Working with Docker Containers

Posted November 4, 2016 84.7k DOCKER

By: Melissa Anderson

Introduction

Docker is a popular containerization tool used to provide software applications with a filesystem that contains everything they need to run. Using Docker containers ensures that the software will behave the same way, regardless of where it is deployed, because its run-time environment is ruthlessly consistent.

In this tutorial, we'll provide a brief overview of the relationship between Docker images and Docker containers. Then, we'll take a more detailed look at how to run, start, stop, and remove containers.

Overview

We can think of a **Docker image** as an inert template used to create Docker containers. Images typically start with a root filesystem and add filesystem changes and their corresponding execution parameters in ordered, read-only layers. Unlike a typical Linux distribution, a Docker image normally contains only the bare essentials necessary for running the application. The images do not have state and they do not change. Rather, they form the starting point for Docker containers.

Images come to life with the `docker run` command, which creates a **container** by adding a read-write layer on top of the image. This combination of read-only layers topped with a read-write layer is known as a **union file system**. When a change is made to an existing file in a running container, the file is copied out of the read-only space into the read-write layer, where the changes are applied. The version in the read-write layer hides the original file but doesn't remove it. Changes in the read-write layer exist only within an individual container instance. When a container is deleted, any changes are lost unless steps are taken to preserve them.

Working with Containers

Each time you use the `docker run` command, it creates a new container from the image you specify. This can be a source of confusion, so let's take a look with some examples:

Step 1: Creating Two Containers

The following `docker run` command will create a new container using the base `ubuntu` image. `-t` will give us a terminal, and `-i` will allow us to interact with it. We'll rely on the default command in the Ubuntu base image's Docker file, `bash`, to drop us into a shell.

```
$ docker run -ti ubuntu
```

The command-line prompt changes to indicate we're inside the container as the root user, followed by the 12 character container ID.

```
root@11cc47339ee1:/#
```

We'll make a change by echoing some text into the container's `/tmp` directory, then use `cat` to verify that it was successfully saved.

```
root@11cc47339ee1:/# echo "Example1" > /tmp/Example1.txt
root@11cc47339ee1:/# cat /tmp/Example1.txt
```

```
Output
Example1
```

Now, let's exit the container.

```
root@11cc47339ee1:/# exit
```

Docker containers stop running as soon as the command they issued is complete, so our container stopped when we exited the bash shell. If we run `docker ps`, the command to display running containers, we won't see ours.

```
$ docker ps
```

```
Output
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              P
```

If we add the `-a` flag, which shows *all* containers, stopped or running, then our container will appear on the list:

```
$ docker ps -a
```

```
Output
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              P
11cc47339ee1        ubuntu              "/bin/bash"         6 minutes ago       Exited (127) 8 second
```

When the container was created, it was given its container ID and a randomly-generated name. In this case, 11cc47339ee1 is the container ID and `small_sinoussi` is the randomly-generated name. `ps -a` shows those values, as well as the image from which the container was built (`ubuntu`), when the container was created (`six minutes ago`), and the command that was run in it (`/bin/bash`). The output also provides the status of the container (`Exited`) and how long ago the container entered that state (`6 seconds ago`). If the container were still running, we'd see the status "Up," followed by how long it had been running.

If we re-run the same command, an entirely new container is created:

```
$ docker run -ti ubuntu
```

We can tell it's a new container because the ID in the command prompt is different, and when we look for our Example1 file, we won't find it:

```
root@6e4341887b69:/# cat /tmp/Example1
```

```
Output
cat: /tmp/Example1: No such file or directory
```

This can make it seem like the data has disappeared, but that's not the case. We'll exit the second container now to see that it, and our first container with the file we created, are both on the system.

```
root@6e4341887b69:/# exit
```

When we list the containers again, both appear:

```
$ docker ps -a
```

```
Output
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
6e4341887b69        ubuntu              "/bin/bash"         About a minute ago   Exited (1) 6 se
11cc47339ee1        ubuntu              "/bin/bash"         13 minutes ago       Exited (127) 6
```

Step 2: Restarting the First Container

To restart an existing container, we'll use the `start` command with the `-a` flag to attach to it and the `-i` flag to make it interactive, followed by either the container ID or name. Be sure to substitute the ID of your container in the command below:

```
$ docker start -ai 11cc47339ee1
```

We find ourselves at the container's bash prompt once again and when we `cat` the file we previously created, it's still there.

```
root@11cc47339ee1:/# cat /tmp/Example1.txt
```

```
Output
Example1
```

We can exit the container now:

```
root@11cc47339ee1:/# exit
```

This output shows that changes made inside the container persist through stopping and starting it. It's only when the container is removed that the content is deleted. This example also illustrates that the changes were limited to the individual container. When we started a second container, it reflected the original state of the image.

Step 3: Deleting Both Containers

We've created two containers, and we'll conclude our brief tutorial by deleting them. The `docker rm` command, which works only on stopped containers, allows you to specify the name or the ID of one or more containers, so we can delete both with the following:

```
$ docker rm 11cc47339ee1 kickass_borg
```

```
Output
11cc47339ee1
kickass_borg
```

Both of the containers, and any changes we made inside them, are now gone.

Conclusion

We've taken a detailed look at the `docker run` command to see how it automatically creates a new container each time it is run. We've also seen how to locate a stopped container, start it, and connect to it. If

you'd like to learn more about managing containers, you might be interested in the guide, Naming Docker Containers: 3 Tips for Beginners.

By: Melissa Anderson

Upvote (25) Subscribe Share

Related Tutorials

How To Install Docker Compose on Ubuntu 18.04

How To Remove Docker Images, Containers, and Volumes

Naming Docker Containers: 3 Tips for Beginners

How to Manually Set Up a Prisma Server on Ubuntu 18.04

How To Use Traefik as a Reverse Proxy for Docker Containers on Debian 9

3 Comments

Leave a comment...

Log In to Comment

linuxman1 *June 30, 2017*

1 Thank you, The tutorial is really informative.

MelissaAnderson MOD *June 30, 2017*

0 Thanks for the feedback! I'm glad it was helpful.

rnmkr *May 31, 2018*

0 Thank you.