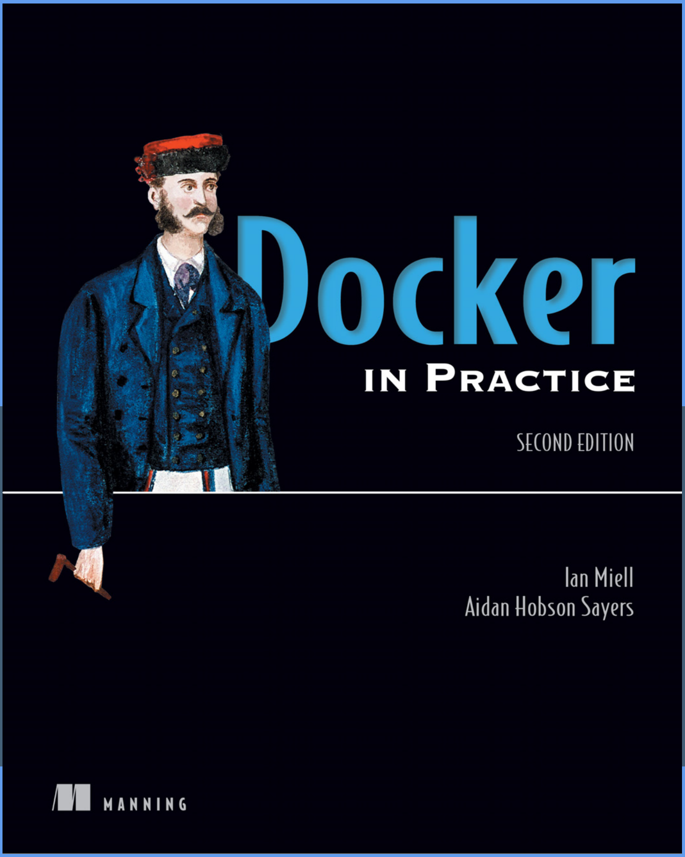
CHAPTER 3

Docker in Practice

SECOND EDITION

Ian Miell
Aidan Hobson Sayers

MANNING



Docker in Practice, Second Edition
by Ian Miell and Aidan Hobson Sayers

ISBN 9781617294808
384 pages
$39.99

Docker in Practice

Second Edition

Ian Miell and Aidan Hobson Sayers

Chapter 3

To pre-order or learn more about these books go to www.manning.com

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: Erin Twohey, corp-sales@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Marija Tudor

ISBN: 9781617294808

contents

3

Using Docker as a lightweight virtual machine

This chapter covers

- Converting a virtual machine to a Docker image
- Managing the startup of your container's services
- Saving your work as you go
- Managing Docker images on your machine
- Sharing images on the Docker Hub
- Playing—and winning—at 2048 with Docker

Virtual machines (VMs) have become ubiquitous in software development and deployment since the turn of the century. The abstraction of machines to software has made the movement and control of software and services in the internet age easier and cheaper.

> **TIP** A virtual machine is an application that emulates a computer, usually to run an operating system and applications. It can be placed on any (compatible) physical resources that are available. The end user experiences the software as though it were on a physical machine, but those managing the hardware can focus on larger-scale resource allocation.

Docker isn't a VM technology. It doesn't simulate a machine's hardware and it doesn't include an operating system. A Docker container is not, by default, constrained to specific hardware limits. If Docker virtualizes anything, it virtualizes the environment in which services run, not the machine. Moreover, Docker can't easily run Windows software (or even that written for other Unix-derived operating systems).

From some standpoints, though, Docker can be used much as a VM. For developers and testers in the internet age, the fact that there's no init process or direct hardware interaction is not usually of great significance. And there are significant commonalities, such as its isolation from the surrounding hardware and its amenability to more fine-grained approaches to software delivery.

This chapter will take you through the scenarios in which you could use Docker as you previously might have used a VM. Using Docker won't give you any obvious functional advantages over a VM, but the speed and convenience Docker brings to the movement and tracking of environments can be a game-changer for your development pipeline.

3.1 From VM to container

In an ideal world, moving from VMs to containers would be a simple matter of running configuration management scripts against a Docker image from a distribution similar to the VM's. For those of us who aren't in that happy state of affairs, this section will show how you can convert a VM to a container—or containers.

TECHNIQUE 11 Converting your VM to a container

The Docker Hub doesn't have all possible base images, so for some niche Linux distributions and use cases, people need to create their own. For example, if you have an existing application state in a VM, you may want to put that state inside a Docker image so that you can iterate further on that, or to benefit from the Docker ecosystem by using tooling and related technology that exists there.

Ideally you'd want to build an equivalent of your VM from scratch using standard Docker techniques, such as Dockerfiles combined with standard configuration management tools (see chapter 7). The reality, though, is that many VMs aren't carefully configuration-managed. This might happen because a VM has grown organically as people have used it, and the investment needed to recreate it in a more structured way isn't worth it.

PROBLEM

You have a VM you want to convert to a Docker image.

SOLUTION

Archive and copy the filesystem of the VM and package it into a Docker image.

First we're going to divide VMs into two broad groups:

- *Local VM*—VM disk image lives on and VM execution happens on your computer.
- *Remote VM*—VM disk image storage and VM execution happen somewhere else.

The principle for both groups of VMs (and anything else you want to create a Docker image from) is the same—you get a TAR of the filesystem and `ADD` the TAR file to / of the `scratch` image.

TIP The `ADD` Dockerfile command (unlike its sibling command `COPY`) unpacks TAR files (as well as gzipped files and other similar file types) when they're placed in an image like this.

TIP The *scratch* image is a zero-byte pseudo-image you can build on top of. Typically it's used in cases like this where you want to copy (or add) a complete filesystem using a Dockerfile.

We'll now look at a case where you have a local VirtualBox VM.

Before you get started, you need to do the following:

1. Install the `qemu-nbd` tool (available as part of the `qemu-utils` package on Ubuntu).
2. Identify the path to your VM disk image.
3. Shut down your VM.

If your VM disk image is in the .vdi or .vmdk format, this technique should work well. Other formats may experience mixed success. The following code demonstrates how you can turn your VM file into a virtual disk, which allows you to copy all the files from it.

Listing 3.1 Extracting the filesystem of a VM image

```
$ VMDISK="$HOME/VirtualBox VMs/myvm/myvm.vdi"
 $ sudo modprobe nbd
 $ sudo qemu-nbd -c /dev/nbd0 -r $VMDISK3((CO1-3))
 $ ls /dev/nbd0p*
 /dev/nbd0p1 /dev/nbd0p2
$ sudo mount /dev/nbd0p2 /mnt
 $ sudo tar cf img.tar -C /mnt .
 $ sudo umount /mnt && sudo qemu-nbd -d /dev/nbd0
```

Sets up a variable pointing to your VM disk image

Initializes a kernel module required by qemu-nbd

Connects the VM disk to a virtual device node

Lists the partition numbers available to mount on this disk

Mounts the selected partition at /mnt with qemu-nbd

Creates a TAR file called img.tar from /mnt

Unmounts and cleans up after qemu-nbd

NOTE To choose which partition to mount, run `sudo cfdisk /dev/nbd0` to see what's available. Note that if you see LVM anywhere, your disk has a nontrivial partitioning scheme—you'll need to do some additional research into how to mount LVM partitions.

If your VM is kept remotely, you have a choice: either shut down the VM and ask your operations team to perform a dump of the partition you want, or create a TAR of your VM while it's still running.

If you get a partition dump, you can mount this fairly easily and then turn it into a TAR file as follows:

Listing 3.2 Extracting a partition

```
$ sudo mount -o loop partition.dump /mnt
$ sudo tar cf $(pwd)/img.tar -C /mnt .
$ sudo umount /mnt
```

Alternatively, you can create a TAR file from a running system. This is quite simple after logging into the system:

Listing 3.3 Extracting the filesystem of a running VM

```
$ cd /
$ sudo tar cf /img.tar --exclude=/img.tar --one-file-system /
```

You now have a TAR of the filesystem image that you can transfer to a different machine with `scp`.

> **WARNING** Creating a TAR from a running system may seem like the easiest option (no shutdowns, installing software, or making requests to other teams), but it has a severe downside—you could copy a file in an inconsistent state and hit strange problems when trying to use your new Docker image. If you must go this route, first stop as many applications and services as possible.

Once you've got the TAR of your filesystem, you can add it to your image. This is the easiest step of the process and consists of a two-line Dockerfile.

Listing 3.4 Adding an archive to a Docker image

```
FROM scratch
ADD img.tar /
```

You can now run `docker build .` and you have your image!

> **NOTE** Docker provides an alternative to `ADD` in the form of the `docker import` command, which you can use with `cat img.tar | docker import - new_image _name`. But building on top of the image with additional instructions will require you to create a Dockerfile anyway, so it may be simpler to go the `ADD` route, so you can easily see the history of your image.

You now have an image in Docker, and you can start experimenting with it. In this case, you might start by creating a new Dockerfile based on your new image, to experiment with stripping out files and packages.

Once you've done this and are happy with your results, you can use `docker export` on a running container to export a new, slimmer TAR that you can use as the basis for a newer image, and repeat the process until you get an image you're happy with.

The flow chart in figure 3.1 demonstrates this process.

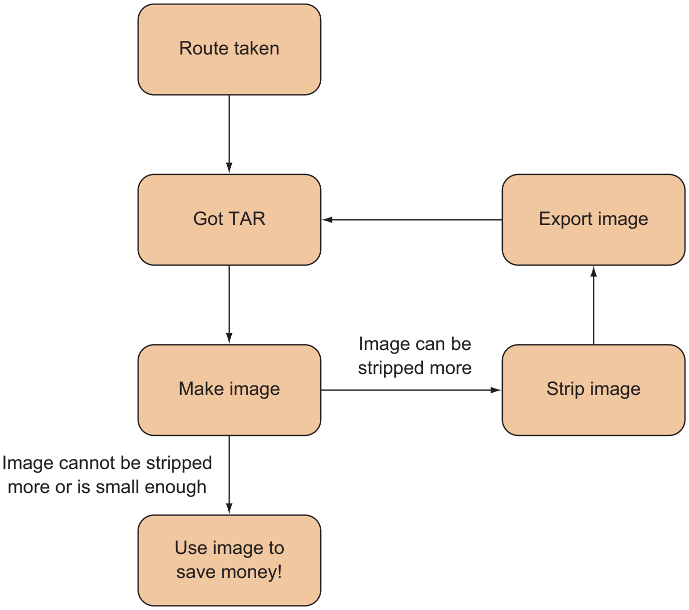


Figure 3.1 Image-stripping flowchart

DISCUSSION

This technique demonstrates a few fundamental principles and techniques that are useful in contexts other than converting a VM to a Docker image.

Most broadly, it shows that a Docker image is essentially a set of files and some metadata: the `scratch` image is an empty filesystem over which a TAR file can be laid. We'll return to this theme when we look at *slim* Docker images.

More specifically, you've seen how you can `ADD` a TAR file to a Docker image, and how you can use the `qemu-nbd` tool.

Once you have your image, you might need to know how to run it like a more traditional host. Because Docker containers typically run one application process only, this is somewhat against the grain, and it's covered in the next technique.

TECHNIQUE 12 A host-like container

We'll now move on to one of the more contentious areas of discussion within the Docker community—running a host-like image, with multiple processes running from the start.

This is considered bad form in parts of the Docker community. Containers are not virtual machines—there are significant differences—and pretending there aren't can cause confusion and issues down the line.

For good or ill, this technique will show you how to run a host-like image and discuss some of the issues around doing this.

NOTE Running a host-like image can be a good way to persuade Docker refuseniks that Docker is useful. As they use it more, they'll understand the paradigm better and the microservices approach will make more sense to them. At the company we introduced Docker into, we found that this monolithic approach was a great way to move people from developing on dev servers and laptops to a more contained and manageable environment. From there, moving Docker into testing, continuous integration, escrow, and DevOps workflows was trivial.

Differences between VMs and Docker containers

These are a few of the differences between VMs and Docker containers:

- Docker is application-oriented, whereas VMs are operating-system oriented.
- Docker containers share an operating system with other Docker containers. In contrast, VMs each have their own operating system managed by a hypervisor.
- Docker containers are designed to run one principal process, not manage multiple sets of processes.

PROBLEM

You want a normal host-like environment for your container with multiple processes and services set up.

SOLUTION

Use a base container designed to run multiple processes.

For this technique you're going to use an image designed to simulate a host, and provision it with the applications you need. The base image will be the phusion/baseimage Docker image, an image designed to run multiple processes.

The first steps are to start the image and jump into it with `docker exec`.

Listing 3.5 Running the phusion base image

```
user@docker-host$ docker run -d phusion/baseimage
 3c3f8e3fb05d795edf9d791969b21f7f73e99eb1926a6e3d5ed9e1e52d0b446e
 user@docker-host$ docker exec -i -t 3c3f8e3fb05d795 /bin/bash
 root@3c3f8e3fb05d:/#
```

Starts the image in the background

Returns the ID of the new container

Passes the container ID to docker exec and allocates an interactive terminal

The prompt to the started container terminal

In this code, `docker run` will start the image in the background, starting the default command for the image and returning the ID of the newly created container.

You then pass this container ID to `docker exec`, which is a command that starts a new process inside an already running container. The `-i` flag allows you to interact with the new process, and `-t` indicates that you want to set up a TTY to allow you to start a terminal (`/bin/bash`) inside the container.

If you wait a minute and then look at the processes table, your output will look something like the following.

Listing 3.6 Processes running in a host-like container

The bash process started by docker exec and acting as your shell

Runs a ps command to list all the running processes

A simple init process designed to run all the other services

```
root@3c3f8e3fb05d:/# ps -ef
 UID  PID  PPID  C STIME TTY     TIME CMD
root    1     0  0 13:33 ?   00:00:00 /usr/bin/python3 -u /sbin/my_init
 root    7     0  0 13:33 ?   00:00:00 /bin/bash
 root  111     1  0 13:33 ?   00:00:00 /usr/bin/runsvdir -P /etc/service
 root  112   111  0 13:33 ?   00:00:00 runsv cron
 root  113   111  0 13:33 ?   00:00:00 runsv sshd
root  114   111  0 13:33 ?   00:00:00 runsv syslog-ng
root  115   112  0 13:33 ?   00:00:00 /usr/sbin/cron -f
root  116   114  0 13:33 ?   00:00:00 syslog-ng -F -p /var/run/syslog-ng.pid
➥ --no-caps
root  117   113  0 13:33 ?   00:00:00 /usr/sbin/sshd -D
root  125     7  0 13:38 ?   00:00:00 ps -ef
```

runsvdir runs the services defined in the passed-in /etc/service directory.

The three standard services (cron, sshd, and syslog) are started here with the runsv command.

The ps command currently being run

You can see that the container starts up much like a host, initializing services such as cron and sshd that make it appear similar to a standard Linux host.

DISCUSSION

Although this can be useful in initial demos for engineers new to Docker or genuinely useful for your particular circumstances, it's worth being aware that it's a somewhat controversial idea.

The history of container use has tended toward using them to isolate workloads to "one service per container." Proponents of the host-like image approach argue that this doesn't violate that principle, because the container can still fulfill a single discrete function for the system within which it runs.

More recently, the growing popularity of both the Kubernetes' pod and docker-compose concepts has made the host-like container relatively redundant—separate containers can be conjoined into a single entity at a higher level, rather than managing multiple processes using a traditional init service.

The next technique looks at how you can break up such a monolithic application into microservice-style containers.

TECHNIQUE 13 Splitting a system into microservice containers

We've explored how to use a container as a monolithic entity (like a classical server) and explained that it can be a great way to quickly move a system architecture onto Docker. In the Docker world, however, it's generally considered a best practice to split up your system as much as possible until you have one service running per container and have all containers connected by networks.

The primary reason for using one service per container is the easier separation of concerns through the single-responsibility principle. If you have one container doing one job, it's easier to put that container through the software development lifecycle of development, test, and production while worrying less about its interactions with other components. This makes for more agile deliveries and more scalable software projects. It does create management overhead, though, so it's good to consider whether it's worth it for your use case.

Putting aside the discussion of which approach is better for you right now, the best-practice approach has one clear advantage—experimentation and rebuilds are much faster when using Dockerfiles, as you'll see.

PROBLEM

You want to break your application up into distinct and more manageable services.

SOLUTION

Create a container for each discrete service process.

As we've touched upon already, there's some debate within the Docker community about how strictly the "one service per container" rule should be followed, with part of this stemming from a disagreement over the definitions—is it a single process, or a collection of processes that combine to fulfill a need? It often boils down to a statement that, given the ability to redesign a system from scratch, microservices is the route most would chose. But sometimes practicality beats idealism—when evaluating Docker for our organization, we found ourselves in the position of having to go the monolithic route in order get Docker working as quickly and easily as possible.

Let's take a look at one of the concrete disadvantages of using monoliths inside Docker. First, the following listing shows you how you'd build a monolith with a database, application, and web server.

> **NOTE** These examples are for explanation purposes and have been simplified accordingly. Trying to run them directly won't necessarily work.

Listing 3.7 Setting up a simple PostgreSQL, NodeJS, and Nginx application

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install postgresql nodejs npm nginx
WORKDIR /opt
COPY . /opt/                                  # {*}
```

```
RUN service postgresql start && \
    cat db/schema.sql | psql && \
    service postgresql stop
RUN cd app && npm install
RUN cp conf/mysite /etc/nginx/sites-available/ && \
    cd /etc/nginx/sites-enabled && \
    ln -s ../sites-available/mysite
```

TIP Each Dockerfile command creates a single new layer on top of the previous one, but using `&&` in your `RUN` statements effectively ensures that several commands get run as one command. This is useful because it can keep your images small. If you run a package update command like `apt-get update` with an install command in this way, you ensure that whenever the packages are installed, they'll be from an updated package cache.

The preceding example is a conceptually simple Dockerfile that installs everything you need inside the container and then sets up the database, application, and web server. Unfortunately, there's a problem if you want to quickly rebuild your container—any change to any file under your repository will rebuild everything starting from the {*} onwards, because the cache can't be reused. If you have some slow steps (database creation or `npm install`), you could be waiting for a while for the container to rebuild.

The solution to this is to split up the `COPY . /opt/` instruction into the individual aspects of the application (database, app, and web setup).

Listing 3.8 Dockerfile for a monolithic application

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install postgresql nodejs npm nginx
WORKDIR /opt
COPY db /opt/db                                          -+
RUN service postgresql start && \                         |- db setup
    cat db/schema.sql | psql && \                         |
    service postgresql stop                              -+
COPY app /opt/app                                        -+
RUN cd app && npm install                                 |- app setup
RUN cd app && ./minify_static.sh                         -+
COPY conf /opt/conf                                      -+
RUN cp conf/mysite /etc/nginx/sites-available/ && \       +
    cd /etc/nginx/sites-enabled && \                      |- web setup
    ln -s ../sites-available/mysite                      -+
```

In the preceding code, the `COPY` command is split into two separate instructions. This means the database won't be rebuilt every time code changes, as the cache can be reused for the unchanged files delivered before the code. Unfortunately, because the caching functionality is fairly simple, the container still has to be completely rebuilt every time a change is made to the schema scripts. The only way to resolve this is to move away from sequential setup steps and create multiple Dockerfiles, as shown in listings 3.9 through 3.11.

Listing 3.9 Dockerfile for the postgres service

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install postgresql
WORKDIR /opt
COPY db /opt/db
RUN service postgresql start && \
    cat db/schema.sql | psql && \
    service postgresql stop
```

Listing 3.10 Dockerfile for the nodejs service

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install nodejs npm
WORKDIR /opt
COPY app /opt/app
RUN cd app && npm install
RUN cd app && ./minify_static.sh
```

Listing 3.11 Dockerfile for the nginx service

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install nginx
WORKDIR /opt
COPY conf /opt/conf
RUN cp conf/mysite /etc/nginx/sites-available/ && \
    cd /etc/nginx/sites-enabled && \
    ln -s ../sites-available/mysite
```

Whenever one of the `db`, `app`, or `conf` folders changes, only one container will need to be rebuilt. This is particularly useful when you have many more than three containers or there are time-intensive setup steps. With some care, you can add the bare minimum of files necessary for each step and get more useful Dockerfile caching as a result.

In the app Dockerfile (listing 3.10), the operation of `npm install` is defined by a single file, `package.json`, so you can alter your Dockerfile to take advantage of Dockerfile layer caching and only rebuild the slow `npm install` step when necessary, as follows.

Listing 3.12 Faster Dockerfile for the nginx service

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install nodejs npm
WORKDIR /opt
COPY app/package.json /opt/app/package.json
RUN cd app && npm install
COPY app /opt/app
RUN cd app && ./minify_static.sh
```

Now you have three discrete, separate Dockerfiles where formerly you had one.

DISCUSSION

Unfortunately, there's no such thing as a free lunch—you've traded a single simple Dockerfile for multiple Dockerfiles with duplication. You can address this partially by adding another Dockerfile to act as your base image, but some duplication is not uncommon. Additionally, there's now some complexity in starting your image—in addition to `EXPOSE` steps making appropriate ports available for linking and altering of Postgres configuration, you need to be sure to link the containers every time they start up. Fortunately there's tooling for this called *Docker Compose*, which we'll cover in technique 76.

So far in this section you've taken a VM, turned it into a Docker image, run a host-like container, and broken a monolith into separate Docker images.

If, after reading this book, you still want to run multiple processes from within a container, there are specific tools that can help you do that. One of these—Supervisord—is treated in the next technique.

TECHNIQUE 14 Managing the startup of your container's services

As is made clear throughout the Docker literature, a Docker container is *not* a VM. One of the key differences between a Docker container and a VM is that a container is designed to run one process. When that process finishes, the container exits. This is different from a Linux VM (or any Linux OS) in that it doesn't have an init process.

The init process runs on a Linux OS with a process ID of 1 and a parent process ID of 0. This init process might be called "init" or "systemd." Whatever it's called, its job is to manage the housekeeping for all other processes running on that operating system.

If you start to experiment with Docker, you may find that you want to start multiple processes. You might want to run cron jobs to tidy up your local application log files, for example, or set up an internal memcached server within the container. If you take this path, you may end up writing shell scripts to manage the startup of these sub-processes. In effect, you'll be emulating the work of the init process. Don't do that! The many problems arising from process management have been encountered by others before and have been solved in prepackaged systems.

Whatever your reason for running multiple processes inside a container, it's important to avoid reinventing the wheel.

PROBLEM

You want to manage multiple processes within a container.

SOLUTION

Use Supervisor to manage the processes in your container.

We'll show you how to provision a container running both Tomcat and an Apache web server, and have it start up and run in a managed way, with the Supervisor application (http://supervisord.org/) managing process startup for you.

First, create your Dockerfile in a new and empty directory, as the following listing shows.

Listing 3.13 Example Supervisor Dockerfile

Installs python-pip (to install Supervisor), apache2, and tomcat7

Starts from ubuntu:l4.04

Sets an environment variable to indicate that this session is non-interactive

Installs Supervisor with pip

Copies the Apache and Tomcat supervisord configuration settings into the image, ready to add to the default configuration

```
FROM ubuntu:14.04
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update && apt-get install -y python-pip apache2 tomcat7
RUN pip install supervisor
RUN mkdir -p /var/lock/apache2
RUN mkdir -p /var/run/apache2
RUN mkdir -p /var/log/tomcat
RUN echo_supervisord_conf > /etc/supervisord.conf
ADD ./supervisord_add.conf /tmp/supervisord_add.conf
RUN cat /tmp/supervisord_add.conf >> /etc/supervisord.conf
RUN rm /tmp/supervisord_add.conf
CMD ["supervisord","-c","/etc/supervisord.conf"]
```

Creates housekeeping directories needed to run the applications

Creates a default supervisord configuration file with the echo_supervisord _conf utility

Appends the Apache and Tomcat supervisord configuration settings to the supervisord configuration file

You only need to run Supervisor now on container startup

Removes the file you uploaded, as it's no longer needed

You'll also need configuration for Supervisor, to specify what applications it needs to start up, as shown in the next listing.

Listing 3.14 supervisord_add.conf

Declares the global configuration section for supervisord

Doesn't daemonize the Supervisor process, as it's the foreground process for the container

```
[supervisord]
nodaemon=true

# apache
[program:apache2]
command=/bin/bash -c "source /etc/apache2/envvars && exec /usr/sbin/apache2
➥ -DFOREGROUND"

# tomcat
[program:tomcat]
command=service start tomcat
redirect_stderr=true
stdout_logfile=/var/log/tomcat/supervisor.log
stderr_logfile=/var/log/tomcat/supervisor.error_log
```

Section declaration for a new program

Commands to start up the programs declared in the section

Section declaration for a new program

Configuration pertaining to logging

Commands to start up the programs declared in the section

You build the image using the standard single-command Docker process because you're using a Dockerfile. Run this command to perform the build:

```
docker build -t supervised .
```

You can now run your image!

Listing 3.15 Run the supervised container

Maps the container's port 80 to the host's port 9000, gives the container a name, and specifies the image name you're running, as tagged with the build command previously

Starts up the Supervisor process

Starts up the Supervisor process

```
$ docker run -p 9000:80 --name supervised supervised
 2015-02-06 10:42:20,336 CRIT Supervisor running as root (no user in config
➥  file)
 2015-02-06 10:42:20,344 INFO RPC interface 'supervisor' initialized
2015-02-06 10:42:20,344 CRIT Server 'unix_http_server' running without any
➥  HTTP authentication checking
2015-02-06 10:42:20,344 INFO supervisord started with pid 1
 2015-02-06 10:42:21,346 INFO spawned: 'tomcat' with pid 12
 2015-02-06 10:42:21,348 INFO spawned: 'apache2' with pid 13
 2015-02-06 10:42:21,368 INFO reaped unknown pid 29
2015-02-06 10:42:21,403 INFO reaped unknown pid 30
2015-02-06 10:42:22,404 INFO success: tomcat entered RUNNING state, process
➥  has stayed up for > than 1 seconds (startsecs)
 2015-02-06 10:42:22,404 INFO success: apache2 entered RUNNING state, process
➥  has stayed up for > than 1 seconds (startsecs)
```

Starts up the managed processes

Managed processes have been deemed by Supervisor to have successfully started.

If you navigate to http://localhost:9000, you should see the default page of the Apache server you started up.

To clean up the container, run the following command:

```
docker rm -f supervised
```

DISCUSSION

This technique used Supervisor to manage multiple processes in your Docker container.

If you're interested in alternatives to Supervisor, there's also `runit`, which was used by the phusion base image covered in technique 12.

3.2 Saving and restoring your work

Some people say that code isn't written until it's committed to source control—it doesn't always hurt to have the same attitude about containers. It's possible to save state with VMs by using snapshots, but Docker takes a much more active approach in encouraging the saving and reusing of your existing work.

We'll cover the "save game" approach to development, the niceties of tagging, using the Docker Hub, and referring to specific images in your builds. Because these operations are considered so fundamental, Docker makes them relatively simple and quick. Nonetheless, this can still be a confusing topic for Docker newbies, so in the next section we'll take you through the steps to a fuller understanding of this subject.

TECHNIQUE 15 The "save game" approach: Cheap source control

If you've ever developed any kind of software, you've likely exclaimed, "I'm sure it was working before!" at least once. Perhaps your language was not as sober as this. The inability to restore a system to a known good (or maybe only "better") state when you're hurriedly hacking away at code to hit a deadline or fix a bug is the cause of many broken keyboards.

Source control has helped significantly with this, but there are two problems in this particular case:

- The source may not reflect the state of your "working" environment's filesystem.
- You may not be willing to commit the code yet.

The first problem is more significant than the second. Although modern source control tools like Git can easily create local throwaway branches, capturing the state of your entire development filesystem isn't the purpose of source control.

Docker provides a cheap and quick way to store the state of your container's development filesystem through its commit functionality, and that's what we're going to explore here.

PROBLEM

You want to save the state of your development environment.

SOLUTION

Regularly commit your container so you can recover state at that point.

Let's imagine you want to make a change to your to-do application from chapter 1. The CEO of ToDoCorp isn't happy and wants the title of the browser to show "ToDoCorp's ToDo App" instead of "Swarm+React - TodoMVC."

You're not sure how to achieve this, so you probably want to fire up your application and experiment by changing files to see what happens.

Listing 3.16 Debugging the application in a terminal

```
$ docker run -d -p 8000:8000 --name todobug1 dockerinpractice/todoapp
 3c3d5d3ffd70d17e7e47e90801af7d12d6fc0b8b14a8b33131fc708423ee4372
$ docker exec -i -t todobug1 /bin/bash 2((CO7-2))
```

The `docker  run` command starts the to-do application in a container in the background (`-d`), mapping the container's port 8000 to port 8000 on the host (`-p  8000:8000`), naming it todobug1 (`--name  todobug1`) for easy reference, and returning the container ID. The command started in the container will be the default command specified when the `dockerinpractice/todoapp` image was built. We've built it for you and made it available on the Docker Hub.

The second command will start `/bin/bash` in the running container. The name `todobug1` is used, but you can also use the container ID. The `-i` makes this `exec` run interactively, and `-t` makes sure that the `exec` will work as a terminal would.

Now you're in the container, so the first step in experimenting is to install an editor. We prefer vim, so we used these commands:

```
apt-get update
apt-get install vim
```

After a little effort, you realize that the file you need to change is local.html. You therefore change line 5 in that file as follows:

```
<title>ToDoCorp's ToDo App</title>
```

Then word comes through that the CEO might want the title to be in lowercase, as she's heard that it looks more modern. You want to be ready either way, so you commit what you have at the moment. In another terminal you run the following command.

Listing 3.17 Committing container state

```
$ docker commit todobug1
 ca76b45144f2cb31fda6a31e55f784c93df8c9d4c96bbeacd73cad9cd55d2970
```

Turns the container you created earlier into an image

The new image ID of the container you've committed

You've now committed your container to an image that you can run from later.

> **NOTE** Committing a container only stores the state of the filesystem at the time of the commit, not the processes. Docker containers aren't VMs, remember! If your environment's state depends on the state of running processes that aren't recoverable through standard files, this technique won't store the state as you need it. In this case, you'll probably want to look at making your development processes recoverable.

Next, you change local.html to the other possible required value:

```
<title>todocorp's todo app</title>
```

Commit again:

```
$ docker commit todobug1
071f6a36c23a19801285b82eafc99333c76f63ea0aa0b44902c6bae482a6e036
```

You now have two image IDs that represent the two options (ca76b45144f2 cb31fda6a31e55f784c93df8c9d4c96bbeacd73cad9cd55d2970 and 071f6a36c23a19801285b82eafc99333c76f63ea0aa0b44902c6bae482a6e036 in our example, but yours will be different). When the CEO comes in to evaluate which one she wants, you can run up either image and let her decide which one to commit.

You do this by opening up new terminals and running the following commands.

Listing 3.18 Running up both committed images as containers

```
$ docker run -p 8001:8000 \
 ca76b45144f2cb31fda6a31e55f784c93df8c9d4c96bbeacd73cad9cd55d2970
 $ docker run -p 8002:8000 \
 071f6a36c23a19801285b82eafc99333c76f63ea0aa0b44902c6bae482a6e036
```

Maps the container's port 8000 to the host's port 8001 and specifies the lowercase image ID

Maps the container's port 8000 to the host's port 8002 and specifies the uppercase image ID

In this way you can present the uppercase option as available on http://localhost:8001 and the lowercase option on http://localhost:8002.

> **NOTE** Any dependencies external to the container (such as databases, Docker volumes, or other services called) aren't stored on commit. This technique doesn't have any external dependencies so you don't need to worry about that.

DISCUSSION

This technique demonstrated `docker commit`'s functionality, and how it might be used in a development workflow. Docker users tend to be directed toward using `docker commit` only as part of a formal `commit-tag-push` workflow, so it's good to remember that it has other uses too.

We find this to be a useful technique when we've negotiated a tricky sequence of commands to set up an application. Committing the container, once successful, also records your bash session history, meaning that a set of steps for regaining the state of your system is available. This can save a *lot* of time! It's also useful when you're experimenting with a new feature and are unsure whether you're finished, or when you've recreated a bug and want to be as sure as possible that you can return to the broken state.

You may well be wondering whether there's a better way to reference images than with long random strings of characters. The next technique will look at giving these containers names you can more easily reference.

TECHNIQUE 16 Docker tagging

You've now saved the state of your container by committing, and you have a random string as the ID of your image. It's obviously difficult to remember and manage the large numbers of these image IDs. It would be helpful to use Docker's tagging functionality to give readable names (and tags) to your images and remind you what they were created for.

Mastering this technique will allow you to see what your images are for at a glance, making image management on your machine far simpler.

PROBLEM

You want to conveniently reference and store a Docker commit.

SOLUTION

Use the `docker tag` command to name your commits.

In its basic form, tagging a Docker image is simple.

Listing 3.19 A simple `docker tag` command

```
$ docker tag \
   071f6a36c23a19801285b82eafc99333c76f63ea0aa0b44902c6bae482a6e036 \
   imagename
```

The docker tag command

The image ID you want to give a name to

The name you want to give your image

This gives your image a name that you can refer to, like so:

```
docker run imagename
```

This is much easier than remembering random strings of letters and numbers!

If you want to share your images with others, there's more to tagging than this, though. Unfortunately, the terminology around tags can be rather confusing. Terms such as *image name* and *repository* are used interchangeably. Table 3.1 provides some definitions.

Table 3.1 Docker tagging terms

Term	Meaning
Image	A read-only layer.
Name	The name of your image, such as "todoapp."
Tag	As a verb, it refers to giving an image a name. As a noun, it's a modifier for your image name.
Repository	A hosted collection of tagged images that together create the filesystem for a container.

Perhaps the most confusing terms in this table are "image" and "repository." We've been using the term *image* loosely to mean a collection of layers that we spawn a container from, but technically an image is a single layer that refers to its parent layer recursively. A *repository* is hosted, meaning that it's stored somewhere (either on your Docker daemon or on a registry). In addition, a repository is a collection of tagged images that make up the filesystem for a container.

An analogy with Git can be helpful here. When cloning a Git repository, you check out the state of the files at the point you requested. This is analogous to an image. The repository is the entire history of the files at each commit, going back to the initial commit. You therefore check out the repository at the head's "layer." The other "layers" (or commits) are all there in the repository you've cloned.

In practice, the terms "image" and "repository" are used more or less interchangeably, so don't worry too much about this. But be aware that these terms exist and are used similarly.

What you've seen so far is how to give an image ID a name. Confusingly, this name isn't the image's "tag," although people often refer to it as that. We distinguish between the action "to tag" (verb) and the "tag" (noun) you can give to the image name. This tag (noun) allows you to name a specific version of the image. You might add a tag to manage references to different versions of the same image. For example, you could tag an image with a version name or the date of commit.

A good example of a repository with multiple tags is the Ubuntu image. If you pull the Ubuntu image and then run `docker images`, you'll get output similar to the following listing.

Listing 3.20 An image with multiple tags

```
$ docker images
REPOSITORY      TAG        IMAGE ID       CREATED       VIRTUAL SIZE
ubuntu          trusty     8eaa4ff06b53   4 weeks ago   192.7 MB
ubuntu          14.04      8eaa4ff06b53   4 weeks ago   192.7 MB
ubuntu          14.04.1    8eaa4ff06b53   4 weeks ago   192.7 MB
ubuntu          latest     8eaa4ff06b53   4 weeks ago   192.7 MB
```

The Repository column lists the hosted collection of layers called "ubuntu". Often this is referred to as the "image." The Tag column here lists four different names (trusty, 14.04, 14.04.1, and latest). The Image ID column lists identical image IDs. This is because these differently tagged images are identical.

This shows that you can have a repository with multiple tags from the same image ID. In theory, though, these tags could later point to different image IDs. If "trusty" gets a security update, for example, the image ID may be changed with a new commit by the maintainers and tagged with "trusty", "14.04.2", and "latest".

The default is to give your image a tag of "latest" if no tag is specified.

NOTE The "latest" tag has no special significance in Docker—it's a default for tagging and pulling. It *doesn't* necessarily mean that this was the last tag set for this image. The "latest" tag of your image may be an old version of the image, as versions built later may have been tagged with a specific tag like "v1.2.3".

DISCUSSION

In this section we covered Docker image tagging. In itself, this technique is relatively simple. The real challenge we've found—and focused on here—is navigating the loose use of terminology among Docker users. It's worth re-emphasizing that when people talk about an image, they might be referring to a tagged image, or even a repository. Another particularly common mistake is referring to an image as a container: "Just download the container and run it." Colleagues at work who have been using Docker for a while still frequently ask us, "What's the difference between a container and an image?"

In the next technique you'll learn how to share your now-tagged image with others using a Docker image hub.

TECHNIQUE 17 Sharing images on the Docker Hub

Tagging images with descriptive names would be even more helpful if you could share these names (and images) with other people. To satisfy this need, Docker comes with the ability to easily move images to other places, and Docker Inc. created the Docker Hub as a free service to encourage this sharing.

> **NOTE** To follow this technique, you'll need a Docker Hub account that you have logged into previously by running `docker login` on your host machine. If you haven't set one up, you can do so at http://hub.docker.com. Just follow the instructions to register.

PROBLEM

You want to share a Docker image publicly.

SOLUTION

Use the Docker Hub registry to share your image.

As with tagging, the terminology around registries can be confusing. Table 3.2 should help you understand how the terms are used.

Table 3.2 Docker registry terms

Term	Meaning
Username	Your Docker registry username.
Registry	Registries hold images. A registry is a store you can upload images to or download them from. Registries can be public or private.
Registry host	The host on which the Docker registry runs.
Docker Hub	The default public registry hosted at https://hub.docker.com.
Index	The same as a registry host. It appears to be a deprecated term.

As you've seen previously, it's possible to tag an image as many times as you like. This is useful for "copying over" an image so that you have control of it.

Let's say your username on the Docker Hub is "adev". The following three commands show how to copy the "debian:wheezy" image from the Docker Hub to be under your own account.

Listing 3.21 Copying a public image and pushing to adev's Docker Hub account

```
docker pull debian:wheezy
 docker tag debian:wheezy adev/debian:mywheezy1
 docker push adev/debian:mywheezy1
```

Pulls the Debian image from the Docker Hub

Tags the wheezy image with your own username (adev) and tag (mywheezyl)

Pushes the newly created tag

You now have a reference to the Debian wheezy image you downloaded that you can maintain, refer to, and build on.

If you have a private repository to push to, the process is identical, except that you must specify the address of the registry before the tag. Let's say you have a repository that's served from http://mycorp.private.dockerregistry. The following listing will tag and push the image.

Listing 3.22 Copying a public image and pushing to adev's private registry

```
docker pull debian
 docker tag debian:wheezy \
mycorp.private.dockerregistry/adev/debian:mywheezy1
 docker push mycorp.private.dockerregistry/adev/debian:mywheezy1
```

Pulls the Debian image from the Docker Hub

Tags the wheezy image with your registry (mycorp.private.dockerregistry), username (adev), and tag (mywheezyl)

Pushes the newly created tag to the private registry. Note that the private registry server's address is required both when tagging and pushing, so that Docker can be sure it's pushing to the right place.

The preceding commands won't push the image to the public Docker Hub but will push it to the private repository, so that anyone with access to resources on that service can pull it.

DISCUSSION

You now have the ability to share your images with others. This is a great way to share work, ideas, or even issues you're facing with other engineers.

Just as GitHub isn't the only publicly available Git server, Docker Hub isn't the only publicly available Docker registry. But like GitHub, it's the most popular. For example, RedHat has a hub at https://access.redhat.com/containers.

Again, like Git servers, public and private Docker registries might have different features and characteristics that make one or the other appeal to you. If you're evaluating them, you might want to think about things like cost (to buy, subscribe to, or maintain), adherence to APIs, security features, and performance.

In the next technique, we'll look at how specific images can be referenced to help avoid issues that arise when the image reference you're using is unspecific.

TECHNIQUE 18 Referring to a specific image in builds

Most of the time you'll be referring to generic image names in your builds, such as "node" or "ubuntu" and will proceed without problem.

If you refer to an image name, it's possible that the image can change while the tag remains the same, as paradoxical as it sounds. The repository name is only a reference, and it may be altered to point to a different underlying image. Specifying a tag with the colon notation (such as `ubuntu:trusty`) doesn't remove this risk either, as security updates can use the same tag to automatically rebuild vulnerable images.

Most of the time you'll want this—the maintainers of the image may have found an improvement, and patching security holes is generally a good thing. Occasionally, though, this can cause you pain. And this is not merely a theoretical risk: this has happened to us on a number of occasions, breaking continuous delivery builds in a way that's difficult to debug. In the early days of Docker, packages would be added to and removed from the most popular images regularly (including, on one memorable occasion, the disappearance of the `passwd` command!), making builds that previously worked suddenly break.

PROBLEM

You want to be sure that your build is from a specific and unchanging image.

SOLUTION

To be absolutely certain that you're building against a given set of files, specify a specific image ID in your Dockerfile.

Here's an example (which will likely not work for you):

Listing 3.23 Dockerfile with a specific image ID

```
FROM 8eaa4ff06b53
 RUN echo "Built from image id:" > /etc/buildinfo
 RUN echo "8eaa4ff06b53" >> /etc/buildinfo
 RUN echo "an ubuntu 14.4.01 image" >> /etc/buildinfo
 CMD ["echo","/etc/buildinfo"]
```

Builds from a specific image (or layer) ID

Runs commands within this image to record the image you built from within a file in the new image

The built image will by default output the information you recorded in the /etc/buildinfo file.

To build from a specific image (or layer) ID like this, the image ID and its data must be stored locally on your Docker daemon. The Docker registry won't perform any kind of lookup to find the image ID in layers of images available to you on the Docker Hub, or in any other registry you may be configured to use.

Note that the image you refer to need not be tagged—it could be any layer you have locally. You can begin your build from any layer you wish. This might be useful for certain surgical or experimental procedures you want to perform for Dockerfile build analysis.

If you want to persist the image remotely, it's best to tag and push the image to a repository that's under your control in a remote registry.

> **WARNING** It's worth pointing out that almost the opposite problem can occur when a Docker image that was previously working suddenly does not. Usually this is because something on the network has changed. One memorable example of this was when our builds failed to `apt-get update` one morning. We assumed it was a problem with our local deb cache and tried debugging without success until a friendly sysadmin pointed out that the particular version of Ubuntu we were building from was no longer supported. This meant that the network calls to `apt-get update` were returning an HTTP error.

DISCUSSION

Although it might sound a little theoretical, it's important to understand the advantages and disadvantages of being more specific about the image you want to build or run.

Being more specific makes the result of your action more predictable and debuggable, as there's less ambiguity about which Docker image is or was downloaded. The disadvantage is that your image may not be the latest available, and you may therefore miss out on critical updates. Which state of affairs you prefer will depend on your particular use case and what you need to prioritize in your Docker environment.

In the next section, you'll apply what you've learned to a somewhat playful real-world scenario: winning at 2048.

3.3 Environments as processes

One way of viewing Docker is to see it as turning environments into processes. VMs can be treated in the same way, but Docker makes this much more convenient and efficient.

To illustrate this, we'll show you how the speedy spin-up, storage, and recreation of container state can allow you to do something otherwise (almost) impossible—winning at 2048!

TECHNIQUE 19 The "save game" approach: Winning at 2048

This technique is designed to provide you with a little light relief while showing you how Docker can be used to revert state easily. If you're not familiar with 2048, it's an addictive game where you push numbers around a board. The original version is available online at http://gabrielecirulli.github.io/2048 if you want to get acquainted with it first.

PROBLEM

You want to save container state regularly in order to revert to a known state if necessary.

SOLUTION

Use `docker commit` to "save game" whenever you are unsure whether you will survive in 2048.

We've created a monolithic image on which you can play 2048 within a Docker container that contains a VNC server and Firefox.

To use this image you'll need to install a VNC client. Popular implementations include TigerVNC and VNC Viewer. If you don't have one, a quick search for "vnc client" on the package manager on your host should yield useful results.

To start up the container, run the following commands.

Listing 3.24 Start the 2048 container

Run the imiell/win2048 image as a daemon

```
$ docker run -d -p 5901:5901 -p 6080:6080 --name win2048 imiell/win2048
 $ vncviewer localhost:1
```

Use VNC to get GUI access to the container

First you run a container from the `imiell/win2048` image that we've prepared for you. You start this in the background and specify that it should open two ports (5901 and 6080) to the host. These ports will be used by the VNC server started automatically inside the container. You also give the container a name for easy use later: `win2048`.

You can now run your VNC viewer (the executable may differ depending on what you have installed) and instruct it to connect to your local computer. Because the appropriate ports have been exposed from the container, connecting to localhost will actually connect to the container. The `:1` after `localhost` is appropriate if you have no X displays on your host, other than a standard desktop—if you do, you may need to choose a different number and look at the documentation for your VNC viewer to manually specify the VNC port as `5901`.

Once you're connected to the VNC server, you'll be prompted for a password. The password for VNC on this image is "vncpass". You'll then see a window with a Firefox tab and a 2048 table preloaded. Click on it to give it focus, and play until you're ready to save the game.

To save the game, you tag the named container after committing it:

Listing 3.25 Commit and tag the game state

Commits the **`win2048`** container

The tag that references your commit

```
$ docker commit win2048 1((CO14-1))
 4ba15c8d337a0a4648884c691919b29891cbbe26cb709c0fde74db832a942083
 $ docker tag 4ba15c8d337 my2048tag:$(date +%s)
```

Tags the commit with the current time as an integer

An image ID was generated by committing the `win2048` container, and now you want to give it a unique name (because you may be creating a number of these images). To do this, you can use the output of `date +%s` as part of the image name. This outputs the number of seconds since the first day of 1970, providing a unique (for our purposes), constantly increasing value. The `$(command)` syntax just substitutes the output of `command` at that position. If you prefer, you can run `date +%s` manually and paste the output as part of the image name instead.

You can then continue playing until you lose. Now comes the magic! You can return to your save point with the following commands.

Listing 3.26 Return to the saved game

```
$ docker rm -f win2048
$ docker run -d -p 5901:5901 -p 6080:6080 --name win2048 my2048tag:$mytag
```

`$mytag` is a tag selected from the `docker images` command. Repeat the `tag`, `rm`, and `run` steps until you complete 2048.

DISCUSSION

We hope that was fun. This example is more fun than practical, but we have used—and seen other developers use—this technique to great effect, especially when their environments are complex and the work they're doing is somewhat forensic and tricky.

Summary

- You can create a Docker container that looks like a "normal" host. Some consider this to be bad practice, but it may benefit your business or fit your use case.
- It's relatively easy to convert a VM to a Docker image to make the initial move to Docker.
- You can supervise services on your containers to mimic their previous VM-like operation.
- Committing is the correct way to save your work as you go.
- You can specify a particular Docker image to build from by using its build ID.
- You can name your images and share them with the world on the Docker Hub for free.
- You can even use Docker's commit capability to win at games like 2048!