Topic 5: Creating and flattening Docker images

Dr. Daniel Yellin

Building image from images, flattening images

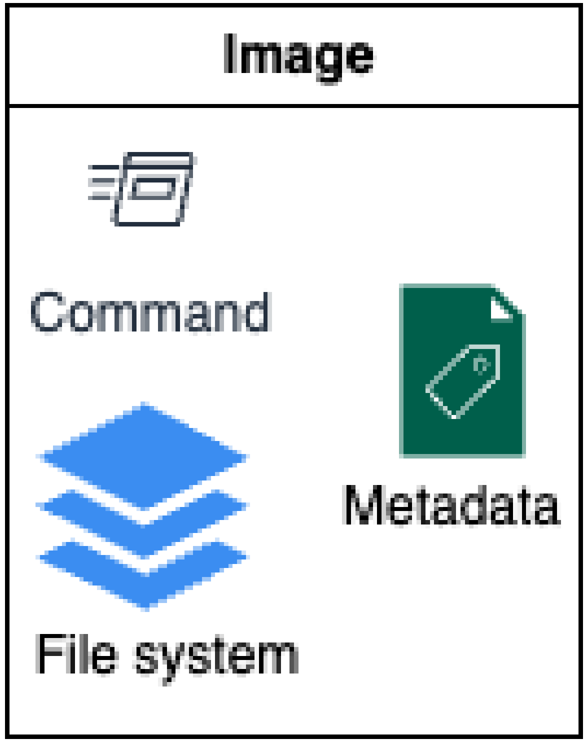
Overview

- Docker review
- Producing images from containers
- Making your container skinny

Docker review

Images

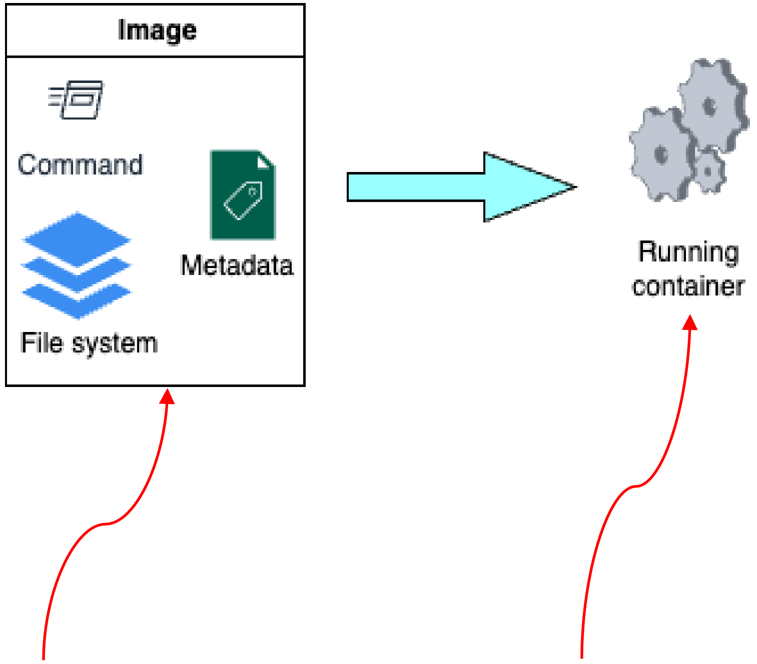
An *image* is a collection of files, metadata, and a command. It contains executable application source code as well as all the tools, libraries, and dependencies that the application code needs to run as a container.



Containers

A *container* is created from an image.

- It is an isolated running process (or processes) started using the command, with access only to the files in the image. The metadata defines properties of the process such as network ports accessible to the the process.
- You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.



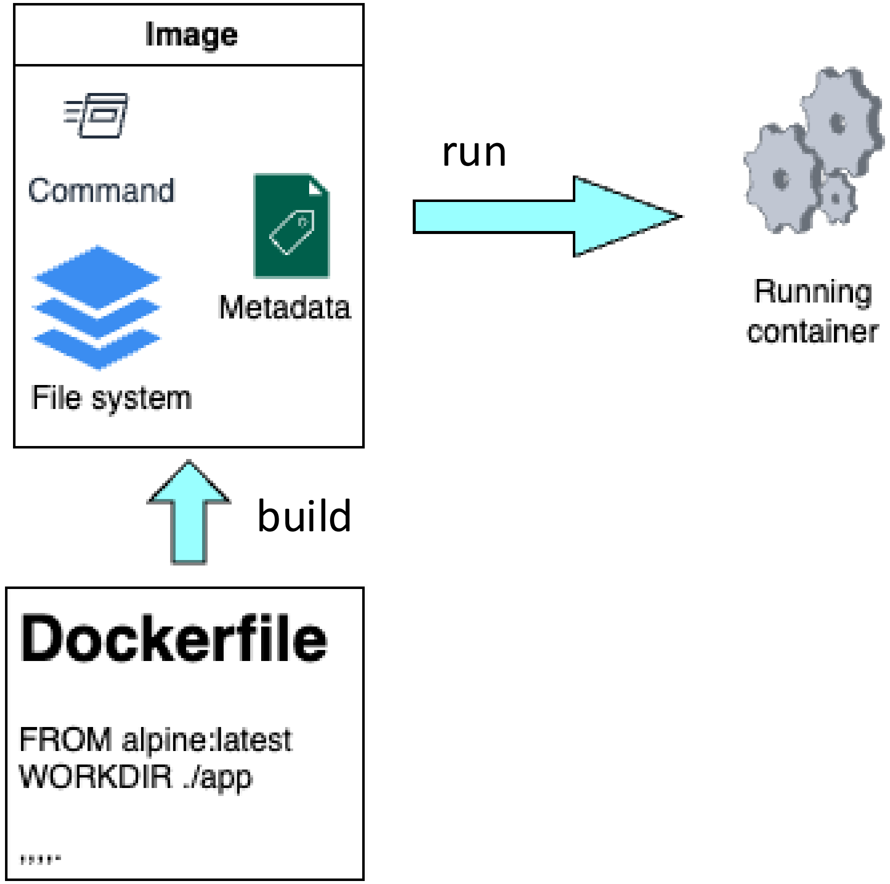
Read-only file

Live, executable content

Dockerfiles

A *Dockerfile* automates the process of Docker image creation. It’s a list of command-line interface (CLI) instructions that Docker Engine will run in order to assemble the image.

Often, an image is based upon another image. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create your desired image “on top of” the other image.



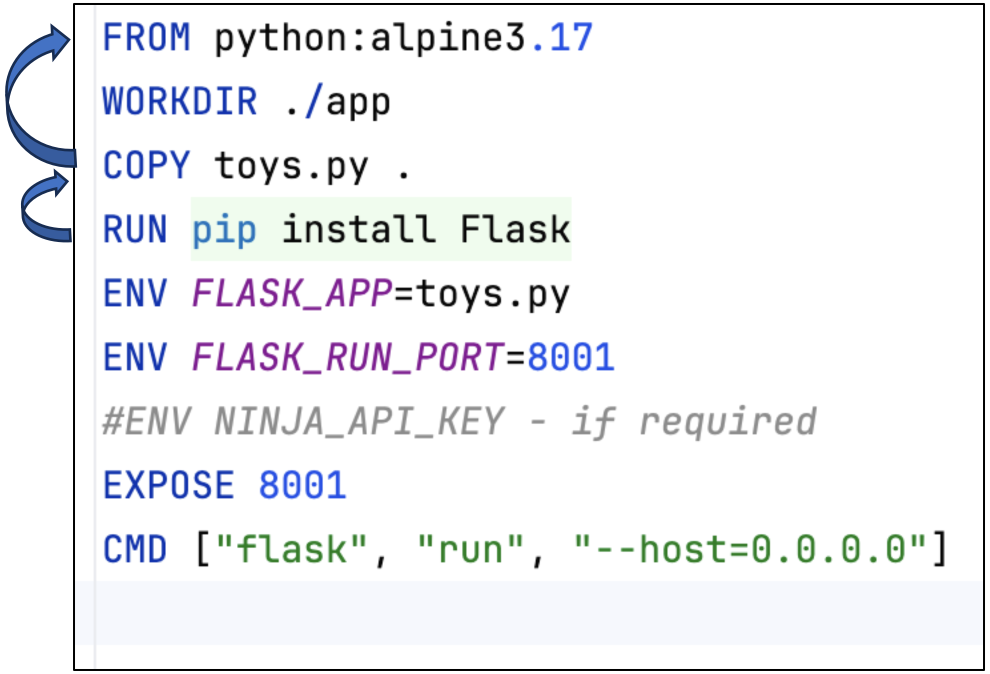
```
FROM alpine:latest
WORKDIR ./app

'''''
```

Docker image layers example

This Docker file results in several layers:

- A layer for the base image, python:alpine3.17 (this itself is comprised of layers).
- A layer with the toys.py files
- A layer with the Flask files installed

The other commands, such as “WORKDIR”, is metadata that does not modify the filesystem and is therefore not part of the image.



```
FROM python:alpine3.17
WORKDIR ./app
COPY toys.py .
RUN pip install Flask
ENV FLASK_APP=toys.py
ENV FLASK_RUN_PORT=8001
#ENV NINJA_API_KEY - if required
EXPOSE 8001
CMD ["flask", "run", "--host=0.0.0.0"]
```

Image layers

- **Read-only and Immutable:** Once a layer is created, it becomes read-only and cannot be modified.
- **Layered Construction:** Each instruction in a Dockerfile that modifies the filesystem (e.g., RUN, COPY, ADD) creates a new layer.

  These layers are stacked sequentially, with the base image forming the bottom layer and subsequent instructions adding layers on top. Any Dockerfile instruction that modifies the file system creates a new layer.

Layer caching and Copy-on-Write

- **Layer Caching:** Docker utilizes layer caching during the image build process. If a layer and its preceding layers remain unchanged from a previous build, Docker can reuse the existing cached layer, significantly speeding up subsequent builds.
    - This is particularly beneficial when only application code changes, as dependency installation layers can often be reused.
- **Copy-on-Write (CoW):** Docker employs a copy-on-write strategy for layers. When a container modifies a file that exists in a lower, read-only layer, a copy of that file is made into the container's writable layer, and the modification is applied to the copy. The original file in the lower layer remains untouched.

The “-it” flags in the Docker run command

docker run <image>

Creates and runs a new container from the given image.   You cannot directly interact with the container from your shell.

docker run -it <image>

Creates and runs a new container from the given image. The “-it” flags (interactive, terminal device) allows you to interact with the image via the shell.

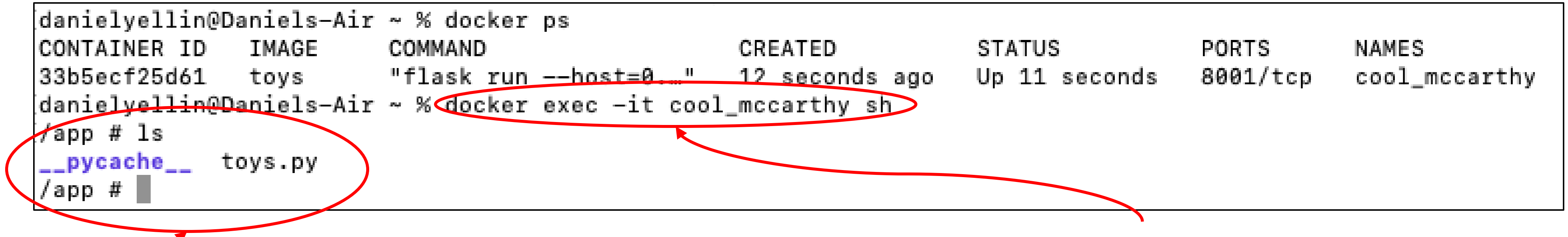
The Docker exec command

docker exec –it <container> <cmd>

Executes the given command in the specified running container.

Example: First I start up a toys container via the cmd **docker run toys**

At this point, it is running but I cannot "get inside" the container.



```
danielyellin@Daniels-Air ~ % docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS      NAMES
33b5ecf25d61   toys      "flask run --host=0.…"   12 seconds ago   Up 11 seconds   8001/tcp   cool_mccarthy
danielyellin@Daniels-Air ~ % docker exec -it cool_mccarthy sh
/app # ls
__pycache__  toys.py
/app # 
```

Runs the sh cmd inside the container

Now I have an interactive shell and I can issue cmds inside the container (like ls).

Producing images from containers

How to make an image without a Dockerfile

Container state

We learnt previously how to make a container from an image. In this section we will show how to make an image from a container!

Recall that:

- Everytime you create and run a container it has the *same filesystem* as the image it is created from.
- If a container makes changes to its file system and then the image is *stopped*, the changes to the file system are preserved when the container is *started* up again. However, when the image is removed (killed), the changes it made are lost.

Committing a container to yield a new image

1. Run a container (built on a base image)
2. Run some commands inside the container changing the file system
3. **Commit** the container

The result is a new image that contains the changes you made.

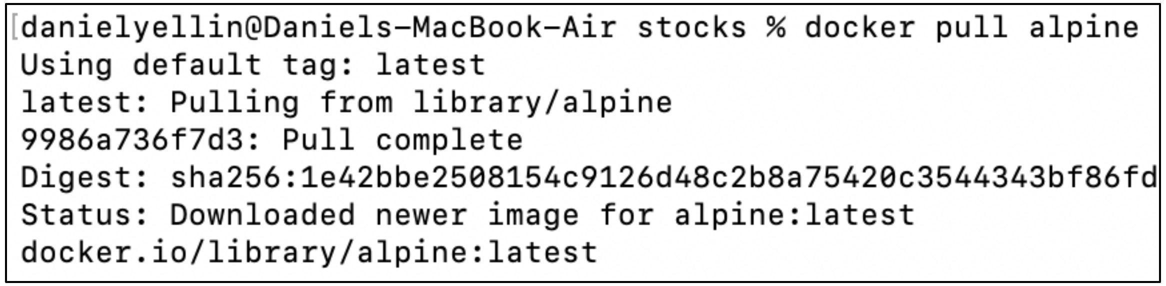
Let’s see an example

Making an image via container commit

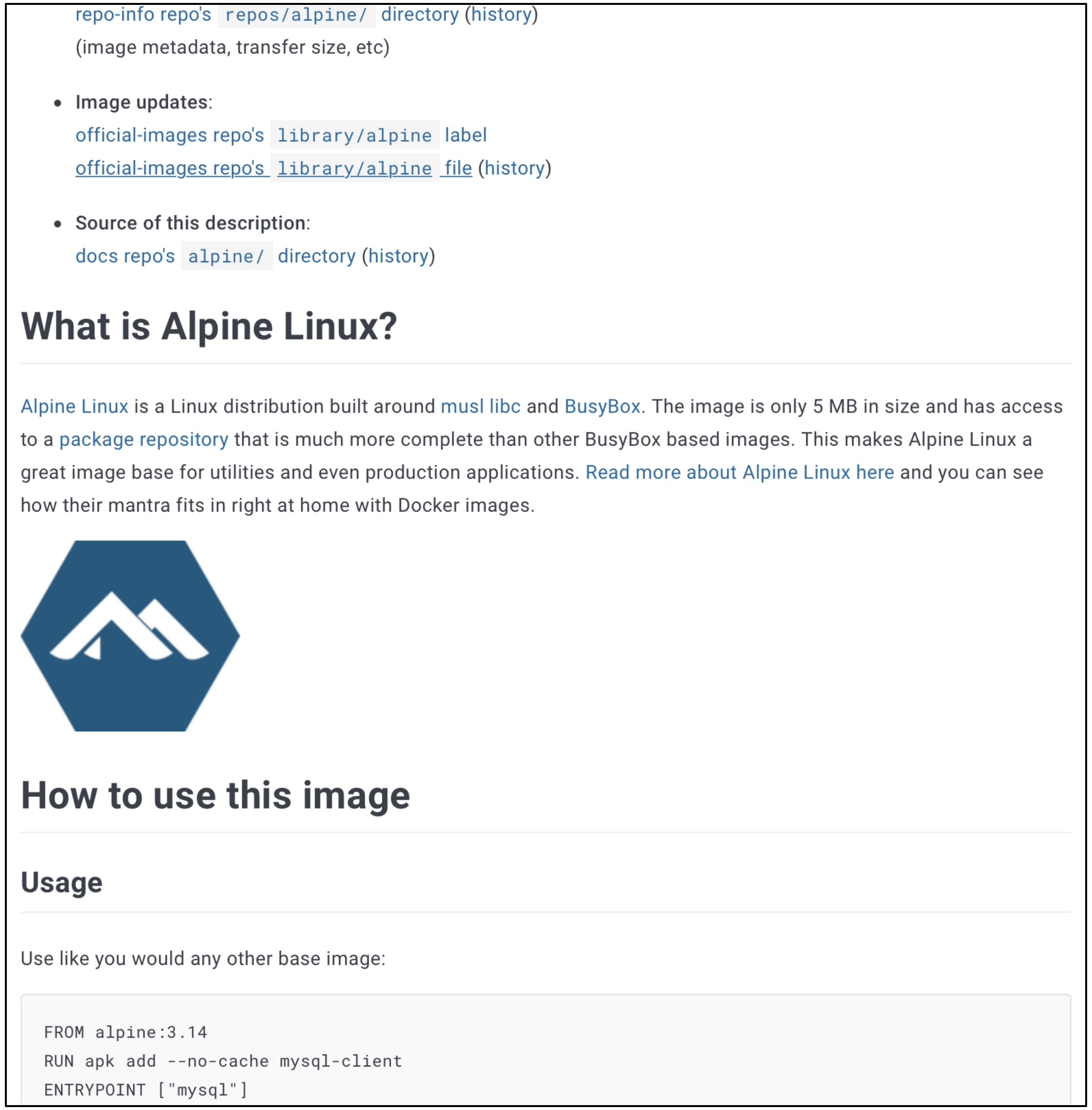
Start off by the command

```
1. docker pull
alpine:latest
```

This retrieves the latest alpine image from Docker Hub



```
[danielyellin@Daniels-MacBook-Air stocks % docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine
9986a736f7d3: Pull complete
Digest: sha256:1e42bbe2508154c9126d48c2b8a75420c3544343bf86fd
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
```



repo-info repo's `repos/alpine/` directory (history)
(image metadata, transfer size, etc)

- **Image updates**:
  official-images repo's `library/alpine` label
  official-images repo's `library/alpine` file (history)

- **Source of this description**:
  docs repo's `alpine/` directory (history)

What is Alpine Linux?

Alpine Linux is a Linux distribution built around musl libc and BusyBox. The image is only 5 MB in size and has access to a package repository that is much more complete than other BusyBox based images. This makes Alpine Linux a great image base for utilities and even production applications. Read more about Alpine Linux here and you can see how their mantra fits in right at home with Docker images.

How to use this image

Usage

Use like you would any other base image:

```
FROM alpine:3.14
RUN apk add --no-cache mysql-client
ENTRYPOINT ["mysql"]
```

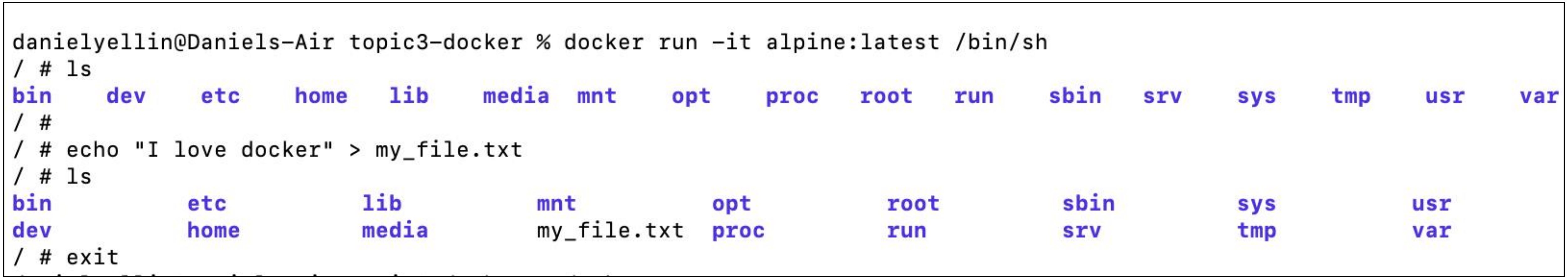
Making an image via container commit (cont)

```
2. docker run -it alpine:latest /bin/sh
```

This creates a container running the alpine image. The flag "-it" makes the container an interactive session connected to the terminal window. "/bin/sh" tells it to run the shell when it starts up. We issue the following cmd **inside** the container.

```
3. echo "I love docker" > my_file.txt
```

This creates a file named "my_file.txt" in the container. The file contains the string "I love Docker"



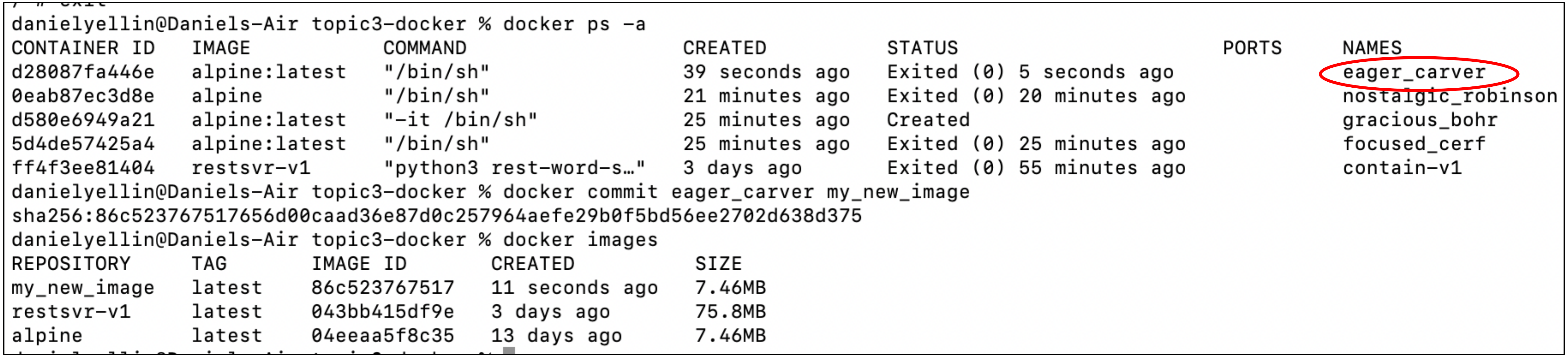
```
danielyellin@Daniels-Air topic3-docker % docker run -it alpine:latest /bin/sh
/ # ls
bin     dev     etc     home    lib     media   mnt     opt     proc    root    run     sbin    srv     sys     tmp     usr     var
/ #
/ # echo "I love docker" > my_file.txt
/ # ls
bin            etc            lib            mnt            opt            root           sbin           sys            usr
dev            home           media          my_file.txt    proc           run            srv            tmp            var
/ # exit
```

Making an image via container commit (cont)

4. Exit the container. This stops the container but it still exists.

5. `docker ps -a`

We see that since we did not specify a name for this container, Docker gave it the name “eager_carver”



```
/ # exit
danielyellin@Daniels-Air topic3-docker % docker ps -a
CONTAINER ID   IMAGE           COMMAND                  CREATED          STATUS                      PORTS     NAMES
d28087fa446e   alpine:latest   "/bin/sh"                39 seconds ago   Exited (0) 5 seconds ago              eager_carver
0eab87ec3d8e   alpine          "/bin/sh"                21 minutes ago   Exited (0) 20 minutes ago             nostalgic_robinson
d580e6949a21   alpine:latest   "-it /bin/sh"            25 minutes ago   Created                               gracious_bohr
5d4de57425a4   alpine:latest   "/bin/sh"                25 minutes ago   Exited (0) 25 minutes ago             focused_cerf
ff4f3ee81404   restsvr-v1      "python3 rest-word-s…"   3 days ago       Exited (0) 55 minutes ago             contain-v1
danielyellin@Daniels-Air topic3-docker % docker commit eager_carver my_new_image
sha256:86c523767517656d00caad36e87d0c257964aefe29b0f5bd56ee2702d638d375
danielyellin@Daniels-Air topic3-docker % docker images
REPOSITORY     TAG       IMAGE ID       CREATED          SIZE
my_new_image   latest    86c523767517   11 seconds ago   7.46MB
restsvr-v1     latest    043bb415df9e   3 days ago       75.8MB
alpine         latest    04eeaa5f8c35   13 days ago      7.46MB
danielyellin@Daniels-Air topic3-docker %
```

Making an image via container commit (cont)

```
6.docker commit eager_carver my_new_image
```

This cmd makes a new image from the container “eager_carver” and we give this image the name “my_new_image”. It stores the state of the file system (not the process).

```
7.docker run -it my_new_image /bin/sh
```

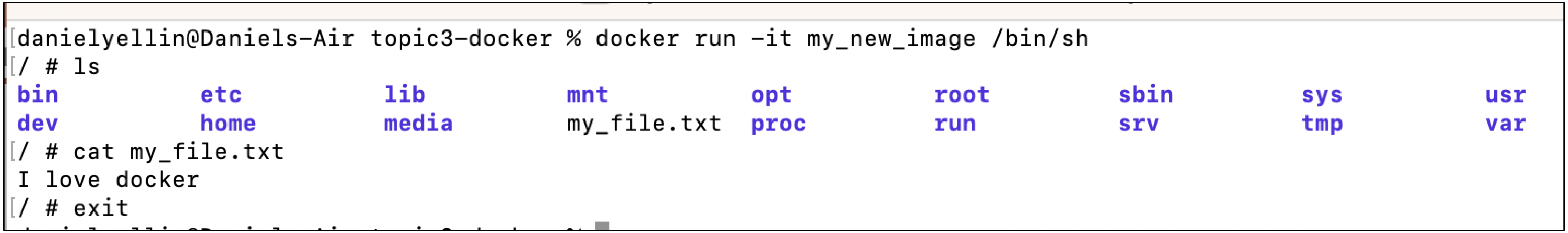
This starts up a container using the new image we just created - the alpine image plus a modified file system (a new file my_file.txt).

The `-it` flag runs the image iteractively and connects it to your terminal window

Making an image via container commit (cont)

8. Inside the running container issue the command ls
9. Then issue the command cat myfile.txt

We see that the file we created is in the image. We successfully modified an existing image and created a new one via the commit command.

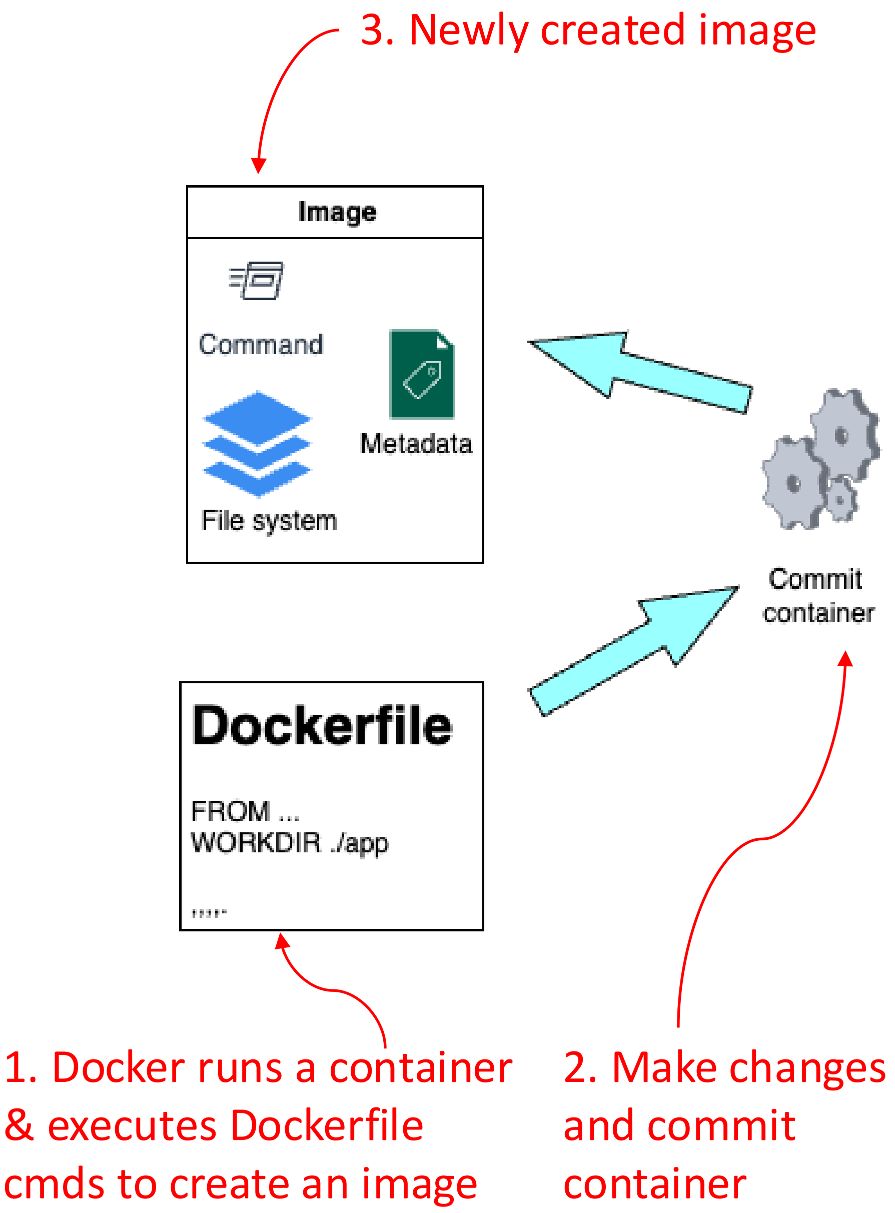


```
[danielyellin@Daniels-Air topic3-docker % docker run -it my_new_image /bin/sh
[/ # ls
bin          etc          lib          mnt          opt          root         sbin         sys          usr
dev          home         media        my_file.txt  proc         run          srv          tmp          var
[/ # cat my_file.txt
I love docker
[/ # exit
```

Images and containers

An *image* is created from a container

- By committing a container, you get an image.
- And that is how Dockerfile actually builds containers
  1. It builds a container with the base image
  2. It executes the next Dockerfile cmd, and commits to create a new image
  3. It uses that new image to build a new container
  4. It repeats steps 2-3 until there are no more commands



```
FROM ...
WORKDIR ./app
.....
```

3. Newly created image

1. Docker runs a container & executes Dockerfile cmds to create an image

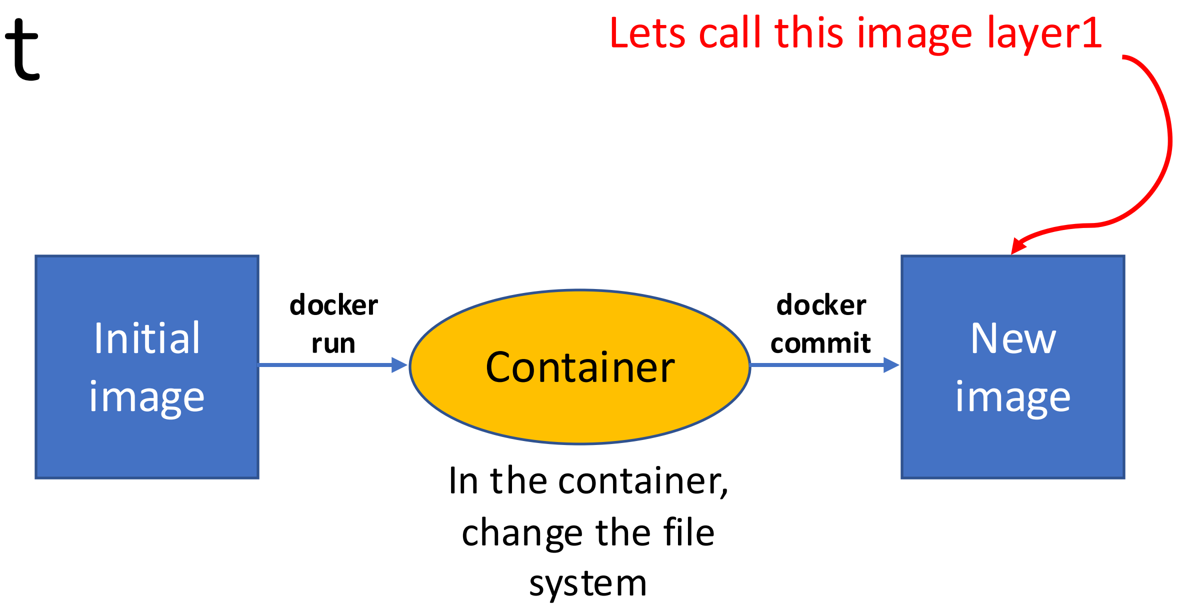
2. Make changes and commit container

Making your container skinny

Flattening a container - How to get rid of layers

Summary: Docker commit

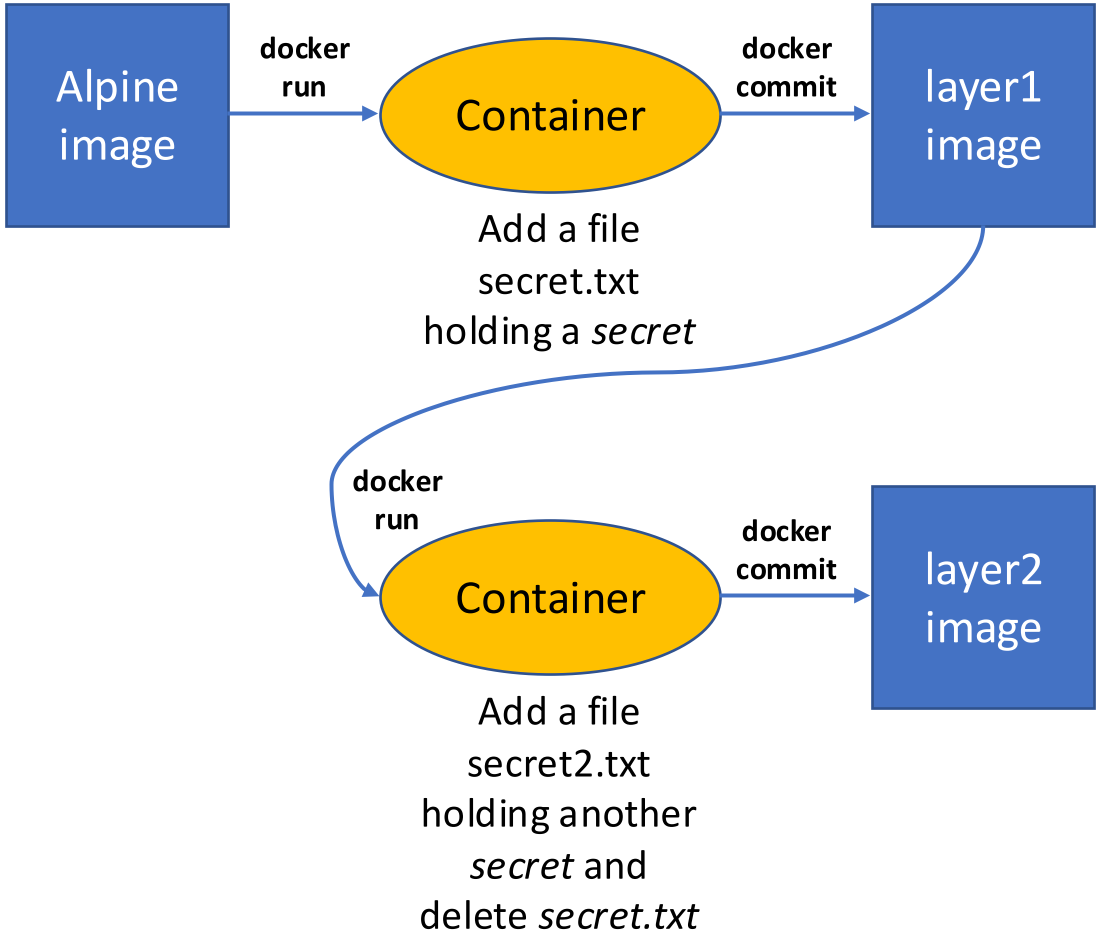
We just saw that you can create a new image by modifying the file system of a container and then *committing* that container. It becomes a new image.


Lets call this image layer1
Initial image
docker run
Container
In the container, change the file system
docker commit
New image

Example

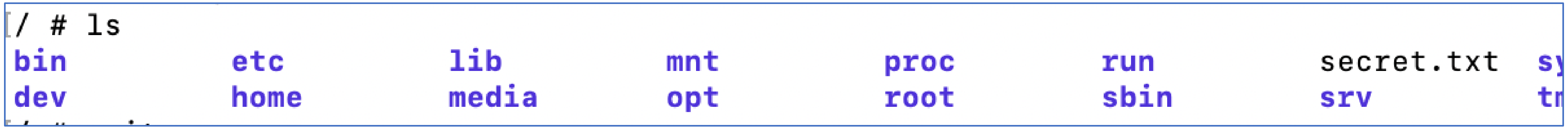
1. Initial image: alpine
2. Run the alpine image in a container and add a file secret.txt
3. Commit that container to an image called layer1
4. Run the layer1 image in a container, add a file secret2.txt and delete secret.txt
5. Commit that container to an image called layer2

What is the difference in the file systems in the two images?


Alpine image
docker run
Container
docker commit
layer1 image
Add a file secret.txt holding a *secret*
docker run
Container
docker commit
layer2 image
Add a file secret2.txt holding another *secret* and delete *secret.txt*

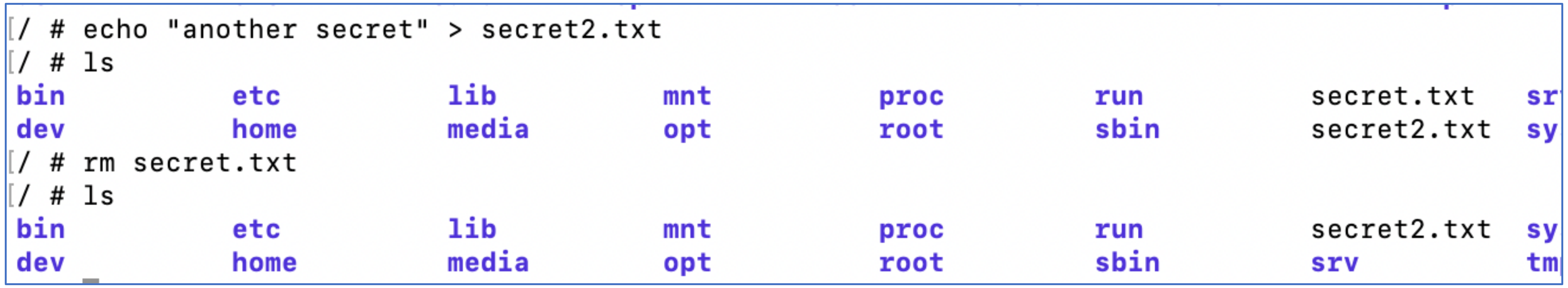
Example (cont)

- The file system in layer1 looks like this:



```
[/ # ls
bin          etc          lib          mnt          proc         run          secret.txt   sy
dev          home         media        opt          root         sbin         srv          tr
```

- The file system in layer2 looks like this:

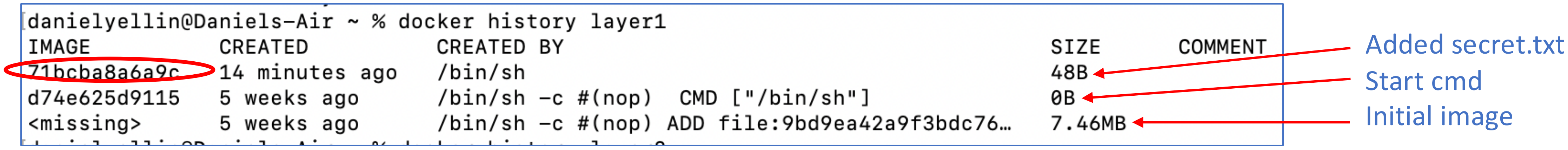


```
[/ # echo "another secret" > secret2.txt
[/ # ls
bin          etc          lib          mnt          proc         run          secret.txt   sr
dev          home         media        opt          root         sbin         secret2.txt  sy
[/ # rm secret.txt
[/ # ls
bin          etc          lib          mnt          proc         run          secret2.txt  sy
dev          home         media        opt          root         sbin         srv          tm
```

- *What do you think the different image layers will be in these images?*

Example (cont)

- docker history layer1



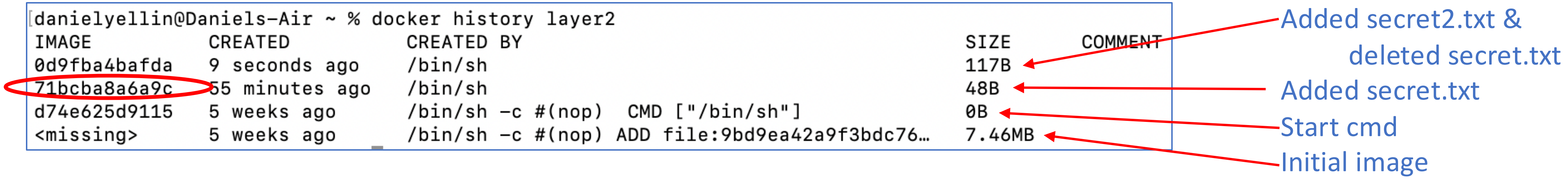
```
danielyellin@Daniels-Air ~ % docker history layer1
IMAGE          CREATED          CREATED BY                                      SIZE      COMMENT
71bcba8a6a9c   14 minutes ago   /bin/sh                                         48B
d74e625d9115   5 weeks ago      /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
<missing>      5 weeks ago      /bin/sh -c #(nop) ADD file:9bd9ea42a9f3bdc76…   7.46MB
```

Added secret.txt

Start cmd

Initial image

- docker history layer2



```
danielyellin@Daniels-Air ~ % docker history layer2
IMAGE          CREATED          CREATED BY                                      SIZE      COMMENT
0d9fba4bafda   9 seconds ago    /bin/sh                                         117B
71bcba8a6a9c   55 minutes ago   /bin/sh                                         48B
d74e625d9115   5 weeks ago      /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
<missing>      5 weeks ago      /bin/sh -c #(nop) ADD file:9bd9ea42a9f3bdc76…   7.46MB
```

Added secret2.txt & deleted secret.txt

Added secret.txt

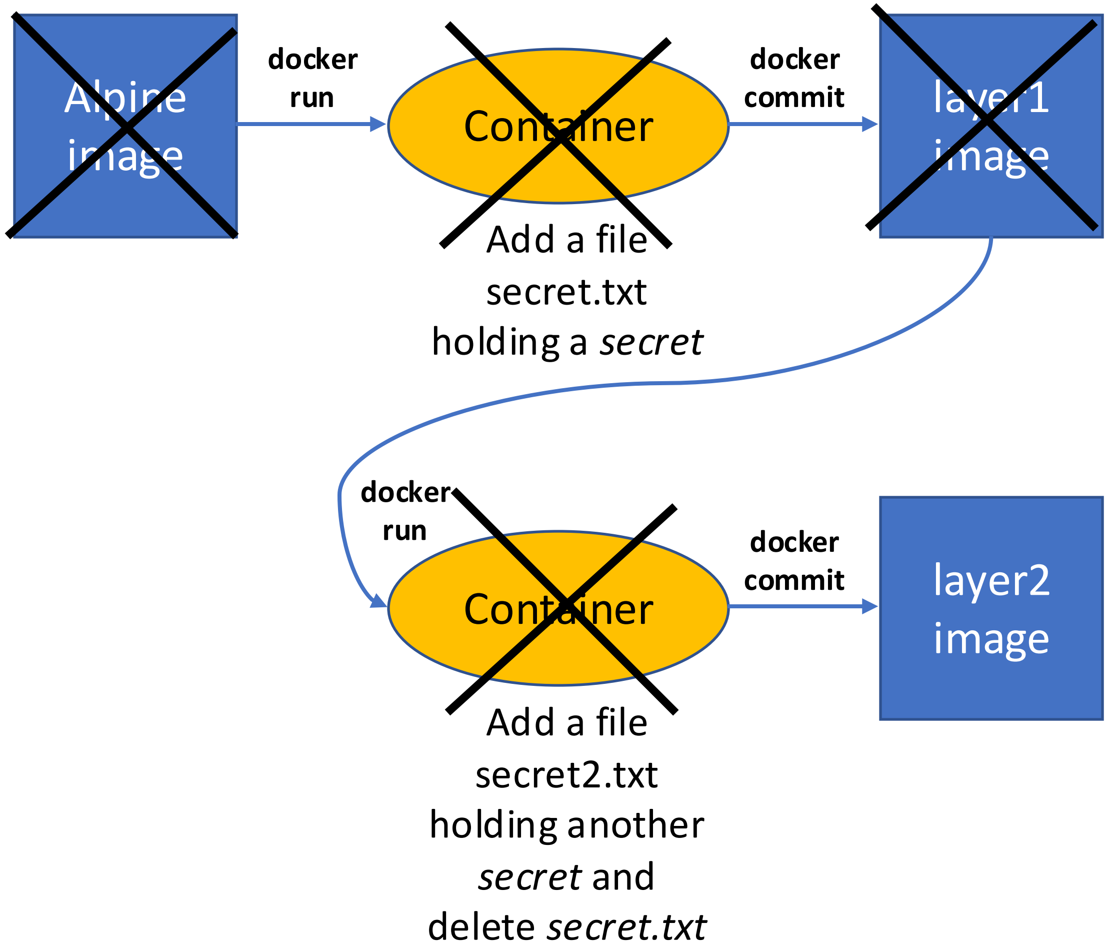
Start cmd

Initial image

Example (cont)

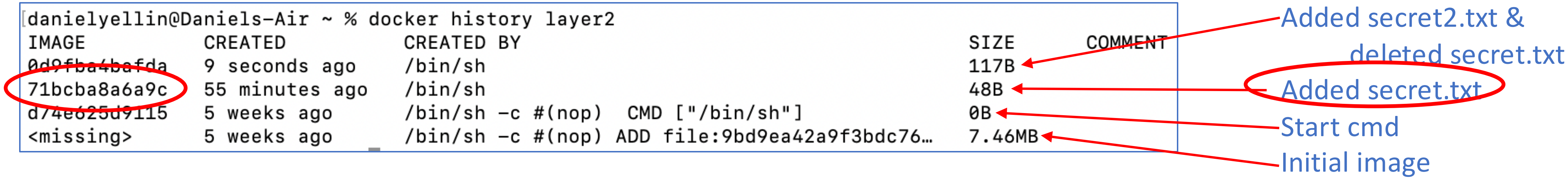
- Next remove the containers and delete all the images exempt *layer2*

- ***QUESTION: can we recover secret.txt?***


Alpine image
docker run
Container
Add a file secret.txt holding a *secret*
docker commit
layer1 image
docker run
Container
Add a file secret2.txt holding another *secret* and delete *secret.txt*
docker commit
layer2 image

Example (cont)

- Recall the history of layer2, all the layers are available in the image



```
[danielyellin@Daniels-Air ~ % docker history layer2
IMAGE          CREATED          CREATED BY                                      SIZE      COMMENT
0d9fba4bafda   9 seconds ago    /bin/sh                                         117B
71bcba8a6a9c   55 minutes ago   /bin/sh                                         48B
d74e625d9115   5 weeks ago      /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
<missing>      5 weeks ago      /bin/sh -c #(nop) ADD file:9bd9ea42a9f3bdc76…   7.46MB
```

Added secret2.txt & deleted secret.txt

Added secret.txt

Start cmd

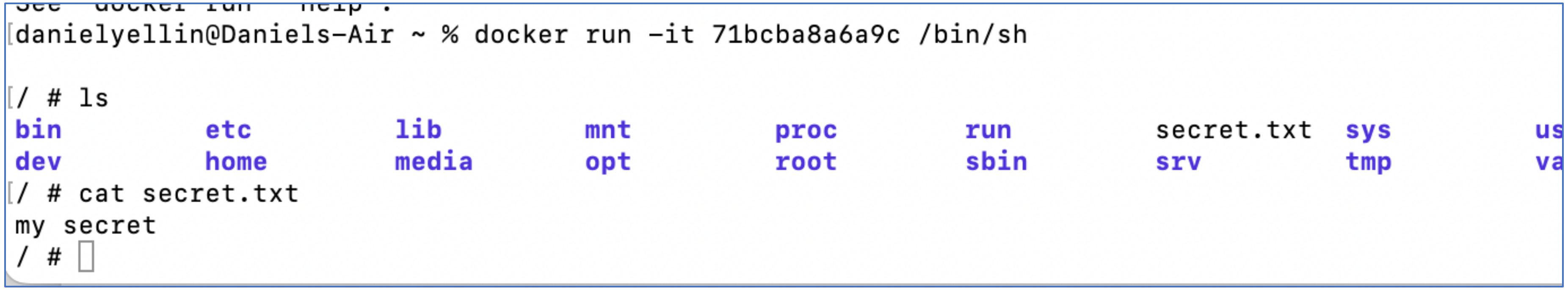
Initial image

Lets see what happens when we issue the command:

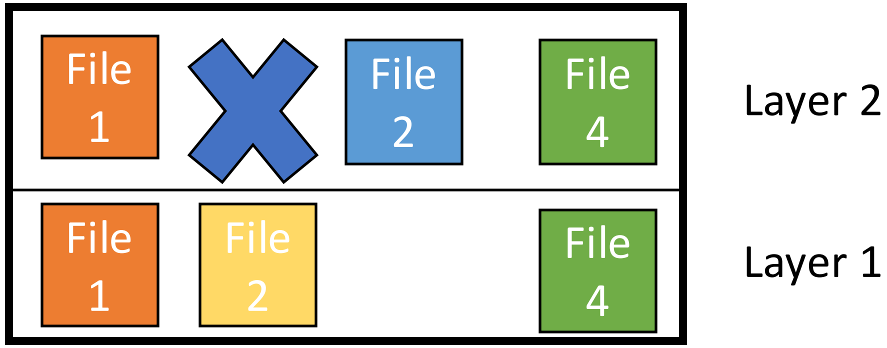
```
docker run -it 71bcba8a6a9c /bin/sh
```

Example (cont)

- Because layer2 is on top of layer1, we can “recover” the filesystem snapshop from the time of layer1 !



```
See 'docker run --help'.
[danielyellin@Daniels-Air ~ % docker run -it 71bcba8a6a9c /bin/sh

[/ # ls
bin          etc          lib          mnt          proc         run          secret.txt  sys          us
dev          home         media        opt          root         sbin         srv          tmp          va
[/ # cat secret.txt
my secret
/ # 
```

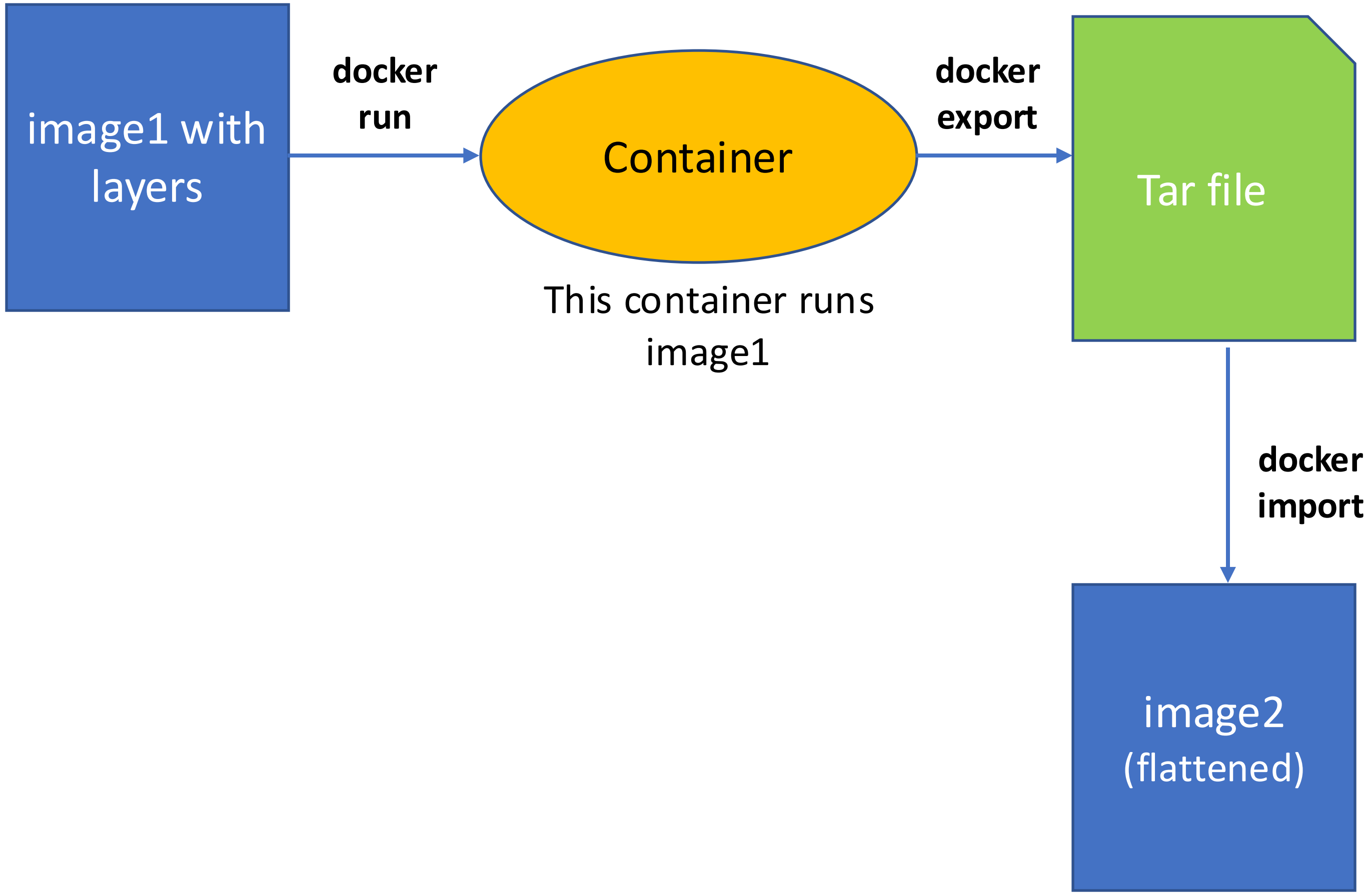


Example (cont)

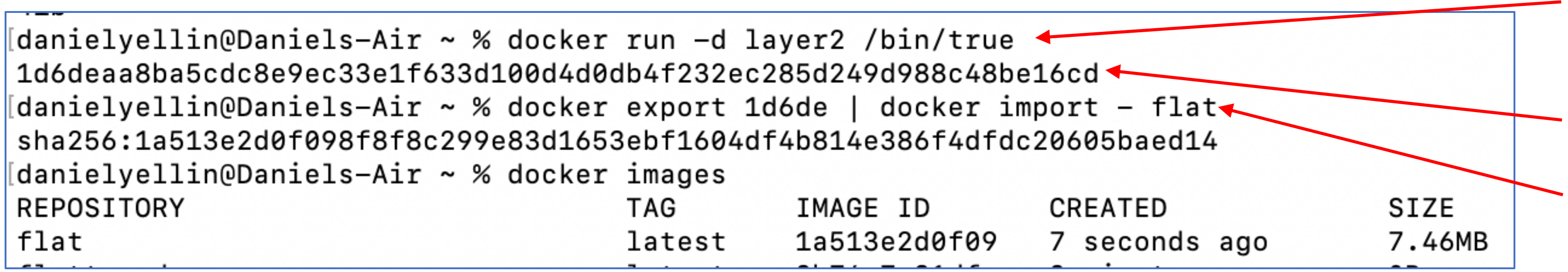
- We often do not want this behavior because:
  - It exposes secrets we do not want others to see
  - It blows up the size of our image (if there is layer that has a lot of large files that is no longer needed)

- There are multiple ways to *flatten* an image
- One way is to use **`docker export`** to create a *tar file*, then to use **`docker import`** to create an image from that tar file. When you do this, all the intermediate images are removed.

Docker export and import



Example (cont)



```
[danielyellin@Daniels-Air ~ % docker run -d layer2 /bin/true
1d6deaa8ba5cdc8e9ec33e1f633d100d4d0db4f232ec285d249d988c48be16cd
[danielyellin@Daniels-Air ~ % docker export 1d6de | docker import - flat
sha256:1a513e2d0f098f8f8c299e83d1653ebf1604df4b814e386f4dfdc20605baed14
[danielyellin@Daniels-Air ~ % docker images
REPOSITORY                                  TAG       IMAGE ID       CREATED          SIZE
flat                                        latest    1a513e2d0f09   7 seconds ago    7.46MB
```

Creates a container from layer2 image and exits

This is the ID of the container running layer2

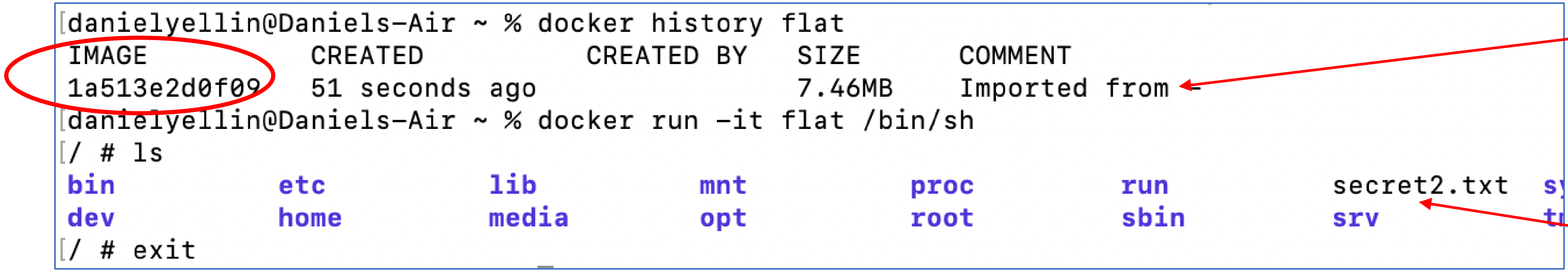
See explanation of this cmd below

```
docker export 1d6de | docker import - flat
```

- The first part of this line is a command that exports the image in the container beginning with ID 1d6de (layer2 image)
- It then *pipes* the output of this command, a *tar* file, into the next command
- The last part of this line is a that command takes a tar file and outputs an image of the given name; in our case we call this image “flat”
- This is the same as issuing the commands:
  - `docker export 1d6de my_image_tar`
  - `docker import my_image_tar flat`

Example (cont)

- Let's look at the history of the flattened image. Let’s also run the flattened image and look inside at the file system.



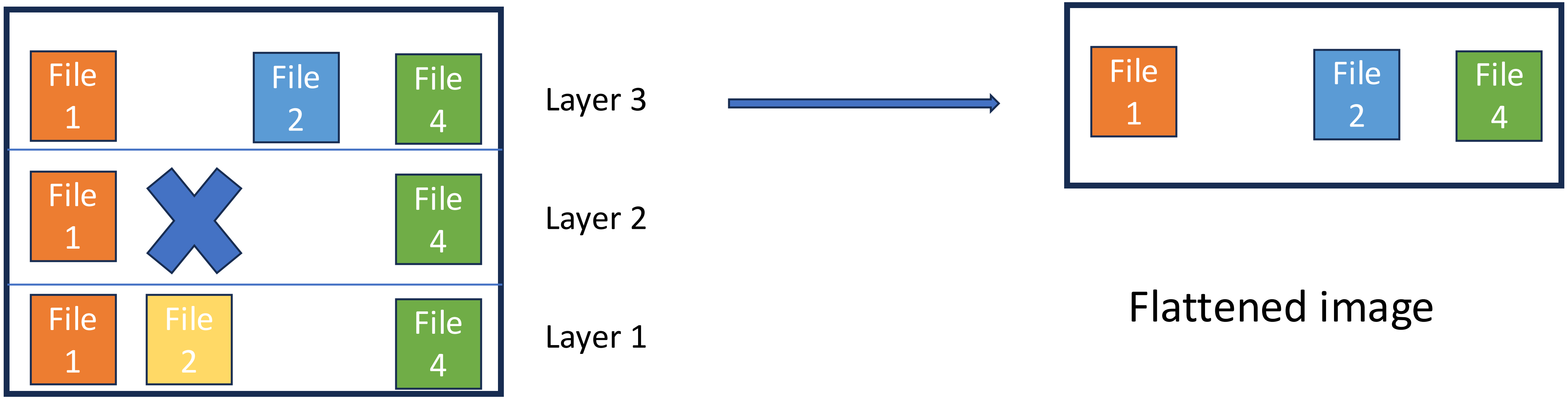
```
[danielyellin@Daniels-Air ~ % docker history flat
 IMAGE          CREATED          CREATED BY    SIZE       COMMENT
 1a513e2d0f09   51 seconds ago                 7.46MB     Imported from -
[danielyellin@Daniels-Air ~ % docker run -it flat /bin/sh
[/ # ls
 bin          etc          lib          mnt          proc         run          secret2.txt  sy
 dev          home         media        opt          root         sbin         srv          tr
[/ # exit
```

Only a single layer is visible in the history

Only secret2.txt, not secret.txt

- Run the history cmd and you will not see any additional layers – they are gone.

Flattening images



Reasons to flatten images

- **Reduced size** − Flattening an image can significantly reduce its size, as it removes the overhead of multiple layers.
- **Improved performance** − Flattening an image can improve the performance of the image, as it reduces the number of layers that must be processed during runtime.
- **Simplified distribution** − Flattening an image can simplify the process of distributing the image, as it removes the need to manage multiple layers and their dependencies.
- **Remove “secrets”** – Flattening an image removes information that might be in early layers of the image, but not intended for distribution.

Ordering of Dockerfile commands to reduce image size

- Docker caches images, so when you build a Docker image from a buildfile, it will look at each step and see if an existing image up to that step exists in the cache.
- If changes cause a step to be redone (e.g., a file it relies on has changes), then a new image is built, and all the steps forward will create new images.
- For Dockerfiles with steps that require a lot of time to execute, you want to order commands that may change Docker image state to be as late as late as possible in the Dockerfile.

For example, if you place an ADD instruction towards the top of the Dockerfile and anything in the added file or directory has changed, then all subsequent layers associated with commands below the ADD will have their cached images invalidated.

Source1 and Source2