by Adrian Mouat
Apr 13, 2015

You can get by running Docker containers with shell scripts, or with Docker Compose (if you don't mind ignoring the "don't use in production" warnings), but for some use cases, it's preferable to take advantage of the host init system/process manager. It seems that every major distro is moving to systemd these days, so that's what I'll look at in this post.

Using systemd or an equivalent is particularly useful if you have another, possibly non-containerized service that is dependent on the container. However, even developers of pure container applications may find advantages in using systemd and it's worth noting that CoreOS is built around systemd and Docker.

If you follow the **official Docker documentation for using systemd**, you'll see that they advise creating the containers manually with `docker create` and only using `docker start` and `docker stop` in the service file. I'm not a huge fan of this advice, as it makes it more difficult to migrate the setup between hosts or to restart the service with a fresh container — it would be better if the service file included all its dependencies. This is the approach taken by CoreOS, and the one I want to show in this blog post.

As an example, we'll consider systemdizing a dockerized redis. I'll be using a CentOS 7 distro for this, but it should be very similar on other systemd distros. Pretty much all you need is the following service file:

```
[Unit]
Description=Redis Container
After=docker.service
Requires=docker.service

[Service]
TimeoutStartSec=0
Restart=always
ExecStartPre=-/usr/bin/docker stop %n
ExecStartPre=-/usr/bin/docker rm %n
ExecStartPre=/usr/bin/docker pull redis
ExecStart=/usr/bin/docker run --rm --name %n redis

[Install]
WantedBy=multi-user.target
```

There's a few things worth pointing out:

- The container is clearly dependent on having Docker running, hence the `Requires` line. The `After` line is also needed to avoid race conditions.
- Before we start the container, we first stop and remove any existing container with the same name and then pull the latest version of the image. The "-" at the start means systemd won't abort if the command fails.
- This means that our container will be started from scratch each time. If you want to persist data then you'll need to do something with volumes or volume containers, or change the code to restart the old container if it exists.
- We've used `TimeoutStartSec=0` to turn off timeouts, as the `docker pull` may take a while.

If you save this file to `/etc/systemd/system/docker.redis.service` and run `systemctl start docker.redis`, systemd will start up the Redis container (remember that this may take some time if it needs to pull the redis image). We can then access it manually, or set up another service that is dependent on it. For example, if we have an application foo which is running in a container and dependent on the redis service, we can use the following service file:

```
[Unit]
Description=Foo Service
After=docker.service
Requires=docker.service
After=docker.redis.service
Requires=docker.redis.service

[Service]
TimeoutStartSec=0
Restart=always
ExecStartPre=-/usr/bin/docker stop foo
ExecStartPre=-/usr/bin/docker rm foo
ExecStartPre=/usr/bin/docker pull foo
ExecStart=/usr/bin/docker run --name foo
  --link docker.redi.service:redis --rm
  foo

[Install]
WantedBy=multi-user.target
```

Now if the redis container fails, systemd will automatically restart both the redis service and the dependent foo service.

This setup works pretty well *most* of the time. But there is a major problem. systemd isn't monitoring the container itself, it's really monitoring the *client*. If the client detaches from the container for whatever reason (e.g. a network problem), systemd will kill the container, even though it may be functioning fine. Conversely, if the container dies but the client remains running, systemd won't do anything. What we really want is for systemd to monitor the container instead of the client[1]. And there is a solution that does just that, **systemd-docker**.

systemd-docker works by wrapping the docker command and moving the container process into the cgroup of the systemd service unit when it starts. Our redis example would look something like:

```
1
2  [Unit]
3  Description=Redis
4  After=docker.service
5  Requires=docker.service
6
7  [Service]
8  TimeoutStartSec=0
9  ExecStartPre=/usr/bin/docker pull redis
10 ExecStart=/usr/local/bin/systemd-docker --cgroups name=systemd run --rm --name %n redis
11 Restart=always
12 RestartSec=10s
13 Type=notify
14 NotifyAccess=all
15
16 [Install]
17 WantedBy=multi-user.target
18
```

This is simpler, but one issue is that the systemd-docker command is hiding the fact that stopped containers with the same name will be removed. I had to add the argument `--cgroups name=systemd` to work around an **issue with CentOS 7 and cgroups** (other distros shouldn't need this argument). If you do want to try systemd-docker, note that you currently have to build from source as the "go get" functionality is broken due to a docker dependency at the time of writing.

So, in short, if you want to use systemd, you need to think about it a bit more than perhaps was first obvious. If you're just playing with some tooling and don't need super-reliable up-time, the code in this blog post should work just fine. Otherwise, you should be using systemd-docker or addressing the container monitoring issue in some other way.

---

1. This situation is currently being worked and we are likely to see a solution that doesn't require hacks or third-party tools soon. See the following issues for more info:

- https://github.com/docker/docker/issues/7245
- https://github.com/docker/docker/pull/10427
- https://github.com/ibuildthecloud/systemd-docker/issues/25

- Share:
-