

Day-to-day Docker

This chapter covers

- Keeping a handle on your container and volume space usage
- Detaching from containers without stopping them
- Visualizing your Docker image lineage in a graph
- Running commands directly on your containers from the host

As with any moderately complex software project, Docker has a lot of nooks and crannies that are important to know about if you want to keep your experience as smooth as possible.

This chapter's techniques will show you some of the more important of these, as well as introduce some external tools built by third parties to scratch their own itches. Think of it as your Docker toolbox.

6.1 *Staying ship-shape*

If you're anything like us (and if you're following this book studiously), your growing Docker addiction will mean that you start up numerous containers on, and download a variety of images to, your chosen host.

As time goes on, Docker will take up more and more resources, and some housekeeping of containers and volumes will be required. We'll show you the how and why of this. We'll also introduce some visual tools for keeping your Docker environment clean and tidy, in case you want an escape from the command line.

Running containers is all very well, but you'll fairly quickly find yourself wanting to do more than just start a single command in the foreground. We'll take a look at escaping a running container without killing it, and at executing commands inside a running container.

TECHNIQUE 41 **Running Docker without sudo**

The Docker daemon runs in the background of your machine as the root user, giving it a significant amount of power, which it exposes to you, the user. Needing to use `sudo` is a result of this, but it can be inconvenient and make some third-party Docker tools impossible to use.

PROBLEM

You want to be able to run the `docker` command without having to use `sudo`.

SOLUTION

The official solution is to add yourself to the `docker` group.

Docker manages permissions around the Docker Unix domain socket through a user group. For security reasons, distributions don't make you part of that group by default, as it effectively grants full root access to the system.

By adding yourself to this group, you'll be able to use the `docker` command as yourself:

```
$ sudo addgroup -a username docker
```

Restart Docker and fully log out and in again, or reboot your machine if that's easier. Now you don't need to remember to type `sudo` or set up an alias to run Docker as yourself.

DISCUSSION

This is an extremely important technique for a number of tools used later in the book. In general, anything that wants to talk to Docker (without being started in a container) will need access to the Docker socket, requiring either `sudo` or the setup described in this technique. Docker Compose, introduced in technique 76, is an official tool from Docker Inc. and is an example of such a tool.

TECHNIQUE 42 **Housekeeping containers**

A frequent gripe of new Docker users is that in a short space of time you can end up with many containers on your system in various states, and there are no standard tools for managing this on the command line.

PROBLEM

You want to prune the containers on your system.

SOLUTION

Set up aliases to run the commands that tidy up old containers.

The simplest approach here is to delete all containers. Obviously, this is something of a nuclear option that should only be used if you're certain it's what you want.

The following command will remove all containers on your host machine.

```
$ docker ps -a -q | \
  xargs --no-run-if-empty docker rm -f
```

Get a list of all container IDs, both running and stopped, and pass them to...

...the `docker rm -f` command, which will remove any containers passed, even if they're running.

To briefly explain `xargs`, it takes each line of the input and passes them all as arguments to the subsequent command. We've passed an additional argument here, `--no-run-if-empty`, which avoids running the command at all if there's no output from the previous command, in order to avoid an error.

If you have containers running that you may want to keep, but you want to remove all those that have exited, you can filter the items returned by the `docker ps` command:

```
docker ps -a -q --filter status=exited | \
  xargs --no-run-if-empty docker rm
```

The `--filter` flag tells the `docker ps` command which containers you want returned. In this case you're restricting it to containers that have exited. Other options are running and restarting.

This time you don't force the removal of containers because they shouldn't be running, based on the filter you've given.

In fact, removing all stopped containers is such a common use case that Docker added a command specifically for it: `docker container prune`. However, this command is limited to just that use case, and you'll need to refer back to the commands in this technique for any more complex manipulation of containers.

As an example of a more advanced use case, the following command will list all containers with a nonzero error code. You may need this if you have many containers on your system and you want to automate the examination and removal of any containers that exited unexpectedly:

```
comm -3 \
  <(docker ps -a -q --filter=status=exited | sort) \
  <(docker ps -a -q --filter=exited=0 | sort) | \
  xargs --no-run-if-empty docker inspect > error_containers
```

Finds exited container IDs, sorts them, and passes them as a file to `comm`

Runs the `comm` command to compare the contents of two files. The `-3` argument suppresses lines that appear in both files (in this example, those with a zero exit code) and outputs any others.

Finds containers with an exit code of 0, sorts them, and passes them as a file to `comm`

Runs `docker inspect` against containers with a nonzero exit code (as piped in by `comm`) and saves the output to the `error_containers` file

TIP If you’ve not seen it before, the `<(command)` syntax is called *process substitution*. It allows you to treat the output of a command as a file and pass it to another command, which can be useful where piping output isn’t possible.

The preceding example is rather complicated, but it shows the power you can get from combining different utilities. It outputs all stopped container IDs, and then picks just those that have a nonzero exit code (those that exited in an unexpected way). If you’re struggling to follow this, running each command separately and understanding them that way first can be helpful in learning the building blocks.

Such a command could be useful for gathering container information on production. You may want to adapt it to run a cron to clear out containers that exited in expected ways.

Make these one-liners available as commands

You can add commands as aliases so that they’re more easily run whenever you log in to your host. To do this, add lines like the following to the end of your `~/.bashrc` file:

```
alias dockernuke='docker ps -a -q | \
xargs --no-run-if-empty docker rm -f'
```

When you next log in, running `dockernuke` from the command line will delete any Docker containers found on your system.

We’ve found that this saves a surprising amount of time. But be careful! It’s all too easy to remove production containers this way, as we can attest. And even if you’re careful enough not to remove running containers, you still might remove non-running but still useful data-only containers.

DISCUSSION

Many of the techniques in this book end up creating containers, particularly when introducing Docker Compose in technique 76 and in the chapters devoted to orchestration—after all, orchestration is all about managing multiple containers. You may find the commands discussed here useful for cleaning up your machines (local or remote) to get a fresh start when you finish each technique.

TECHNIQUE 43 **Housekeeping volumes**

Although volumes are a powerful feature of Docker, they come with a significant operational downside. Because volumes can be shared between different containers, they can’t be deleted when a container that mounted them is deleted. Imagine the scenario outlined in figure 6.1.

“Easy!” you might think. “Delete the volume when the last-referencing container is removed!” Indeed, Docker could have taken that option, and this approach is the one

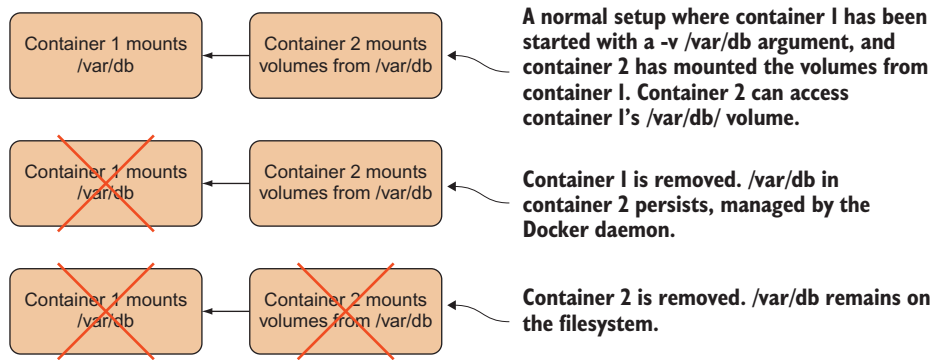


Figure 6.1 What happens to `/var/db` when containers are removed?

that garbage-collected programming languages take when they remove objects from memory: when no other object references it, it can be deleted.

But Docker judged that this could leave people open to losing valuable data accidentally and preferred to make it a user decision as to whether a volume should be deleted on removal of the container. An unfortunate side effect of this is that, by default, volumes remain on your Docker daemon's host disk until they're removed manually. If these volumes are full of data, your disk can fill up, so it's useful to be aware of ways to manage these orphaned volumes.

PROBLEM

You're using too much disk space because orphaned Docker mounts exist in your host.

SOLUTION

Use the `-v` flag when calling `docker rm`, or use the `docker volume` subcommands to destroy them if you forget.

In the scenario in figure 6.1, you can ensure that `/var/db` is deleted if you always call `docker rm` with the `-v` flag. The `-v` flag removes any associated volumes if no other container still has it mounted. Fortunately, Docker is smart enough to know whether any other container has the volume mounted, so there are no nasty surprises.

The simplest approach is to get into the habit of typing `-v` whenever you remove a container. That way you retain control of whether volumes are removed. But the problem with this approach is that you might not want to always delete volumes. If you're writing a lot of data to these volumes, it's quite likely that you won't want to lose the data. Additionally, if you get into such a habit, it's likely to become automatic, and you'll only realize you've deleted something important when it's too late.

In these scenarios you can use a command that was added to Docker after much griping and many third-party solutions: `docker volume prune`. This will remove any unused volumes:

Volumes that exist on the machine, whether or not they're in use

```
$ docker volume ls
DRIVER          VOLUME NAME
local           80a40d34a2322f505d67472f8301c16dc75f4209b231bb08faa8ae48f
➡ 36c033f
local           b40a19d89fe89f60d30b3324a6ea423796828a1ec5b613693a740b33
➡ 77fd6a7b
local           bceef6294fb5b62c9453fcbba4b7100fc4a0c918d11d580f362b09eb
➡ 58503014
```

Runs the command to list the volumes Docker is aware of

```
$ docker volume prune
WARNING! This will remove all volumes not used by at least one container.
```

Runs the command to delete unused volumes

```
Are you sure you want to continue? [y/N] y
```

Confirms the deletion of volumes

```
Deleted Volumes:
80a40d34a2322f505d67472f8301c16dc75f4209b231bb08faa8ae48f36c033f
b40a19d89fe89f60d30b3324a6ea423796828a1ec5b613693a740b3377fd6a7b
```

```
Total reclaimed space: 230.7MB
```

Volumes that have been deleted

If you want to skip the confirmation prompt, perhaps for an automated script, you can pass `-f` to `docker volume prune` to skip it.

TIP If you want to recover data from an undeleted volume that's no longer referenced by any containers, you can use `docker volume inspect` to discover the directory a volume lives in (likely under `/var/lib/docker/volumes/`). You can then browse it as the root user.

DISCUSSION

Deleting volumes is likely not something you'll need to do very often, as large files in a container are usually mounted from the host machine and don't get stored in the Docker data directory. But it's worth doing a cleanup every week or so, to avoid them piling up, particularly if you're using data containers from technique 37.

TECHNIQUE 44 Detaching containers without stopping them

When working with Docker, you'll often find yourself in a position where you have an interactive shell, but exiting from the shell would terminate the container, as it's the container's principal process. Fortunately there's a way to detach from a container (and, if you want, you can use `docker attach` to connect to the container again).

PROBLEM

You want to detach from a container interaction without stopping it.

SOLUTION

Use the built-in key combination in Docker to escape from the container.

Docker has helpfully implemented a key sequence that's unlikely to be needed by any other application and that's also unlikely to be pressed by accident.

Let's say you started up a container with `docker run -t -i -p 9005:80 ubuntu /bin/bash`, and then `apt-get` installed an Nginx web server. You want to test that it's accessible from your host with a quick `curl` command to `localhost:9005`.

Press Ctrl-P and then Ctrl-Q. Note that it's not all three keys pressed at once.

NOTE If you're running with `--rm` and `detach`, the container will still be removed once it terminates, either because the command finishes or you stop it manually.

DISCUSSION

This technique is useful if you've started a container but perhaps forgot to start it in the background, as shown in technique 2. It also allows you to freely attach and detach from containers if you want to check how they're doing or provide some input.

TECHNIQUE 45 Using Portainer to manage your Docker daemon

When demonstrating Docker, it can be difficult to demonstrate how containers and images differ—lines on a terminal aren't visual. In addition, the Docker command-line tools can be unfriendly if you want to kill and remove specific containers out of many. This problem has been solved with the creation of a point-and-click tool for managing the images and containers on your host.

PROBLEM

You want to manage containers and images on your host without using the CLI.

SOLUTION

Use Portainer, a tool created by one of the core contributors to Docker.

Portainer started out life as DockerUI, and you can read about it and find the source at <https://github.com/portainer/portainer>. Because there are no prerequisites, you can jump straight to running it:

```
$ docker run -d -p 9000:9000 \
-v /var/run/docker.sock:/var/run/docker.sock \
portainer/portainer -H unix:///var/run/docker.sock
```

This will start the portainer container in the background. If you now visit <http://localhost:9000>, you'll see the dashboard giving you at-a-glance information for Docker on your computer.

Container management functionality is probably one of the most useful pieces of functionality here—go to the Containers page, and you'll see your running containers listed (including the portainer container), with an option to display all containers. From here you can perform bulk operations on containers (such as killing them) or click on a container name to dive into more detail about the container and perform individual operations relevant to that container. For example, you'll be shown the option to remove a running container.

The Images page looks fairly similar to the Containers page and also allows you to select multiple images and perform bulk operations on them. Clicking on the image ID offers some interesting options, such as creating a container from the image and tagging the image.

Remember that Portainer may lag behind official Docker functionality—if you want to use the latest and greatest functionality, you may be forced to resort to the command line.

DISCUSSION

Portainer is one of many interfaces available for Docker, and it's one of the most popular, with many features and active development. As one example, you can use it to manage remote machines, perhaps after starting containers on them with technique 32.

TECHNIQUE 46 **Generating a dependency graph of your Docker images**

The file-layering system in Docker is an immensely powerful idea that can save space and make building software much quicker. But once you start using a lot of images, it can be difficult to understand how your images are related. The `docker images -a` command will return a list of all the layers on your system, but this isn't a user-friendly way to comprehend these relationships—it's much easier to visualize the relationships between your images by creating a tree of them as an image using Graphviz.

This is also a demonstration of Docker's power to make complicated tasks simpler. Installing all the components to produce the image on a host machine would previously have involved a long series of error-prone steps, but with Docker it can be turned into a single portable command that's far less likely to fail.

PROBLEM

You want to visualize a tree of the images stored on your host.

SOLUTION

Use an image that we've created (based on one by CenturyLink Labs) with this functionality to output a PNG or get a web view. This image contains scripts that use Graphviz to generate the PNG image file.

This technique uses the Docker image at `dockerinpractice/docker-image-graph`. This image may go out of date over time and stop working, so you may want to run the following commands to ensure it's up to date.

Listing 6.1 Building an up-to-date `docker-image-graph` image (optional)

```
$ git clone https://github.com/docker-in-practice/docker-image-graph
$ cd docker-image-graph
$ docker build -t dockerinpractice/docker-image-graph
```

All you need to do in your run command is mount the Docker server socket and you're good to go, as the next listing shows.

Listing 6.2 Generating an image of your layer tree

Specifies an image and produces a PNG as an artifact	Removes the container when the image is produced	Mounts the Docker server's Unix domain socket so you can access the Docker server from within the container. If you've changed the default for the Docker daemon, this won't work.
<pre>\$ docker run --rm \ -v /var/run/docker.sock:/var/run/docker.sock \ dockerinpractice/docker-image-graph > docker_images.png</pre>		

Figure 6.2 shows a PNG of an image tree from one of our machines. You can see from this figure that the node and golang:1.3 images share a common root, and that the golang:runtime only shares the global root with the golang:1.3 image. Similarly, the mesosphere image is built from the same root as the ubuntu-upstart image.

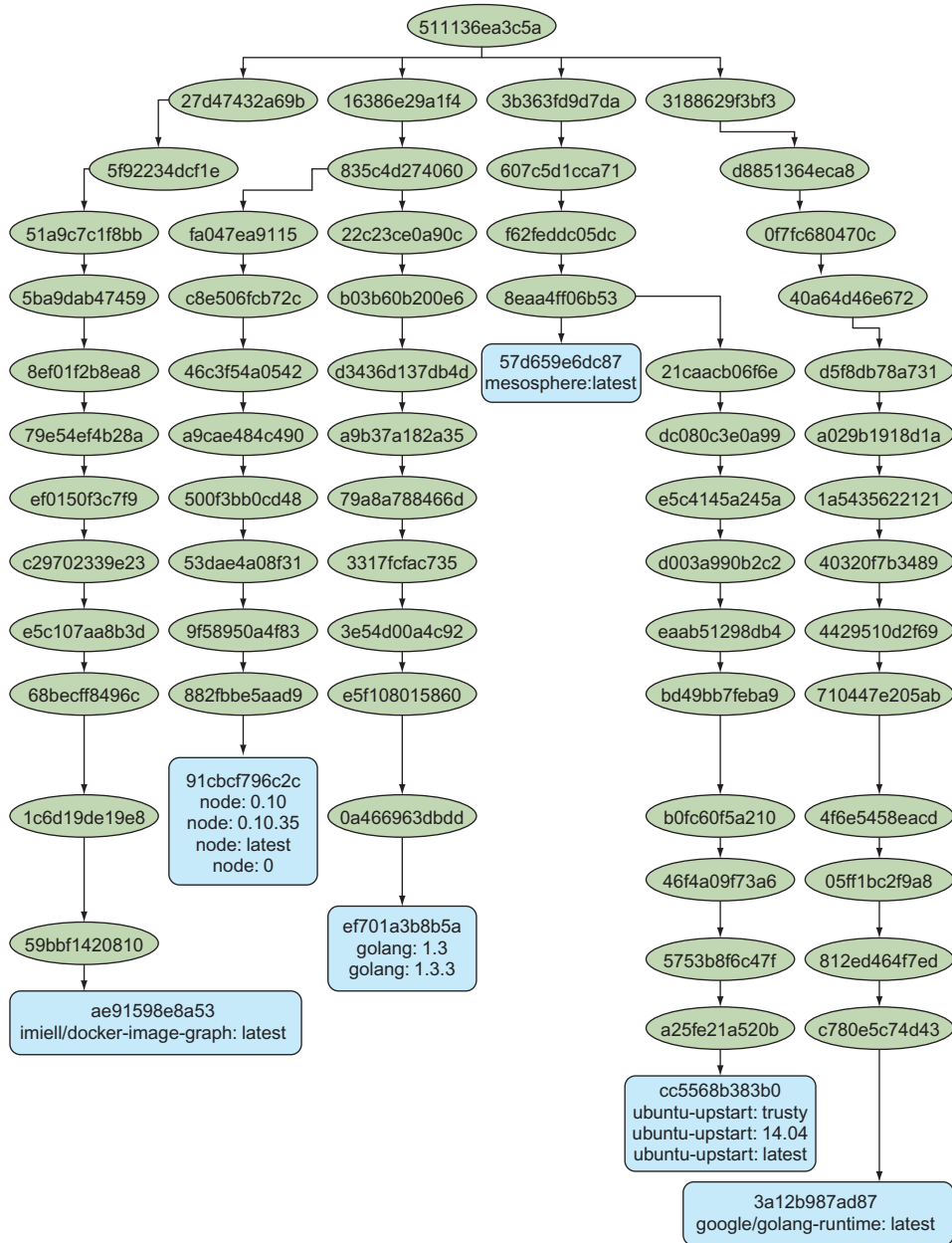


Figure 6.2 Image tree diagram

You may be wondering what the global root node on the tree is. This is the *scratch* pseudo-image, which is exactly 0 bytes in size.

DISCUSSION

When you start building more Docker images, perhaps as part of continuous delivery in chapter 9, it can be overwhelming to keep track of the history of an image and what it's built on. This can be particularly important if you're trying to speed up your delivery by sharing more layers to optimize for size. Periodically pulling all your images and generating a graph can be a great way to keep track.

TECHNIQUE 47 Direct action: Executing commands on your container

In the early days of Docker, many users added SSH servers to their images so that they could access them with a shell from outside. This was frowned upon by Docker, as it treated the container as a VM (and we know that containers aren't VMs) and added process overhead to a system that shouldn't need it. Many objected that once started, there was no easy way to get into a container. As a result, Docker introduced the `exec` command, which was a much neater solution to the problem of affecting and inspecting the internals of containers once started. It's this command that we'll discuss here.

PROBLEM

You want to perform commands on a running container.

SOLUTION

Use the `docker exec` command.

The following command starts a container in the background (with `-d`) and tells it to sleep forever (do nothing). We'll name this command `sleeper`.

```
docker run -d --name sleeper debian sleep infinity
```

Now that you've started a container, you can perform various actions on it using Docker's `exec` command. The command can be viewed as having three basic modes, listed in table 6.1.

Table 6.1 Docker `exec` modes

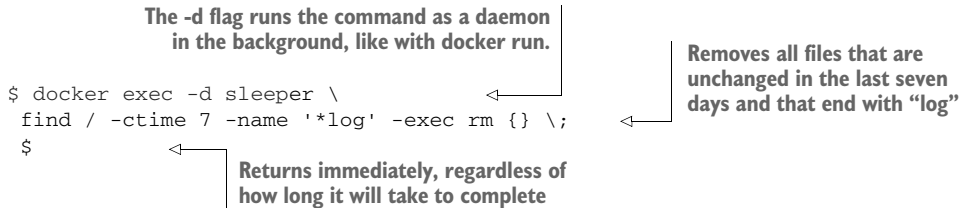
Mode	Description
Basic	Runs the command in the container synchronously on the command line
Daemon	Runs the command in the background on the container
Interactive	Runs the command and allows the user to interact with it

First we'll cover the basic mode. The following command runs an `echo` command inside our `sleeper` container.

```
$ docker exec sleeper echo "hello host from container"
hello host from container
```

Note that the structure of this command is very similar to the `docker run` command, but instead of the ID of an image, we give it the ID of a running container. The `echo` command refers to the `echo` binary within the container, not outside.

Daemon mode runs the command in the background; you won't see the output in your terminal. This might be useful for regular housekeeping tasks, where you want to fire the command and forget, such as cleaning up log files:



```
$ docker exec -d sleeper \
  find / -ctime 7 -name '*log' -exec rm {} \;
$
```

The `-d` flag runs the command as a daemon in the background, like with `docker run`.

Removes all files that are unchanged in the last seven days and that end with “log”

Returns immediately, regardless of how long it will take to complete

Finally, we have interactive mode. This allows you to run whatever commands you like from within the container. To enable this, you'll usually want to specify that the shell should run interactively, which in the following code is `bash`:

```
$ docker exec -i -t sleeper /bin/bash
root@d46dc042480f:/#
```

The `-i` and `-t` arguments do the same thing you're familiar with from `docker run`—they make the command interactive and set up a TTY device so shells will function correctly. After running this, you'll have a prompt running inside the container.

DISCUSSION

Jumping into a container is an essential debugging step when something is going wrong, or if you want to figure out what a container is doing. It's often not possible to use the `attach` and `detach` method enabled by technique 44 because processes in containers are typically run in the foreground, making it impossible to get access to a shell prompt. Because `exec` allows you to specify the binary you want to run, this isn't an issue ... as long as the container filesystem actually has the binary you want to run.

In particular, if you've used technique 58 to create a container with a single binary, you won't be able to start a shell. In this case you may want to stick with technique 57 as a low-overhead way to permit `exec`.

TECHNIQUE 48 Are you in a Docker container?

When creating containers, it's common to put logic inside shell scripts rather than trying to directly write the script in a `Dockerfile`. Or you may have assorted scripts for use while the container is running. Either way, the tasks these perform are often carefully customized for use inside a container and can be damaging to run on a “normal” machine. In situations like this, it's useful to have some safety rails to prevent accidental execution outside a container.

PROBLEM

Your code needs to know whether you're operating from within a Docker container.

SOLUTION

Check for the existence of the `/.dockerenv` file. If it exists, you're likely in a Docker container.

Note that this isn't a cast-iron guarantee—if anyone, or anything, removed the `/.dockerenv` file, this check could give misleading results. These scenarios are unlikely, but at worst you'll get a false positive with no ill effects; you'll think you're not in a Docker container and at worst *won't* run a potentially destructive piece of code.

A more realistic scenario is that this undocumented behavior of Docker has been altered or removed in a newer version of Docker (or you're using a version from before the behavior was first implemented).

The code might be part of a startup bash script, as in the following listing, followed by the remainder of your startup script code.

Listing 6.3 Shell script fails if it's run outside a container

```
#!/bin/bash
if ! [ -f /.dockerenv ]
then
    echo 'Not in a Docker container, exiting.'
    exit 1
fi
```

Of course, the opposite logic could be used to determine that you are *not* running within a container, if that's your need:

Listing 6.4 Shell script fails if it's run inside a container

```
#!/bin/bash
if [ -f /.dockerenv ]
then
    echo 'In a Docker container, exiting.'
    exit 1
fi
```

This example uses bash to determine the existence of the file, but the vast majority of programming languages will have their own ways to determine the existence of files on the container (or host) filesystem.

DISCUSSION

You may be wondering how often this situation arises. It happens often enough for it to be a regular discussion point on Docker forums, where somewhat religious arguments flare up about whether this is a valid use case, or whether something else in your application's design is amiss.

Leaving these discussions aside, you can easily end up in a situation where you need to switch your code path depending on whether you're in a Docker container or not. One such example we've experienced is when using a Makefile to build a container.

Summary

- You can configure your machine to let you run Docker without `sudo`.
- Use the built-in Docker commands to clean up unused containers and volumes.
- External tools can be used to expose information about your containers in new ways.
- The `docker exec` command is the correct way to get inside a running container—resist installing SSH.