

directly. The following command will print `true` if the container named `wp` is running and `false` otherwise.

```
docker inspect --format "{{.State.Running}}" wp
```

The `docker inspect` command will display all the metadata (a JSON document) that Docker maintains for a container. The format option transforms that metadata, and in this case it filters everything except for the field indicating the running state of the container. This command should simply output `false`.

In this case, the container isn't running. To determine why, examine the logs for the container:

```
docker logs wp
```

That should output something like:

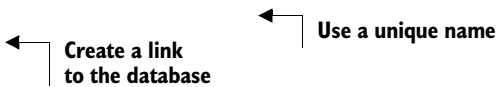
```
error: missing WORDPRESS_DB_HOST and MYSQL_PORT_3306_TCP environment
variables
Did you forget to --link some_mysql_container:mysql or set an external db
with -e WORDPRESS_DB_HOST=hostname:port?
```

It appears that WordPress has a dependency on a MySQL database. A database is a program that stores data in such a way that it's retrievable and searchable later. The good news is that you can install MySQL using Docker just like WordPress:

```
docker run -d --name wpdb \
  -e MYSQL_ROOT_PASSWORD=ch2demo \
  mysql:5
```

Once that is started, create a different WordPress container that's linked to this new database container (linking is covered in depth in chapter 5):

```
docker run -d --name wp2 \
  --link wpdb:mysql \
  -p 80 --read-only \
  wordpress:4
```



Check one more time that WordPress is running correctly:

```
docker inspect --format "{{.State.Running}}" wp2
```

You can tell that WordPress failed to start again. Examine the logs to determine the cause:

```
docker logs wp2
```

There should be a line in the logs that is similar to the following:

```
... Read-only file system: AH00023: Couldn't create the rewrite-map mutex
(file /var/lock/apache2/rewrite-map.1)
```

You can tell that WordPress failed to start again, but this time the problem is that it's trying to write a lock file to a specific location. This is a required part of the startup

process and is not a specialization. It's appropriate to make an exception to the read-only file system in this case. You need to use a volume to make that exception. Use the following to start WordPress without any issues:

```
# Start the container with specific volumes for read only exceptions
docker run -d --name wp3 --link wpdb:mysql -p 80 \
    -v /run/lock/apache2/ \
    -v /run/apache2/ \
    --read-only wordpress:4
```

**Create specific volumes
for writeable space**

An updated version of the script you've been working on should look like this:

```
SQL_CID=$(docker create -e MYSQL_ROOT_PASSWORD=ch2demo mysql:5)
docker start $SQL_CID

MAILER_CID=$(docker create dockerinaction/ch2_mailer)
docker start $MAILER_CID

WP_CID=$(docker create --link $SQL_CID:mysql -p 80 \
    -v /run/lock/apache2/ -v /run/apache2/ \
    --read-only wordpress:4)

docker start $WP_CID

AGENT_CID=$(docker create --link $WP_CID:insideweb \
    --link $MAILER_CID:insidemailer \
    dockerinaction/ch2_agent)

docker start $AGENT_CID
```

Congratulations, at this point you should have a running WordPress container! By using a read-only file system and linking WordPress to another container running a database, you can be sure that the container running the WordPress image will never change. This means that if there is ever something wrong with the computer running a client's WordPress blog, you should be able to start up another copy of that container elsewhere with no problems.

But there are two problems with this design. First, the database is running in a container on the same computer as the WordPress container. Second, WordPress is using several default values for important settings like database name, administrative user, administrative password, database salt, and so on. To deal with this problem, you could create several versions of the WordPress software, each with a special configuration for the client. Doing so would turn your simple provisioning script into a monster that creates images and writes files. A better way to inject that configuration would be through the use of environment variables.

2.5.2 *Environment variable injection*

Environment variables are key-value pairs that are made available to programs through their execution context. They let you change a program's configuration without modifying any files or changing the command used to start the program.

Docker uses environment variables to communicate information about dependent containers, the host name of the container, and other convenient information for programs running in containers. Docker also provides a mechanism for a user to inject environment variables into a new container. Programs that know to expect important information through environment variables can be configured at container-creation time. Luckily for you and your client, WordPress is one such program.

Before diving into WordPress specifics, try injecting and viewing environment variables on your own. The UNIX command `env` displays all the environment variables in the current execution context (your terminal). To see environment variable injection in action, use the following command:

Inject an
environment
variable

```
docker run --env MY_ENVIRONMENT_VAR="this is a test" \
  busybox:latest \
  env
```

Execute the `env` command
inside the container

The `--env` flag—or `-e` for short—can be used to inject any environment variable. If the variable is already set by the image or Docker, then the value will be overridden. This way programs running inside containers can rely on the variables always being set. WordPress observes the following environment variables:

- `WORDPRESS_DB_HOST`
- `WORDPRESS_DB_USER`
- `WORDPRESS_DB_PASSWORD`
- `WORDPRESS_DB_NAME`
- `WORDPRESS_AUTH_KEY`
- `WORDPRESS_SECURE_AUTH_KEY`
- `WORDPRESS_LOGGED_IN_KEY`
- `WORDPRESS_NONCE_KEY`
- `WORDPRESS_AUTH_SALT`
- `WORDPRESS_SECURE_AUTH_SALT`
- `WORDPRESS_LOGGED_IN_SALT`
- `WORDPRESS_NONCE_SALT`

TIP This example neglects the `KEY` and `SALT` variables, but any real production system should absolutely set these values.

To get started, you should address the problem that the database is running in a container on the same computer as the WordPress container. Rather than using linking to satisfy WordPress's database dependency, inject a value for the `WORDPRESS_DB_HOST` variable:

```
docker create --env WORDPRESS_DB_HOST=<my database hostname> wordpress:4
```

This example would create (not start) a container for WordPress that will try to connect to a MySQL database at whatever you specify at `<my database hostname>`.

Because the remote database isn't likely using any default user name or password, you'll have to inject values for those settings as well. Suppose the database administrator is a cat lover and hates strong passwords:

```
docker create \
  --env WORDPRESS_DB_HOST=<my database hostname> \
  --env WORDPRESS_DB_USER=site_admin \
  --env WORDPRESS_DB_PASSWORD=MeowMix42 \
  wordpress:4
```

Using environment variable injection this way will help you separate the physical ties between a WordPress container and a MySQL container. Even in the case where you want to host the database and your customer WordPress sites all on the same machine, you'll still need to fix the second problem mentioned earlier. All the sites are using the same default database name. You'll need to use environment variable injection to set the database name for each independent site:

```
docker create --link wpdb:mysql \
  -e WORDPRESS_DB_NAME=client_a_wp wordpress:4
docker create --link wpdb:mysql \
  -e WORDPRESS_DB_NAME=client_b_wp wordpress:4
```

← For client A

← For client B

Now that you've solved these problems, you can revise the provisioning script. First, set the computer to run only a single MySQL container:

```
DB_CID=$(docker run -d -e MYSQL_ROOT_PASSWORD=ch2demo mysql:5)
MAILER_CID=$(docker run -d dockerinaction/ch2_mailer)
```

Then the site provisioning script would be this:

```
if [ ! -n "$CLIENT_ID" ]; then
  echo "Client ID not set"
  exit 1
fi

WP_CID=$(docker create \
  --link $DB_CID:mysql \
  --name wp_$CLIENT_ID \
  -p 80 \
  -v /run/lock/apache2/ -v /run/apache2/ \
  -e WORDPRESS_DB_NAME=$CLIENT_ID \
  --read-only wordpress:4)

docker start $WP_CID

AGENT_CID=$(docker create \
  --name agent_$CLIENT_ID \
  --link $WP_CID:insideweb \
  --link $MAILER_CID:insidemailer \
  dockerinaction/ch2_agent)

docker start $AGENT_CID
```

← Assume \$CLIENT_ID variable
is set as input to script

← Create link using DB_CID

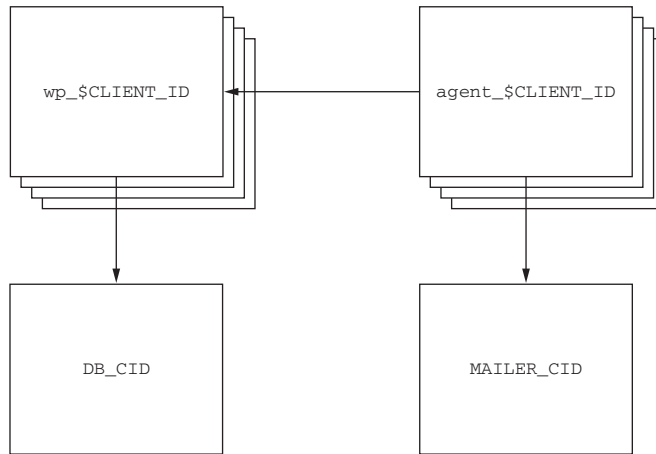


Figure 2.4 Each WordPress and agent container uses the same database and mailer.

This new script will start an instance of WordPress and the monitoring agent for each customer and connect those containers to each other as well as a single mailer program and MySQL database. The WordPress containers can be destroyed, restarted, and upgraded without any worry about loss of data. Figure 2.4 shows this architecture.

The client should be pleased with what is being delivered. But one thing might be bothering you. In earlier testing you found that the monitoring agent correctly notified the mailer when the site was unavailable, but restarting the site and agent required manual work. It would be better if the system tried to automatically recover when a failure was detected. Docker provides restart policies to help deal with that, but you might want something more robust.

2.6 Building durable containers

There are cases where software fails in rare conditions that are temporary in nature. Although it's important to be made aware when these conditions arise, it's usually at least as important to restore the service as quickly as possible. The monitoring system that you built in this chapter is a fine start for keeping system owners aware of problems with a system, but it does nothing to help restore service.

When all the processes in a container have exited, that container will enter the exited state. Remember, a Docker container can be in one of four states:

- Running
- Paused
- Restarting
- Exited (also used if the container has never been started)

A basic strategy for recovering from temporary failures is automatically restarting a process when it exits or fails. Docker provides a few options for monitoring and restarting containers.

2.6.1 *Automatically restarting containers*

Docker provides this functionality with a restart policy. Using the `--restart` flag at container-creation time, you can tell Docker to do any of the following:

- Never restart (default)
- Attempt to restart when a failure is detected
- Attempt for some predetermined time to restart when a failure is detected
- Always restart the container regardless of the condition

Docker doesn't always attempt to immediately restart a container. If it did, that would cause more problems than it solved. Imagine a container that does nothing but print the time and exit. If that container was configured to always restart and Docker always immediately restarted it, the system would do nothing but restart that container. Instead, Docker uses an exponential backoff strategy for timing restart attempts.

A backoff strategy determines how much time should pass between successive restart attempts. An exponential backoff strategy will do something like double the previous time spent waiting on each successive attempt. For example, if the first time the container needs to be restarted Docker waits 1 second, then on the second attempt it would wait 2 seconds, 4 seconds on the third attempt, 8 on the fourth, and so on. Exponential backoff strategies with low initial wait times are a common service-restoration technique. You can see Docker employ this strategy yourself by building a container that always restarts and simply prints the time:

```
docker run -d --name backoff-detector --restart always busybox date
```

Then after a few seconds use the trailing logs feature to watch it back off and restart:

```
docker logs -f backoff-detector
```

The logs will show all the times it has already been restarted and will wait until the next time it is restarted, print the current time, and then exit. Adding this single flag to the monitoring system and the WordPress containers you've been working on would solve the recovery issue.

The only reason you might not want to adopt this directly is that during backoff periods, the container isn't running. Containers waiting to be restarted are in the restarting state. To demonstrate, try to run another process in the backoff-detector container:

```
docker exec backoff-detector echo Just a Test
```

Running that command should result in an error message:

```
Cannot run exec command ... in container ...: No active container exists  
with ID ...
```

That means you can't do anything that requires the container to be in a running state, like execute additional commands in the container. That could be a problem if you need to run diagnostic programs in a broken container. A more complete strategy is to use containers that run init or supervisor processes.

2.6.2 Keeping containers running with supervisor and startup processes

A supervisor process, or init process, is a program that's used to launch and maintain the state of other programs. On a Linux system, PID #1 is an init process. It starts all the other system processes and restarts them in the event that they fail unexpectedly. It's a common practice to use a similar pattern inside containers to start and manage processes.

Using a supervisor process inside your container will keep the container running in the event that the target process—a web server, for example—fails and is restarted. There are several programs that might be used inside a container. The most popular include `init`, `systemd`, `runit`, `upstart`, and `supervisord`. Publishing software that uses these programs is covered in chapter 8. For now, take a look at a container that uses `supervisord`.

A company named Tutum provides software that produces a full LAMP (Linux, Apache, MySQL PHP) stack inside a single container. Containers created this way use `supervisord` to make sure that all the related processes are kept running. Start an example container:

```
docker run -d -p 80:80 --name lamp-test tutum/lamp
```

You can see what processes are running inside this container by using the `docker top` command:

```
docker top lamp-test
```

The `top` subcommand will show the host PID for each of the processes in the container. You'll see `supervisord`, `mysql`, and `apache` included in the list of running programs. Now that the container is running, you can test the `supervisord` restart functionality by manually stopping one of the processes inside the container.

The problem is that to kill a process inside of a container from within that container, you need to know the PID in the container's PID namespace. To get that list, run the following `exec` subcommand:

```
docker exec lamp-test ps
```

The process list generated will have listed `apache2` in the `CMD` column:

PID	TTY	TIME	CMD
1	?	00:00:00	supervisord
433	?	00:00:00	mysqld_safe
835	?	00:00:00	apache2
842	?	00:00:00	ps

The values in the `PID` column will be different when you run the command. Find the PID on the row for `apache2` and then insert that for `<PID>` in the following command:

```
docker exec lamp-test kill <PID>
```

Running this command will run the Linux `kill` program inside the `lamp-test` container and tell the `apache2` process to shut down. When `apache2` stops, the `supervisord`

process will log the event and restart the process. The container logs will clearly show these events:

```
...
... exited: apache2 (exit status 0; expected)
... spawned: 'apache2' with pid 820
... success: apache2 entered RUNNING state, process has stayed up for >
    than 1 seconds (startsecs)
```

A common alternative to the use of `init` or supervisor programs is using a startup script that at least checks the preconditions for successfully starting the contained software. These are sometimes used as the default command for the container. For example, the WordPress containers that you've created start by running a script to validate and set default environment variables before starting the WordPress process. You can view this script by overriding the default command and using a command to view the contents of the startup script:

```
docker run wordpress:4 cat /entrypoint.sh
```

Running that command will result in an error messages like:

```
error: missing WORDPRESS_DB_HOST and MYSQL_PORT_3306_TCP environment
variables
...
```

This failed because even though you set the command to run as `cat /entrypoint.sh`, Docker containers run something called an `entrypoint` before executing the command. Entrypoints are perfect places to put code that validates the preconditions of a container. Although this is discussed in depth in part 2 of this book, you need to know how to override or specifically set the `entrypoint` of a container on the command line. Try running the last command again but this time using the `--entrypoint` flag to specify the program to run and using the command section to pass arguments:

```
docker run --entrypoint="cat" \
    wordpress:4 /entrypoint.sh
```

Use "cat" as the entrypoint
 Pass /entrypoint.sh as
 the argument to cat

If you run through the displayed script, you'll see how it validates the environment variables against the dependencies of the software and sets default values. Once the script has validated that WordPress can execute, it will start the requested or default command.

Startup scripts are an important part of building durable containers and can always be combined with Docker restart policies to take advantage of the strengths of each. Because both the MySQL and WordPress containers already use startup scripts, it's appropriate to simply set the restart policy for each in an updated version of the example script.

With that final modification, you've built a complete WordPress site-provisioning system and learned the basics of container management with Docker. It has taken

considerable experimentation. Your computer is likely littered with several containers that you no longer need. To reclaim the resources that those containers are using, you need to stop them and remove them from your system.

2.7 Cleaning up

Ease of cleanup is one of the strongest reasons to use containers and Docker. The isolation that containers provide simplifies any steps that you'd have to take to stop processes and remove files. With Docker, the whole cleanup process is reduced to one of a few simple commands. In any cleanup task, you must first identify the container that you want to stop and/or remove. Remember, to list all of the containers on your computer, use the `docker ps` command:

```
docker ps -a
```

Because the containers you created for the examples in this chapter won't be used again, you should be able to safely stop and remove all the listed containers. Make sure you pay attention to the containers you're cleaning up if there are any that you created for your own activities.

All containers use hard drive space to store logs, container metadata, and files that have been written to the container file system. All containers also consume resources in the global namespace like container names and host port mappings. In most cases, containers that will no longer be used should be removed.

To remove a container from your computer, use the `docker rm` command. For example, to delete the stopped container named `wp` you'd run:

```
docker rm wp
```

You should go through all the containers in the list you generated by running `docker ps -a` and remove all containers that are in the exited state. If you try to remove a container that's running, paused, or restarting, Docker will display a message like the following:

```
Error response from daemon: Conflict, You cannot remove a running container.
    Stop the container before attempting removal or use -f
FATA[0000] Error: failed to remove one or more containers
```

The processes running in a container should be stopped before the files in the container are removed. You can do this with the `docker stop` command or by using the `-f` flag on `docker rm`. The key difference is that when you stop a process using the `-f` flag, Docker sends a `SIG_KILL` signal, which immediately terminates the receiving process. In contrast, using `docker stop` will send a `SIG_HUP` signal. Recipients of `SIG_HUP` have time to perform finalization and cleanup tasks. The `SIG_KILL` signal makes for no such allowances and can result in file corruption or poor network experiences. You can issue a `SIG_KILL` directly to a container using the `docker kill` command. But you should use `docker kill` or `docker rm -f` only if you must stop the container in less than the standard 30-second maximum stop time.

In the future, if you're experimenting with short-lived containers, you can avoid the cleanup burden by specifying `--rm` on the command. Doing so will automatically remove the container as soon as it enters the exited state. For example, the following command will write a message to the screen in a new BusyBox container, and the container will be removed as soon as it exits:

```
docker run --rm --name auto-exit-test busybox:latest echo Hello World
docker ps -a
```

In this case, you could use either `docker stop` or `docker rm` to properly clean up, or it would be appropriate to use the single-step `docker rm -f` command. You should also use the `-v` flag for reasons that will be covered in chapter 4. The docker CLI makes it is easy to compose a quick cleanup command:

```
docker rm -vf $(docker ps -a -q)
```

This concludes the basics of running software in containers. Each chapter in the remainder of part 1 will focus on a specific aspect of working with containers. The next chapter focuses on installing and uninstalling images, how images relate to containers, and working with container file systems.

2.8 **Summary**

The primary focus of the Docker project is to enable users to run software in containers. This chapter shows how you can use Docker for that purpose. The ideas and features covered include the following:

- Containers can be run with virtual terminals attached to the user's shell or in detached mode.
- By default, every Docker container has its own PID namespace, isolating process information for each container.
- Docker identifies every container by its generated container ID, abbreviated container ID, or its human-friendly name.
- All containers are in any one of four distinct states: running, paused, restarting, or exited.
- The `docker exec` command can be used to run additional processes inside a running container.
- A user can pass input or provide additional configuration to a process in a container by specifying environment variables at container-creation time.
- Using the `--read-only` flag at container-creation time will mount the container file system as read-only and prevent specialization of the container.
- A container restart policy, set with the `--restart` flag at container-creation time, will help your systems automatically recover in the event of a failure.
- Docker makes cleaning up containers with the `docker rm` command as simple as creating them.

Software installation *simplified*

This chapter covers

- Identifying software
- Finding and installing software with Docker Hub
- Installing software from alternative sources
- Understanding file system isolation
- How images and layers work
- Benefits of images with layers

Chapters 1 and 2 introduce all-new concepts and abstractions provided by Docker. This chapter dives deeper into container file systems and software installation. It breaks down software installation into three steps, as illustrated in figure 3.1.

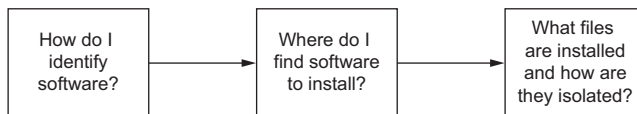


Figure 3.1 Flow of topics covered in this chapter

The first step in installing any software is identifying the software you want to install. You know that software is distributed using images, but you need to know how to tell Docker exactly which image you want to install. I've already mentioned that repositories hold images, but in this chapter I show how repositories and tags are used to identify images in order to install the software you want.

This chapter goes into detail on the three main ways to install Docker images:

- Docker Hub and other registries
- Using image files with `docker save` and `docker load`
- Building images with Dockerfiles

In the course of reading this material you'll learn how Docker isolates installed software and you'll be exposed to a new term, *layer*. Layers are an important concept when dealing with images and have an important impact on software users. This chapter closes with a section about how images work. That knowledge will help you evaluate the image quality and establish a baseline skillset for part 2 of this book.

3.1 *Identifying software*

Suppose you want to install a program called TotallyAwesomeBlog 2.0. How would you tell Docker what you wanted to install? You would need a way to name the program, specify the version that you want to use, and specify the source that you want to install it from. Learning how to identify specific software is the first step in software installation, as illustrated in figure 3.2.

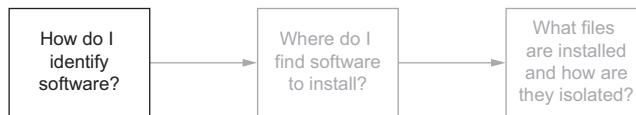


Figure 3.2 Step 1—Software identification

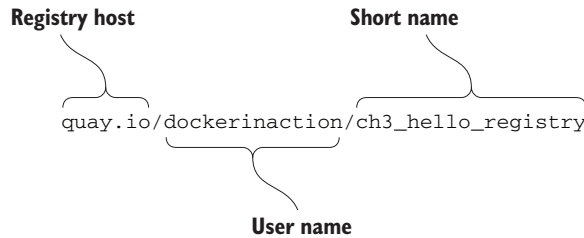
You've learned that Docker creates containers from images. An image is a file. It holds files that will be available to containers created from it and metadata about the image. This metadata contains information about relationships between images, the command history for an image, exposed ports, volume definitions, and more.

Images have identifiers, so they could be used as a name and version for the software, but in practice it's rare to actually work with raw image identifiers. They are long, unique sequences of letters and numbers. Each time a change is made to an image, the image identifier changes. Image identifiers are difficult to work with because they're unpredictable. Instead, users work with repositories.

3.1.1 *What is a repository?*

A *repository* is a named bucket of images. The name is similar to a URL. A repository's name is made up of the name of the host where the image is located, the user account that owns the image, and a short name. For example, later in this chapter

you will install an image from the repository named `quay.io/dockerinaction/ch3_hello_registry`.



Just as there can be several versions of software, a repository can hold several images. Each of the images in a repository is identified uniquely with tags. If I were to release a new version of `quay.io/dockerinaction/ch3_hello_registry`, I might tag it “v2” while tagging the old version with “v1.” If you wanted to download the old version, you could specifically identify that image by its v1 tag.

In chapter 2 you installed an image from the NGINX repository on Docker Hub that was identified with the “latest” tag. A repository name and tag form a composite key, or a unique reference made up of a combination of non-unique components. In that example, the image was identified by `nginx:latest`. Although identifiers built in this fashion may occasionally be longer than raw image identifiers, they’re predictable and communicate the intention of the image.

3.1.2 Using tags

Tags are both an important way to uniquely identify an image and a convenient way to create useful aliases. Whereas a tag can only be applied to a single image in a repository, a single image can have several tags. This allows repository owners to create useful versioning or feature tags.

For example, the Java repository on Docker Hub maintains the following tags: `7`, `7-jdk`, `7u71`, `7u71-jdk`, `openjdk-7`, and `openjdk-7u71`. All these tags are applied to the same image. But as the current minor version of Java 7 increases, and they release `7u72`, the `7u71` tag will likely go away and be replaced with `7u72`. If you care about what minor version of Java 7 you’re running, you have to keep up with those tag changes. If you just want to make sure you’re always running the most recent version of Java 7, just use the image tagged with `7`. It will always be assigned to the newest minor revision of Java 7. These tags give users great flexibility.

It’s also common to see different tags for images with different software configurations. For example, I’ve released two images for an open source program called `freegeoip`. It’s a web application that can be used to get the rough geographical location associated with a network address. One image is configured to use the default configuration for the software. It’s meant to run by itself with a direct link to the world. The second is configured to run behind a web load balancer. Each image has a distinct tag that allows the user to easily identify the image with the features required.

TIP When you're looking for software to install, always pay careful attention to the tags offered in a repository. If you're not sure which one you need, you can download all the tagged images in a repository by simply omitting the tag qualifier when you pull from the repository. I occasionally do this by accident, and it can be annoying. But it's easy to clean up.

This is all there is to identifying software for use with Docker. With this knowledge, you're ready to start looking for and installing software with Docker.

3.2 *Finding and installing software*

You can identify software by a repository name, but how do you find the repositories that you want to install? Discovering trustworthy software is complex, and it is the second step in learning how to install software with Docker, as shown in figure 3.3.

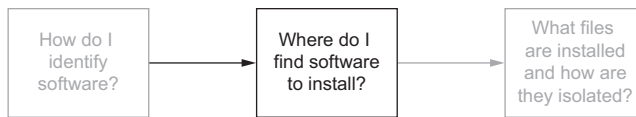


Figure 3.3 Step 2—Locating repositories

To find repositories, you could either keep guessing until you get lucky or use an index. Indexes are search engines that catalog repositories. There are several public Docker indexes, but by default Docker is integrated with an index named Docker Hub.

Docker Hub is a registry and index with a website run by Docker Inc. It's the default registry and index used by Docker. When you issue a `docker pull` or `docker run` command without specifying an alternative registry, Docker will default to looking for the repository on Docker Hub. Docker Hub makes Docker more useful out of the box.

Docker Inc. has made efforts to ensure that Docker is an open ecosystem. It publishes a public image to run your own registry, and the `docker` command-line tool can be easily configured to use alternative registries. Later in this chapter I cover alternative image installation and distribution tools included with Docker. But first, the next section covers how to use Docker Hub so you can get the most from the default toolset.

3.2.1 *Docker Hub from the command line*

Almost anything worth doing with Docker can be done from the command line. This includes searching Docker Hub for repositories.

The `docker` command line will search the Docker Hub index for you and display the results, including details like the number of times each repository has been starred, a flag to indicate that a particular repository is official (the `OFFICIAL` column), and a flag to indicate if the repository is what they call a trusted image (the `TRUSTED` column). The Docker Hub website allows registered users to star a repository in a similar fashion to other community development sites like GitHub. A repository's star count can act as a proxy metric for image quality and popularity or trust by the

community. Docker Hub also provides a set of official repositories that are maintained by Docker Inc. or the current software maintainers. These are often called *libraries*.

There are two ways that an image author can publish their images on Docker Hub:

- *Use the command line to push images that they built independently and on their own systems.* Images pushed this way are considered by some to be less trustworthy because it's not clear how exactly they were built.
- *Make a Dockerfile publicly available and use Docker Hub's continuous build system.* Dockerfiles are scripts for building images. Images created from these automated builds are preferred because the Dockerfile is available for examination prior to installing the image. Images published in this second way will be marked as trusted.

Working with private Docker Hub registries or pushing into registries that you control on Docker Hub does require that you authenticate. In this case, you can use the `docker login` command to log in to Docker Hub. Once you've logged in, you'll be able to pull from private repositories, push to any repository that you control, and tag images in your repositories. Chapter 7 covers pushing and tagging images.

Running `docker login` will prompt you for your Docker Hub credentials. Once you've provided them, your command-line client will be authenticated, and you'll be able to access your private repositories. When you've finished working with your account, you can log out with the `docker logout` command.

If you want to find software to install, you'll need to know where to begin your search. The next example demonstrates how to search for repositories using the `docker search` command. This command may take a few seconds, but it has a timeout built in, so it will eventually return. When you run this command, it will only search the index; nothing will be installed.

Suppose Bob, a software developer, decided that the project he was working on needed a database. He had heard about a popular program named Postgres. He wondered if it was available on Docker Hub, so he ran the following command:

```
docker search postgres
```

After a few seconds several results were returned. At the top of the list he identified a very popular repository with hundreds of stars. He also liked that it was an official repository, which meant that the Docker Hub maintainers had carefully selected the owners of the repository. He used `docker pull` to install the image and moved on with his project.

This is a simple example of how to search for repositories using the `docker` command line. The command will search Docker Hub for any repositories with the term `postgres`. Because Docker Hub is a free public service, users tend to build up lots of public but personal copies. Docker Hub lets users star a repository, similar to a Facebook Like. This is a reasonable proxy indicator for image quality, but you should be careful not to use it as an indicator of trustworthy code.

Imagine if someone builds up a repository with several hundred stars by providing some high-quality open source software. One day a malicious hacker gains control of their repository and publishes an image to the repository that contains a virus. Although containers might be effective for containing malicious code, that notion does not hold true for malicious images. If an attacker controls how an image is built or has targeted an attack specifically to break out of a weakened image, an image can cause serious harm. For this reason, images that are built using publicly available scripts are considered much more trustworthy. In the search results from running `docker search`, you can tell that an image was built from a public script by looking for an `[OK]` in the column label `AUTOMATED`.

Now you've seen how to find software on Docker Hub without leaving your terminal. Although you can do most things from the terminal, there are some things that you can do only through the website.

3.2.2 **Docker Hub from the website**

If you have yet to stumble upon it while browsing `docker.com`, you should take a moment to check out <https://hub.docker.com>. Docker Hub lets you search for repositories, organizations, or specific users. User and organization profile pages list the repositories that the account maintains, recent activity on the account, and the repositories that the account has starred. On repository pages you can see the following:

- General information about the image provided by the image publisher
- A list of the tags available in the repository
- The date the repository was created
- The number of times it has been downloaded
- Comments from registered users

Docker Hub is free to join, and you'll need an account later in this book. When you're signed in, you can star and comment on repositories. You can create and manage your own repositories. We will do that in part 2. For now, just get a feel for the site and what it has to offer.

Activity: a Docker Hub scavenger hunt

It's good to practice finding software on Docker Hub using the skills you learned in chapter 2. This activity is designed to encourage you to use Docker Hub and practice creating containers. You will also be introduced to three new options on the `docker run` command.

In this activity you're going to create containers from two images that are available through Docker Hub. The first is available from the `dockerinaction/ch3_ex2_hunt` repository. In that image you'll find a small program that prompts you for a password. You can only find the password by finding and running a container from the second

mystery repository on Docker Hub. To use the programs in these images, you'll need to attach your terminal to the containers so that the input and output of your terminal are connected directly to the running container. The following command demonstrates how to do that and run a container that will be removed automatically when stopped:

```
docker run -it --rm dockerinaction/ch3_ex2_hunt
```

When you run this command, the scavenger hunt program will prompt you for the password. If you know the answer already, go ahead and enter it now. If not, just enter anything and it will give you a hint. At this point you should have all the tools you need to complete the activity. Figure 3.4 illustrates what you need to do from this point.

Still stuck? I can give you one more hint. The mystery repository is one that was created for this book. Maybe you should try searching for this book's Docker Hub repositories. Remember, repositories are named with a username/repository pattern.

When you get the answer, pat yourself on the back and remove the images using the `docker rmi` command. Concretely, the commands you run should look something like these:

```
docker rmi dockerinaction/ch3_ex2_hunt
docker rmi <mystery repository>
```

If you were following the examples and using the `--rm` option on your `docker run` commands, you should have no containers to clean up. You've learned a lot in this example. You've found a new image on Docker Hub and used the `docker run` command in a new way. There's a lot to know about running interactive containers. The next section covers that in greater detail.

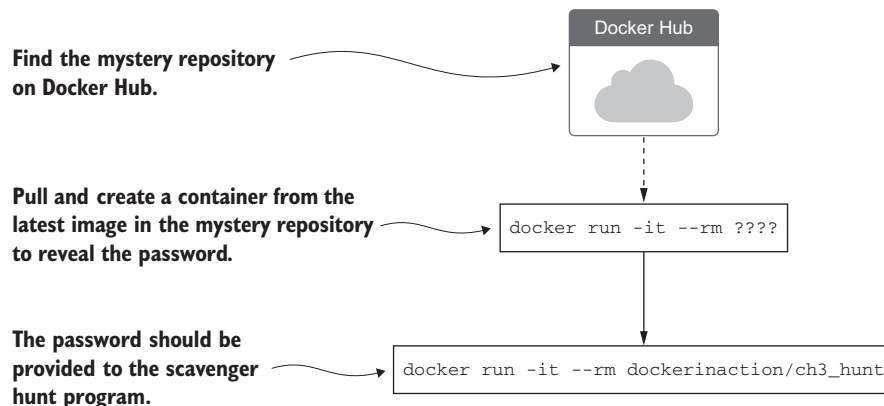


Figure 3.4 The steps required to complete the Docker Hub scavenger hunt. Find the mystery repository on Docker Hub. Install the latest image from that repository and run it interactively to get the password.

Docker Hub is by no means the only source for software. Depending on the goals and perspective of software publishers, Docker Hub may not be an appropriate distribution point. Closed source or proprietary projects may not want to risk publishing their software through a third party. There are three other ways to install software:

- You can use alternative repository registries or run your own registry.
- You can manually load images from a file.
- You can download a project from some other source and build an image using a provided Dockerfile.

All three of these options are viable for private projects or corporate infrastructure. The next few subsections cover how to install software from each alternative source.

3.2.3 *Using alternative registries*

As mentioned earlier, Docker makes the registry software available for anyone to run. Hosting companies have integrated it into their offerings, and companies have begun running their own internal registries. I'm not going to cover running a registry until chapter 8, but it's important that you learn how to use them early.

Using an alternative registry is simple. It requires no additional configuration. All you need is the address of the registry. The following command will download another “Hello World” type example from an alternative registry:

```
docker pull quay.io/dockerinaction/ch3_hello_registry:latest
```

The registry address is part of the full repository specification covered in section 3.1. The full pattern is as follows:

```
[REGISTRYHOST/] [USERNAME/] NAME [:TAG]
```

Docker knows how to talk to Docker registries, so the only difference is that you specify the registry host. In some cases, working with registries will require an authentication step. If you encounter a situation where this is the case, consult the documentation or the group that configured the registry to find out more. When you're finished with the hello-registry image you installed, remove it with the following command:

```
docker rmi quay.io/dockerinaction/ch3_hello_registry
```

Registries are powerful. They enable a user to relinquish control of image storage and transportation. But running your own registry can be complicated and may create a potential single point of failure for your deployment infrastructure. If running a custom registry sounds a bit complicated for your use case, and third-party distribution tools are out of the question, you might consider loading images directly from a file.

3.2.4 *Images as files*

Docker provides a command to load images into Docker from a file. With this tool, you can load images that you acquired through other channels. Maybe your company

has chosen to distribute images through a central file server or some type of version-control system. Maybe the image is small enough that your friend just sent it to you over email or shared it via flash drive. However you came upon the file, you can load it into Docker with the `docker load` command.

You'll need an image file to load before I can show you the `docker load` command. Because it's unlikely that you have an image file lying around, I'll show you how to save one from a loaded image. For the purposes of this example, you'll pull `busybox:latest`. That image is small and easy to work with. To save that image to a file, use the `docker save` command. Figure 3.5 demonstrates `docker save` by creating a file from BusyBox.

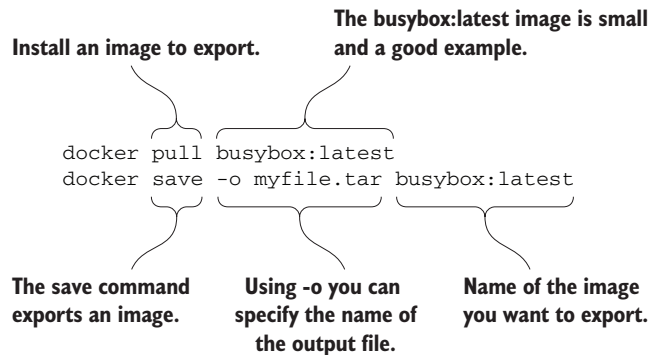


Figure 3.5 Parts of the pull and save subcommands

I used the `.tar` filename suffix in this example because the `docker save` command creates TAR archive files. You can use any filename you want. If you omit the `-o` flag, the resulting file will be streamed to the terminal.

TIP Other ecosystems that use TAR archives for packing define custom file extensions. For example, Java uses `.jar`, `.war`, and `.ear`. In cases like these, using custom file extensions can help hint at the purpose and content of the archive. Although there are no defaults set by Docker and no official guidance on the matter, you may find using a custom extension useful if you work with these files often.

After running the `save` command, the `docker` program will terminate unceremoniously. Check that it worked by listing the contents of your current working directory. If the specified file is there, use this command to remove the image from Docker:

```
docker rmi busybox
```

After removing the image, load it again from the file you created using the `docker load` command. Like `docker save`, if you run `docker load` without the `-i` command, Docker will use the standard input stream instead of reading the archive from a file:

```
docker load -i myfile.tar
```

Once you've run the `docker load` command, the image should be loaded. You can verify this by running the `docker images` command again. If everything worked correctly, BusyBox should be included in the list.

Working with images as files is as easy as working with registries, but you miss out on all the nice distribution facilities that registries provide. If you want to build your own distribution tools, or you already have something else in place, it should be trivial to integrate with Docker using these commands.

Another popular project distribution pattern uses bundles of files with installation scripts. This approach is popular with open source projects that use public version-control repositories for distribution. In these cases you work with a file, but the file is not an image; it is a Dockerfile.

3.2.5 *Installing from a Dockerfile*

A Dockerfile is a script that describes steps for Docker to take to build a new image. These files are distributed along with software that the author wants to be put into an image. In this case, you're not technically installing an image. Instead, you're following instructions to build an image. Working with Dockerfiles is covered in depth in chapter 7.

Distributing a Dockerfile is similar to distributing image files. You're left to your own distribution mechanisms. A common pattern is to distribute a Dockerfile with software from common version-control systems like Git or Mercurial. If you have Git installed, you can try this by running an example from a public repository:

```
git clone https://github.com/dockerinaction/ch3_dockerfile.git
docker build -t dia_ch3/dockerfile:latest ch3_dockerfile
```

In this example you copy the project from a public source repository onto your computer and then build and install a Docker image using the Dockerfile included with that project. The value provided to the `-t` option of `docker build` is the repository where you want to install the image. Building images from Dockerfiles is a light way to move projects around that fits into existing workflows. There are two disadvantages to taking this approach. First, depending on the specifics of the project, the build process might take some time. Second, dependencies may drift between the time when the Dockerfile was authored and when an image is built on a user's computer. These issues make distributing build files less than an ideal experience for a user. But it remains popular in spite of these drawbacks.

When you're finished with this example, make sure to clean up your workspace:

```
docker rmi dia_ch3/dockerfile
rm -rf ch3_dockerfile
```

After reading this section you should have a complete picture of your options to install software with Docker. But when you install software, you should have an idea about what changes are being made to your computer.

3.3 Installation files and isolation

Understanding how images are identified, discovered, and installed is a minimum proficiency for a Docker user. If you understand what files are actually installed and how those files are built and isolated at runtime, you'll be able to answer more difficult questions that come up with experience, such as these:

- What image properties factor into download and installation speeds?
- What are all these unnamed images that are listed when I use the `docker images` command?
- Why does output from the `docker pull` command include messages about pulling dependent layers?
- Where are the files that I wrote to my container's file system?

Learning this material is the third and final step to understanding software installation with Docker, as illustrated in figure 3.6.

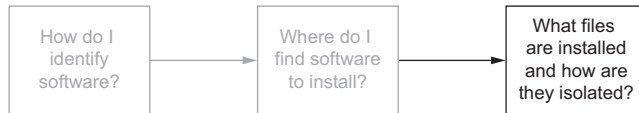


Figure 3.6 Step 3—Understanding how software is installed

So far, when I've written about installing software, I've used the term *image*. This was to infer that the software you were going to use was in a single image and that an image was contained within a single file. Although this may occasionally be accurate, most of the time what I've been calling an image is actually a collection of image layers. A *layer* is an image that's related to at least one other image. It is easier to understand layers when you see them in action.

3.3.1 Image layers in action

In this example you're going to install the two images. Both depend on Java 6. The applications themselves are simple Hello World-style programs. What I want you to keep an eye on is what Docker does when you install each. You should notice how long it takes to install the first compared to the second and read what it's printing to the terminal. When an image is being installed, you can watch Docker determine which dependencies it needs to download and then see the progress of the individual image layer downloads. Java is great for this example because the layers are quite large, and that will give you a moment to really see Docker in action.

The two images you're going to install are `dockerinaction/ch3_myapp` and `dockerinaction/ch3_myotherapp`. You should just use the `docker pull` command because you only need to see the images install, not start a container from them. Here are the commands you should run:

```
docker pull dockerinaction/ch3_myapp
docker pull dockerinaction/ch3_myotherapp
```

Did you see it? Unless your network connection is far better than mine, or you had already installed Java 6 as a dependency of some other image, the download for `dockerinaction/ch3_myapp` should have been much slower than `dockerinaction/ch3_myotherapp`.

When you installed `ch3_myapp`, Docker determined that it needed to install the `openjdk-6` image because it's the direct dependency (parent layer) of the requested image. When Docker went to install that dependency, it discovered the dependencies of that layer and downloaded those first. Once all the dependencies of a layer are installed, that layer is installed. Finally, `openjdk-6` was installed, and then the tiny `ch3_myapp` layer was installed.

When you issued the command to install `ch3_myotherapp`, Docker identified that `openjdk-6` was already installed and immediately installed the image for `ch3_myotherapp`. This was simpler, and because less than one megabyte of data was transferred, it was faster. But again, to the user it was an identical process.

From the user perspective this ability is nice to have, but you wouldn't want to have to try to optimize for it. Just take the benefits where they happen to work out. From the perspective of a software or image author, this ability should play a major factor in your image design. I cover that more in chapter 7.

If you run `docker images` now, you'll see the following repositories listed:

- `dockerinaction/ch3_myapp`
- `dockerinaction/ch3_myotherapp`
- `java:6`

By default, the `docker images` command will only show you repositories. Similar to other commands, if you specify the `-a` flag, the list will include every installed intermediate image or layer. Running `docker images -a` will show a list that includes several repositories listed as `<none>`. The only way to refer to these is to use the value in the `IMAGE ID` column.

In this example you installed two images directly, but a third parent repository was installed as well. You'll need to clean up all three. You can do so more easily if you use the condensed `docker rmi` syntax:

```
docker rmi \  
    dockerinaction/ch3_myapp \  
    dockerinaction/ch3_myotherapp \  
    java:6
```

The `docker rmi` command allows you to specify a space-separated list of images to be removed. This comes in handy when you need to remove a small set of images after an example. I'll be using this when appropriate throughout the rest of the examples in this book.

3.3.2 Layer relationships

Images maintain parent/child relationships. In these relationships they build from their parents and form layers. The files available to a container are the union of all of the layers in the lineage of the image the container was created from. Images can have relationships with any other image, including images in different repositories with different owners. The two images in section 3.3.1 use a Java 6 image as their parent. Figure 3.7 illustrates the full image ancestry of both images.

The layers shown in figure 3.7 are a sample of the `java:6` image at the time of this writing. An image is named when its author tags and publishes it. A user can create aliases, as you did in chapter 2 using the `docker tag` command. Until an image is tagged, the only way to refer to it is to use its unique identifier (UID) that was generated when the image was built. In figure 3.7, the parents of the common Java 6 image are labeled using the first 12 digits of their UID. These layers contain common libraries and dependencies of the Java 6 software. Docker truncates the UID from 65 (base 16) digits to 12 for the benefit of its human users. Internally and through API access, Docker uses the full 65. It's important to be aware of this when you've installed images along with similar unnamed images. I wouldn't want you to think something bad happened or some malicious software had made it into your computer when you see these images included when you use the `docker images` command.

The Java images are sizable. At the time of this writing, the `openjdk-6` image is 348 MB, and the `openjdk-7` image is 590 MB. You get some space savings when you use the runtime-only images, but even `openjre-6` is 200 MB. Again, Java was chosen here because its images are particularly large for a common dependency.

3.3.3 Container file system abstraction and isolation

Programs running inside containers know nothing about image layers. From inside a container, the file system operates as though it's not running in a container or operating on an image. From the perspective of the container, it has exclusive copies of the files provided by the image. This is made possible with something called a union file system. Docker uses a variety of union file systems and will select the best fit for your system. The details of how the union file system works are beyond what you need to know to use Docker effectively.

A union file system is part of a critical set of tools that combine to create effective file system isolation. The other tools are MNT namespaces and the `chroot` system call.

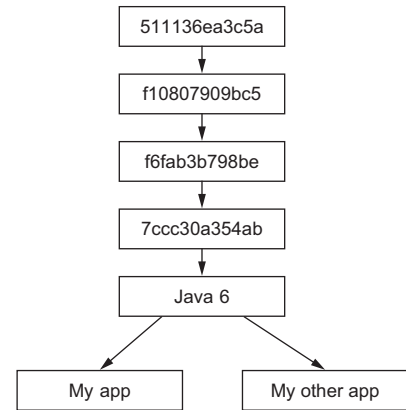


Figure 3.7 The full lineage of the two Docker images used in section 3.3.1

The file system is used to create mount points on your host's file system that abstract the use of layers. The layers created are what are bundled into Docker image layers. Likewise, when a Docker image is installed, its layers are unpacked and appropriately configured for use by the specific file system provider chosen for your system.

The Linux kernel provides a namespace for the MNT system. When Docker creates a container, that new container will have its own MNT namespace, and a new mount point will be created for the container to the image.

Lastly, `chroot` is used to make the root of the image file system the root in the container's context. This prevents anything running inside the container from referencing any other part of the host file system.

Using `chroot` and MNT namespaces is common for container technologies. By adding a union file system to the recipe, Docker containers have several benefits.

3.3.4 Benefits of this toolset and file system structure

The first and perhaps most important benefit of this approach is that common layers need to be installed only once. If you install any number of images and they all depend on some common layer, that common layer and all of its parent layers will need to be downloaded or installed only once. This means you might be able to install several specializations of a program without storing redundant files on your computer or downloading redundant layers. By contrast, most virtual machine technologies will store the same files as many times as you have redundant virtual machines on a computer.

Second, layers provide a coarse tool for managing dependencies and separating concerns. This is especially handy for software authors, and chapter 7 talks more about this. From a user perspective, this benefit will help you quickly identify what software you're running by examining which images and layers you're using.

Lastly, it's easy to create software specializations when you can layer minor changes on top of some basic image. That's another subject covered in detail in chapter 7. Providing specialized images helps users get exactly what they need from software with minimal customization. This is one of the best reasons to use Docker.

3.3.5 Weaknesses of union file systems

Docker will choose the best file system for the system it's running on, but no implementation is perfect for every workload. In fact, there are some specific use cases when you should pause and consider using another Docker feature.

Different file systems have different rules about file attributes, sizes, names, and characters. Union file systems are in a position where they often need to translate between the rules of different file systems. In the best cases they're able to provide acceptable translations. In the worst cases features are omitted. For example, neither `btrfs` nor `OverlayFS` provides support for the extended attributes that make `SELinux` work.

Union file systems use a pattern called copy-on-write, and that makes implementing memory-mapped files (the `mmap()` system call) difficult. Some union file systems

provide implementations that work under the right conditions, but it may be a better idea to avoid memory-mapping files from an image.

The backing file system is another pluggable feature of Docker. You can determine which file system your installation is using with the `info` subcommand. If you want to specifically tell Docker which file system to use, do so with the `--storage-driver` or `-s` option when you start the Docker daemon. Most issues that arise with writing to the union file system can be addressed without changing the storage provider. These can be solved with volumes, the subject of chapter 4.

3.4 Summary

The task of installing and managing software on a computer presents a unique set of challenges. This chapter explains how you can use Docker to address them. The core ideas and features covered by this chapter are as follows:

- Human Docker users use repository names to communicate which software they would like Docker to install.
- Docker Hub is the default Docker registry. You can find software on Docker Hub through either the website or the `docker` command-line program.
- The `docker` command-line program makes it simple to install software that's distributed through alternative registries or in other forms.
- The image repository specification includes a registry host field.
- The `docker load` and `docker save` commands can be used to load and save images from TAR archives.
- Distributing a Dockerfile with a project simplifies image builds on user machines.
- Images are usually related to other images in parent/child relationships. These relationships form layers. When we say that we have installed an image, we are saying that we have installed a target image and each image layer in its lineage.
- Structuring images with layers enables layer reuse and saves bandwidth during distribution and storage space on your computer.

Persistent storage and shared state with volumes

This chapter covers

- An introduction to volumes
- The two types of volumes
- How to share data between the host and a container
- How to share data between containers
- The volume life cycle
- Data management and control patterns with volumes

At this point in the book, you've installed and run a few programs. You've seen a few toy examples but haven't run anything that resembles the real world. The difference between the examples in the first three chapters and the real world is that in the real world, programs work with data. This chapter introduces Docker volumes and strategies that you'll use to manage data with containers.

Consider what it might look like to run a database program inside a container. You could package the software with the image, and when you start the container it

might initialize an empty database. When programs connect to the database and enter data, where is that data stored? Is it in a file inside the container? What happens to that data when you stop the container or remove it? How would you move your data if you wanted to upgrade the database program?

Consider another situation where you're running a couple of different web applications inside different containers. Where would you write log files so that they will outlive the container? How would you get access to those logs to troubleshoot a problem? How can other programs such as log digest tools get access to those files? The answer to all these questions involves the use of volumes.

4.1 Introducing volumes

A host or container's directory tree is created by a set of mount points that describe how to piece together one or more file systems. A *volume* is a mount point on the container's directory tree where a portion of the host directory tree has been mounted. Most people are only minimally familiar with file systems and mount points and rarely customize them. People have a more difficult time with volumes than with any other Docker topic. That lack of familiarity with mount points is a contributing factor.

Without volumes, container users are limited to working with the union file system that provides image mounts. Figure 4.1 shows a program running in a container and writing to files. The first file is written to the root file system. The operating system directs root file system changes to the top layer of the mounted union file system. The second file is written to a volume that has been mounted on the container's directory tree at /data. That change is made directly on the host's file system through the volume.

Although the union file system works for building and sharing images, it's less than ideal for working with persistent or shared data. Volumes fill those use cases and play a critical role in containerized system design.

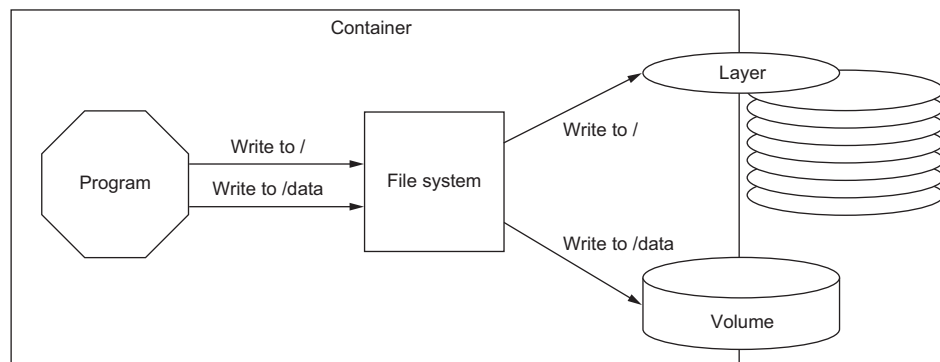


Figure 4.1 A container with a mounted volume and writeable top layer of the union file system

4.1.1 **Volumes provide container-independent data management**

Semantically, a volume is a tool for segmenting and sharing data that has a scope or life cycle that's independent of a single container. That makes volumes an important part of any containerized system design that shares or writes files. Examples of data that differs in scope or access from a container include the following:

- Database software versus database data
- Web application versus log data
- Data processing application versus input and output data
- Web server versus static content
- Products versus support tools

Volumes enable separation of concerns and create modularity for architectural components. That modularity helps you understand, build, support, and reuse parts of larger systems more easily.

Think about it this way: images are appropriate for packaging and distributing relatively static files like programs; volumes hold dynamic data or specializations. This distinction makes images reusable and data simple to share. This separation of relatively static and dynamic file space allows application or image authors to implement advanced patterns such as polymorphic and composable tools.

A *polymorphic* tool is one that maintains a consistent interface but might have several implementations that do different things. Consider an application such as a general application server. Apache Tomcat, for example, is an application that provides an HTTP interface on a network and dispatches any requests it receives to pluggable programs. Tomcat has polymorphic behavior. Using volumes, you can inject behavior into containers without modifying an image. Alternatively, consider a database program like MongoDB or MySQL. The value of a database is defined by the data it contains. A database program always presents the same interface but takes on a wholly different value depending on the data that can be injected with a volume. The polymorphic container pattern is the subject of section 4.5.3.

More fundamentally, volumes enable the separation of application and host concerns. At some point an image will be loaded onto a host and a container created from it. Docker knows little about the host where it's running and can only make assertions about what files should be available to a container. That means Docker alone has no way to take advantage of host-specific facilities like mounted network storage or mixed spinning and solid-state hard drives. But a user with knowledge of the host can use volumes to map directories in a container to appropriate storage on that host.

Now that you're familiar with what volumes are and why they're important, you can get started with them in a real-world example.

4.1.2 **Using volumes with a NoSQL database**

The Apache Cassandra project provides a column database with built-in clustering, eventual consistency, and linear write scalability. It's a popular choice in modern

system designs, and an official image is available on Docker Hub. Cassandra is like other databases in that it stores its data in files on disk. In this section you'll use the official Cassandra image to create a single-node Cassandra cluster, create a keyspace, delete the container, and then recover that keyspace on a new node in another container.

Get started by creating a single container that defines a volume. This is called a volume container. Volume containers are one of the advanced patterns discussed later in this chapter:

```
docker run -d \
  --volume /var/lib/cassandra/data \
  --name cass-shared \
  alpine echo Data Container
```

← **Specify volume mount point inside the container**

The volume container will immediately stop. That is appropriate for the purposes of this example. Don't remove it yet. You're going to use the volume it created when you create a new container running Cassandra:

```
docker run -d \
  --volumes-from cass-shared \
  --name cass1 \
  cassandra:2.2
```

← **Inherit volume definitions**

After Docker pulls the `cassandra:2.2` image from Docker Hub, it creates a new container and copies the volume definitions from the volume container. After that, both containers have a volume mounted at `/var/lib/cassandra/data` that points to the same location on the host's directory tree. Next, start a container from the `cassandra:2.2` image, but run a Cassandra client tool and connect to your running server:

```
docker run -it --rm \
  --link cass1:cass \
  cassandra:2.2 cqlsh cass
```

Now you can inspect or modify your Cassandra database from the CQLSH command line. First, look for a keyspace named `docker_hello_world`:

```
select *
from system.schema_keyspaces
where keyspace_name = 'docker_hello_world';
```

Cassandra should return an empty list. This means the database hasn't been modified by the example. Next, create that keyspace with the following command:

```
create keyspace docker_hello_world
with replication = {
  'class' : 'SimpleStrategy',
  'replication_factor': 1
};
```

Now that you've modified the database, you should be able to issue the same query again to see the results and verify that your changes were accepted. The following command is the same as the one you ran earlier:

```
select *
from system.schema_keyspaces
where keyspace_name = 'docker_hello_world';
```

This time Cassandra should return a single entry with the properties you specified when you created the keyspace. If you're satisfied that you've connected to and modified your Cassandra node, quit the CQLSH program to stop the client container:

```
# Leave and stop the current container
quit
```

The client container was created with the `--rm` flag and was automatically removed when the command stopped. Continue cleaning up the first part of this example by stopping and removing the Cassandra node you created:

```
docker stop cass1
docker rm -vf cass1
```

Both the Cassandra client and server you created will be deleted after running those commands. If the modifications you made are persisted, the only place they could remain is the volume container. If that is true, then the scope of that data has expanded to include two containers, and its life cycle has extended beyond the container where the data originated.

You can test this by repeating these steps. Create a new Cassandra node, attach a client, and query for the keyspace. Figure 4.2 illustrates the system and what you will have built.

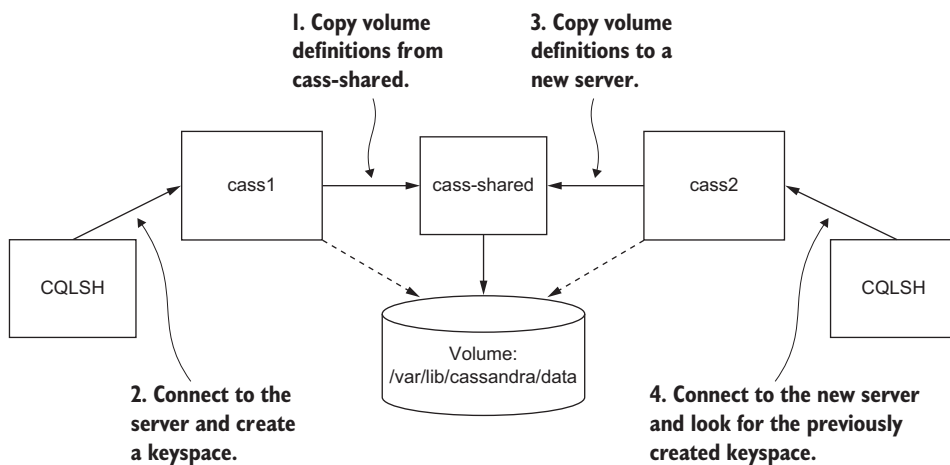


Figure 4.2 Key steps in creating and recovering data persisted to a volume with Cassandra

The next three commands will test recovery of the data:

```
docker run -d \
  --volumes-from cass-shared \
  --name cass2 \
  cassandra:2.2

docker run -it --rm \
  --link cass2:cass \
  cassandra:2.2 \
  cqlsh cass

select *
from system.schema_keyspaces
where keyspace_name = 'docker_hello_world';
```

The last command in this set returns a single entry, and it matches the keyspace you created in the previous container. This confirms the previous claims and demonstrates how volumes might be used to create durable systems. Before moving on, quit the CQLSH program and clean up your workspace. Make sure to remove that volume container as well:

```
quit

docker rm -vf cass2 cass-shared
```

This example demonstrates one way to use volumes without going into how they work, the patterns in use, or how to manage volume life cycle. The remainder of this chapter dives deeper into each facet of volumes, starting with the different types available.

4.2 Volume types

There are two types of volume. Every volume is a mount point on the container directory tree to a location on the host directory tree, but the types differ in where that location is on the host. The first type of volume is a bind mount. Bind mount volumes use any user-specified directory or file on the host operating system. The second type is a managed volume. Managed volumes use locations that are created by the Docker daemon in space controlled by the daemon, called Docker managed space. The volume types are illustrated in figure 4.3.

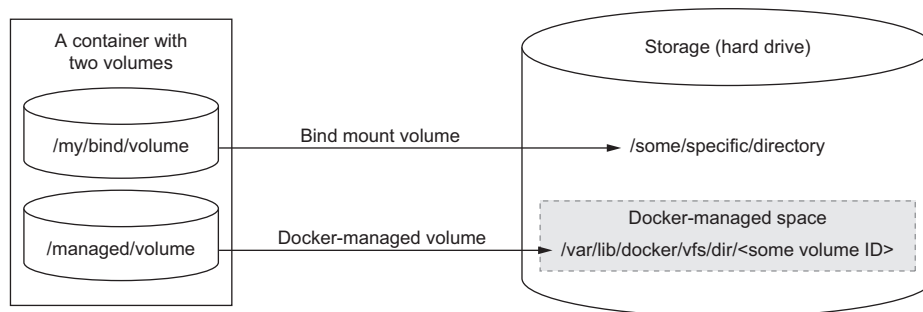


Figure 4.3 Docker provides both bind mount and managed volumes.

Each type of volume has advantages and disadvantages. Depending on your specific use case, you may need to use one or be unable to use the other. This section explores each type in depth.

4.2.1 **Bind mount volumes**

A bind mount volume is a volume that points to a user-specified location on the host file system. Bind mount volumes are useful when the host provides some file or directory that needs to be mounted into the container directory tree at a specific point, as shown in figure 4.4.

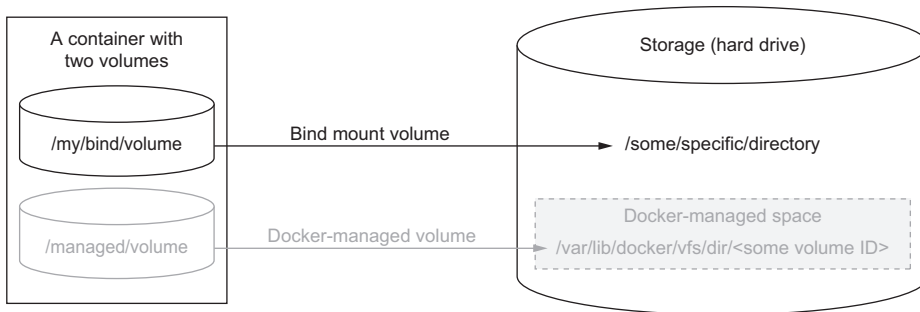


Figure 4.4 A host directory as a bind mount volume

Bind mount volumes are useful if you want to share data with other processes running outside a container, such as components of the host system itself. They also work if you want to share data that lives on your host at some known location with a specific program that runs in a container.

For example, suppose you’re working on a document or web page on your local computer and want to share your work with a friend. One way to do so would be to use Docker to launch a web server and serve content that you’ve copied into the web server image. Although that would work and might even be a best practice for production environments, it’s cumbersome to rebuild the image every time you want to share an updated version of the document.

Instead, you could use Docker to launch the web server and bind mount the location of your document into the new container at the web server’s document root. You can try this for yourself. Create a new directory in your home directory called `example-docs`. Now create a file named `index.html` in that directory. Add a nice message for your friend to the file. The following command will start an Apache HTTP server where your new directory is bind mounted to the server’s document root:

```
docker run -d --name bmweb \
  -v ~/example-docs:/usr/local/apache2/htdocs \
```



```
-p 80:80 \  
httpd:latest
```

With this container running, you should be able to point your web browser at the IP address where your Docker engine is running and see the file you created.

In this example you used the `-v` option and a location map to create the bind mount volume. The map is delimited with a colon (as is common with Linux-style command-line tools). The map key (the path before the colon) is the absolute path of a location on the host file system, and the value (the path after the colon) is the location where it should be mounted inside the container. You must specify locations with absolute paths.

This example touches on an important attribute or feature of volumes. When you mount a volume on a container file system, it replaces the content that the image provides at that location. In this example, the `httpd:latest` image provides some default HTML content at `/usr/local/apache2/htdocs/`, but when you mounted a volume at that location, the content provided by the image was overridden by the content on the host. This behavior is the basis for the polymorphic container pattern discussed later in the chapter.

Expanding on this use case, suppose you want to make sure that the Apache HTTP web server can't change the contents of this volume. Even the most trusted software can contain vulnerabilities, and it's best to minimize the impact of an attack on your website. Fortunately, Docker provides a mechanism to mount volumes as read-only. You can do this by appending `:ro` to the volume map specification. In the example, you should change the `run` command to something like the following:

```
docker rm -vf bmweb  
  
docker run --name bmweb_ro \  
  --volume ~/example-docs:/usr/local/apache2/htdocs/:ro \  
  -p 80:80 \  
  httpd:latest
```

By mounting the volume as read-only, you can prevent any process inside the container from modifying the content of the volume. You can see this in action by running a quick test:

```
docker run --rm \  
  -v ~/example-docs:/testspace:ro \  
  alpine \  
  /bin/sh -c 'echo test > /testspace/test'
```

This command starts a container with a similar read-only bind mount as the web server. It runs a command that tries to add the word `test` to a file named `test` in the volume. The command fails because the volume is mounted as read-only.

Finally, note that if you specify a host directory that doesn't exist, Docker will create it for you. Although this can come in handy, relying on this functionality isn't the

best idea. It's better to have more control over the ownership and permissions set on a directory.

```
ls ~/example-docs/absent

docker run --rm -v ~/example-docs/absent:/absent alpine:latest \
/bin/sh -c 'mount | grep absent'

ls ~/example-docs/absent
```

Examine the created directory

Examine the volume mount definition

Verify that "absent" does not exist

Bind mount volumes aren't limited to directories, though that's how they're frequently used. You can use bind mount volumes to mount individual files. This provides the flexibility to create or link resources at a level that avoids conflict with other resources. Consider when you want to mount a specific file into a directory that contains other files. Concretely, suppose you only wanted to serve a single additional file alongside the web content that shipped with some image. If you use a bind mount of a whole directory over that location, the other files will be lost. By using a specific file as a volume, you can override or inject individual files.

The important thing to note in this case is that the file must exist on the host before you create the container. Otherwise, Docker will assume that you wanted to use a directory, create it on the host, and mount it at the desired location (even if that location is occupied by a file).

The first problem with bind mount volumes is that they tie otherwise portable container descriptions to the file system of a specific host. If a container description depends on content at a specific location on the host file system, then that description isn't portable to hosts where the content is unavailable or available in some other location.

The next big problem is that they create an opportunity for conflict with other containers. It would be a bad idea to start multiple instances of Cassandra that all use the same host location as a volume. In that case, each of the instances would compete for the same set of files. Without other tools such as file locks, that would likely result in corruption of the database.

Bind mount volumes are appropriate tools for workstations or machines with specialized concerns. It's better to avoid these kinds of specific bindings in generalized platforms or hardware pools. You can take advantage of volumes in a host-agnostic and portable way with Docker-managed volumes.

4.2.2 **Docker-managed volumes**

Managed volumes are different from bind mount volumes because the Docker daemon creates managed volumes in a portion of the host's file system that's owned by Docker, as shown in figure 4.5. Using managed volumes is a method of decoupling volumes from specialized locations on the file system.

Managed volumes are created when you use the `-v` option (or `--volume`) on `docker run` but only specify the mount point in the container directory tree. You

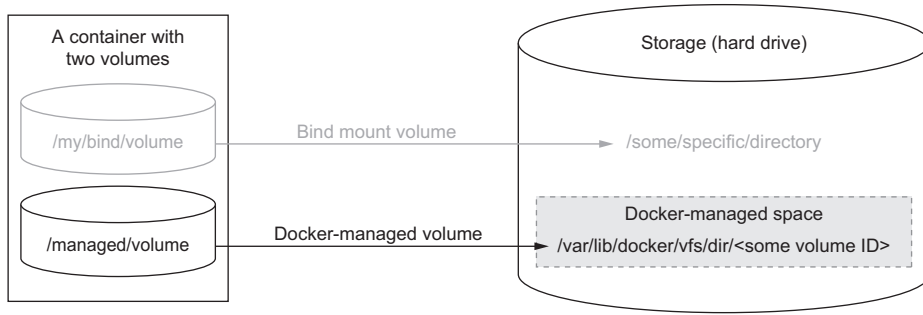


Figure 4.5 A directory in Docker-managed space mounted as a volume

created a managed volume in the Cassandra example in section 4.1.2. The container named `cass-shared` specified a volume at `/var/lib/cassandra/data`:

```
docker run -d \
  -v /var/lib/cassandra/data \
  --name cass-shared \
  alpine echo Data Container
```

← Specify volume mount point inside container

When you created this container, the Docker daemon created directories to store the contents of the three volumes somewhere in a part of the host file system that it controls. To find out exactly where this folder is, you can use the `docker inspect` command filtered for the `Volumes` key. The important thing to take away from this output is that Docker created each of the volumes in a directory controlled by the Docker daemon on the host:

```
docker inspect -f "{{json .Volumes}}" cass-shared
```

The `inspect` subcommand will output a list of container mount points and the corresponding path on the host directory tree. The output will look like this:

```
{"/var/lib/cassandra/data":"/mnt/sda1/var/lib/docker/vfs/dir/632fa59c..."}
```

The `Volumes` key points to a value that is itself a map. In this map each key is a mount point in the container, and the value is the location of the directory on the host file system. Here we've inspected a container with one volume. The map is sorted by the lexicographical ordering of its keys and is independent of the ordering specified when the container is created.

TIP VirtualBox (Docker Machine or Boot2Docker) users should keep in mind that the host path specified in each value is relative to their virtual machine root file system and not the root of their host. Managed volumes are created on the machine that's running the Docker daemon, but VirtualBox will create bind mount volumes that reference directories or files on the host machine.

Docker-managed volumes may seem difficult to work with if you're manually building or linking tools together on your desktop, but in larger systems where specific locality of the data is less important, managed volumes are a much more effective way to organize your data. Using them decouples volumes from other potential concerns of the system. By using Docker-managed volumes, you're simply stating, "I need a place to put some data that I'm working with." This is a requirement that Docker can fill on any machine with Docker installed. Further, when you're finished with a volume and you ask Docker to clean things up for you, Docker can confidently remove any directories or files that are no longer being used by a container. Using volumes in this way helps manage clutter. As Docker middleware or plugins evolve, managed volume users will be able to adopt more advanced features like portable volumes.





Sharing access to data is a key feature of volumes. If you have decoupled volumes from known locations on the file system, you need to know how to share volumes between containers without exposing the exact location of managed containers. The next section describes two ways to share data between containers using volumes.


4.3 **Sharing volumes**

Suppose you have a web server running inside a container that logs all the requests it receives to `/logs/access`. If you want to move those logs off your web server into storage that's more permanent, you might do that with a script inside another container. Sharing volumes between containers is where their value becomes more obvious. Just as there are two types of volume, there are two ways to share volumes between containers.

4.3.1 **Host-dependent sharing**

You've already read about the tools needed to implement host-dependent sharing. Two or more containers are said to use host-dependent sharing when each has a bind mount volume for a single known location on the host file system. This is the most obvious way to share some disk space between containers. You can see it in action in the following example:

<pre>mkdir ~/web-logs-example docker run --name plath -d \ -v ~/web-logs-example:/data \ dockerinaction/ch4_writer_a docker run --rm \ -v ~/web-logs-example:/reader-data \ alpine:latest \ head /reader-data/logA cat ~/web-logs-example/logA docker stop plath</pre>	<div style="margin-bottom: 20px;">  Set up a known location </div> <div style="margin-bottom: 20px;">  Bind mount the location into a log-writing container </div> <div style="margin-bottom: 20px;">  Bind mount the same location into a container for reading </div> <div>  View the logs from the host </div>
--	---

Stop the writer


```
docker stop plath
```

In this example you created two containers: one named `plath` that writes lines to a file and another that views the top part of the file. These containers share a common bind

mount volume. Outside any container you can see the changes by listing the contents of the directory you created or viewing the new file.

Explore ways that containers might be linked together in this way. The next example starts four containers—two log writers and two readers:

```
docker run --name woolf -d \
  --volume ~/web-logs-example:/data \
  dockerinaction/ch4_writer_a

docker run --name alcott -d \
  -v ~/web-logs-example:/data \
  dockerinaction/ch4_writer_b

docker run --rm --entrypoint head \
  -v ~/web-logs-example:/towatch:ro \
  alpine:latest \
  /towatch/logA

docker run --rm \
  -v ~/web-logs-example:/toread:ro \
  alpine:latest \
  head /toread/logB
```

In this example, you created four containers, each of which mounted the same directory as a volume. The first two containers are writing to different files in that volume. The third and fourth containers mount the volume at a different location and as read-only. This is a toy example, but it clearly demonstrates a feature that could be useful given the variety of ways that people build images and software.

Host-dependent sharing requires you to use bind mount volumes but—for the reasons mentioned at the end of section 4.2.1—bind mount volumes and therefore host-dependent sharing might cause problems or be too expensive to maintain if you’re working with a large number of machines. The next section demonstrates a shortcut to share both managed volumes and bind mount volumes with a set of containers.

4.3.2 Generalized sharing and the volumes-from flag

The `docker run` command provides a flag that will copy the volumes from one or more containers to the new container. The flag `--volumes-from` can be set multiple times to specify multiple source containers.

You used this flag in section 4.1.2 to copy the managed volume defined by a volume container into each of the containers running Cassandra. The example is realistic but fails to illustrate a few specific behaviors of the `--volumes-from` flag and managed containers:

```
docker run --name fowler \
  -v ~/example-books:/library/PoEAA \
  -v /library/DSL \
  alpine:latest \
  echo "Fowler collection created."
```

```
docker run --name knuth \
  -v /library/TAoCP.vol1 \
  -v /library/TAoCP.vol2 \
  -v /library/TAoCP.vol3 \
  -v /library/TAoCP.vol4.a \
  alpine:latest \
  echo "Knuth collection created"
```

```
docker run --name reader \
  --volumes-from fowler \
  --volumes-from knuth \
  alpine:latest ls -l /library/
```

**List all volumes as they were
copied into new container**

```
docker inspect --format "{{json .Volumes}}" reader
```

**Checkout volume
list for reader**

In this example you created two containers that defined Docker-managed volumes as well as a bind mount volume. To share these with a third container without the `--volumes-from` flag, you'd need to inspect the previously created containers and then craft bind mount volumes to the Docker-managed host directories. Docker does all this on your behalf when you use the `--volumes-from` flag. It copies any volume present on a referenced source container into the new container. In this case, the container named `reader` copied all the volumes defined by both `fowler` and `knuth`.

You can copy volumes directly or transitively. This means that if you're copying the volumes from another container, you'll also copy the volumes that it copied from some other container. Using the containers created in the last example yields the following:

```
docker run --name aggregator \
  --volumes-from fowler \
  --volumes-from knuth \
  alpine:latest \
  echo "Collection Created."
```

Create an aggregation

```
docker run --rm \
  --volumes-from aggregator \
  alpine:latest \
  ls -l /library/
```

**Consume volumes from a
single source and list them**

Copied volumes always have the same mount point. That means that you can't use `--volumes-from` in three situations.

In the first situation, you can't use `--volumes-from` if the container you're building needs a shared volume mounted to a different location. It offers no tooling for remapping mount points. It will only copy and union the mount points specified by the specified containers. For example, if the student in the last example wanted to mount the library to a location like `/school/library`, they wouldn't be able to do so.

The second situation occurs when the volume sources conflict with each other or a new volume specification. If one or more sources create a managed volume with the same mount point, then a consumer of both will receive only one of the volume definitions:

```
docker run --name chomsky --volume /library/ss \
  alpine:latest echo "Chomsky collection created."
```

```
docker run --name lampport --volume /library/ss \
  alpine:latest echo "Lampport collection created."

docker run --name student \
  --volumes-from chomsky --volumes-from lampport \
  alpine:latest ls -l /library/

docker inspect -f "{{json .Volumes}}" student
```

When you run the example, the output of `docker inspect` will show that the last container has only a single volume listed at `/library/ss` and its value is the same as one of the other two. Each source container defines the same mount point, and you create a race condition by copying both to the new container. Only one of the two copy operations can succeed.

A real-world example where this would be limiting is if you were copying the volumes of several web servers into a single container for inspection. If those servers are all running the same software or share common configuration (which is more likely than not in a containerized system), then all those servers might use the same mount points. In that case, the mount points would conflict, and you'd be able to access only a subset of the required data.

The third situation where you can't use `--volumes-from` is if you need to change the write permission of a volume. This is because `--volumes-from` copies the full volumes definition. For example, if your source has a volume mounted with read/write access, and you want to share that with a container that should have only read access, using `--volumes-from` won't work.

Sharing volumes with the `--volumes-from` flag is an important tool for building portable application architectures, but it does introduce some limitations. Using Docker-managed volumes decouples containers from the data and file system structure of the host machine, and that's critical for most production environments. The files and directories that Docker creates for managed volumes still need to be accounted for and maintained. To understand how Docker works with these files and how to keep your Docker environment clean, you need to understand the managed volume life cycle.

4.4 The managed volume life cycle

By this point in the chapter you should have quite a few containers and volumes to clean up. I've omitted cleanup instructions thus far so that you have a wealth of material to use in this section. Managed volumes have life cycles that are independent of any container, but as of this writing you can only reference them by the containers that use them.

4.4.1 Volume ownership

Managed volumes are second-class entities. You have no way to share or delete a specific managed volume because you have no way to identify a managed volume.

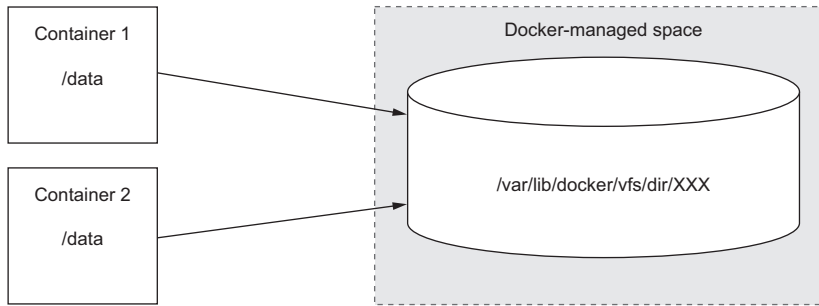


Figure 4.6 These two containers have an ownership relationship with a single managed volume.

Managed volumes are only created when you omit a bind mount source, and they're only identifiable by the containers that use them.

The highest fidelity way to identify volumes is to define a single container for each managed volume. In doing so, you can be very specific about which volumes you consume. More importantly, doing so helps you delete specific volumes. Unless you resort to examining volume mappings on a container and manually cleaning up the Docker-managed space, removing volumes requires a referencing container, and that makes it important to understand which containers own each managed volume. See figure 4.6.

A container owns all managed volumes mounted to its file system, and multiple containers can own a volume like in the fowler, knuth, and reader example. Docker tracks these references on managed volumes to ensure that no currently referenced volume is deleted.

4.4.2 *Cleaning up volumes*

Cleaning up managed volumes is a manual task. This default functionality prevents accidental destruction of potentially valuable data. Docker can't delete bind mount volumes because the source exists outside the Docker scope. Doing so could result in all manner of conflicts, instability, and unintentional data loss.

Docker can delete managed volumes when deleting containers. Running the `docker rm` command with the `-v` option will attempt to delete any managed volumes referenced by the target container. Any managed volumes that are referenced by other containers will be skipped, but the internal counters will be decremented. This is a safe default, but it can lead to the problematic scenario shown in figure 4.7.

If you delete every container that references a managed volume but fail to use the `-v` flag, you'll make that volume an orphan. Removing orphaned volumes requires messy manual steps, but depending on the size of the volumes it may be worth the effort. Alternatively, there are orphan volume cleanup scripts that you might consider using. You should carefully check those before running them. You'll need to run those scripts as a privileged user, and if they contain malware, you could be handing over full control of your system.

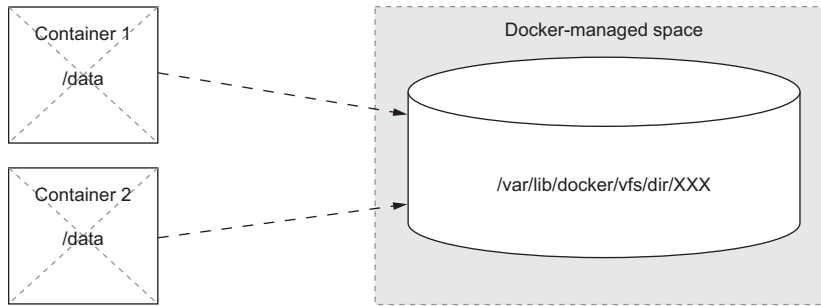


Figure 4.7 The user created an orphan volume by deleting the two owners of that volume without instructing Docker to remove the volumes attached to those containers.

It's a better idea to avoid the situation by getting into the habit of using the `-v` option and using the volume container pattern discussed in section 4.5 for critical data.

Docker creates volumes in another way that we haven't discussed. Image metadata can provide volume specifications. Chapter 7 includes details on this mechanism. In these cases, you may not even be aware of the volumes created for new containers. This is the primary reason to train yourself to use the `-v` option.

Orphan volumes render disk space unusable until you've cleaned them up. You can minimize this problem by remembering to clean them up and using a volume container pattern.

CLEANUP Before reading further, take a few moments to clean up the containers that you've created. Use `docker ps -a` to get a list of those containers and remember to use the `-v` flag on `docker rm` to prevent orphan volumes.

The following is a concrete example of removing a container from one of the earlier examples:

```
docker rm -v student
```

Alternatively, if you're using a POSIX-compliant shell, you can remove all stopped containers and their volumes with the following command:

```
docker rm -v $(docker ps -aq)
```

However you accomplish the task, cleaning up volumes is an important part of resource management. Now that you have a firm grasp on the volume life cycle, sharing mechanisms, and use cases, you should be ready to learn about advanced volume patterns.

4.5 Advanced container patterns with volumes

In the real world, volumes are used to accomplish a wide range of file system customizations and container interactions. This section focuses on a couple of advanced but common patterns that you may encounter or have a reason to employ in your own systems.

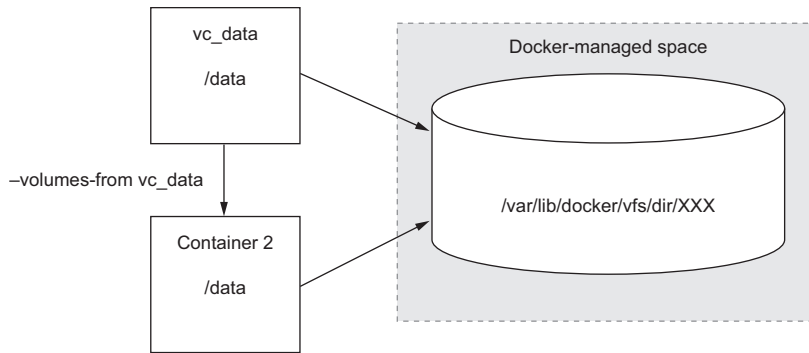


Figure 4.8 Container 2 copied `vc_data`'s volume references.

4.5.1 Volume container pattern

Sections 4.1.3 and 4.3.2 use a pattern called a volume container, which is a container that does little more than provide a handle to volumes. This is useful if you come across a case for sharing a set of volumes with many containers, or if you can categorize a set of volumes that fit a common use case; see figure 4.8.

A volume container doesn't need to be running because stopped containers maintain their volume references. Several of the examples you've read so far used the volume container pattern. The example containers `cass-shared`, `fowler`, `knuth`, `chomsky`, and `lampport` all ran a simple `echo` command to print something to the terminal and then exited. Then you used the stopped containers as sources for the `--volumes-from` flag when creating consumer containers.

Volume containers are important for keeping a handle on data even in cases where a single container should have exclusive access to some data. These handles make it possible to easily back up, restore, and migrate data.

Suppose you wanted to update your database software (use a new image). If your database container writes its state to a volume and that volume was defined by a volume container, the migration would be as simple as shutting down the original database container and starting the new one with the volume container as a volume source. Backup and restore operations could be handled similarly. This, of course, assumes that the new database software is able to read the storage format of the old software, and it looks for the data at the same location.

TIP Using a container name prefix such as `vc_` would be a great hint for humans or scripts not to use the `-v` option when deleting a container. The specific prefix is not as important as establishing some convention that people on your team and the tools you build can rely on.

Volume containers are most useful when you control and are able to standardize on mount point naming conventions. This is because every container that copies volumes

from a volume container inherits its mount point definitions. For example, a volume container that defines a volume mounted at `/logs` will only be useful to other containers that expect to be able to access a volume mounted at `/logs`. In this way, a volume and its mount point become a sort of contract between containers. For this reason, images that have specific volume requirements should clearly communicate those in their documentation or find a way to do so programmatically.

An example where two containers disagree might be where a volume container contributes a volume mounted at `/logs`, but the container that uses `--volumes-from` is expecting to find logs at `/var/logs`. In this case, the consuming container would be unable to access the material it needs, and the system would fail.

Consider another example with a volume container named `vc_data` that contributes two volumes: `/data` and `/app`. A container that has a dependency on the `/data` volume provided by `vc_data` but uses `/app` for something else would break if both volumes were copied in this way. These two containers are incompatible, but Docker has no way of determining intent. The error wouldn't be discovered until after the new container was created and failed in some way.

The volume container pattern is more about simplicity and convention than anything else. It's a fundamental tool for working with data in Docker and can be extended in a few interesting ways.

4.5.2 Data-packed volume containers

You can extend the volume container pattern and value added by packing containers with data, as illustrated in figure 4.9. Once you've adapted your containers to use volumes, you'll find all sorts of occasions to share volumes. Volume containers are in a unique position to seed volumes with data. The data-packed volume container extension formalizes that notion. It describes how images can be used to distribute static resources like configuration or code for use in containers created with other images.

A data-packed volume container is built from an image that copies static content from its image to volumes it defines. In doing so, these containers can be used to distribute critical architecture information like configuration, key material, and code.

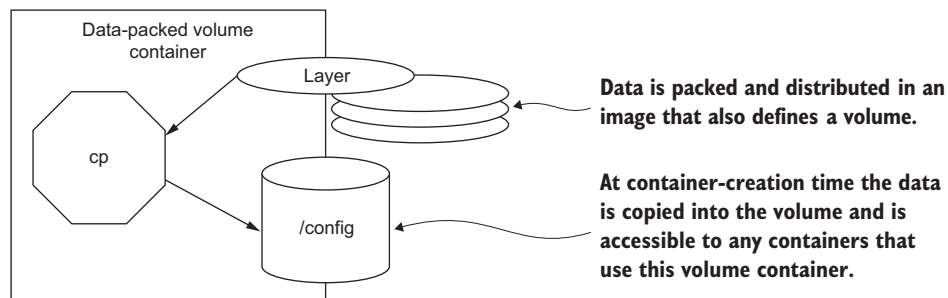


Figure 4.9 A data-packed volume container that contributes and populates a volume mounted at `/config`

You can build these by hand if you have an image that has the data you'd like to make available by running and defining the volume and running a `cp` command at container-creation time:

Copy image
content into
a volume

```
docker run --name dpvc \
  -v /config \
  dockerinaction/ch4_packed /bin/sh -c 'cp /packed/* /config/'
```

```
docker run --rm --volumes-from dpvc \
  alpine:latest ls /config
```

List shared material

```
docker run --rm --volumes-from dpvc \
  alpine:latest cat /config/packedData
```

View shared material

```
docker rm -v dpvc
```

Remember to use
-v when you clean up

The commands in this code share files distributed by a single image. You created three containers: one data-packed volume container and two that copied its volume and inspected the contents of the volume. Again, this is a toy example, but it demonstrates the way that you might consider distributing configuration in your own situations. Using data-packed volume containers to inject material into a new container is the basis for the polymorphic container pattern discussed in the next section.

4.5.3 Polymorphic container pattern

As I stated earlier in the chapter, a polymorphic tool is one that you interact with in a consistent way but might have several implementations that do different things. Using volumes, you can inject different behavior into containers without modifying an image. A polymorphic container is one that provides some functionality that's easily substituted using volumes. For example, you may have an image that contains the binaries for Node.js and by default executes a command that runs the Node.js program located at `/app/app.js`. The image might contain some default implementation that simply prints "This is a Node.js application" to the terminal.

You can change the behavior of containers created from this image by injecting your own `app.js` implementation using a volume mounted at `/app/app.js`. It might make more sense to layer that new functionality in a new image, but there are some cases when this is the best solution. The first is during development when you might not want to build a new image each time you iterate. The second is during operational events.

Consider a situation where an operational issue has occurred. In order to triage the issue, you might need tools available in an image that you had not anticipated when the image was built. But if you mount a volume where you make additional tools available, you can use the `docker exec` command to run additional processes in a container:

```
docker run --name tools dockerinaction/ch4_tools
```

Create data-packed volume
container with tools

```
docker run --rm \
  --volumes-from tools \
  alpine:latest \
  ls /operations/*
```

List shared tools

```

docker run -d --name important_application \
    --volumes-from tools \
    dockerinaction/ch4_ia

```

Start another container with shared tools

```

docker exec important_application /operations/tools/someTool

```

Use shared tool in running container

```

docker rm -vf important_application

```

Shut down the application

```

docker rm -v tools

```

Clean up the tools

You can inject files into otherwise static containers to change all types of behavior. Most commonly, you'll use polymorphic containers to inject application configuration. Consider a multi-state deployment pipeline where an application's configuration would change depending on where you deploy it. You might use data-packed volume containers to contribute environment-specific configuration at each stage, and then your application would look for its configuration at some known location:

```

docker run --name devConfig \
    -v /config \
    dockerinaction/ch4_packed_config:latest \
    /bin/sh -c 'cp /development/* /config/'

docker run --name prodConfig \
    -v /config \
    dockerinaction/ch4_packed_config:latest \
    /bin/sh -c 'cp /production/* /config/'

docker run --name devApp \
    --volumes-from devConfig \
    dockerinaction/ch4_polyapp

docker run --name prodApp \
    --volumes-from prodConfig \
    dockerinaction/ch4_polyapp

```

In this example, you start the same application twice but with a different configuration file injected. Using this pattern you can build a simple version-controlled configuration distribution system.

4.6 Summary

One of the first major hurdles in learning how to use Docker is understanding volumes and the file system. This chapter covers volumes in depth, including the following:

- Volumes allow containers to share files with the host or other containers.
- Volumes are parts of the host file system that Docker mounts into containers at specified locations.
- There are two types of volumes: Docker-managed volumes that are located in the Docker part of the host file system and bind mount volumes that are located anywhere on the host file system.
- Volumes have life cycles that are independent of any specific container, but a user can only reference Docker-managed volumes with a container handle.

- The orphan volume problem can make disk space difficult to recover. Use the `-v` option on `docker rm` to avoid the problem.
- The volume container pattern is useful for keeping your volumes organized and avoiding the orphan volume problem.
- The data-packed volume container pattern is useful for distributing static content for other containers.
- The polymorphic container pattern is a way to compose minimal functional components and maximize reuse.

5

Network exposure

This chapter covers

- Network container archetypes
- How Docker works with the computer's network
- How Docker builds network containers
- Ways to customize a container network
- Making containers available to the network
- Discovering other containers

In the previous chapter you read about how to use volumes and work with files in a container. This chapter deals with another common form of input and output: network access.

If you want to run a website, database, email server, or any software that depends on networking, like a web browser inside a Docker container, then you need to understand how to connect that container to the network. After reading this chapter you'll be able to create containers with network exposure appropriate for the application you're running, use network software in one container from another, and understand how containers interact with the host and the host's network.

This chapter is focused on single-host Docker networking. Multi-host Docker is the subject of chapter 12. That chapter describes strategies for service discovery

and the role container linking plays in that situation. You'll need the information in this chapter before any of that will make sense.

5.1 **Networking background**

A quick overview of relevant networking concepts will be helpful for understanding the topics in this chapter. This section includes only high-level detail; so if you're an expert, feel free to skip ahead.

Networking is all about communicating between processes that may or may not share the same local resources. To understand the material in this chapter you only need to consider a few basic network abstractions that are commonly used by processes. The better understanding you have of networking, the more you'll learn about the mechanics at work. But a deep understanding isn't required to use the tools provided by Docker. If anything, the material contained herein should prompt you to independently research selected topics as they come up. Those basic abstractions used by processes include protocols, network interfaces, and ports.

5.1.1 **Basics: protocols, interfaces, and ports**

A *protocol* with respect to communication and networking is a sort of language. Two parties that agree on a protocol can understand what each other is communicating. This is key to effective communication. Hypertext Transfer Protocol (HTTP) is one popular network protocol that many people have heard of. It's the protocol that provides the World Wide Web. A huge number of network protocols and several layers of communication are created by those protocols. For now, it's only important that you know what a protocol is so that you can understand network interfaces and ports.

A network *interface* has an address and represents a location. You can think of interfaces as analogous to real-world locations with addresses. A network interface is like a mailbox. Messages are delivered to a mailbox for recipients at that address, and messages are taken from a mailbox to be delivered elsewhere.

Whereas a mailbox has a postal address, a network interface has an *IP address*, which is defined by the Internet Protocol. The details of IP are interesting but outside of the scope of this book. The important thing to know about IP addresses is that they are unique in their network and contain information about their location on their network.

It's common for computers to have two kinds of *interfaces*: an Ethernet interface and a loopback interface. An Ethernet interface is what you're likely most familiar with. It's used to connect to other interfaces and processes. A loopback interface isn't connected to any other interface. At first this might seem useless, but it's often useful to be able to use network protocols to communicate with other programs on the same computer. In those cases a loopback is a great solution.

In keeping with the mailbox metaphor, a *port* is like a recipient or a sender. There might be several people who receive messages at a single address. For example, a single address might receive messages for Wendy Webserver, Deborah Database, and

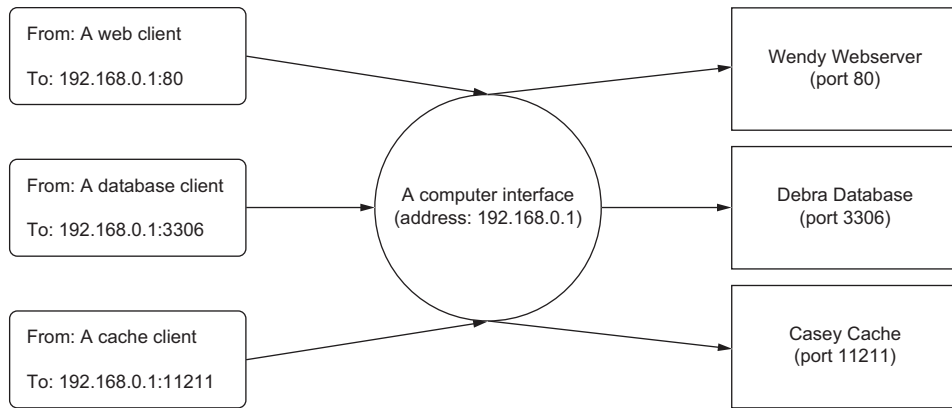


Figure 5.1 Processes use the same interface and are uniquely identified in the same way multiple people might use the same mailbox.

Casey Cache, as illustrated in figure 5.1. Each recipient should only open his or her own messages.

In reality, ports are just numbers and defined as part of the Transmission Control Protocol (TCP). Again the details of the protocol are beyond the scope of this book, but I encourage you to read about it some time. People who created standards for protocols, or companies that own a particular product, decide what port number should be used for specific purposes. For example, web servers provide HTTP on port 80 by default. MySQL, a database product, serves its protocol on port 3306 by default. Memcached, a fast cache technology, provides its protocol on port 11211. Ports are written on TCP messages just like names are written on envelopes.

Interfaces, protocols, and ports are all immediate concerns for software and users. By learning about these things, you develop a better appreciation for the way programs communicate and how your computer fits into the bigger picture.

5.1.2 Bigger picture: networks, NAT, and port forwarding

Interfaces are single points in larger networks. Networks are defined in the way that interfaces are linked together, and that linkage determines an interface's IP address.

Sometimes a message has a recipient that an interface is not directly linked to, so instead it's delivered to an intermediary that knows how to route the message for delivery. Coming back to the mail metaphor, this is similar to how real-world mail carriers operate.

When you place a message in your outbox, a mail carrier picks it up and delivers it to a local routing facility. That facility is itself an interface. It will take the message and send it along to the next stop on the route to a destination. A local routing facility for a mail carrier might forward a message to a regional facility, and then to a local facility for the destination, and finally to the recipient. It's common for network routes to

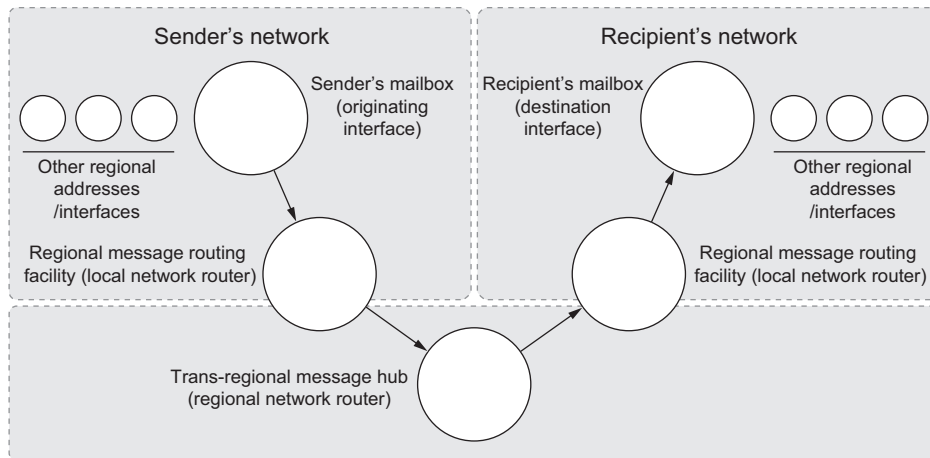


Figure 5.2 The path of a message in a postal system and a computer network

follow a similar pattern. Figure 5.2 illustrates the described route and draws the relationships between physical message routing and network routing.

This chapter is concerned with interfaces that exist on a single computer, so the networks and routes we consider won't be anywhere near that complicated. In fact, this chapter is about two specific networks and the way containers are attached to them. The first network is the one that your computer is connected to. The second is a virtual network that Docker creates to connect all of the running containers to the network that the computer is connected to. That second network is called a *bridge*.

Just as the name implies, a bridge is an interface that connects multiple networks so that they can function as a single network, as shown in figure 5.3. Bridges work by selectively forwarding traffic between the connected networks based on another type of network address. To understand the material in this chapter, you only need to be comfortable with this abstract idea.

This has been a very rough introduction to some nuanced topics. I've really only scratched the surface in order to help you understand how to use Docker and the networking facilities that it simplifies.

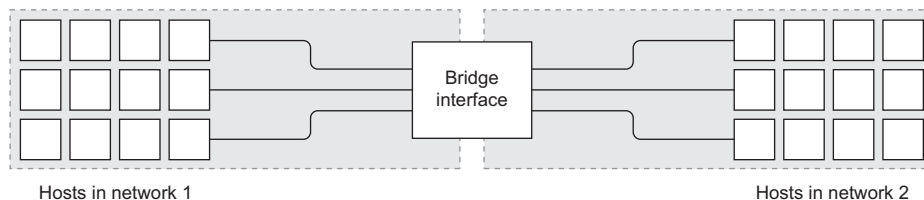


Figure 5.3 A bridge interface connecting two distinct networks

5.2 Docker container networking

Docker is concerned with two types of networking: single-host virtual networks and multi-host networks. Local virtual networks are used to provide container isolation. Multi-host virtual networks provide an overlay where any container on a participating host can have its own routable IP address from any other container in the network.

This chapter covers single-host virtual networks in depth. Understanding how Docker isolates containers on the network is critical for the security-minded. People building networked applications need to know how containerization will impact their deployment requirements.

Multi-host networking is still in beta at the time of this writing. Implementing it requires a broader understanding of other ecosystem tools in addition to understanding the material covering single-host networking. Until multi-host networking settles, it's best to get started by understanding how Docker builds local virtual networks.

5.2.1 The local Docker network topology

Docker uses features of the underlying operating system to build a specific and customizable virtual network topology. The virtual network is local to the machine where Docker is installed and is made up of routes between participating containers and the wider network where the host is attached. You can change the behavior of that network structure and in some cases change the structure itself by using command-line options for starting the Docker daemon and each container. Figure 5.4 illustrates two containers attached to the virtual network and its components.

Containers have their own private loopback interface and a separate Ethernet interface linked to another virtual interface in the host's namespace. These two linked interfaces form a link between the host's network stack and the stack created for each container. Just like typical home networks, each container is assigned a unique private IP address that's not directly reachable from the external network. Connections are routed through the Docker bridge interface called `docker0`. You can think of the

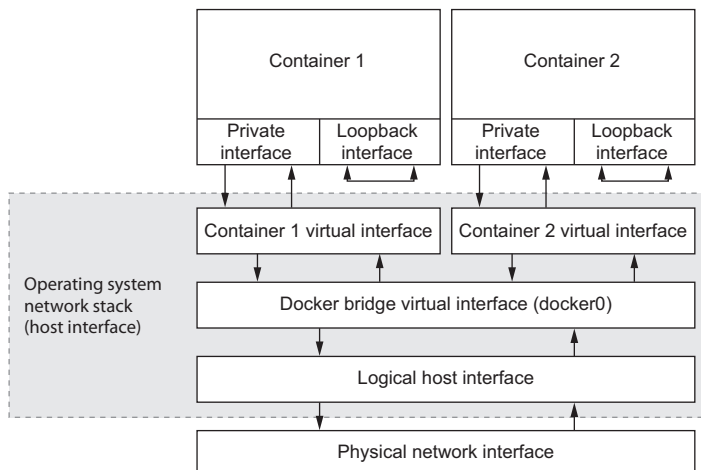


Figure 5.4 The default local Docker network topology and two attached containers