Size Matters

We've learned how to create images and we've even learned how to publish them. In this lesson, we will look at images in a bit more detail and discuss why the size of an image is important.

WE'LL COVER THE FOLLOWING

When you create an image, you want it to be as small as possible for several reasons:

- Reduce pull and push times
- Use a minimum amount of space in the Registry
- Use a minimum amount of space on the machines that will run the containers
- Use a minimum amount of space on the machine that creates the image

In order to reduce the size of an image, you need to understand that it is influenced by several factors:

- The files included in your image
- The base image size
- Image layers

Ideally, you want to reduce these factors. Let’s see how.

Files Included in Your Image

Include only necessary files in your image. That may sound obvious, but it’s

easy to include unwanted files.

First of all, avoid *COPY* instructions that are too broad. A typical example that should be avoided is:

```
COPY . .
```

Try to be as precise as possible. If necessary, split them into separate *COPY* instructions such as:

```
COPY ./Project/src/*.ts ./src
COPY ./Project/package.json .
```

Obviously, there are times when you will need to use *COPY* instructions that copy whole folders, as below:

```
COPY ./js/built /app/js
```

However, you may want to exclude files from that copy. You can use a *.dockerignore* file for that purpose. Simply add a *.dockerignore* file at the root of your build context that lists files and folders that should be excluded from the build like a *.gitignore* file.

Here is an example *.dockerignore* file:

```
# Ignore .git folder
.git
# Ignore Typescript files in any folder or subfolder
**/*.ts
```

Second, when using package managers like NPM, NuGet, apt, and so on, make sure you run them while building the image. It will avoid sending them as the context of the image, and it will allow the layer caching system to cache the result of running them. As long as the definition file doesn't change, Docker will reuse its cache.

Base Image Size

The base image you choose in your *FROM* instruction at the top of your

Dockerfile file is part of the image you build.

There are optimizations in which a machine will not pull the base image when pulling your image, as long as it already pulled that base image before. But oftentimes such optimizations cannot run, so it's better to reference small base images.

Many images you can use as base images have smaller variants. For instance, the *debian* image I used in the *Create Docker Images* chapter is quite big. Here is the definition I used:

Dockerfile

```
FROM debian:8

CMD ["echo", "Hello world"]
```

That definition results in an image that weights **127 MB**. The size can be checked using the *docker image ls* command.

If I head to the description page of that image, I can see that there is a smaller variant using the *8-slim* tag. Here is an optimized definition:

Dockerfile

```
FROM debian:8-slim

CMD ["echo", "Hello world"]
```

That revised definition yields an image that weights **79.3 MB**. We just saved **38%** of the original image size.

The same can be done using a *node:alpine* image instead of the default *node* image. For instance, the image definition I used for my *webserver* image results in a **109 MB** image:

Dockerfile

```
FROM nginx:1.15

COPY index.html /usr/share/nginx/html
```

If I switch the base image from *nginx:1.15* to *nginx:1.15-alpine*, the resulting image weights only **16.1 MB**. That's an **85%** saving over the original image size! Here's the optimized image definition:

Dockerfile

```
FROM nginx:1.15-alpine

COPY index.html /usr/share/nginx/html
```

Image Layers

When creating an image, Docker reads each instruction in order and the resulting partial image is kept separate; it is cached and labeled with a unique ID. Such caching is very effective because it is used at different moments of an image life:

- In a future build, Docker will use the cached part instead of recreating it as long as it is possible.
- When pushing a new version of the image to a Registry, the common part is not pushed.
- When pulling an image from a registry, the common part you already have is not pulled.

The caching mechanism can be summed up as follows: when building a new image, Docker will try its best to skip all instructions up to the first instruction that actually changes the resulting image. All prior instructions result in the cached layers being used. Let's see some examples.

Suppose I want to create an image that included PHP, so I create the following *Dockerfile* file:

Dockerfile file:

```
FROM debian:8

COPY . .

RUN apt-get update && apt-get upgrade && apt-get dist-upgrade -y
RUN apt-get install -y php5
```

When I build it using the *docker build* command, it takes quite some time because the two *RUN* steps need to download files and manipulate them. That's normal and expected. Here's the truncated output:

```
Step 1/4 : FROM debian:8
 ---> ec0727c65ed3
Step 2/4 : COPY . .
 ---> cb8153c1f09a
Step 3/4 : RUN apt-get update && apt-get upgrade && apt-get dist-upgrade -
y
 ---> Running in eff335d4aeba
Get:1 http://security.debian.org jessie/updates InRelease [44.9 kB][...]
[...]
Removing intermediate container eff335d4aeba
 ---> f4a652ba0849
Step 4/4 : RUN apt-get install -y php5
 ---> Running in d43a7a64606d
Reading package lists...
Building dependency tree...
Reading state information...
The following extra packages will be installed:
  apache2 apache2-bin apache2-data apache2-utils file ifupdown isc-dhcp-cl
ient
  isc-dhcp-common krb5-locales libalgorithm-c3-perl libapache2-mod-php5
[...]
Need to get 23.2 MB of archives.
After this operation, 88.3 MB of additional disk space will be used.
Get:1 http://deb.debian.org/debian/ jessie/main libgdbm3 amd64 1.8.3-13.1
[30.0 kB]
Get:2 http://deb.debian.org/debian/ jessie/main libjson-c2 amd64 0.11-4 [2
4.8 kB]
[...]
Preparing to unpack .../libkrb5support0_1.12.1+dfsg-19+deb8u5_amd64.deb
...
Unpacking libkrb5support0:amd64 (1.12.1+dfsg-19+deb8u5) ...
Selecting previously unselected package libk5crypto3:amd64.
[...]
```

```
[...]
Removing intermediate container d43a7a64606d

 ---> 3499256d8527
Successfully built 3499256d8527
```

Since this is the first build, each instruction is processed. However, each of the 4 steps has been cached.

If I run the same *docker build* command without changing anything, I can benefit from the cached layers. The whole build process takes less than a second and here's the output:

```
Step 1/4 : FROM debian:8
 ---> ec0727c65ed3
Step 2/4 : COPY . .
 ---> Using cache
 ---> cb8153c1f09a
Step 3/4 : RUN apt-get update && apt-get upgrade && apt-get dist-upgrade -
y
 ---> Using cache
 ---> f4a652ba0849
Step 4/4 : RUN apt-get install -y php5
 ---> Using cache
 ---> 3499256d8527
Successfully built 3499256d8527
```

Note that each layer ID matches the ID of a layer that has been previously cached. For instance, the *apt-get update* command created the *5fa6ecc854bb* layer, and running the same command gets us the same layer. If I had pushed the previous image to a Registry and pushed that one, almost nothing would actually be pushed.

That's efficient but unrealistic. In real life I would rebuild my image because some of the files used in the *COPY* step changed. Let's change a file in the current directory and run the exact same *docker build* command. We can see that the cache isn't used, and the build process takes several seconds again because it needs to download, unpack, and install all the packages:

```
Step 1/4 : FROM debian:8
 ---> ec0727c65ed3
Step 2/4 : COPY . .
 ---> edec0c4e4746
```

```
Step 3/4 : RUN apt-get update && apt-get upgrade && apt-get dist-upgrade -
y

 ---> Running in ddc4ad55ab7d
Ign http://deb.debian.org jessie InRelease
Get:1 http://deb.debian.org jessie-updates InRelease [145 kB]
Get:2 http://security.debian.org jessie/updates InRelease [44.9 kB]
[...]
Removing intermediate container ddc4ad55ab7d
 ---> 6d484f51c8f8
Step 4/4 : RUN apt-get install -y php5
 ---> Running in b6ba3afb3cc9
Reading package lists...
Need to get 23.2 MB of archives.
After this operation, 88.3 MB of additional disk space will be used.
[...]
Removing intermediate container b6ba3afb3cc9
 ---> 48c3c3bf2fad
Successfully built 48c3c3bf2fad
```

Why wasn't any cache used this time? Simply because the *COPY* instruction was impacted by us changing a file. If we push that image to a Registry, the whole image will be pushed since no cache was used.

How can we improve this? Simply by changing the order of the instructions in the *Dockerfile* file. Indeed, the files copied by the *COPY* instruction are more likely to change than the packages needed to install PHP5. We can first install PHP and then copy the files. Here's the improved *Dockerfile* file:

```
FROM debian:8

RUN apt-get update && apt-get upgrade && apt-get dist-upgrade -y
RUN apt-get install -y php5

COPY . .
```

Note that the *COPY* instruction has been placed last. That's the only change.

If I run the *docker build* instruction, then change some files in the folder (as we did earlier), and run the *docker build* command again, here's what we get. On the first *docker build*, the whole build is done, taking up several seconds:

```
Step 1/4 : FROM debian:8
 ---> ec0727c65ed3
Step 2/4 : RUN apt-get update && apt-get upgrade && apt-get dist-upgrade -
```

```
y
[...]
 ---> 219277a22ec2
Step 3/4 : RUN apt-get install -y php5
[...]
 ---> e493f018171f
Step 4/4 : COPY . .
[...]
 ---> 5d1f58a397c7
Successfully built 5d1f58a397c7
```

Now, if I change a file and run the *docker build* command again, the whole process only takes a few seconds because cached layers are used for everything except the *COPY* instruction:

```
Step 1/4 : FROM debian:8
 ---> ec0727c65ed3
Step 2/4 : RUN apt-get update && apt-get upgrade && apt-get dist-upgrade -
y
 ---> Using cache
 ---> 219277a22ec2
Step 3/4 : RUN apt-get install -y php5
 ---> Using cache
 ---> e493f018171f
Step 4/4 : COPY . .
 ---> 08c87dcf7267
Successfully built 08c87dcf7267
```

Note how the ID of the layers resulting from steps **1** to **3** are the same. Only step **4** produces a different layer.

As you can see, we had the exact same instructions in both cases. The only change was the order of those instructions, which yielded a fantastic boost. In order to benefit from caching, do the steps in the *Dockerfile* that are likely to change, or have their input change, as late as possible.

Let's wrap up this chapter with a quiz to test what you have learned so far.