What Is a Container?

Last updated November 11, 2019

Module Development

Containers are sandboxed applications that can be run anywhere, no matter the underlying host OS. Docker containers are quite different from VMs in a number of ways that need to be understood before we can use them to develop Drupal sites.

In this tutorial, we'll:

- Define a container
- Explain how a container is a more limited environment than a VM

Goal

Be able to explain what a container is and how it differs from a VM.

Prerequisites

- What Is Docker?
- Run a Container Interactively

Not a lightweight VM

When using Docker, it's pretty easy to trick yourself into thinking that a container is merely a lightweight virtual machine. After all, when running a container interactively, it appears to be another machine, running a completely different operating system, with a different set of resources.

Let's test that assumption. First let's start a new container interactively:

```
$ docker run -i -t debian /bin/bash

root@9055b116a5d1:/#
```

Great! We're inside a running container. If you know your way around a Linux server, you could start by installing Apache. We're using the Debian distribution here as an example, so we use Debian's package manager, `apt-get`:

```
root@9055b116a5d1:/# apt-get update
...
root@9055b116a5d1:/# apt-get install apache2
...
root@9055b116a5d1:/#
```

For brevity, we've removed much of the command output above. We might see a few warnings, but nothing that prevents Apache from being installed properly. We should be able to start it now, as we usually would:

```
root@9055b116a5d1:/# apachectl start
AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.17.0.2. Set the 'ServerName' directive
root@9055b116a5d1:/#
```

Wonderful! For reasons we'll get into later, we can't just point a browser at the container yet. Instead, let's use another command line utility in the container, `curl`. The command isn't provided out of the box with Debian, so we need to install it too:

```
root@9055b116a5d1:/# apt-get install curl -y
...
root@9055b116a5d1:/# curl localhost

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
...
</body>
</html>

root@9055b116a5d1:/#
```

So far, so good. In addition to the `docker run` command, the `docker` executable also provides a number of additional subcommands for us to use. One of those is the `ps` subcommand, which lists running containers.

Let's open a *new* terminal window, and use it to list all of our running containers:

docker ps command at this point gives me a reply of co

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
9055b116a5d1        debian              "/bin/bash"         25 minutes ago      Up 25 minutes                           adoring_torvalds
```

Okay, that makes sense. We only have one container running, and so one container is listed. We'll get into what each of these columns mean in a later tutorial.

Let's return to the first terminal window, the one where we have the container running interactively. We should be able to leave the container now by using the `exit` command, just as we would if we had logged into a VM or remote server via SSH:

```
root@9055b116a5d1:/# exit

$
```

If this were a VM or a remote server, we would expect the container to still be there, happily running Apache. Let's run `docker ps` again:

```
root@9055b116a5d1:/# exit
exit

$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES

$
```

Huh!? Where'd our container go? We started Apache, and we verified it was running, so what happened?

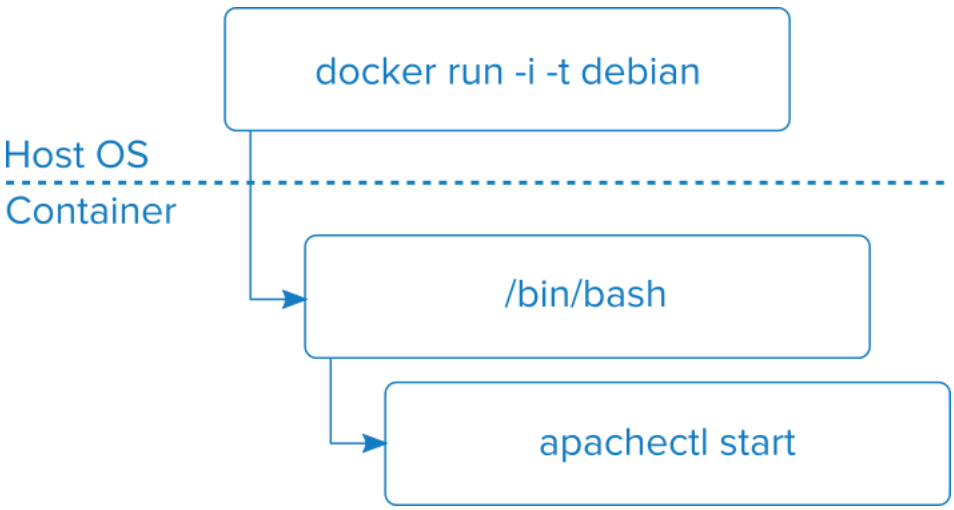
One container, one process

What happened is that we ran into one of the limitations of Docker containers. A container is less like a VM and more like a sandboxed application. Let's take another look at the `docker run` command we used:

```
$ docker run -i -t debian /bin/bash
```

We can see we told Docker to start a pre-made container, `debian`, interactively, and to run the `bash` command to give us an interactive command shell. Due to Linux's process model, when we start a process from a command shell such as `apachectl`, we are *forking* the process.

Docker handles forked processes well. When we started Apache using `apachectl`, Linux created a child process of our `bash` shell:



In a Linux VM or a physical server, the parent process of everything is usually the *init* process. On most Linux distributions today, that's *systemd*. Docker containers don't (usually) use *systemd*, but instead rely on whatever command you pass to `docker run`. Furthermore, if the parent process of a child process is stopped, the child process is stopped too. When we exited our container, it took down Apache with it.

A Docker container can only run one tree of processes at a time. If we stop the parent process of a container, everything else goes with it. This sounds like a huge limitation, but it's an intentional design choice. By restricting a container to only one process, it's easier for Docker to manage all the resources related to the container. Furthermore, it makes it easier to create and reuse containers as the scope is much smaller for each.

Background commands and Docker

What would happen if we tried running `apachectl start` right away? Instead of using `debian`, we'll use the `httpd` image which already has Apache installed. Let's use `docker run` to start Apache directly. Unlike when we ran the `debian` container, we do **not** want to run the `httpd` container interactively. Instead, we'll run the container without specifying any switches -- no `-i` or `-t`:

```
$ docker run httpd apachectl start
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.2. Set the 'ServerName' directive gl

$
```

The heck? Why did it stop?

It looked like Docker started the `httpd` container, downloading ("pulling") it if necessary. Apache did give us a `ServerName` warning, but this alone doesn't prevent Apache from functioning.

If we check with `docker ps`, we can see no container is running:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES

$
```

So what happened?

When we started Apache in this way, we ran into another subtle problem with Docker's restriction on processes. The `apachectl` command is used by administrators to start the Apache service in the background. It starts Apache, and returns control immediately to us. Once that happened, Docker assumed the container was finished and stopped the entire process tree. Again, this took Apache with it.

Running services in the foreground

What we need to do is force the `apachectl` command to stay running as long as Apache is running. In general, this is referred to as running a service in the *foreground*. If we dig around in the Apache documentation, we find that we can run it in the foreground with the following command:

```
apachectl -D FOREGROUND
```

Let's use that with Docker. Again, we do **not** want to run the container in interactively, so no `-i` or `-t`:

```
$ docker run httpd apachectl -D FOREGROUND
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.2. Set the 'ServerName' directive gl
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.2. Set the 'ServerName' directive gl
[Sun Dec 17 20:56:05.797912 2017] [mpm_event:notice] [pid 8:tid 140031787014016] AH00489: Apache/2.4.29 (Unix) configured -- resuming norm
[Sun Dec 17 20:56:05.798132 2017] [core:notice] [pid 8:tid 140031787014016] AH00094: Command line: '/usr/local/apache2/bin/httpd -D FOREGR
```

This time, the command does **not** immediately return, so we can assume that Apache is still up.

(If you've been following along, you can stop the container by first finding its container ID with `docker ps` then `docker kill CONTAINER_ID`, replacing `CONTAINER_ID` with the appropriate ID listed in `docker ps`. These commands are covered in-depth in Use a Detached Container.)

Recap

Docker containers are not lightweight VMs. They are instead sandboxed applications that can be run anywhere no matter the underlying operating system. Each container can only run one tree of processes. If the parent process stops for whatever reason, all of the child processes stop with it.

Further your understanding

- If we can only run one process in a container, how do we run multiple services to support LAMP?

Additional resources

- What is a container? (docker.com)

Was this helpful? Yes No

◀ Previous tutorial Next tutorial ▶

Drupal Development with Docker

Powered by: