5 Running containers

This chapter covers

- Using GUI applications within Docker
- Retrieving information about containers
- The different ways you can terminate containers
- Starting containers on a remote machine
- Using and managing Docker volumes for persistent shared data
- Learning your first Docker patterns: the data and dev tools containers

You can't get very far without running containers when using Docker, and there's a lot to understand if you want to use the full power they make available.

This chapter will look at some of the details involved in running containers, examine some concrete use cases, and provide a thorough treatment of the possibilities enabled by volumes along the way.

5.1 Running containers

Although much of this book is about running containers, there are some practical techniques related to running containers on your host that may not be immediately obvious. We'll look at how you can get GUI applications working, start a container on a remote machine, inspect the state of containers and their source images, shut down containers, manage Docker daemons on remote machines, and use a wildcard DNS service to make testing easier.

TECHNIQUE 29 Running GUIs within Docker

You've already seen a GUI served from within a Docker container using a VNC server in technique 19. That's one way to view applications within your Docker container, and it's self-contained, requiring only a VNC client to use.

Fortunately there's a more lightweight and well-integrated way to run GUIs on your desktop, but it requires more setup on your part. It mounts the directory on the host that manages communications with the X server, so that it's accessible to the container.

PROBLEM

You want to run GUIs in a container as though they were normal desktop apps.

SOLUTION

Create an image with your user credentials and the program, and bind mount your X server to it.

Figure 5.1 shows how the final setup will work.

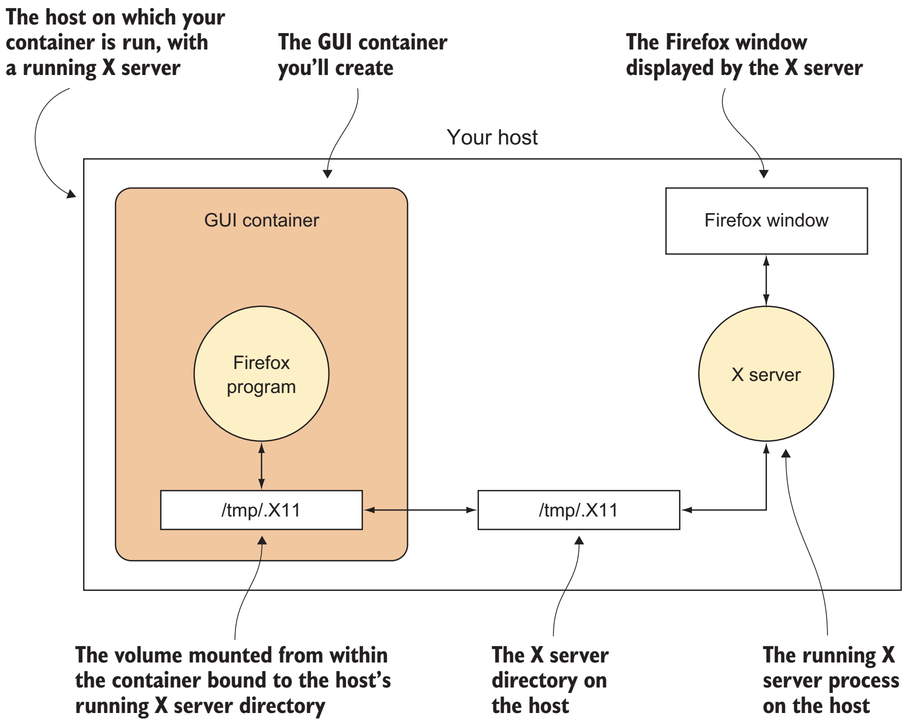
The host on which your container is run, with a running X server
The GUI container you'll create
The Firefox window displayed by the X server
Your host
GUI container
Firefox window
Firefox program
X server
/tmp/.X11
/tmp/.X11
The volume mounted from within the container bound to the host's running X server directory
The X server directory on the host
The running X server process on the host

Figure 5.1 Communicating with the host's X server

The container is linked to the host via the mount of the host's /tmp/.X11 directory, and this is how the container can perform actions on the host's desktop.

First make a new directory somewhere convenient, and determine your user and group IDs with the `id` command, as shown in the following listing.

Listing 5.1 Setting up a directory and finding out your user details

Gets information about your user that you'll need for the Dockerfile

Note your user ID (uid). In this case, it's 1000.

Note your group ID (gid). In this case, it's 1000.

```
$ mkdir dockergui
$ cd dockergui
$ id
 uid=1000(dockerinpractice) \
 gid=1000(dockerinpractice) \
 groups=1000(dockerinpractice),10(wheel),989(vboxusers),990(docker)
```

Now create a file called Dockerfile as follows.

Listing 5.2 Firefox in a Dockerfile

```
FROM ubuntu:14.04

RUN apt-get update
RUN apt-get install -y firefox

RUN groupadd -g GID USERNAME
 RUN useradd -d /home/USERNAME -s /bin/bash \
-m USERNAME -u UID -g GID
 USER USERNAME
 ENV HOME /home/USERNAME
 CMD /usr/bin/firefox
```

Installs Firefox as the GUI app. You can change this to whatever application(s) you may want.

Adds your host's group to the image. Replace GID with your group ID and USERNAME with your username.

The image should run as the user you've created. Replace USERNAME with your username.

Adds your user account to the image. Replace USERNAME with your username, UID with your user ID, and GID with your group ID.

Sets the HOME variable. Replace USERNAME with your username.

Runs Firefox on startup by default

Now you can build from that Dockerfile and tag the result as "gui":

```
$ docker build -t gui .
```

Run it as follows:

Bind mounts the X server directory to the container

Sets the DISPLAY variable in the container to be the same as that used in the host, so the program knows which X server to talk to

```
docker run -v /tmp/.X11-unix:/tmp/.X11-unix \
 -h $HOSTNAME -v $HOME/.Xauthority:/home/$USER/.Xauthority \
 -e DISPLAY=$DISPLAY gui
```

Gives the container the appropriate credentials

You'll see a Firefox window pop up!

DISCUSSION

You can use this technique to avoid mixing up your desktop work with your development work. With Firefox, for example, you might want to see how your application behaves with no web cache, bookmarks, or search history in a repeatable way for testing purposes. If you see error messages about being unable to open a display when trying to start the image and run Firefox, see technique 65 for other ways to allow containers to start graphical applications that are displayed on the host.

We understand that some people run almost all their applications inside Docker, including games! Although we don't go quite that far, it's useful to know that somebody has probably already encountered any problems you see.

TECHNIQUE 30 Inspecting containers

Although the Docker commands give you access to information about images and containers, sometimes you'll want to know more about the internal metadata of these Docker objects.

PROBLEM

You want to find out a container's IP address.

SOLUTION

Use the `docker inspect` command.

The `docker inspect` command gives you access to Docker's internal metadata in JSON format, including the IP address. This command produces a lot of output, so only a brief snippet of an image's metadata is shown here.

Listing 5.3 Raw `inspect` output on an image

```
$ docker inspect ubuntu | head
[{
    "Architecture": "amd64",
    "Author": "",
    "Comment": "",
    "Config": {
        "AttachStderr": false,
        "AttachStdin": false,
        "AttachStdout": false,
        "Cmd": [
            "/bin/bash"
$
```

You can inspect images and containers by name or ID. Obviously, their metadata will differ—for example, a container will have runtime fields such as "State" that the image will lack (an image has no state).

In this case, you want to find out a container's IP address on your host. To do this, you can use the `docker inspect` command with the `format` flag.

Listing 5.4 Determining a container's IP address

```
docker inspect \
 --format '{{.NetworkSettings.IPAddress}}' \
 0808ef13d450
```

The docker inspect command

The format flag. This uses Go templates (not covered here) to format the output. Here, the IPAddress field is taken from the NetworkSettings field in the inspect output.

The ID of the Docker item you want to inspect

This technique can be useful for automation, as the interface is likely to be more stable than that of other Docker commands.

The following command gives you the IP addresses of all running containers and pings them.

Listing 5.5 Getting IP addresses of running containers and pinging each in turn

```
$ docker ps -q | \
 xargs docker inspect --format='{{.NetworkSettings.IPAddress}}' | \
 xargs -l1 ping -c1
 PING 172.17.0.5 (172.17.0.5) 56(84) bytes of data.
64 bytes from 172.17.0.5: icmp_seq=1 ttl=64 time=0.095 ms

--- 172.17.0.5 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.095/0.095/0.095/0.000 ms
```

Gets the container IDs of all running containers

Runs the inspect command against all container IDs to get their IP addresses

Takes each IP address and runs ping against each in turn

Note that because `ping` only accepts one IP address, we had to pass an additional argument to `xargs` telling it to run the command for each individual line.

> **TIP** If you have no running containers, run this command to get one going: `docker run -d ubuntu sleep 1000`.

DISCUSSION

Inspecting containers and the method of jumping into containers in technique 47 are likely the two most important tools in your inventory for debugging why containers aren't working. Inspect shines most when you believe you've started a container configured in a particular way but it behaves unexpectedly—your first step should be to inspect the container to verify that Docker agrees with your expectation of the port and volume mappings of the container, among other things.

TECHNIQUE 31 Cleanly killing containers

If the state of a container is important to you when it terminates, you may want to understand the distinction between `docker kill` and `docker stop`. This distinction can also be important if you need your applications to close gracefully in order to save data.

PROBLEM

You want to cleanly terminate a container.

SOLUTION

Use `docker stop` rather than `docker kill` to cleanly terminate the container.

The crucial point to understand is that `docker kill` doesn't behave in the same way as the standard command-line `kill` program.

The `kill` program works by sending a `TERM` (a.k.a. signal value 15) signal to the process specified, unless directed otherwise. This signal indicates to the program that it should terminate, but it doesn't force the program. Most programs will perform some kind of cleanup when this signal is handled, but the program can do what it likes—including ignoring the signal.

A `KILL` signal (a.k.a. signal value 9), by contrast, forces the specified program to terminate.

Confusingly, `docker kill` uses a `KILL` signal on the running process, giving the processes within it no chance to handle the termination. This means that stray files, such as files containing running process IDs, may be left in the filesystem. Depending on the application's ability to manage state, this may or may not cause problems for you if you start up the container again.

Even more confusingly, the `docker stop` command acts like the standard `kill` command, sending a `TERM` signal (see table 5.1), except it will wait for 10 seconds and then send the `KILL` signal if the container hasn't stopped.

Table 5.1 Stopping and killing

Command	Default signal	Default signal value
`kill`	`TERM`	15
`docker kill`	`KILL`	9
`docker stop`	`TERM`	15

In summary, don't use `docker kill` as you'd use `kill`. You're probably best off getting into the habit of using `docker stop`.

DISCUSSION

Although we recommend `docker stop` for everyday use, `docker kill` has some additional configurability that allows you to choose the signal sent to the container via the `--signal` argument. As discussed, the default is `KILL`, but you can also send `TERM` or one of the less common Unix signals.

If you're writing your own application that you'll start in a container, the `USR1` signal may be of interest. This is explicitly reserved for applications to do whatever they want with it, and in some places it's used as an indication to print out progress information, or the equivalent—you could use it for whatever you see fit. `HUP` is another popular one, conventionally interpreted by servers and other long-running applications to trigger the reloading of configuration files and a "soft" restart. Of course, make sure you check the documentation of the application you're running before you start sending random signals to it!

TECHNIQUE 32 Using Docker Machine to provision Docker hosts

Setting up Docker on your local machine was probably not too difficult—there's a script you can use for convenience, or you can use a few commands to add the appropriate sources for your package manager. But this can get tedious when you're trying to manage Docker installs on other hosts.

PROBLEM

You want to spin up containers on a separate Docker host from your machine.

SOLUTION

Docker Machine is the official solution for managing Docker installs on remote machines.

This technique will be useful if you need to run Docker containers on multiple external hosts. You may want this for a number of reasons: to test networking between Docker containers by provisioning a VM to run within your own physical host; to provision containers on a more powerful machine through a VPS provider; to risk trashing a host with some kind of crazy experiment; to have the choice of running on multiple cloud providers. Whatever the reason, Docker Machine is probably the answer for you. It's also the gateway to more sophisticated orchestration tools like Docker Swarm.

WHAT DOCKER MACHINE IS

Docker Machine is mainly a convenience program. It wraps a lot of potentially tortuous instructions around provisioning external hosts and turns them into a few easy-to-use commands. If you're familiar with Vagrant, it has a similar feel: provisioning and managing other machine environments is made simpler with a consistent interface. If you cast your mind back to our architecture overview in chapter 2, one way of viewing Docker Machine is to imagine that it's facilitating the management of different Docker daemons from one client (see figure 5.2).

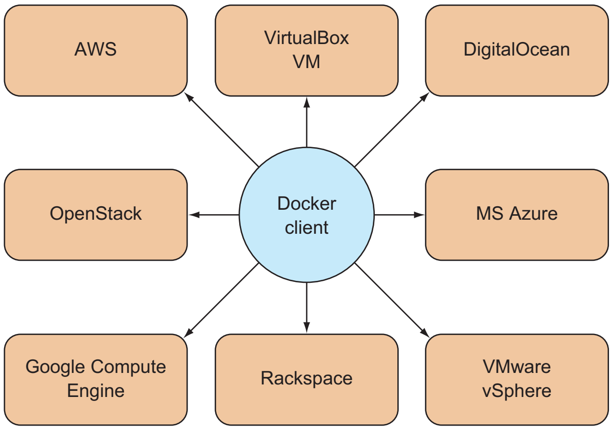

Figure 5.2 Docker Machine as a client of external hosts

The list of Docker host providers in figure 5.2 isn't exhaustive, and it's likely to grow. At the time of writing, the following drivers are available, which allow you to provision the given host provider:

- Amazon Web Services
- DigitalOcean
- Google Compute Engine
- IBM SoftLayer
- Microsoft Azure
- Microsoft Hyper-V
- OpenStack
- Oracle VirtualBox
- Rackspace
- VMware Fusion
- VMware vCloud Air
- VMware vSphere

The options that must be specified to provision a machine will vary greatly depending on the functionality provided by the driver. At one end, provisioning an Oracle VirtualBox VM on your machine has only 3 flags available to `create`, compared with OpenStack's 17.

NOTE It's worth clarifying that Docker Machine is not any kind of clustering solution for Docker. Other tools, such as Docker Swarm, fulfill that function, and we'll look at them later.

INSTALLATION

Installation involves a straightforward binary. Download links and installation instructions for different architectures are available here: https://github.com/docker/machine/releases.

NOTE You may want to move the binary to a standard location, like /usr/bin, and ensure it's renamed or symlinked to `docker-machine` before continuing, as the downloaded file may have a longer name suffixed with the binary's architecture.

USING DOCKER MACHINE

To demonstrate Docker Machine's use, you can start by creating a VM with a Docker daemon on it that you can work with.

NOTE You'll need to have Oracle's VirtualBox installed for this to work. It's widely available in most package managers.

Use docker-machine's create subcommand to create a new host and specify its type with the --driver flag. The host has been named `host1`.

```
$ docker-machine create --driver virtualbox host1
 INFO[0000] Creating CA: /home/imiell/.docker/machine/certs/ca.pem
INFO[0000] Creating client certificate:
➥ /home/imiell/.docker/machine/certs/cert.pem
INFO[0002] Downloading boot2docker.iso to /home/imiell/.docker/machine/cache/
➥ boot2docker.iso...
INFO[0011] Creating VirtualBox VM...
INFO[0023] Starting VirtualBox VM...
INFO[0025] Waiting for VM to start...
```

```
INFO[0043] "host1" has been created and is now the active machine.
 INFO[0043] To point your Docker client at it, run this in your shell:

➥ $(docker-machine env host1)
```

Your machine is now created.

Run this command to set the DOCKER_HOST environment variable, which sets the default host that Docker commands will be run on

Vagrant users will feel right at home here. By running these commands, you've created a machine that you can now manage Docker on. If you follow the instructions given in the output, you can SSH directly to the new VM:

These variables handle the security side of connections to the new host.

The $() takes the output of the docker-machine env command and applies it to your environment. docker-machine env outputs a set of commands that you can use to set the default host for Docker commands.

The environment variable names are all prefixed with DOCKER_.

The DOCKER_HOST variable is the endpoint of the Docker daemon on the VM.

```
$ eval $(docker-machine env host1)
 $ env | grep DOCKER
 DOCKER_HOST=tcp://192.168.99.101:2376
 DOCKER_TLS_VERIFY=yes
 DOCKER_CERT_PATH=/home/imiell/.docker/machine/machines/host1
 DOCKER_MACHINE_NAME=host1
$ docker ps -a
 CONTAINER ID   IMAGE    COMMAND   CREATED     STATUS   PORTS      NAMES
$ docker-machine ssh host1
                        ##        .
                  ## ## ##       ==
               ## ## ## ##      ===
           /""""""""""""""""\___/ ===
      ~~~ {~~ ~~~~ ~~~ ~~~~ ~~ ~ /  ===- ~~~
           \______ o          __/
             \    \        __/
              \____\______/
 _                 _   ____     _            _
| |__   ___   ___ | |_|___ \ __| | ___   ___| | _____ _ __
| '_ \ / _ \ / _ \| __| __) / _` |/ _ \ / __| |/ / _ \ '__|
| |_) | (_) | (_) | |_ / __/ (_| | (_) | (__|   <  __/ |
|_.__/ \___/ \___/ \__|_____\__,_|\___/ \___|_|\_\___|_|
Boot2Docker version 1.5.0, build master : a66bce5 -
      Tue Feb 10 23:31:27 UTC 2015
Docker version 1.5.0, build a8a31ef
docker@host1:~$
```

The docker command is now pointed at the VM host you've created, not at your previously used host machine. You've created no containers on the new VM, so there's no output.

The ssh subcommand will take you directly to the new VM itself.

MANAGING HOSTS

Managing multiple Docker hosts from one client machine can make it difficult to track what's going on. Docker Machine comes with various management commands to make this simpler, as shown in table 5.2.

Table 5.2 List of docker-machine commands

Subcommand	Action
`create`	Creates a new machine
`ls`	Lists the Docker host machines
`stop`	Stops the machine
`start`	Starts a machine
`restart`	Stops and starts a machine
`rm`	Destroys a machine
`kill`	Kills a machine off
`inspect`	Returns a JSON representation of the machine's metadata
`config`	Returns the configuration required to connect to the machine
`ip`	Returns the IP address of the machine
`url`	Returns a URL for the Docker daemon on the machine
`upgrade`	Upgrades the Docker version on the host to the latest

The following example lists two machines. The active machine is listed with an asterisk, and it has a state associated with it, analogous to the state of containers or processes:

```
$ docker-machine ls
NAME   ACTIVE   DRIVER      STATE     URL                         SWARM
host1           virtualbox  Running   tcp://192.168.99.102:2376
host2  *        virtualbox  Running   tcp://192.168.99.103:2376
```

TIP You may be wondering how to switch back to your original host machine Docker instance. At the time of writing we haven't found a simple way to do this. You can either `docker-machine rm` all the machines, or if that's not an option you can manually unset the environment variables previously set with `unset DOCKER_HOST DOCKER_TLS_VERIFY DOCKER_CERT_PATH`.

DISCUSSION

You can look at this as turning machines into processes, much like Docker itself can be seen as turning environments into processes.

It may be tempting to use a Docker Machine setup to manually manage containers across multiple hosts, but if you find yourself manually taking down containers, rebuilding them, and starting them back up again on code changes, we encourage you to look at part 4 of this book. Tedious tasks like this can be done by computers perfectly well. Technique 87 covers the official solution from Docker Inc. for creating an automatic cluster of containers. Technique 84 may be appealing if you like the idea of the unified view of a cluster, but also prefer to retain ultimate control over where your containers end up running.

TECHNIQUE 33 Wildcard DNS

When working with Docker, it's very common to have many containers running that need to refer to a central or external service. When testing or developing such systems, it's normal to use a static IP address for these services. But for many Docker-based systems, such as OpenShift, an IP address isn't sufficient. Such applications demand that there's a DNS lookup.

The usual solution to this is to edit your /etc/hosts file on the hosts you're running your services on. But this isn't always possible. For example, you might not have access to edit the file. Neither is it always practical. You might have too many hosts to maintain, or other bespoke DNS lookup caches might get in the way.

In these cases, there's a solution that uses "real" DNS servers.

PROBLEM

You need a DNS-resolvable URL for a specific IP address.

SOLUTION

Use the NIP.IO web service to resolve an IP address to a DNS-resolvable URL without any DNS setup.

This one is really simple. NIP.IO is a web-based service that turns an IP address into a URL automatically for you. You just need to replace the "IP" section of the URL "http://IP.nip.io" with your desired IP address.

Let's say the IP address you want a URL to resolve to is "10.0.0.1". Your URL could look like this,

http://myappname.10.0.0.1.nip.io

where `myappname` refers to your preferred name for your application, `10.0.0.1` refers to the IP address you want the URL to resolve to, and `nip.io` is the "real" domain on the internet that manages this DNS lookup service.

The `myappname.` part is optional, so this URL would resolve to the same IP address:

http://10.0.0.1.nip.io

DISCUSSION

This technique is handy in all sorts of contexts, not just when using Docker-based services.

It should be obvious that this technique isn't suitable for production or proper UAT environments, because it submits DNS requests to a third party and reveals information about your internal IP address layout. But it can be a very handy tool for development work.

If you're using this service with HTTPS, make sure that the URL (or a suitable wildcard) is baked into the certificate you use.

5.2 Volumes—a persistent problem

Containers are a powerful concept, but sometimes not everything you want to access is ready to be encapsulated. You may have a reference Oracle database stored on a large cluster that you want to connect to for testing. Or maybe you have a large legacy server already set up with binaries that can't easily be reproduced.

When you begin working with Docker, most of the things you'll want to access will likely be data and programs external to your container. We'll take you from the straightforward mounting of files from your host to more sophisticated container patterns: the data container and the dev tools container. We'll also demonstrate a pragmatic favorite of ours for remote mounting across a network that requires only an SSH connection to work, and we'll look at a means of sharing data with other users via the BitTorrent protocol.

Volumes are a core part of Docker, and the issue of external data reference is yet another fast-changing area of the Docker ecosystem.

TECHNIQUE 34 Docker volumes: Problems of persistence

Much of the power of containers comes from the fact that they encapsulate as much of the state of the environment's filesystem as is useful.

Sometimes, though, you don't want to put files into a container. You might have some large files that you want to share across containers or manage separately. The classic example is a large centralized database that you want your container to access, but you also want other (perhaps more traditional) clients to access alongside your newfangled containers.

The solution is *volumes*, Docker's mechanism for managing files outside the lifecycle of the container. Although this goes against the philosophy of containers being "deployed anywhere" (you won't be able to deploy your database-dependent container where there's no compatible database available to mount, for example), it's a useful feature for real-world Docker use.

PROBLEM

You want to access files on the host from within a container.

SOLUTION

Use Docker's volume flag to access host files from within the container. Figure 5.3 illustrates the use of a volume flag to interact with the host's filesystem.

The following command shows the host's /var/db/tables directory being mounted on /var/data1, and it could be run to start the container in figure 5.3.

```
$ docker run -v /var/db/tables:/var/data1 -it debian bash
```

The -v flag (`--volume` in longhand) indicates that a volume external to the container is required. The subsequent argument gives the volume specification in the form of two directories separated by a colon, instructing Docker to map the external /var/db/tables directory to the container's /var/data1 directory. Both the external and container directories will be created if they don't exist.

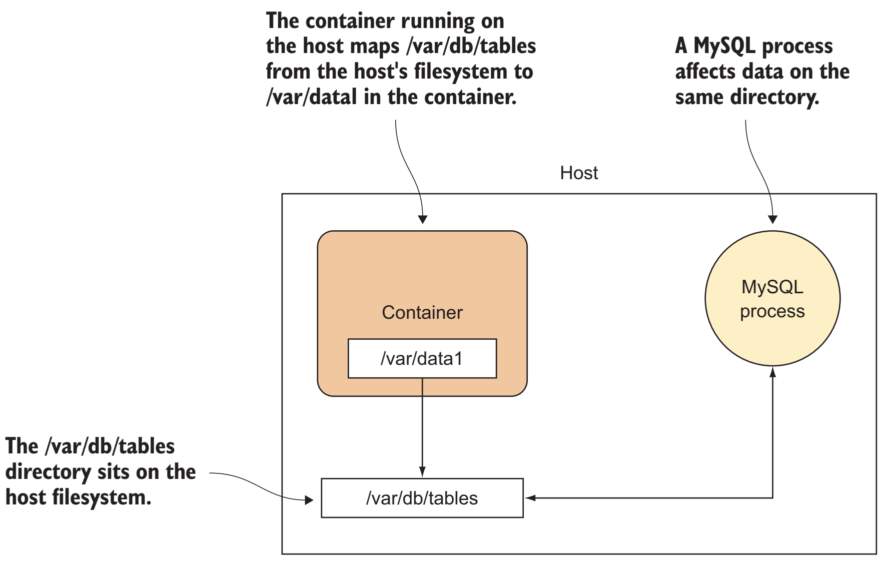
The container running on the host maps /var/db/tables from the host's filesystem to /var/datal in the container.
A MySQL process affects data on the same directory.
Host
Container
/var/data1
MySQL process
The /var/db/tables directory sits on the host filesystem.
/var/db/tables

Figure 5.3 A volume inside a container

Beware of mapping over existing directories. The container's directory will be mapped even if it already exists in the image. This means that the directory you're mapping to within the container will effectively disappear. Fun things happen if you try to map a key directory! Try mounting an empty directory over /bin, for example.

Also note that volumes are assumed not to persist in Dockerfiles. If you add a volume and then make changes to that folder within a Dockerfile, the changes won't be persisted to the resulting image.

> **WARNING** You may run into difficulties if your host runs SELinux. If SELinux policy is enforced, the container may not be able to write to the /var/db/tables directory. You'll see a "permission denied" error. If you need to work around this, you'll have to talk to your sysadmin (if you have one) or switch off SELinux (for development purposes only). See technique 113 for more on SELinux.

DISCUSSION

Exposing files from the host in a container is one of the most common operations we perform when experimenting with individual containers—containers are intended to be ephemeral, and it's all too easy to blow one away after spending a significant amount of time working on some files within one. Better to be confident that the files are safe, come what may.

There's also the advantage that the normal overhead of copying files into containers with the method in technique 114 is simply not present. Databases like the one in technique 77 are the obvious beneficiary if they grow large.

Finally, you'll see a number of techniques that use `-v /var/run/docker.sock:/var/run/docker.sock`, one of the many being technique 45. This exposes the special Unix socket file to the container and demonstrates an important capability of this technique—you aren't limited to so-called "regular" files—you can also permit more unusual filesystem-based use cases. But if you encounter permissions issues with device nodes (for example), you may need to refer to technique 93 to get a handle on what the `--privileged` flag does.

TECHNIQUE 35 Distributed volumes with Resilio Sync

When experimenting with Docker in a team, you may want to share large quantities of data among team members, but you may not be allocated the resources for a shared server with sufficient capacity. The lazy solution to this is copying the latest files from other team members when you need them—this quickly gets out of hand for a larger team.

The solution is to use a decentralized tool for sharing files—no dedicated resource required.

PROBLEM

You want to share volumes across hosts over the internet.

SOLUTION

Use a technology called Resilio to share volumes over the internet.

Figure 5.4 illustrates the setup you're aiming for.

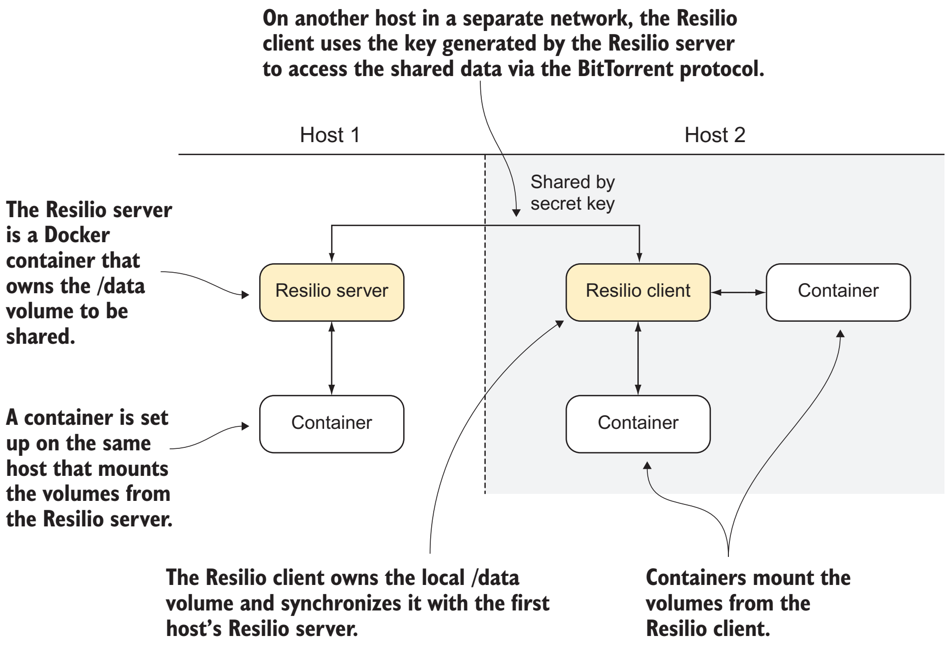
On another host in a separate network, the Resilio client uses the key generated by the Resilio server to access the shared data via the BitTorrent protocol.
Host 1
Host 2
Shared by secret key
The Resilio server is a Docker container that owns the /data volume to be shared.
Resilio server
Resilio client
Container
Container
Container
A container is set up on the same host that mounts the volumes from the Resilio server.
The Resilio client owns the local /data volume and synchronizes it with the first host's Resilio server.
Containers mount the volumes from the Resilio client.

Figure 5.4 Using Resilio

The end result is a volume (`/data`) conveniently synchronized over the internet without requiring any complicated setup.

On your primary server, run the following commands to set up the containers on the first host:

Runs the published ctlc/btsync image as a daemon container, calls the btsync binary, and opens the required ports

Make a note of this key—it will be different for your run.

Gets the output of the resilio container so you can make a note of the key

```
[host1]$ docker run -d -p 8888:8888 -p 55555:55555 \
 --name resilio ctlc/btsync
$ docker logs resilio
 Starting btsync with secret: \
ALSVEUABQQ5ILRS2OQJKAOKCU5SIIP6A3
 By using this application, you agree to our Privacy Policy and Terms.
http://www.bittorrent.com/legal/privacy
http://www.bittorrent.com/legal/terms-of-use

total physical memory 536870912 max disk cache 2097152
Using IP address 172.17.4.121

[host1]$ docker run -i -t --volumes-from resilio \
 ubuntu /bin/bash
$ touch /data/shared_from_server_one
 $ ls /data
shared_from_server_one
```

Adds a file to the /data volume

Starts up an interactive container with the volumes from the resilio server

On the second server, open up a terminal and run these commands to synchronize the volume:

Starts an interactive container that mounts the volumes from your client daemon

Starts a resilio client container as a daemon with the key generated by the daemon run on host1

```
[host2]$ docker run -d --name resilio-client -p 8888:8888 \
-p 55555:55555 \
ctlc/btsync ALSVEUABQQ5ILRS2OQJKAOKCU5SIIP6A3
 [host2]$ docker run -i -t --volumes-from resilio-client \
ubuntu bash
 $ ls /data
shared_from_server_one
 $ touch /data/shared_from_server_two
 $ ls /data
shared_from_server_one  shared_from_server_two
```

The file created on host1 has been transferred to host2.

Creates a second file on host2

Back on host1's running container, you should see that the file has been synchronized between the hosts exactly as the first file was:

```
[host1]$ ls /data
shared_from_server_one  shared_from_server_two
```

DISCUSSION

The synchronization of files comes with no timing guarantees, so you may have to wait for the data to sync. This is particularly the case for larger files.

WARNING Because the data may be sent over the internet and is processed by a protocol over which you have no control, you shouldn't rely on this technique if you have any meaningful security, scalability, or performance constraints.

We've only demonstrated that this technique works between two containers, as mentioned at the beginning, but it should also work across many members of a team. Aside from the obvious use case of large files that don't fit in version control, candidates for distribution include backups and possibly Docker images themselves, particularly if this technique is used in combination with an efficient compression mechanism like the one shown in technique 72. To avoid conflicts, make sure that images are always going in one direction (for example, from a build machine to many servers), or follow an agreed-upon process for performing updates.

TECHNIQUE 36 Retaining your container's bash history

Experimenting inside a container, knowing that you can wipe everything out when you're done, can be a liberating experience. But there are some conveniences that you lose when doing this. One that we've hit many times is forgetting a sequence of commands we've run inside a container.

PROBLEM

You want to share your container's bash history with your host's history.

SOLUTION

Use the `-e` flag, Docker mounts, and a bash alias to automatically share your container's bash history with the host's.

To understand this problem, we'll show you a simple scenario where losing this history is plain annoying.

Imagine you're experimenting in Docker containers, and in the midst of your work you do something interesting and reusable. We'll use a simple `echo` command for this example, but it could be a long and complex concatenation of programs that results in useful output:

```
$ docker run -ti --rm ubuntu /bin/bash
$ echo my amazing command
$ exit
```

After some time, you want to recall the incredible `echo` command you ran earlier. Unfortunately you can't remember it, and you no longer have the terminal session on your screen to scroll to. Out of habit, you try looking through your bash history on the host:

```
$ history | grep amazing
```

Nothing comes back, because the bash history is kept within the now-removed container and not the host you were returned to.

To share your bash history with the host, you can use a volume mount when running your Docker images. Here's an example:

```
$ docker run -e HIST_FILE=/root/.bash_history \
  -v=$HOME/.bash_history:/root/.bash_history \
  -ti ubuntu /bin/bash
```

Sets the environment variable picked up by bash. This ensures the bash history file used is the one you mount.

Maps the container's root's bash history file to the host's

TIP You may want to separate the container's bash history from your host's. One way to do this is to change the value for the first part of the preceding `-v` argument.

This is quite a handful to type every time, so to make this more user-friendly you can set up an alias by putting this into your ~/.bashrc file:

```
$ alias dockbash='docker run -e HIST_FILE=/root/.bash_history \
 -v=$HOME/.bash_history:/root/.bash_history
```

This still isn't seamless, because you have to remember to type `dockbash` if you want to perform a `docker run` command. For a more seamless experience, you can add these to your ~/.bashrc file:

Listing 5.6 Function alias to auto-mount host bash history

```
function basher() {
  if [[ $1 = 'run' ]]
  then
   shift
    /usr/bin/docker run \
      -e HIST_FILE=/root/.bash_history \
     -v $HOME/.bash_history:/root/.bash_history "$@"
  else
   /usr/bin/docker "$@"
  fi
}
alias docker=basher
```

Creates a bash function called basher that will handle the docker command

Determines whether the first argument to basher/docker is "run"

Removes that argument from the list of arguments you've passed in

Runs the docker run command you ran earlier, invoking the absolute path to the Docker runtime to avoid confusion with the following docker alias. The absolute path is discovered by running the "which docker" command on your host before implementing this solution.

Passes the arguments after "run" to the Docker runtime

Runs the docker command with the original arguments intact

Aliases the docker command when it's invoked on the command line to the basher function you've created. This ensures that the call to docker is caught before bash finds the docker binary on the path.

DISCUSSION

Now, when you next open a bash shell and run any `docker run` command, the commands that are run within that container will be added to your host's bash history. Make sure the path to Docker is correct. It might be located in /bin/docker, for example.

NOTE You'll need to log out of your host's original bash session for the history file to be updated. This is due to a subtlety of bash and how it updates the bash history it keeps in memory. If in doubt, exit all bash sessions you're aware of, and then start one up to ensure your history is as up-to-date as possible.

A number of command-line tools with prompts also store history, SQLite being one example (storing history in a .sqlite_history file). If you don't want to use the integrated logging solutions available in Docker described in technique 102, you could use a similar practice to make your application write to a file that ends up outside the container. Be aware that the complexities of logging, such as log rotation, mean that it may be simpler to use a log directory volume rather than just a file.

TECHNIQUE 37 Data containers

If you use volumes a lot on a host, managing the container's startup can get tricky. You may also want the data to be managed by Docker exclusively, and not be generally accessible on the host. One way to manage these things more cleanly is to use the data-only container design pattern.

PROBLEM

You want to use an external volume within a container, but you only want Docker to access the files.

SOLUTION

Start up a data container and use the `--volumes-from` flag when running other containers.

Figure 5.5 shows the structure of the data container pattern and explains how it works. The key thing to note is that in the second host, the containers don't need to know where the data is located on disk. All they need to know is the name of the data container, and they're good to go. This can make the operation of containers more portable.

Another benefit of this approach over the straightforward mapping of host directories is that access to these files is managed by Docker, which means that a non-Docker process is less likely to affect the contents.

NOTE People are commonly confused about whether the data-only container needs to run. It doesn't! It merely needs to exist, to have been run on the host, and not been deleted.

Let's go through a simple example so you can get a feel for how to use this technique.

First you run your data container:

```
$ docker run -v /shared-data --name dc busybox \
  touch /shared-data/somefile
```

The `-v` argument doesn't map the volume to a host directory, so it creates the directory within the scope of this container's responsibility. This directory is populated with a single file with `touch`, and the container immediately exists—a data container need

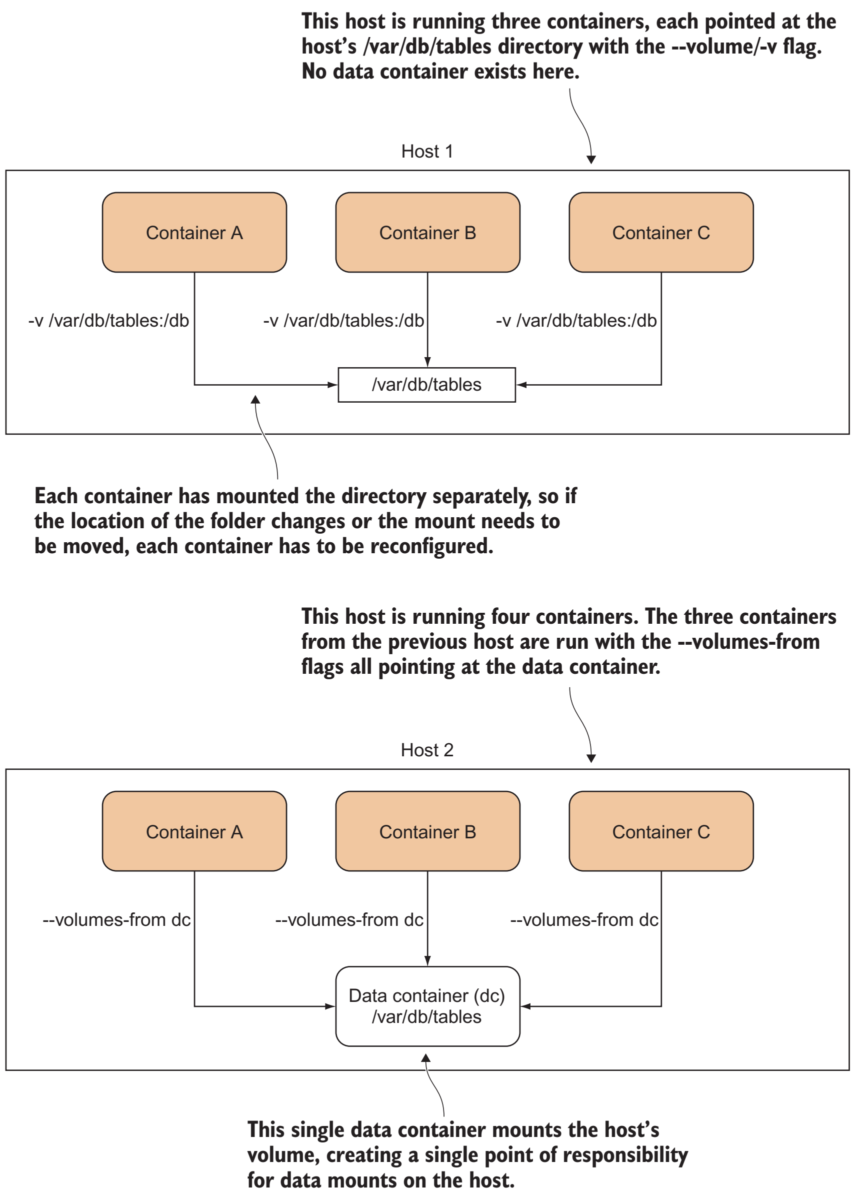
This host is running three containers, each pointed at the host's /var/db/tables directory with the --volume/-v flag. No data container exists here.
Host 1
Container A
Container B
Container C
-v /var/db/tables:/db
-v /var/db/tables:/db
-v /var/db/tables:/db
/var/db/tables
Each container has mounted the directory separately, so if the location of the folder changes or the mount needs to be moved, each container has to be reconfigured.
This host is running four containers. The three containers from the previous host are run with the --volumes-from flags all pointing at the data container.
Host 2
Container A
Container B
Container C
--volumes-from dc
--volumes-from dc
--volumes-from dc
Data container (dc) /var/db/tables
This single data container mounts the host's volume, creating a single point of responsibility for data mounts on the host.

Figure 5.5 The data container pattern

not be running to be used. We've used the small but functional busybox image to reduce the amount of extra baggage our data container needs.

Then you run up another container to access the file you just created:

```
docker run -t -i --volumes-from dc busybox /bin/sh
/ # ls /shared-data
somefile
```

DISCUSSION

The `--volumes-from` flag allows you to reference the files from the data container by mounting them in the current container—you just need to pass it the ID of a container with volumes defined. The busybox image doesn't have bash, so you need to start up a simpler shell to verify that the /shared-data folder from the `dc` container is available to you.

You can start up any number of containers, all reading from and writing to the specified data container's volumes.

You don't need to use this pattern in order to use volumes—you may find this approach harder to manage than a straightforward mount of a host directory. If, however, you like to cleanly delegate responsibility for managing data to a single point managed within Docker and uncontaminated by other host processes, data containers may be useful for you.

> **WARNING** If your application is logging from multiple containers to the same data container, it's important to ensure that each container log file writes to a unique file path. If you don't, different containers might overwrite or truncate the file, resulting in lost data, or they might write interleaved data, which is less easy to analyze. Similarly, if you invoke `--volumes-from` from a data container, you allow that container to potentially overlay directories over yours, so be careful of name clashes here.

It's important to understand that this pattern can result in heavy disk usage that can be relatively difficult to debug. Because Docker manages the volume within the data-only container and doesn't delete the volume when the last container referencing it has exited, any data on a volume will persist. This is to prevent undesired data loss. For advice on managing this, see technique 43.

TECHNIQUE 38 Remote volume mounting using SSHFS

We've discussed mounting local files, but soon the question of how to mount remote filesystems arises. Perhaps you want to share a reference database on a remote server and treat it as if it were local, for example.

Although it's theoretically possible to set up NFS on your host system and the server, and then access the filesystem by mounting that directory, there's a quicker and simpler way for most users that requires no setup on the server side (as long as there is SSH access).

> **NOTE** You'll need root privileges for this technique to work, and you'll need FUSE (Linux's "Filesystem in Userspace" kernel module) installed. You can determine whether you have the latter by running `ls /dev/fuse` in a terminal to see whether the file exists.

PROBLEM

You want to mount a remote filesystem without requiring any server-side configuration.

SOLUTION

Use a technology called SSHFS to mount the remote filesystem so that it appears to be local to your machine.

This technique works by using a FUSE kernel module with SSH to give you a standard interface to a filesystem, while in the background doing all communications via SSH. SSHFS also provides various behind-the-scenes features (such as remote file read-ahead) to facilitate the illusion that the files are local. The upshot is that once a user is logged into the remote server, they'll see the files as if they were local. Figure 5.6 helps explain this.

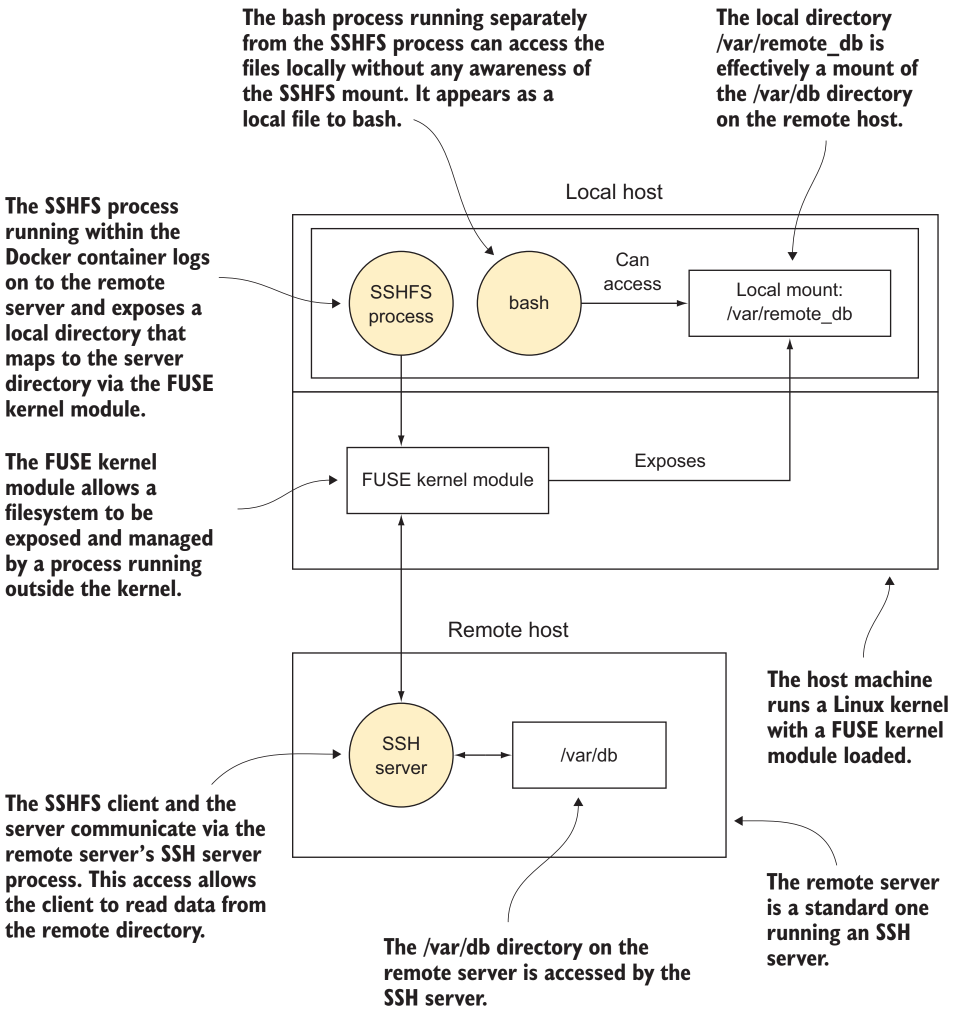
The bash process running separately from the SSHFS process can access the files locally without any awareness of the SSHFS mount. It appears as a local file to bash.
The local directory /var/remote_db is effectively a mount of the /var/db directory on the remote host.
Local host
The SSHFS process running within the Docker container logs on to the remote server and exposes a local directory that maps to the server directory via the FUSE kernel module.
SSHFS process
bash
Can access
Local mount: /var/remote_db
The FUSE kernel module allows a filesystem to be exposed and managed by a process running outside the kernel.
FUSE kernel module
Exposes
Remote host
The host machine runs a Linux kernel with a FUSE kernel module loaded.
SSH server
/var/db
The SSHFS client and the server communicate via the remote server's SSH server process. This access allows the client to read data from the remote directory.
The /var/db directory on the remote server is accessed by the SSH server.
The remote server is a standard one running an SSH server.

Figure 5.6 Mounting a remote filesystem with SSHFS

WARNING Although this technique doesn't use the Docker volumes functionality, and the files are visible through the filesystem, this technique doesn't give you any container-level persistence. Any changes made take place on the remote server's filesystem only.

You can get started by running the following commands, adjusted for your environment.

The first step is to start up a container with `--privileged` on your host machine:

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

Then, when it's started up, run `apt-get update && apt-get install sshfs` from within the container to install SSHFS.

When SSHFS is successfully installed, log on to the remote host as follows:

```
$ LOCALPATH=/path/to/local/directory
 $ mkdir $LOCALPATH
 $ sshfs user@host:/path/to/remote/directory $LOCALPATH
```

Choose a directory to mount the remote location into

Create the local directory to mount into

Replace the values here with your remote host username, remote host address, and remote path

You'll now see the contents of the path on the remote server in the folder you've just created.

TIP It's simplest to mount to a directory that you've newly created, but it's also possible to mount a pre-existing directory with files already present if you use the `-o nonempty` option. See the SSHFS man page for more information.

To cleanly unmount the files, use the `fusermount` command as follows, replacing the path as appropriate:

```
fusermount -u /path/to/local/directory
```

DISCUSSION

This is a great way to quickly get remote mounts working from within containers (and on standard Linux machines) with minimal effort.

Although we've only talked about SSHFS in this technique, successfully managing this opens up the wonderful (and sometimes weird) world of FUSE filesystems inside Docker. From storing your data inside Gmail to the distributed GlusterFS filesystem for storing petabytes of data across many machines, a number of opportunities open up to you.

TECHNIQUE 39 Sharing data over NFS

In a larger company, NFS shared directories will likely already be in use—NFS is a well-proven option for serving files out of a central location. For Docker to get traction, it's usually fairly important to be able to get access to these shared files.

Docker doesn't support NFS out of the box, and installing an NFS client on every container so you can mount the remote folders isn't considered a best practice. Instead, the suggested approach is to have one container act as a translator from NFS to a more Docker-friendly concept: volumes.

PROBLEM

You want seamless access to a remote filesystem over NFS.

SOLUTION

Use an infrastructure data container to broker access to your remote NFS filesystem.

This technique builds on technique 37, where we created a data container to manage data in a running system.

Figure 5.7 shows the idea of this technique in the abstract. The NFS server exposes the internal directory as the /export folder, which is bind-mounted on the host. The Docker host then mounts this folder using the NFS protocol to its /mnt folder. Then a so-called "infrastructure container" is created, which binds the mount folder.

This may seem a little over-engineered at first glance, but the benefit is that it provides a level of indirection as far as the Docker containers are concerned: all they need to do is mount the volumes from a pre-agreed infrastructure container, and whoever is responsible for the infrastructure can worry about the internal plumbing, availability, network, and so on.

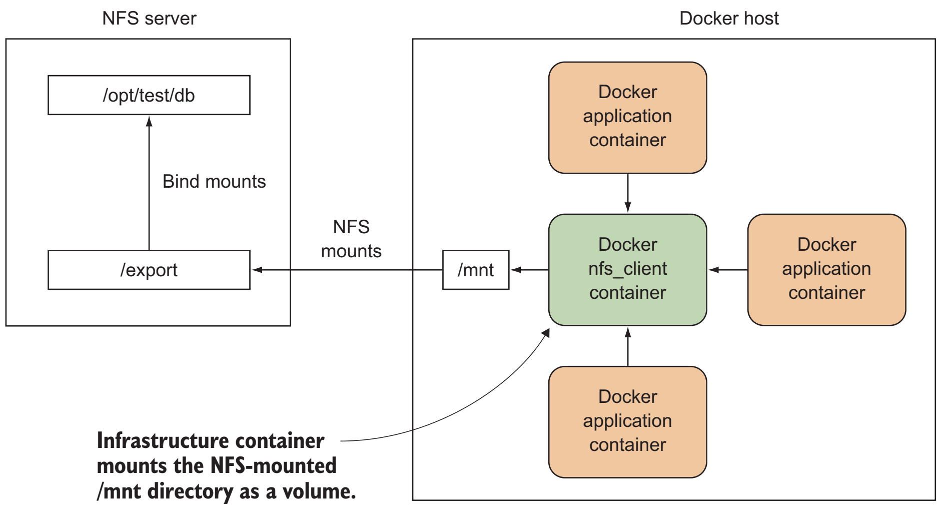
NFS server
Docker host
/opt/test/db
Bind mounts
/export
NFS mounts
/mnt
Docker application container
Docker nfs_client container
Docker application container
Docker application container
Infrastructure container mounts the NFS-mounted /mnt directory as a volume.

Figure 5.7 An infrastructure container that brokers NFS access

A thorough treatment of NFS is beyond the scope of this book. In this technique, we're just going to go through the steps of setting up such a share on a single host by having the NFS server's components on the same host as the Docker containers. This has been tested on Ubuntu 14.04.

Suppose you want to share the contents of your host's /opt/test/db folder, which contains the file mybigdb.db.

As root, install the NFS server and create an export directory with open permissions:

```
# apt-get install nfs-kernel-server
# mkdir /export
# chmod 777 /export
```

NOTE We've created the NFS share with open permissions, which is not a secure way to proceed for a production system. We've taken this approach in the interest of simplifying this tutorial. NFS security is a complicated and varied topic, which is beyond the scope of this book. For more on Docker and security, see chapter 14.

Now bind mount the db directory to your export directory:

```
# mount --bind /opt/test/db /export
```

You should now be able to see the contents of the /opt/test/db directory in /export:

TIP If you want this to persist following a reboot, add this line to your /etc/fstab file: `/opt/test/db /export none bind 0 0`

Now add this line to your /etc/exports file:

```
/export       127.0.0.1(ro,fsid=0,insecure,no_subtree_check,async)
```

For this proof of concept example, we're mounting locally on `127.0.0.1`, which defeats the object a little. In a real-world scenario, you'd lock this down to a class of IP addresses such as `192.168.1.0/24`. If you like playing with fire, you can open it up to the world with `*` instead of `127.0.0.1`. For safety, we're mounting read-only (`ro`) here, but you can mount read-write by replacing `ro` with `rw`. Remember that if you do this, you'll need to add a `no_root_squash` flag after the `async` flag there, but think about security before going outside this sandpit.

Mount the directory over NFS to the /mnt directory, export the filesystems you specified previously in /etc/exports, and then restart the NFS service to pick up the changes:

```
# mount -t nfs 127.0.0.1:/export /mnt
# exportfs -a
# service nfs-kernel-server restart
```

Now you're ready to run your infrastructure container:

```
# docker run -ti --name nfs_client --privileged
➥ -v /mnt:/mnt busybox /bin/true
```

And now you can run—without privileges, or knowledge of the underlying implementation—the directory you want to access:

```
# docker run -ti --volumes-from nfs_client debian /bin/bash
root@079d70f79d84:/# ls /mnt
myb
root@079d70f79d84:/# cd /mnt
root@079d70f79d84:/mnt# touch asd
touch: cannot touch `asd': Read-only file system
```

DISCUSSION

This pattern of centrally mounting a shared resource with privileged access for use by others in multiple containers is a powerful one that can make development workflows much simpler.

> **TIP** If you have a lot of these containers to manage, you can make this easier to manage by having a naming convention such as `--name nfs_client_opt_database_live` for a container that exposes the /opt/database/live path.

> **TIP** Remember that this technique only provides security through obscurity (which is no security at all). As you'll see later, anyone who can run the Docker executable effectively has root privileges on the host.

Infrastructure containers for brokering access and abstracting away details are in some ways an equivalent of service-discovery tools for networking—the precise details of how the service runs or where it lives aren't important. You just need to know its name.

As it happens, you've seen `--volumes-from` being used before in technique 35. The details are a little different because the access is being brokered to infrastructure running *inside* the container rather than on the host, but the principle of using names to refer to available volumes remains. You could even swap out that container for the one in this technique and, if configured correctly, applications wouldn't notice a difference in where they look to retrieve their files.

TECHNIQUE 40 Dev tools container

If you're an engineer who often finds yourself on others' machines, struggling without the programs or configuration you have on your beautiful unique-as-a-snowflake development environment, this technique may be for you. Similarly, if you want to share your pimped-up dev environment with others, Docker can make this easy.

PROBLEM

You want to access your development environment on others' machines.

SOLUTION

Create a Docker image with your setup on it, and place it on a registry.

As a demonstration, we're going to use one of our dev tools images. You can download it by running `docker pull dockerinpractice/docker-dev-tools-image`. The repo is available at https://github.com/docker-in-practice/docker-dev-tools-image if you want to inspect the Dockerfile.

Running up the container is simple—a straightforward `docker run -t -i dockerinpractice/docker-dev-tools-image` will give you a shell in our dev environment. You can root around our dotfiles and maybe send us some advice about the setup.

The real power of this technique can be seen when it's combined with others. In the following listing you can see a dev tools container used to display a GUI on the host's network and IPDC stacks and to mount the host's code.

Listing 5.7 Running dev-tools image with a GUI

```
docker run -t -i \
-v /var/run/docker.sock:/var/run/docker.sock \
 -v /tmp/.X11-unix:/tmp/.X11-unix \
 -e DISPLAY=$DISPLAY \
 --net=host --ipc=host \
 -v /opt/workspace:/home/dockerinpractice \
 dockerinpractice/docker-dev-tools-image
```

Mounts the Docker socket to give access to the host's Docker daemon

Mounts the X server Unix domain socket to allow you to start up GUI-based applications (see technique 29)

Sets an environment variable instructing the container to use the host display

These arguments bypass the container's network bridge and allow you access to the host's interprocess communication files (see technique 109).

Mounts the work area to the container's home directory

The preceding command gives you an environment with access to the host's resources:

- Network
- Docker daemon (to run normal Docker commands as though on the host)
- Interprocess communication (IPC) files
- X server to start GUI-based apps, if needed

NOTE As always when mounting host directories, be careful not to mount any vital directories, as you could do damage. Mounting any host directory under root is generally best avoided.

DISCUSSION

We mentioned that you have access to the X server, so it's worth looking at technique 29 for a reminder of some of the possibilities.

For some more invasive dev tools, perhaps for inspecting processes on the host, you may need to look at technique 109 to understand how to grant permission to view some (by default) restricted parts of your system. Technique 93 is also an important read—just because a container can see parts of your system doesn't necessarily mean it has permission to alter them.

Summary

- You should reach for volumes if you need to get at external data from inside a container.
- SSHFS is a simple way to access data on other machines with no extra setup.
- Running GUI applications in Docker requires only a small amount of preparation of your image.
- You can use data containers to abstract away the location of your data.