Mark Betz Follow
Programmer, system architect, husband, father of three smart kids, two unruly dogs, and a resentful ...
Mar 23, 2016 · 8 min read

Ten tips for debugging Docker containers

Containers are awesome, but sometimes it can feel like your code has been shut up in a black box, stuck off in the cloud where you can't see what it's doing. When your app runs the way it's supposed to this isn't a problem: containers come, containers go, everyone's happy. But then there's that moment when you push a new version of your image and check the cluster status only to see it crashlooping. What's going on? The CTO is pinging you on Slack, and you need answers. Here are ten tips for shining a light into your misbehaving containers.

1—View stdout history with the logs command.

Lots of programs log stuff to stdout, especially when things go south. Anything that gets written to stdout for the process that is pid 1 inside the container will get captured to a history file on the host, where it can be viewed with the **logs** command.

```
$ docker run -d --name=logtest alpine /bin/sh -c "while
true; do sleep 2; df -h; done"
35f6353d2e47ab1f6c34073475014f4ab5e0b131043dca4454f67be9d8ef
1253
$ docker logs logtest
Filesystem Size Used Available Use% Mounted on
none 93.7G 2.0G 87.0G 2% /
tmpfs 3.8G 0 3.8G 0% /dev
tmpfs 3.8G 0 3.8G 0% /sys/fs/cgroup
... etc ...
```

This history is available even after the container exits, as long as its file system is still present on disk (until it is removed with **docker rm**). The data is stored in a json file buried under /var/lib/docker. You can get the complete path using the inspect command, which we'll look at later. The log command takes options that allow you to follow this file, basically tail -f, as well as choose how many lines the command returns (tail -n). By default the command returns all lines.

2—Stream stdout with the attach command.

If you want to see what is written to stdout in real time then the **attach** command is your friend.

```
$ docker run -d --name=logtest alpine /bin/sh -c "while
true; do sleep 2; df -h; done"
26a329f1e7074f0c0f89caf266ad145ab427b1bcb35f82557e78bafe053f
af44
$ docker attach logtest
Filesystem Size Used Available Use% Mounted on
none 93.7G 2.0G 87.0G 2% /
tmpfs 3.8G 0 3.8G 0% /dev
tmpfs 3.8G 0 3.8G 0% /sys/fs/cgroup
… etc …
Filesystem Size Used Available Use% Mounted on
none 93.7G 2.0G 87.0G 2% /
tmpfs 3.8G 0 3.8G 0% /dev
tmpfs 3.8G 0 3.8G 0% /sys/fs/cgroup
… etc …
```

By default this command attaches stdin and proxies signals to the remote process. Options are available to control both of these behaviors. To detach from the process use the default `ctrl-p ctrl-q` sequence. Note that running a command in a loop the way I am above can make detaching a little wonky, but it works fine for normal stuff.

3—Execute arbitrary commands with exec.

Maybe the most powerful all-around tool in your kit, the **exec** command allows you to run arbitrary commands inside a running container.

```
$ docker run -d --name=exectest alpine watch "echo 'This is
a test.' >> /var/log/test.log"
70ddfdf5169e755177a812959808973c92974dba2531c42a21ad9e50c8a4
```

```
804c
$ docker exec exectest cat /var/log/test.log
This is a test.
This is a test.
This is a test.
This is a test.
```

You can even use exec to get an interactive shell in the container.

```
$ docker run -d --name=exectest alpine watch "echo 'This is
a test.' >> /var/log/test.log"
91e46bf5d19d4239a2f30af06669d4263580a01187d2290c33d7dee110f7
6356
$ docker exec -it exectest /bin/sh
/ # ls -al /var/log
total 12
drwxr-xr-x 2 root root 4096 Mar 23 05:37 .
drwxr-xr-x 10 root root 4096 Mar 23 05:37 ..
-rw-r — r — 1 root root 192 Mar 23 05:38 test.log
/ # exit
$
```

Note that exec only works while the container is running. So for a container that is crashing you'll need to fall back on the logs command.

4—Override the ENTRYPOINT.

Every docker image has an entrypoint and command, whether defined in the dockerfile at build time or as an option to the docker run command at run time. The relationship between entrypoint and command can be a little confusing, and there are different ways to use them, but here is one setup that follows best practices and gives you a lot of customization ability.

Let's say you are running a django app. First, you define the python process that runs the app as the entry point. This will make sure it is pid 1 inside the container. You can do this in the dockerfile.

```
ENTRYPOINT ["python", "manage.py", "runserver"]
```

You can also do it on the command line when docker run is called, and that is where the cool comes in. If the entry point is specified on the command line then it overrides any entry point set in the dockerfile when the image was built. This can be a powerful tool. Imagine a situation where the django image in this example was crashing due to a configuration problem. Instead of running manage.py you want to get a shell and poke around.

```
docker run -d -p 80:80 --entrypoint /bin/sh /myrepo
/mydjangoapp
```

Now you can jump in, check the config, even try to run the manage.py command interactively to see what happens.

5—Add options with the CMD.

Back to that confusing relationship between ENTRYPOINT and CMD. Here's one way to look at it: the ENTRYPOINT defines the process that runs as PID 1 in the container, and the CMD adds options to it. If you don't specify an ENTRYPOINT but you do specify a CMD then the implicit entrypoint is /bin/sh -c. CMD, like ENTRYPOINT, can be specified on the command line when docker run is called, which makes it another powerful tool for modifying container behavior. However, the usage is a little different.

```
docker run -d -p 80:8000 /myrepo/mydjangoapp 0.0.0.0:8000
```

Anything that appears after the image name in the docker run command is passed to the container and treated as CMD arguments, basically as if it were specified in the dockerfile like this:

```
ENTRYPOINT [“python”, “manage.py”, “runserver”]
CMD ["0.0.0.0:8000"]
```

Additional arguments can be passed as space-delimited parameters to the docker run call. This is a very convenient and flexible way to reconfigure an image for debugging purposes, by passing an option to increase log verbosity, for example.

6—Pause and unpause a container.

I doubt you'll have use for this one very often, but it's cool so I'm throwing it in here anyway. Using the docker **pause** command you can pause all of the processes inside a container.

```
$ docker run -d --name=pausetest alpine /bin/sh -c “while
true; do sleep 2; date; done”
e81e1bc519e4eb2e30a1c1e57198f0060147fa749ff8f93ba4f69bdf8a11
4311
```

The container is just echoing the date to stdout, so we can watch it with the attach command.

```
$ docker attach pausetest
Wed Mar 23 06:21:40 UTC 2016
Wed Mar 23 06:21:42 UTC 2016
```

Now pause the container, wait a bit, then unpause it.

```
$ docker pause pausetest
pausetest
$ docker unpause pausetest
pausetest
```

And back in our other window where attach is running…

```
mark@viking:~/workspace/cluster-iperf$ docker attach
pausetest
Wed Mar 23 06:21:40 UTC 2016
Wed Mar 23 06:21:42 UTC 2016
Wed Mar 23 06:22:06 UTC 2016
Wed Mar 23 06:22:08 UTC 2016
```

Actually I don’t have to stretch my imagination too much to come up with scenarios where this would be useful. It might be nice to freeze the current state of a server while I eat lunch or something.

7—Get process stats with the top command.

The docker **top** command is exactly what it sounds like: top that runs in the container.

```
$ docker run -d — name=toptest alpine:3.1 watch “echo
‘Testing top’”
fc54369116fe993ae45620415fb5a6376a3069cdab7c206ac5ce3b57006d
4241
$ docker top toptest
UID PID … TIME CMD
root 26339 … 00:00:00 watch “echo ‘Testing top’”
root 26370 … 00:00:00 sleep 2
```

Some columns removed to make it fit here. There is also a docker stats command that is basically top for all the containers running on a host.

8—View container details with the inspect command.

The **inspect** command returns information about a container or an image. Here’s an example of running it on the toptest container from the last example above.

```
$ docker inspect toptest
[
  {
    “Id”: “fdb3008e70892e14d183f8 ... 020cc34fec9703c821”,
    “Created”: “2016-03-23T17:45:01.876121835Z”,
    “Path”: “/bin/sh”,
    “Args”: [
      “-c”,
      “while true; do sleep 2; echo ‘Testing top’; done”
    ],
    … and lots, lots more.
  }
]
```

I’ve skipped the bulk of the output because there’s a lot of it. Some of the more valuable bits of intelligence you can get are:

Current state of the container. (In the “State” property.)

Path to the log history file. (In the “LogPath” field.)

Values of set environment vars. (In the “Config.Env” field.)

Mapped ports. (In the “NetworkSettings.Ports” field.)

Probably the most valuable use of inspect for me in the past has been getting the values of environment vars. Even with largely automated deployments I’ve run into issues in the past where the wrong arg was passed to a command and a container ended up running with vars set to incorrect values. When one of your cloud containers starts choking commands like inspect can be a quick cure.

9—View image layers with the history command.

This one is more of a build time diagnostic tool, but the questions it answers sometimes come up in debugging situations as well. The docker **history** command shows the individual layers that make up an image, along with the commands that created them, their size on disk, and hashes.

```
$ docker history alpine
```

The returned table is quite wide so I won’t try to illustrate it here. If you run into a situation where the contents of your image aren’t what you expect, this command may solve your puzzle for you.

10—Run one process in each container.

This last one is less of a debugging tip, and more of a best practice that will definitely make debugging, and reasoning about the state of your container a lot easier. Docker containers are meant to run a single process that is PID 1 inside the sandbox. You can make that process a shell and spin up a bunch of stuff in the background, or you can make it supervisord and do the same, but it’s not really what containers are good at and it hobbles systems meant to manage and monitor them.

The main benefit of running one process per container is that it’s easier to reason about the state of the container at run time. The container comes up when the process comes up and dies when it dies, and platforms like Docker Compose, kubernetes, and Amazon ECS can see that and restart it. They can also monitor the health (liveness and readiness) of a single-process container, but it is much more difficult to define declaratively what either of those things means when they depend on multiple processes being in good health inside the container.

Conclusion

There are a few other advanced techniques that didn’t fit into my arbitrary limit of ten tips, but perhaps they’ll be a subject for another time. They include things like advanced logging scenarios, exporting and importing container file systems, and commands to diff image layers. In the meantime the tips in this article will hopefully get the top off that black box and make your transition to running containers a little easier.