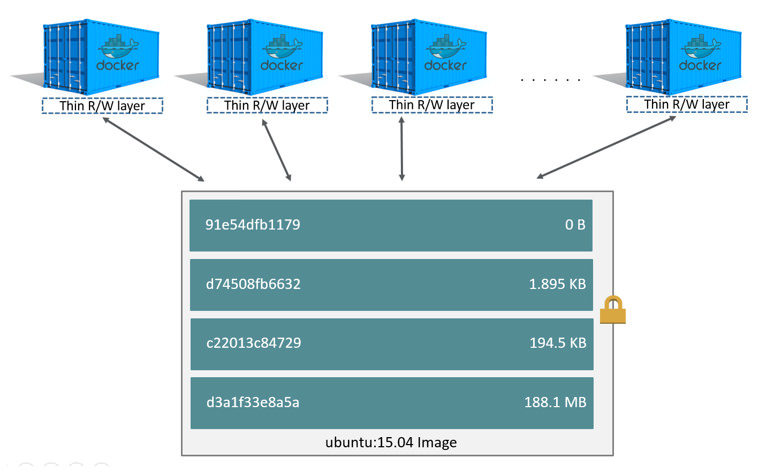


A storage driver handles the details about the way these layers interact with each other. Different storage drivers are available, which have advantages and disadvantages in different situations.

**Container and Layers**

The major difference between a container and an image is the top writable layer. All writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged.

Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state.



Docker uses storage drivers to manage the contents of the image layers and the writable container layer. Each storage driver handles the implementation differently, but all drivers use stackable image layers and the copy-on-write (CoW) strategy.

The copy-on-write (CoW) strategy🔗

* Copy-on-write is a strategy of sharing and copying files for maximum efficiency. If a file or directory exists in a lower layer within the image, and another layer (including the writable layer) needs read access to it, it just uses the existing file. The first time another layer needs to modify the file (when building the image or running the container), the file is copied into that layer and modified. This minimizes I/O and the size of each of the subsequent layers. These advantages are explained in more depth below.
* When you use docker pull to pull down an image from a repository, or when you create a container from an image that does not yet exist locally, each layer is pulled down separately, and stored in Docker’s local storage area, which is usually **/var/lib/docker**/ on Linux hosts. You can see these layers being pulled in this example:

**$ docker pull ubuntu:18.04**

18.04: Pulling from library/ubuntu

f476d66f5408: Pull complete

8882c27f669e: Pull complete

d9af21273955: Pull complete

f5029279ec12: Pull complete

Digest: sha256:ab6cb8de3ad7bb33e2534677f865008535427390b117d7939193f8d1a6613e34

Status: Downloaded newer image for ubuntu:18.04

Each of these layers is stored in its own directory inside the Docker host’s local storage area. To examine the layers on the filesystem, list the contents of /var/lib/docker/<storage-driver>. This example uses the overlay2 storage driver:

$ ls /var/lib/docker/overlay2

**Copying makes containers efficient🔗**

* When you start a container, a thin writable container layer is added on top of the other layers. Any changes the container makes to the filesystem are stored here. Any files the container does not change do not get copied to this writable layer. This means that the writable layer is as small as possible.
* When an existing file in a container is modified, the storage driver performs a copy-on-write operation. The specifics steps involved depend on the specific storage driver. For the aufs, overlay, and overlay2 drivers, the copy-on-write operation follows this rough sequence:
  + Search through the image layers for the file to update. The process starts at the newest layer and works down to the base layer one layer at a time. When results are found, they are added to a cache to speed future operations.
  + Perform a copy\_up operation on the first copy of the file that is found, to copy the file to the container’s writable layer.
  + Any modifications are made to this copy of the file, and the container cannot see the read-only copy of the file that exists in the lower layer.
  + Btrfs, ZFS, and other drivers handle the copy-on-write differently. You can read more about the methods of these drivers later in their detailed descriptions.
  + Containers that write a lot of data consume more space than containers that do not. This is because most write operations consume new space in the container’s thin writable top layer.
  + Note: for write-heavy applications, you should not store the data in the container. Instead, use Docker volumes, which are independent of the running container and are designed to be efficient for I/O. In addition, volumes can be shared among containers and do not increase the size of your container’s writable layer.
  + A copy\_up operation can incur a noticeable performance overhead. This overhead is different depending on which storage driver is in use. Large files, lots of layers, and deep directory trees can make the impact more noticeable. This is mitigated by the fact that each copy\_up operation only occurs the first time a given file is modified.
  + To verify the way that copy-on-write works, the following procedures spins up 5 containers based on the acme/my-final-image:1.0 image we built earlier and examines how much room they take up.

### Docker Image ls | Docker images

*The default docker images will show all top level images, their repository and tags, and their size.*

*Docker images have intermediate layers that increase reusability, decrease disk usage, and speed up docker build by allowing each step to be cached. These intermediate layers are not shown by default.*

*The SIZE is the cumulative space taken up by the image and all its parent images. This is also the disk space used by the contents of the Tar file created when you docker save an image.*

*An image will be listed more than once if it has multiple repository names or tags. This single image (identifiable by its matching IMAGE ID) uses up the SIZE listed only once.*

***List All Images***

*$ docker image ls [OPTIONS] [REPOSITORY[:TAG]]*

*or*

*$ docker images [OPTIONS] [REPOSITORY[:TAG]]*

*--no-trunc*

*-a show all images*

*-f filter output based on conditions provided*

*--digests show digests*

*-q Only show image IDS*

***List images By Name & Tag***

*$ docker images java*

*$ docker images java:8.*

***Filtering***

*The filtering flag (-f or --filter) format is of “key=value”. If there is more than one filter, then pass multiple flags (e.g., --filter "foo=bar" --filter "bif=baz")*

*The currently supported filters are:*

*dangling (boolean - true or false)*

*label (label=<key> or label=<key>=<value>)*

*before*

*since*

*reference*

***Filter Untagged Images [Dangling]***

*$ docker images --filter "dangling=true"*

***Filter images with a given label***

*$ docker images --filter "label=com.example.version=1.0"*

***Filter Images by time***

*$ docker images -f “before=images1”*

*$ docker images -f “since=image3”*

***Filter Images by Reference***

*filter shows only images whose reference matches the specified pattern*

*$ docker images --filter=reference='busy\*:\*libc' # image:tag*

*$ docker images --filter=reference='busy\*:uclibc' --filter=reference='busy\*:glibc'*

***Format the output***

*$ docker images --format "{{.ID}}: {{.Repository}}"*

*$ docker images --format "table {{.ID}}\t{{.Repository}}\t{{.Tag}}"*

*.ID Image ID*

*.Repository Image repo*

*.Tag Image Tag*

*.Digest Image Digest*

*.Size Image Disk Size*

*.CreatedSince Elapsed time since the image was created*

*.CreatedAt Time when the image was created*

### Docker Image rm

***docker image rm | docker image rmi***

***[***Remove one or more images]

*$ docker image rm [OPTIONS] IMAGE [IMAGE...]*

*$ docker image rm test:latest*

*$ docker rmi $(docker images -a -q)*

*-f force*

*--no-prune do not delete untagged parents*

### Docker Image Prune

***docker image prune***

*[Remove unused images]*

*$ docker image prune [OPTIONS]*

*-a Remove all unused images, not just dangling ones.*

*--filter Provide filter values (e.g. 'until=<timestamp>')*

*-f force*

*Clean up dangling images. - This is the default behavior.*

*Clean up all Docker images that have no running containers associated with them.*

***Remove all images without at least one container associated to them***

*$ docker image prune -a*

***Filtering***

*The currently supported filters are:*

*until*

*label*

***Removes images created before 2017-01-04T00:00:00***

*$ docker images --format 'table {{.Repository}}\t{{.Tag}}\t{{.ID}}\t{{.CreatedAt}}\t{{.Size}}'*

*$ docker image prune -a --force --filter "until=2017-01-04T00:00:00"*

***Removes images created more than 10 days (240h) ago***

*$ docker images*

*$ docker image prune -a --force --filter "until=240h"*

***Removes images with the label maintainer set to john***

*$ docker image prune --filter="label=maintainer=john"*

***Removes images with the label deprecated:***

*$ docker image prune --filter="label=deprecated"*

***Removes images which have no maintainer label:***

*$ docker image prune --filter="label!=maintainer"*

***Removes images which have a maintainer label not set to john:***

*$ docker image prune --filter="label!=maintainer=john"*

### What is Build Context ??

The build context is the set of files located at the specified PATH or URL. Those files are sent to the Docker daemon during the build so it can use them in the filesystem of the image.

Docker Build Syntax : $ docker build [OPTIONS] PATH | URL | -

**Things to remember about the build context**:

* Files inside the build context are the only files readable by the instructions specified in the Dockerfile.
* Any symlinks that point to external locations will not be resolved.
* If a .dockerignore file is specified at the root of the build context, it can be used to exclude files from the build context by adding filtering rules

**Using a path**

Let’s suppose I’m in the folder /Users/luc/src/github.com/lucj/genx containing the source code of the genx application

Usually, we use a command like the following one to build the image, the Dockerfile being at the root of the project’s folder:

$ docker image build -t genx:1.0 .

In that case, the build context is the content of the current folder (“.” specified as the last element of the command).

**Using a URL**

This same genx project is managed in GitLab, so it’s possible to build an image locally referencing the GitLab repository:

$ docker image build -t genx:1.0 [git@gitlab.com:lucj/genx.git](mailto:git@gitlab.com:lucj/genx.git)

$ docker build -t automatron <https://github.com/madflojo/automatron.git>

$ docker build -t automatron http://example.com/automatron.tar.gz

In that case, the build context is the set of files in gitlab.com/lucj/genx.

Basically, the build context contains at least the application code which will be copied over to the image filesystem, but it often contains many other things we may or may not need in the image.

**Should I Filter the Build Context?**

Yes, we’d probably be better off making sure the build context only contains the files and folders it really needs.

In a project where source code is handled by Git, we use a .gitignore file to make sure private data is kept locally and not sent out to GitHub/GitLab/BitBucket/etc.

The same thing applies during the build phase of a Docker image as the daemon uses a .dockerignore file to filter out the files and folders that should not be taken into account in the build context.

**What if I Don’t Use a .dockerignore?**

You would then send to the Docker daemon a lot of stuff that it does not need and which could be copied over to the image filesystem.

**Why Is the Build Context Used?**

* The build context is important because the Docker CLI and Docker Engine might not be running on the same machine. When you run docker build, the CLI sends the files and folders to build to the Docker engine. This set of files and folders becomes the build context.
* Furthermore, not every build context is as straightforward as reusing your working directory. Docker also supports Git repository URLs as the path given to docker build. In this case, the build context becomes the content of the specified repository.

**Compressing the Build Context**

You can compress the build context to further improve build performance. Pass the --compress flag to docker build to apply gzip compression. The compression occurs before the context is sent to the Docker daemon.

$ docker build . -t my-image:latest --compress

This can improve performance in some scenarios. The compression adds its own overheads, though—your system now needs to compress the context, and the receiving Docker daemon has to uncompress it. Using compression could actually be slower than copying the original files, in some circumstances. Experiment with each of your images to assess whether you see an improvement.

**Conclusion**

The Docker build context defines the files that will be available for copying in your Dockerfile. The build context is copied over to the Docker daemon before the build begins.

### Dockerignore File

The docker build command builds Docker images from a Dockerfile and a “context”. A build’s context is the set of files located in the specified PATH or URL. The build process can refer to any of the files in the context. For example, your build can use a COPY instruction to reference a file in the context.

The URL parameter can refer to three kinds of resources: Git repositories, pre-packaged tarball contexts and plain text files.

Before the docker CLI sends the context to the docker daemon, it looks for a file named .dockerignore in the root directory of the context. If this file exists, the CLI modifies the context to exclude files and directories that match patterns in it. This helps to avoid unnecessarily sending large or sensitive files and directories to the daemon and potentially adding them to images using ADD or COPY.

Matching is done using Go’s filepath.Match rules.

# 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.

**Reason**

**Security Issues** – Some important files such as passwords, secret keys, .git folders, etc contain a lot of information about your project and you might now want to expose those details to the outside world to prevent intrusion.

**Cache Invalidation** – When you write the Dockerfile, it’s a general practice to use the COPY instruction to copy the files and folders inside the Docker build context. Each statement inside the Dockerfile results in building a new intermediate image layer. Hence, when you make changes in your dockerfile again and again, this might lead to multiple Cache Invalidation and leads to wastage of resources.

Also, excluding unnecessary large files from your Docker Build Context will lead to lower Docker Image size.

It speeds up the process of building the Docker Image.

### Docker Image Build

Build an image from a Dockerfile

Syntax

$ docker image build [OPTIONS] PATH | URL | -

**Description**

The docker build command builds Docker images from a Dockerfile and a “context”. A build’s context is the set of files located in the specified PATH or URL. The build process can refer to any of the files in the context. For example, your build can use a COPY instruction to reference a file in the context.

The URL parameter can refer to three kinds of resources: Git repositories, pre-packaged tarball contexts and plain text files.

**Git repositories**

* When the URL parameter points to the location of a Git repository, the repository acts as the build context. The system recursively fetches the repository and its submodules. The commit history is not preserved. A repository is first pulled into a temporary directory on your local host. After that succeeds, the directory is sent to the Docker daemon as the context. Local copy gives you the ability to access private repositories using local user credentials, VPN’s, and so forth.
* Note: If the URL parameter contains a fragment the system will recursively clone the repository and its submodules using a git clone --recursive command.

**Example**

$ docker build https://github.com/docker/rootfs.git#container:docker

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

|  |  |  |
| --- | --- | --- |
| Build Syntax Suffix | Commit Used | Build Context Used |
| myrepo.git | refs/heads/master | / |
| myrepo.git#mybranch or tag | refs/heads/mybranch or tag | / |
| myrepo.git#:myfolder | refs/heads/master | /myfolder |
| myrepo.git#mybranch:myfolder | refs/heads/mybranch | /myfolder |
|  |  |  |

**Tarball contexts**

If you pass an URL to a remote tarball, the URL itself is sent to the daemon:

$ docker build http://server/context.tar.gz

The download operation will be performed on the host the Docker daemon is running on, which is not necessarily the same host from which the build command is being issued. The Docker daemon will fetch context.tar.gz and use 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.

**Text files**

Instead of specifying a context, you can pass a single Dockerfile in the URL or pipe the file in via STDIN. To pipe a Dockerfile from STDIN:

$ docker build - < Dockerfile

By default the docker build command will look for a Dockerfile at the root of the build context. The -f, --file, option lets you specify the path to an alternative file to use instead. This is useful in cases where the same set of files are used for multiple builds. The path must be to a file within the build context. If a relative path is specified then it is interpreted as relative to the root of the context.

In most cases, it’s best to put each Dockerfile in an empty directory. Then, add to that directory only the files needed for building the Dockerfile. To increase the build’s performance, you can exclude files and directories by adding a .dockerignore file to that directory as well. For information on creating one, see the .dockerignore file.

If the Docker client loses connection to the daemon, the build is canceled. This happens if you interrupt the Docker client with CTRL-c or if the Docker client is killed for any reason. If the build initiated a pull which is still running at the time the build is cancelled, the pull is cancelled as well.

### Docker Image Build Options

***Options***

*--add-host Add a custom host-to-IP mapping (host:ip)*

*--build-arg Set build-time variables*

*--cache-from Images to consider as cache sources*

*--compress Compress the build context using gzip*

*-f Name of the Dockerfile (Default is 'PATH/Dockerfile')*

*--force-rm=true|false Always remove intermediate containers*

*--iidfile Write the image ID to the file*

*--isolation Container isolation technology*

*--label Set metadata for an image*

*-m Memory limit*

*--network Set the networking mode for the RUN instructions during build*

*--no-cache Do not use cache when building the image*

*--pull Always attempt to pull a newer version of the image*

*-q Suppress the build output and print image ID on success*

*--secret Secret file to expose to the build (only if BuildKit enabled): id=mysecret,src=/local/secret*

*-t Name and optionally a tag in the 'name:tag' format*

*--target Set the target build stage to build.*

*--ulimit Ulimit options*

***Build with PATH***

*$ docker build .*

*This example specifies that the PATH is ., and so all the files in the local directory get tard and sent to the Docker daemon. The PATH specifies where to find the files for the “context” of the build on the Docker daemon. That means that all the files at PATH get sent, not just the ones listed to ADD in the Dockerfile.*

***Build with URL***

*$ docker build github.com/creack/docker-firefox*

*This will clone the GitHub repository and use the cloned repository as context. The Dockerfile at the root of the repository is used as Dockerfile. You can specify an arbitrary Git repository by using the git:// or git@ scheme.*

***Build with URL***

*$ docker build -f ctx/Dockerfile http://server/ctx.tar.gz*

*This sends the URL http://server/ctx.tar.gz to the Docker daemon, which downloads and extracts the referenced tarball. The -f ctx/Dockerfile parameter specifies a path inside ctx.tar.gz to the Dockerfile that is used to build the image.*

***Build with -***

*$ docker build - < Dockerfile*

*This will read a Dockerfile from STDIN without context. Due to the lack of a context, no contents of any local directory will be sent to the Docker daemon. Since there is no context, a Dockerfile ADD only works if it refers to a remote URL*

*$ docker build - < context.tar.gz*

*This will build an image for a compressed context read from STDIN. Supported formats are: bzip2, gzip and xz.*

***Tag an image (-t)***

*$ docker build -t vieux/apache:2.0 .*

*This will build like the previous example, but it will then tag the resulting image. The repository name will be vieux/apache and the tag will be 2.0. Read more about valid tags.*

*You can apply multiple tags to an image. For example, you can apply the latest tag to a newly built image and add another tag that references a specific version. For example, to tag an image both as whenry/fedora-jboss:latest and whenry/fedora-jboss:v2.1, use the following:*

*$ docker build -t whenry/fedora-jboss:latest -t whenry/fedora-jboss:v2.1 .*

***Specify a Dockerfile (-f)***

*$ docker build -f Dockerfile.debug .*

*This will use a file called Dockerfile.debug for the build instructions instead of Dockerfile..*

***Set build-time variables (--build-arg)***

*You can use ENV instructions in a Dockerfile to define variable values. These values persist in the built image. However, often persistence is not what you want. Users want to specify variables differently depending on which host they build an image on.*

*A good example is http\_proxy or source versions for pulling intermediate files. The ARG instruction lets Dockerfile authors define values that users can set at build-time using the --build-arg flag:*

*$ docker build --build-arg HTTP\_PROXY=http://10.20.30.2:1234 --build-arg FTP\_PROXY=http://40.50.60.5:4567 .*

*This flag allows you to pass the build-time variables that are accessed like regular environment variables in the RUN instruction of the Dockerfile. Also, these values don’t persist in the intermediate or final images like ENV values do. You must add --build-arg for each build argument.*

***Add entries to container hosts file (--add-host)***

*You can add other hosts into a container’s /etc/hosts file by using one or more --add-host flags. This example adds a static address for a host named docker:*

*$ docker build --add-host=docker:10.180.0.1 .*

*The --add-host flag is used to add a single host to IP mapping within a Docker container. This flag can be useful when you need to connect a service within a container to an external host.*

*An error Could not resolve host. This is the typical error we would see when trying to reach a domain that does not have a DNS entry.*

***Specifying target build stage (--target)***

*When building a Dockerfile with multiple build stages, --target can be used to specify an intermediate build stage by name as a final stage for the resulting image. Commands after the target stage will be skipped.*

*you can tell Docker to only build some of the stages by using the --target argument.*

*FROM debian AS build-env*

*...*

*FROM alpine AS production-env*

*...*

*$ docker build -t mybuildimage --target build-env .*

***Specifying external cache sources [--cache-from]***

*In addition to local build cache, the builder can reuse the cache generated from previous builds with the --cache-from flag pointing to an image in the registry.*

*To use an image as a cache source, cache metadata needs to be written into the image on creation. This can be done by setting --build-arg BUILDKIT\_INLINE\_CACHE=1 when building the image. After that, the built image can be used as a cache source for subsequent builds.*

*Upon importing the cache, the builder will only pull the JSON metadata from the registry and determine possible cache hits based on that information. If there is a cache hit, the matched layers are pulled into the local environment.*

*In addition to images, the cache can also be pulled from special cache manifests generated by buildx or the BuildKit CLI (buildctl). These manifests (when built with the type=registry and mode=max options) allow pulling layer data for intermediate stages in multi-stage builds.*

*The following example builds an image with inline-cache metadata and pushes it to a registry, then uses the image as a cache source on another machine:*

*$ docker build -t myname/myapp --build-arg BUILDKIT\_INLINE\_CACHE=1 .*

*$ docker push myname/myapp*

*After pushing the image, the image is used as cache source on another machine. BuildKit automatically pulls the image from the registry if needed.*

*# on another machine*

*$ docker build --cache-from myname/myapp .*

*Note: This feature requires the BuildKit backend. You can either enable BuildKit or use the buildx plugin. The previous builder has limited support for reusing cache from pre-pulled images.*

***Compressing the Build Context [--compress]***

*You can compress the build context to further improve build performance. Pass the --compress flag to docker build to apply gzip compression. The compression occurs before the context is sent to the Docker daemon.*

*$ docker build . -t my-image:latest --compress*

*This can improve performance in some scenarios. The compression adds its own overheads, though—your system now needs to compress the context, and the receiving Docker daemon has to uncompress it. Using compression could actually be slower than copying the original files, in some circumstances. Experiment with each of your images to assess whether you see an improvement.*

***Always remove intermediate containers [--force-rm=true|false]****Yes, Docker images are layered. When you build a new image, Docker does this for each instruction (RUN, COPY etc.) in your Dockerfile:*

*create a temporary container from the previous image layer (or the base FROM image for the first command;*

*run the Dockerfile instruction in the temporary "intermediate" container;*

*save the temporary container as a new image layer.*

*The final image layer is tagged with whatever you name the image - this will be clear if you run docker history raghavendar/hands-on:2.0, you'll see each layer and an abbreviation of the instruction that created it.*

*Your specific queries:*

*1) 532 is a temporary container created from image ID b17, which is your FROM image, ubuntu:14.04.*

*2) ea6 is the image layer created as the output of the instruction, i.e. from saving intermediate container 532*

### DockerFile Examples

<https://github.com/ricardoandre97/docker-en-resources/tree/master/docker-images>

### Docker Image History

***Docker image history***

*Show the history of an image and its layer.*

*$ docker image history [OPTIONS] IMAGE*

**Options**

* -H=true/false Print sizes and dates in human readable format
* --no-trunc Don't truncate output
* -q Only show image IDs
* --format Pretty-print images using a Go template

.ID Image ID

.CreatedSince Elapsed time since the image was created if --human=true, otherwise timestamp of when image was created

.CreatedAt Timestamp of when image was created

.CreatedBy Command that was used to create the image

.Size Image disk size

.Comment Comment for image

**Example**

#1 To see how the docker:latest image was built:

$ docker image history docker:latest

#2 Using --format  
   
$ docker history <image-id> --format "table{{.ID}}, {{.CreatedBy}}" --no-trunc

$ docker history --format "{{.ID}}: {{.CreatedSince}}" busybox

* f6e427c148a7: 4 weeks ago
* <missing>: 4 weeks ago

### Dangling Images vs Unused Images

**Dangling Images = Image without tags & Image not refernce with by any container**  
Dangling images are layers that have no relationship to any tagged images. They no longer serve a purpose and consume disk space.

An unused image is an image that has not been assigned or used in a container.

**List Dangling images**

$ docker images --filter "dangling=true"

**Delete Dangling Imgaes**

$ docker rmi $(docker images -q -f dangling=true)

**Delete all dangling as well as unused images**

$ docker image prune -a

**How Dangling image occurs ?**

$ docker build -t dangling\_image:latest .

Image is created with tag latest.

$ docker build -t dangling\_image:latest .  
  
If you built the image with same name and same tag then previous image will go to dangling state means no image name and no tag name. newly created image will have the name & tag

Always use different tag to avoid dangling

### Docker image tag

During build  
  
$ docker image build -t tagName:version .

**Tagging Existing image**

$ docker image tag SOURCE\_IMAGE[:TAG] TARGET\_IMAGE[:TAG]

### Docker Image Inspect

**Description**

Display detailed information on one or more images

**Syntax**

$ docker image inspect [OPTIONS] IMAGE [IMAGE...]

**Options**

--format , -f Format the output using the given Go template

**Example using grep**

$ docker image inspect nginx | grep Hostname

**Example using --format**

**Display Image ID**

$ dokcer image inspect nginx --format='{{.ID}}'

**Display Entire data**

$ docker image inspect nginx:latest

**Display the os of the image [Linux]**

$ docker inspect --format='{{.Os}}' nginx

**Parent & child**

$ docker inspect --format='{{.Config.ExposedPorts}}' nginx

"Config": {

"Hostname": "",

"Domainname": "",

"User": "",

"AttachStdin": false,

"AttachStdout": false,

"AttachStderr": false,

"ExposedPorts": {

"80/tcp": {}

},

**Interesting Image Inspect Results**

**ID** This is the unique identifier of the image.

**Parent** A link to the identifier of the parent image. It is very common for an image to have a defined parent.

**Container** A container identifier is interesting because this is meta-data for an image not a container. This container identifier is a temporary container created when the image was built. Docker will create a container during the image construction process, and this identifier is stored in the image data.

ContainerConfig This data again is referring to the temporary container created when the Docker build command was executed.

DockerVersion The version of Docker used to create the image is stored in this value. It could be very useful as the Docker ecosystem continues to advance.

Virtual Size The size of the image reported in bytes.

### Docker Image push

**Syntax**

$ docker image push [OPTIONS] NAME[:TAG]

**Options**

-a Push all tagged images in the repository

-q Suppress verbose output

**Push Docker image to Docker Hub**

$ docker login #1 login to hub.docker.com

$ docker image tag image-name username/image-name #2 Tag Docker Image

$ docker image tag hello-world fvenkat/hello-world

username refers to our dockerid or the username which is used to login.

image-name is the name of our docker image present on our system.

$ docker push username/image-name #3 Push Docker Image

$ docker push fvenkat/helloworld

$ docker logout #4 logout

### Docker Image pull

**Syntax**

$ docker image pull [OPTIONS] NAME[:TAG|@DIGEST]

**Options**

-a Download all tagged images in the repository

-q Suppress verbose output

**Pull an image from Docker Hub**

To download a particular image, or set of images (i.e., a repository), use docker pull. If no tag is provided, Docker Engine uses the :latest tag as a default. This command pulls the debian:latest image:

$ docker pull debian

**Pull from a different registry**

By default, docker pull pulls images from Docker Hub. It is also possible to manually specify the path of a registry to pull from. For example, if you have set up a local registry, you can specify its path to pull from it. A registry path is similar to a URL, but does not contain a protocol specifier (https://).

The following command pulls the testing/test-image image from a local registry listening on port 5000 (myregistry.local:5000):

$ docker pull myregistry.local:5000/testing/test-image

### Docker Image save

**Description**

Save one or more images to a tar archive (streamed to STDOUT by default)

To share or back up our image, we use the docker save command

**Syntax**

$ docker image save [OPTIONS] IMAGE [IMAGE...]

**Options**

-o Write to a file, instead of STDOUT

**Example**

$ docker image save ImageName > ImageName.tar

$ docker image save 15ca549393be > /tmp/phpmyadmin\_image.tar

**Using -o**

$ docker save -o <path for generated tar file> <image name>

$ docker save -o ubuntu.tar myjenk:latest jenkins/jenkins:lts

$ docker save -o c:/myfile.tar centos:16

If you want to export all images at once, create one big tar file:

$ docker save $(docker images -q) -o /path/to/save/mydockersimages.tar

IDS=$(docker images | awk '{if ($1 ~ /^(debian|centos)/) print $3}')

$ docker save $IDS -o /path/to/save/somedockersimages.tar

**Transferring a Docker image via SSH, bzipping the content on the fly:**



$ docker save <image> | bzip2 | pv | \

ssh user@host 'bunzip2 | docker load' pv in the middle of the pipe to see how the transfer is going

$ docker save app:1.0 | gzip | DOCKER\_HOST=ssh://user@remotehost docker load

**Save an image to a tar.gz file using gzip**

$ docker save myimage:latest | gzip > myimage\_latest.tar.gz

### Docker Image Load

**Description**

Load an image from a tar archive or STDIN

**Syntax**

$ docker image load [OPTIONS]

i Read from tar archive file, instead of STDIN

-q Suppress the load output

**Example**

$ docker load < busybox.tar.gz Loaded image: busybox:latest

$ docker load --input fedora.tar # Loaded image: fedora:rawhide Loaded image: fedora:20

$ docker load -i xxx.tar



### Docker Image Import

Description

Import the contents from a tarball to create a filesystem image

**Syntax**

$ docker image import [OPTIONS] file|URL|- [REPOSITORY[:TAG]]

-c Apply Dockerfile instruction to the created image

-m Set commit message for imported image

Docker import is a Docker command to create a Docker image by importing the content from an archive or tarball which is created by exporting a container. We can specify URL or ‘-‘ to import data or content from the archive. The URL can point to a location where archive file is present and ‘-‘ (dash) is used to import data directly from the STDIN i.e. standard input. We can apply Dockerfile instructions while creating the image using Docker import however there are limited Dockerfile instructions that are supported over here.

**How to Import Docker Image?**

In order to import a Docker image, first, we must have an exported archive file of a container. So when we export any container, it actually exports containers as a regular Linux file system in an archive file; then we have to import this archive file as a Docker image and when we run any container using this new imported Docker image then it works in the same way as it was working in the old container. This actually saves our time to build the Docker image from scratch if there is any changes are required. Docker import only supports these Dockerfile instructions: CMD, ENTRYPOINT, ENV, EXPOSE, ONBUILD, USER, VOLUME, WORKDIR

**Example #1 – Exporting a Container**

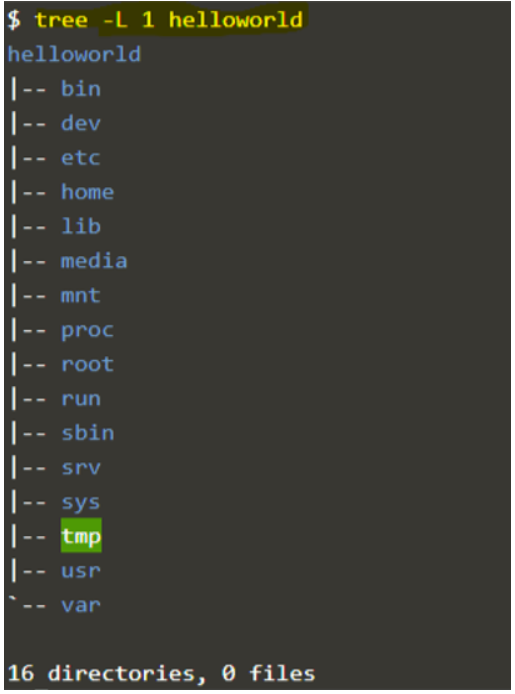
* $ docker build -t my-image:v2 .
* $ docker run my-image:v2
* $ docker ps –a

**Export the container in a tar file named helloworld.tar using the below command:**

* docker export <container\_ID> > <file\_name>
* docker export <container\_name> > <file\_name>
* $ docker export 3673f8996e1a > helloworld.tar

**Example #2 – Investigating the Exported Archive File  
And if we check the folder structure of helloworld, it looks like a regular Linux filesystem.**

* mkdir helloworld
* tar -xf helloworld.tar -C helloworld
* tree –L 1 helloworld



**Importing Images**

While save and load are easy to understand, both accepting and resulting in an image, the relationship between import and export is a little harder to grok.

There’s no way to “import a container” (which wouldn’t make sense, as it’s a running environment). As we saw above, export gives us a file system. import takes this file system and imports it as an image, which can run as-is or serve as a layer for other images.

To import an exported container as an image, we use the docker import command. The documentation describes import as follows:

* $ docker import calc-container.tar calcfs:latest
* $ docker image ls

**Import from a remote location**

$ docker image import http://example.com/exampleimage.tgz example/imagerepo

**Import from a local file**

# cat exampleimage.tgz | docker image import - example/imagelocal

**Import with a commit message.**

# cat exampleimage.tgz | docker image import --message "New image imported from tarball" - exampleimagelocal:new

**Import from a local file and tag**

# cat exampleimageV2.tgz | docker image import - example/imagelocal:V-2.0

**Import from a local directory**

# tar -c . | docker image import - exampleimagedir

### Docker Image save|load|import|export

**save works with Docker images.** It saves everything needed to build a container from scratch. Use this command if you want to share an image with others.

**load works with Docker images.** Use this command if you want to run an image exported with save. Unlike pull, which requires connecting to a Docker registry, load can import from anywhere (e.g. a file system, URLs).

**export works with Docker containers**, and it exports a snapshot of the container’s file system. Use this command if you want to share or back up the result of building an image.

**import works with the file system of an exported container**, and it imports it as a Docker image. Use this command if you have an exported file system you want to explore or use as a layer for a new image.

When I was new to Docker, this caused me some confusion. Had I RTFM’d a little more, digging into the subcommands, I might have noticed that export only applies to containers, while import, load, and save apply to images 🤦‍♂️:

* $ docker container --help | grep -E "(export|import|load|save)"
* export Export a container\'s filesystem as a tar archive
* $ docker image --help | grep -E "(export|import|load|save)"
* import Import the contents from a tarball to create a filesystem image
* load Load an image from a tar archive or STDIN
* save Save one or more images to a tar archive (streamed to STDOUT by default)
* The result of all this learning is that PSPDFKit for Web is now available on both Docker Hub and npm, meaning first-class PDF support for your web apps is only a docker pull or npm install away 🎉.

### Flatenning Docker Image

**Difference between save and export**

As I described in my last post (http://tuhrig.de/difference-between-save-and-export-in-docker), there are two ways to persist a Docker images or container:

A Docker image can be saved to a tarball and loaded back again. This will preserve the history of the image.

# save the image to a tarball

docker save <IMAGE NAME> > /home/save.tar

# load it back

docker load < /home/save.tar

A Docker container can be exported to a tarball and imported back again. This will not preserve the history of the container.

# export the container to a tarball

docker export <CONTAINER ID> > /home/export.tar

# import it back

cat /home/export.tar | docker import - some-name:latest

**No history**

We can use this mechanism to flatten and shrink a Docker container. If we save an image to the disk, its whole history will be preserved, but if we export a container, its history gets lost and the resulting tarball will be much smaller.

We can see the history of a image be running docker tag <LAYER ID> <IMAGE NAMEgt;:

vagrant@ubuntu-13:~$ sudo docker images --tree

├─f502877df6a1 Virtual Size: 2.489 MB Tags: busybox-1-export:latest

└─511136ea3c5a Virtual Size: 0 B

└─bf747efa0e2f Virtual Size: 0 B

└─48e5f45168b9 Virtual Size: 2.489 MB

└─769b9341d937 Virtual Size: 2.489 MB

└─227516d93162 Virtual Size: 2.489 MB Tags: busybox-1:latest

So if we export a container (either an already running one or just start a new one from an image) it will lose its history and all previous layers. This will make it impossible to make a rollback to a certain layer, but it will also shrink the image. My >7 GB image is now >3 GB large, which saves more than 50% of disk space.

**Flatten a Docker container**

So it is only possible to “flatten” a Docker container, not an image. So we need to start a container from an image first. Then we can export and import the container in one line:

$ docker export <CONTAINER ID> | docker import - some-image-name:latest

**What else?**

You can use some common Linux tricks to shrink Docker images. One simple trick is to clear the cache of the package manager. So depending on which base image you use you can do something like this (for an Ubuntu/Debian system, for more see here):

# clean apt cache

apt-get clean

**One**: Think Carefully About Your Application’s Needs

Do you genuinely think about the application that you’re deploying? Do you fully consider the language, framework, extensions, tools, and third-party packages which it needs? Or do you install everything, such as development tools and Linux headers, before installing your application?

Bear in mind that everything that you install increases an image’s size, as you’d expect. While it may be easier to work this way (some may argue that it is quick and efficient) is it useful — over the long term?

Alternatively, if you stop and consider your application’s needs in-depth, you may be surprised as to just how little it may need. Sure, you don’t get the thrill of jumping in straight-away and building something. Moreover, you might even see planning as somewhat dull and tedious.

However, remember the old saying: “Proper Planning Prevents Poor Performance”. It’s an adage well worth remembering!

**Two: Use a Small Base Image**

What base image are you using? For quite some time, the default Docker image used Ubuntu, which made sense for several reasons. Let’s consider two of the biggest.

Firstly, Ubuntu’s based on Debian, which uses the Apt package manager. Apt is one of the most well thought out and most feature-rich package managers available. What’s more, its interface is about as intuitive as you can get.

Secondly, Ubuntu is one of the most popular Linux distributions, so its conventions are well understood by a large percentage of developers and systems administrators.

That said, when using Ubuntu as a foundation, your base image won’t be particularly small. Weighing in at around 188 MB, Ubuntu comes packed with most of the tools that you’re likely to need, even if you don’t need them. It’s an excellent example of size being sacrificed for ease of use.

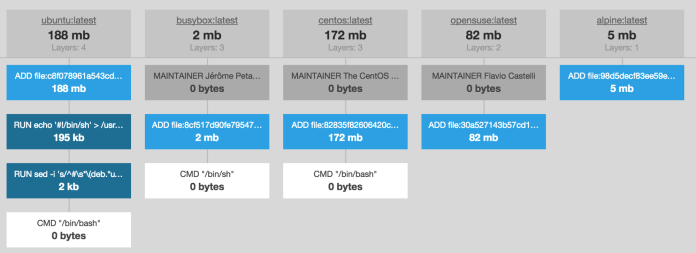
There’s nothing, necessarily, wrong with that, but if you want as small an image as possible, you’re going to have to use a different base image. Luckily, one is readily available and is the new default for Docker images; it’s called Alpine Linux.

Alpine Linux is:

A Small Linux Distribution Based On Musl And BusyBox, Primarily Designed For “Power Users Who Appreciate Security, Simplicity And Resource Efficiency.

As a result, an image based on Alpine Linux is typically around 5 MB in size. That’s a size saving of around 97%!

Compare that to Ubuntu’s size and you can see how making this one change alone is a significant saving. To make it easier, here’s an image size comparison, from Brian Christner:



**Three: Use as Few Layers As Possible**

How many layers compose your Docker image? Layers?, you may be asking. If you’re not familiar, every Docker image is composed of several layers, one for each command in a Dockerfile (and those in its base image).

* FROM php:7.0-apache
* COPY ./ /var/www/html/
* RUN chown -R www-data:www-data /var/www/html \
* && chown www-data:www-data /var/www
* RUN apt-get -y update
* RUN apt-get install -y libmcrypt-dev libzip-dev libpng12-dev libicu-dev libxml2-dev wget build-essential
* RUN apt-get install -y git vim
* RUN apt-get install -y npm nodejs-legacy

In the example Dockerfile above, the resulting Docker image will be composed of 7 layers, in addition to any layers in the base image. Building Docker images in this way allows Docker to be as efficient as possible.

If a change is made in one layer, then only that layer needs to be changed. Every other layer can be fetched from cache.

That way, build times are as quick as possible.

To know how many layers your image is composed of, and how large they are, run the docker history command, as in the following example:

docker history ze-php7.1.11-alpine

IMAGE CREATED CREATED BY SIZE COMMENT

5e0644e052e4 7 days ago /bin/sh -c #(nop) ENTRYPOINT [“/bootstrap… 0B

6791d155bd2e 7 days ago /bin/sh -c #(nop) EXPOSE 80/tcp 0B

6db0e80c9902 7 days ago /bin/sh -c #(nop) COPY dir:67f62ca0a16622f… 12MB

b53ae8739f99 7 days ago /bin/sh -c chmod +x /bootstrap/start.sh 10.1kB

9fe7fa43babf 7 days ago /bin/sh -c #(nop) ADD file:9fef33a371bdaef… 10.1kB

70949f1e3e70 7 days ago /bin/sh -c mkdir /app && mkdir /app/public… 0B

969f39e455b7 7 days ago /bin/sh -c mkdir /run/apache2 && sed -… 17.8kB

6cfc7b7d4a34 7 days ago /bin/sh -c apk — no-cache update && apk … 70.6MB

946e205f9ff5 7 days ago /bin/sh -c echo “http://dl-cdn.alpinelinux… 149B

7bb295030e89 7 days ago /bin/sh -c #(nop) LABEL maintainer=Paul S… 0B

72dfe9e82e16 3 weeks ago /bin/sh -c #(nop) CMD [“/bin/sh”] 0B

<missing> 3 weeks ago /bin/sh -c #(nop) ADD file:682bfba4bfda1ed… 4.15MB

There, you can see that it’s composed of twelve layers. For each layer you can see:

The layer’s unique hash.

When that layer was created.

The command that created it.

The layer’s size.

If you want to see an image’s combined size, use the docker imagescommand, as in this example:

docker images ze-php7.1.11-alpine

REPOSITORY TAG IMAGE ID CREATED SIZE

ze-php7.1.11-alpine latest 5e0644e052e4 7 days ago 86.8MB

Then there’s the next aspect of how layers work. Referring again to the Dockerfile example above, notice the last three RUN commands. I’ve deliberately split up three calls to apt-get install to highlight this point.

It first installs the required development packages, which subsequent packages depend on. It then installs several system binaries. Finally, it installs the application’s software packages.

Working this way might make sense during development, as the Dockerfile’s easy to read and is ordered logically. However, while it might be logical, it will result in a larger image than if those steps were combined.

Moreover, gets even worse. Do you install packages in one RUN command only to remove them after a following RUN command? You may wonder what the problem is, as the packages are ultimately deleted. Well, no they’re not.

They’re deleted in a later layer — but they still exist in the one where they were installed! You see, anything that exists in one layer isn’t removed in a later one, it’s only covered over, so that following layers can’t see it, and that it appears as though it doesn’t exist.

It gets back to the point about caching that I spoke of earlier.

That’s why:

It’s better not to install a package in the first place if you don’t need it.

Don’t install a package in the first place if you can avoid it.

How we could do this? Well, here are four ways:

Develop the Dockerfile in logically separated blocks, but compact it in the final version.

Don’t create files if you don’t have to — use streams and pipes as much as possible.

Remove unnecessary files, such as cache files, or don’t use the cache in the first place.

If you have to install files, use the smallest one possible.

Let’s see some of those in practice:

# Add basics first

RUN apk — no-cache update \

&& apk upgrade \

&& apk — no-cache add apache2 ca-certificates openntpd php7 php7-apache2 php7-sqlite3 php7-tokenizer \

&& cp /usr/bin/php7 /usr/bin/php \

&& rm -f /var/log/apache/\*

Here, you can see that I’ve combined the cache update and upgrade commands with the add command, by doing so, it’s not downloaded any extra files to the filesystem. Then, those commands were combined with the cp and rm command, which took care of some housekeeping and cleared out any unnecessary files.

Do some experimenting and see how you can compact your Dockerfile configurations.

**Four: Use .dockerignore files**

Referring back to the earlier Dockerfile example again, remember that it copied everything in its context into the images /var/www/html directory, in the second command? Are all those files necessary? Could some of them have been avoided?

If so, and it’s likely the case that some could have been, then do so by using [a .dockerignore file. These files, as their name implies, instruct Docker that specific files can be ignored.

If you’re familiar with .gitignore files, then you’ll know what I mean. If not, have a look at the example below.

vendor

.idea

data

mysql

Using this configuration Docker would not copy any files in the vendor, .idea, data, or mysql directories of my project, saving both time and space in the process. .dockerignore files are convenient, and I strongly encourage you to become a whiz with them.

**Five: Squash Docker Images**

* The last tip I want to offer is on squashing Docker images.
* The idea here is that after your image is created, you then flatten it as much as possible, using a tool such as docker-squash.
* Docker-Squash Is A Utility To Squash Multiple Docker Layers Into One To Create An Image With Fewer And Smaller Layers. It Retains Dockerfile Commands Such As PORT, ENV, Etc.. So That Squashed Images Work The Same As They Were Originally Built. In Addition, Deleted Files In Later Layers Are Actually Purged From The Image When Squashed.
* The idea is that your image is as small as possible so that when it’s transferred, it’s as quick as is possible. However, while the concept is quite appealing, I was never able to use it effectively.
* Perhaps it’s something to do with running it on macOS versus on a Linux distribution, but despite leaving it running for over an hour on a small image, there seemed no end in sight as to when the process would complete.
* Given that, while using docker-squash might drastically reduce my Docker image sizes, saving precious time pulling down and pushing up images, this time saving seems outweighed by the time to squash the image in the first place.
* However, your mileage may vary, and I’m entirely new to the tool. So if you’ve found success with it, or a related tool, please share your thoughts in the comments.

<https://blog.codacy.com/five-ways-to-slim-your-docker-images/>

### MultiStage Build

<https://docs.docker.com/develop/develop-images/multistage-build/>

### Building docker images from private git repositories using ssh login

<https://itnext.io/building-docker-images-from-private-git-repositories-using-ssh-login-433edf5a18f2>

<https://rolandsdev.blog/access-private-repos-in-docker/>

<https://blog.lelonek.me/private-dependencies-from-github-in-your-docker-container-92e3b8cbf677>

<https://vsupalov.com/build-docker-image-clone-private-repo-ssh-key/>