Docker Layers Explained

by Gunter Rotsaert MVB · Mar. 18, 19 · Cloud Zone · Tutorial

When you pull a Docker image, you will notice that it is pulled as different layers. Also, when you create your own Docker image, several layers are created. In this post we will try to get a better understanding of Docker layers.

1. What Is a Docker Layer?

A Docker image consists of several layers. Each layer corresponds to certain instructions in your `Dockerfile`. The following instructions create a layer: `RUN`, `COPY`, `ADD`. The other instructions will create intermediate layers and do not influence the size of your image. Let's take a look at an example. We will use a Spring Boot MVC application which we have created beforehand and where the Maven build creates our Docker image. The sources are available at GitHub.

We use the `feature/dockerbenchsecurity` branch, which is a more secure version of the `master` branch. Here is the Dockerfile:

```
1  FROM openjdk:10-jdk
2  VOLUME /tmp
3
4  RUN useradd -d /home/appuser -m -s /bin/bash appuser
5  USER appuser
6
7  HEALTHCHECK --interval=5m --timeout=3s CMD curl -f http://localhost:8080/actuator/health/ ||
8
9  ARG JAR_FILE
10 COPY ${JAR_FILE} app.jar
11 ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

We build the application with `mvn clean install` which will also create the Docker image. We don't list the pulling of all layers of the `openjdk:10-jdk` image for brevity.

```
Image will be built as mydeveloperplanet/mykubernetesplanet:0.0.3-SNAPSHOT

Step 1/8 : FROM openjdk:10-jdk

Pulling from library/openjdk
Image 16e82e17faef: Pulling fs layer
...
Image a9448aba0bc3: Pull complete
Digest: sha256:9f17c917630d5e95667840029487b6561b752f1be6a3c4a90c4716907c1aad65
Status: Downloaded newer image for openjdk:10-jdk
 ---> b11e88dd885d
Step 2/8 : VOLUME /tmp

 ---> Running in 21329898c3a6
Removing intermediate container 21329898c3a6
 ---> b6f9ca000de6
Step 3/8 : RUN useradd -d /home/appuser -m -s /bin/bash appuser

 ---> Running in 82645047e6e7
Removing intermediate container 82645047e6e7
 ---> 04f6b2716819
Step 4/8 : USER appuser

 ---> Running in 697b663dadbb
Removing intermediate container 697b663dadbb
 ---> eaf6b8af5709
Step 5/8 : HEALTHCHECK --interval=5m --timeout=3s CMD curl -f http://localhost:8080/actuator/

 ---> Running in f420b9d060c5
Removing intermediate container f420b9d060c5
 ---> 77f95436a3ff
Step 6/8 : ARG JAR_FILE

 ---> Running in 60b9d25ad2ac
Removing intermediate container 60b9d25ad2ac
 ---> 135fa7df95ac
Step 7/8 : COPY ${JAR_FILE} app.jar

 ---> 63c18567012b
Step 8/8 : ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

```
 ---> Running in 79203446934a
Removing intermediate container 79203446934a
 ---> 8e2b049f9783
Successfully built 8e2b049f9783
Successfully tagged mydeveloperplanet/mykubernetesplanet:0.0.3-SNAPSHOT
```

What happens here? We notice that layers are created and most of them are removed (*removing intermediate container*). So why does it say *removing intermediate container* and not *removing intermediate layer*? That's because a build step is executed in an intermediate container. When the build step is finished executing, the intermediate container can be removed. Besides that, a layer is read-only. A layer contains the differences between the preceding layer and the current layer. On top of the layers, there is a writable layer (the current one) which is called the *container layer*. As mentioned before, only specific instructions will create a new layer. Let's take a look at our Docker images:

```
$ docker image ls
REPOSITORY                                TAG                 IMAGE ID            CREATED
mydeveloperplanet/mykubernetesplanet      0.0.3-SNAPSHOT      8e2b049f9783        About a minute ago
openjdk                                   10-jdk              b11e88dd885d        2 months ago
```

And take a look at the history of our *mykubernetesplanet* Docker image:

```
$ docker history 8e2b049f9783
IMAGE               CREATED             CREATED BY                                      SIZE
8e2b049f9783        About a minute ago  /bin/sh -c #(nop) ENTRYPOINT ["java" "-Djav…    0B
63c18567012b        About a minute ago  /bin/sh -c #(nop) COPY file:2a5b71774c60e0f6…   17.4MB
135fa7df95ac        About a minute ago  /bin/sh -c #(nop) ARG JAR_FILE                  0B
77f95436a3ff        2 minutes ago       /bin/sh -c #(nop) HEALTHCHECK &{["CMD-SHELL…    0B
eaf6b8af5709        2 minutes ago       /bin/sh -c #(nop) USER appuser                  0B
04f6b2716819        2 minutes ago       /bin/sh -c useradd -d /home/appuser -m -s /b…   399kB
b6f9ca000de6        2 minutes ago       /bin/sh -c #(nop) VOLUME [/tmp]                 0B
b11e88dd885d        2 months ago        /bin/sh -c #(nop) CMD ["jshell"]                0B
<missing>           2 months ago        /bin/sh -c set -ex; if [ ! -d /usr/share/m…     697MB
```

```
<missing>       2 months ago          /bin/sh -c #(nop) ENV JAVA_DEBIAN_VERSION=1…    0B
...
```

We notice here, that the intermediate containers do have a size of 0B, just what was expected. Only the `RUN` and `COPY` command from the `Dockerfile` contribute to the size of the Docker image. The layers of the `openjdk:10-jdk` image are also listed and are recognized by the *missing* keyword. This only means that those layers are built on a different system and are not available locally.

2. Recreate the Docker Image

What happens if we run our Maven build again without making any changes to the sources?

```
Image will be built as mydeveloperplanet/mykubernetesplanet:0.0.3-SNAPSHOT

Step 1/8 : FROM openjdk:10-jdk

Pulling from library/openjdk
Digest: sha256:9f17c917630d5e95667840029487b6561b752f1be6a3c4a90c4716907c1aad65
Status: Image is up to date for openjdk:10-jdk
---> b11e88dd885d
Step 2/8 : VOLUME /tmp

---> Using cache
---> b6f9ca000de6
...
Step 6/8 : ARG JAR_FILE

---> Using cache
---> 135fa7df95ac
Step 7/8 : COPY ${JAR_FILE} app.jar

---> 409f2fee0cde
Step 8/8 : ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]

---> Running in 75f07955bbc8
Removing intermediate container 75f07955bbc8
---> e5d7b72aad05
Successfully built e5d7b72aad05
Successfully tagged mydeveloperplanet/mykubernetesplanet:0.0.3-SNAPSHOT
```

We notice that the first layers are identical to our previous build. The layer ID's are the same. In the log, we

notice that the layers are taken from the cache. In step 7 a new layer is created with a new ID. We did create a new JAR file, Docker interprets this as a new file and therefore a new layer is created. In step 8 also a new layer is created, because it is build on top of the new layer.

Let's list the Docker images again:

```
$ docker image ls
REPOSITORY                                TAG                 IMAGE ID            CREATED             S
mydeveloperplanet/mykubernetesplanet      0.0.3-SNAPSHOT      e5d7b72aad05        13 seconds ago      1
<none>                                    <none>              8e2b049f9783        5 minutes ago       1
openjdk                                   10-jdk              b11e88dd885d        2 months ago        9
```

Our tag `0.0.3-SNAPSHOT` has received the *image ID* of our last build. The repository and tag of our old image ID are removed, which is indicated with the *none* keyword. This is called a *dangling* image. We will explain this in more detail at the end of this post.

When we take a look at the history of the newly-created image, we notice that the two top layers are new, just as in the build log:

```
$ docker history e5d7b72aad05
IMAGE               CREATED             CREATED BY                                      SIZE      CO
e5d7b72aad05        38 seconds ago      /bin/sh -c #(nop) ENTRYPOINT ["java" "-Djav…    0B
409f2fee0cde        42 seconds ago      /bin/sh -c #(nop) COPY file:4b04c6500d340c9e…   17.4MB
135fa7df95ac        6 minutes ago       /bin/sh -c #(nop) ARG JAR_FILE                  0B
...
```

When we make a source code change, the result is identical because also, in this case, a new JAR file is generated.

```
$ docker image ls
REPOSITORY                                TAG                 IMAGE ID            CREATED             S
mydeveloperplanet/mykubernetesplanet      0.0.3-SNAPSHOT      eced642d4f5c        30 seconds ago      1
<none>                                    <none>              e5d7b72aad05        3 minutes ago       1
<none>                                    <none>              8e2b049f9783        8 minutes ago       1
openjdk                                   10-jdk              b11e88dd885d        2 months ago        9
```

```
$ docker history eced642d4f5c
IMAGE          CREATED              CREATED BY                                      SIZE
eced642d4f5c   About a minute ago   /bin/sh -c #(nop) ENTRYPOINT ["java" "-Djav…    0B
44a9097b8bad   About a minute ago   /bin/sh -c #(nop) COPY file:1d5276778b53310e…   17.4MB
135fa7df95ac   9 minutes ago        /bin/sh -c #(nop) ARG JAR_FILE                  0B
...
```

3. What About Size?

Let's take a closer look at the latest output of the `docker image ls` command. We notice two dangling images, which are 1 GB in size. But what does this really mean for storage? First, we need to know where the data for our images are stored. Use the following command in order to retrieve the storage location:

```
$ docker image inspect eced642d4f5c
  ...
  "GraphDriver": {
    "Data": {
      "LowerDir": "/var/lib/docker/overlay2/655be8bea8e54c31ebb7e3adf05db227d194a49c1e2f9555
      "MergedDir": "/var/lib/docker/overlay2/205b55ee2f0e06394b6d1706733884541060988 7ccd18f5
      "UpperDir": "/var/lib/docker/overlay2/205b55ee2f0e06394b6d1706733884541060988 7ccd18f53
      "WorkDir": "/var/lib/docker/overlay2/205b55ee2f0e06394b6d1706733884541060988 7ccd18f53b
    },
    "Name": "overlay2"
  },
  ...
```

Our Docker images are stored at `/var/lib/docker/overlay2` . We can simply retrieve the size of the `overlay2` directory in order to have an idea of the allocated storage:

```
$ du -sh -m overlay2
1059 overlay2
```

The `openjdk:10-jdk` image is 987 MB. The JAR file in our image is 17.4 MB. The total size should be somewhere around 987 MB + 3 * 17.4 MB (two dangling images and one real image). This is about 1,040 MB. We can conclude that there is some sort of smart storage for the Docker images and that we cannot

simply add all the sizes of the Docker images in order to retrieve the real storage size. The difference is due to the existence of intermediate images. These can be revealed as follows:

```
$ docker images -a
REPOSITORY                                TAG                 IMAGE ID            CREATED             SIZE
mydeveloperplanet/mykubernetesplanet      0.0.3-SNAPSHOT      eced642d4f5c        7 days ago          1GB
<none>                                    <none>              44a9097b8bad        7 days ago          1GB
<none>                                    <none>              e5d7b72aad05        7 days ago          1GB
<none>                                    <none>              409f2fee0cde        7 days ago          1GB
<none>                                    <none>              8e2b049f9783        7 days ago          1GB
<none>                                    <none>              63c18567012b        7 days ago          1GB
<none>                                    <none>              135fa7df95ac        7 days ago          987MB
<none>                                    <none>              77f95436a3ff        7 days ago          987MB
<none>                                    <none>              eaf6b8af5709        7 days ago          987MB
<none>                                    <none>              04f6b2716819        7 days ago          987MB
<none>                                    <none>              b6f9ca000de6        7 days ago          987MB
openjdk                                   10-jdk              b11e88dd885d        2 months ago        987MB
```

4. Get Rid of Dangling Images

How do we get rid of these dangling images? We don't need them anymore and they only allocate storage. First, list the dangling images:

```
$ docker images -f dangling=true
REPOSITORY    TAG     IMAGE ID        CREATED        SIZE
<none>        <none>  e5d7b72aad05    7 days ago     1GB
<none>        <none>  8e2b049f9783    7 days ago     1GB
```

We can use the `docker rmi` command to remove the images:

```
$ docker rmi e5d7b72aad05
```

```
Deleted: sha256:e5d7b72aad054100d142d99467c218062a2ef3bc2a0994fb589f9fc7ff004afe
Deleted: sha256:409f2fee0cde9b5144f8e92887b61e49f3ccbd2b0e601f536941d3b9be32ff47
Deleted: sha256:2162a2af22ee26f7ac9bd95c39818312dc9714b8fbfbeb892ff827be15c7795b
```

Or you can use the `docker image prune` command to do so.

Now that we have removed the dangling images, let's take a look at the size of the `overlay2` directory:

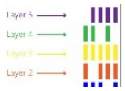
```
$ du -sh -m overlay2
1026 overlay2
```

We got rid of 33 MB. This seems like it's not so much, but when you often build Docker images, this can grow significantly over time.

5. Conclusion

In this post we tried to get a better understanding of Docker layers. We noticed that intermediate layers are created and that dangling images remain on our system if we don't clean them regularly. We also inspected the size of the Docker images on our system.

Like This Article? Read More From DZone



Understanding and Creating Effective Docker Images

Bleeding Edge on Docker

Docker image_build Module: Features for the Power User

Free DZone Refcard
Cloud Native Data Grids: Hazelcast IMDG With Kubernetes

Topics: DOCKER BUILD , DOCKER , LAYERS , DANGLING IMAGES , INTERMEDIATE LAYERS , CLOUD

Published at DZone with permission of Gunter Rotsaert , DZone MVB. See the original article here.
Opinions expressed by DZone contributors are their own.

Deployment Manifests That Work in