7 Configuration management: Getting your house in order

This chapter covers

- Managing the building of images using Dockerfiles
- Building images using traditional configuration management tools
- Managing the secret information required to build images
- Reducing the size of your images for faster, lighter, and safer delivery

Configuration management is the art of managing your environments so that they're stable and predictable. Tools such as Chef and Puppet have attempted to alleviate the sysadmin burden of managing multiple machines. To an extent, Docker also reduces this burden by making the software environment isolated and portable. Even so, configuration management techniques are required to produce Docker images, and it's an important topic to recognize.

By the end of the chapter, you'll know how to integrate your existing tools with Docker, solve some Docker-specific problems like removing secrets from layers, and follow the best practice of minimizing your final image. As you get more experience with Docker, these techniques will give you the ability to build images for whatever configuration needs you're trying to satisfy.

7.1 Configuration management and Dockerfiles

Dockerfiles are considered to be the standard way of building Docker images. Dockerfiles are often confusing in terms of what they mean for configuration management. You may have many questions (particularly if you have experience with other configuration management tools), such as

- What happens if the base image changes?
- What happens if the packages I'm installing change and I rebuild?
- Does this replace Chef/Puppet/Ansible?

In fact, Dockerfiles are quite simple: starting from a given image, a Dockerfile specifies a series of shell commands and meta-instructions to Docker, which will produce the desired final image.

Dockerfiles provide a common, simple, and universal language for provisioning Docker images. Within them, you can use anything you like to reach the desired end state. You could call out to Puppet, copy in another script, or copy in an entire filesystem!

First we'll consider how you can deal with some minor challenges that Dockerfiles bring with them. Then we'll move on to the meatier issues we just outlined.

TECHNIQUE 49 Creating reliable bespoke tools with ENTRYPOINT

Docker's potential for allowing you to run commands *anywhere* means that complex bespoke instructions or scripts that are run on the command line can be preconfigured and wrapped up into a packaged tool.

The easily misunderstood `ENTRYPOINT` instruction is a vital part of this. You're going to see how it enables you to create Docker images as tools that are well-encapsulated, clearly defined, and flexible enough to be useful.

PROBLEM

You want to define the command the container will run, but leave the command's arguments up to the user.

SOLUTION

Use the Dockerfile `ENTRYPOINT` instruction.

As a demonstration, we'll imagine a simple scenario in a corporation where a regular admin task is to clean up old log files. Often this is prone to error, and people accidentally delete the wrong things, so we're going to use a Docker image to reduce the risk of problems arising.

The following script (which you should name "clean_log" when you save it) deletes logs over a certain number of days old, where the number of days is passed in as a

command-line option. Create a new folder anywhere with any name you like, move into it, and place `clean_log` within it.

Listing 7.1 `clean_log` shell script

```
#!/bin/bash
echo "Cleaning logs over $1 days old"
find /log_dir -ctime "$1" -name '*log' -exec rm {} \;
```

Note that the log cleaning takes place on the /log_dir folder. This folder will only exist when you mount it at runtime. You may have also noticed that there's no check for whether an argument has been passed in to the script. The reason for this will be revealed as we go through the technique.

Now let's create a Dockerfile in the same directory to create an image, with the script running as the defined command, or *entrypoint*.

Listing 7.2 Creating an image with the `clean_log` script

```
FROM ubuntu:17.04
ADD clean_log /usr/bin/clean_log
 RUN chmod +x /usr/bin/clean_log
ENTRYPOINT ["/usr/bin/clean_log"]
 CMD ["7"]
```

Adds the previous corporate clean_log script to the image

Defines the entrypoint for this image as being the clean_log script

Defines the default argument for the entrypoint command (7 days)

TIP You'll observe that we generally prefer the array form for `CMD` and `ENTRYPOINT` (for example, `CMD ["/usr/bin/command"]`) over the shell form (`CMD /usr/bin/command`). This is because the shell form automatically prepends a `/bin/bash -c` command to the command you supply, which can result in unexpected behavior. Sometimes, however, the shell form is more useful (see technique 55).

The difference between `ENTRYPOINT` and `CMD` often confuses people. The key point to understand is that an entrypoint will always be run when the image is started, even if a command is supplied to the `docker run` invocation. If you try to supply a command, it will add that as an argument to the entrypoint, replacing the default defined in the `CMD` instruction. You can only override the entrypoint if you explicitly pass in an `--entrypoint` flag to the `docker run` command. This means that running the image with a /bin/bash command won't give you a shell; rather, it will supply /bin/bash as an argument to the clean_log script.

The fact that a default argument is defined by the `CMD` instruction means that the argument supplied need not be checked. Here's how you might build and invoke this tool:

```
docker build -t log-cleaner .
docker run -v /var/log/myapplogs:/log_dir log-cleaner 365
```

After building the image, the image is invoked by mounting `/var/log/myapplogs` into the directory the script will use and passing 365 to remove log files over a year old, rather than a week.

If someone tries to use the image incorrectly by not specifying a number of days, they'll be given an error message:

```
$ docker run -ti log-cleaner /bin/bash
Cleaning logs over /bin/bash days old
find: invalid argument `-name' to `-ctime'
```

DISCUSSION

This example was quite trivial, but you can imagine that a corporation could apply it to centrally manage scripts used across its estate, such that they could be maintained and distributed safely with a private registry.

You can view and use the image we created in this technique at dockerinpractice/log-cleaner on the Docker Hub.

TECHNIQUE 50 Avoiding package drift by specifying versions

Dockerfiles have simple syntax and limited functionality, they can help greatly to clarify your build's requirements, and they can aid the stability of image production, but they can't guarantee repeatable builds. We're going to explore one of the numerous approaches to solving this problem and reducing the risk of nasty surprises when the underlying package management dependencies change.

This technique is helpful for avoiding those "it worked yesterday" moments, and it may be familiar if you've used classic configuration management tools. Building Docker images is fundamentally quite different from maintaining a server, but some hard-learned lessons are still applicable.

> **NOTE** This technique will only work for Debian-based images, such as Ubuntu. Yum users could find analogous techniques to make it work under their package manager.

PROBLEM

You want to ensure that your deb packages are the versions you expect.

SOLUTION

Run a script to capture the versions of all dependent packages on a system that's set up as you desire. Then install the specific versions in your Dockerfile, to ensure the versions are exactly as you expect.

A basic check for versions can be performed with an `apt-cache` call on a system you've verified as OK:

```
$ apt-cache show nginx | grep ^Version:
Version: 1.4.6-1ubuntu3
```

You can then specify the version in your Dockerfile like this:

```
RUN apt-get -y install nginx=1.4.6-1ubuntu3
```

This may be enough for your needs. What this doesn't do is guarantee that all dependencies from this version of nginx have the same versions that you originally verified.

You can get information about all of those dependencies by adding a `--recurse` flag to the argument:

```
apt-cache --recurse depends nginx
```

The output of this command is intimidatingly large, so getting a list of version requirements is tricky. Fortunately, we maintain a Docker image (what else?) to make this easier for you. It outputs the `RUN` line you need to put into your Dockerfile to ensure that the versions of all the dependencies are correct.

```
$ docker run -ti dockerinpractice/get-versions vim
RUN apt-get install -y \
vim=2:7.4.052-1ubuntu3 vim-common=2:7.4.052-1ubuntu3 \
vim-runtime=2:7.4.052-1ubuntu3 libacl1:amd64=2.2.52-1 \
libc6:amd64=2.19-0ubuntu6.5 libc6:amd64=2.19-0ubuntu6.5 \
libgpm2:amd64=1.20.4-6.1 libpython2.7:amd64=2.7.6-8 \
libselinux1:amd64=2.2.2-1ubuntu0.1 libselinux1:amd64=2.2.2-1ubuntu0.1 \
libtinfo5:amd64=5.9+20140118-1ubuntu1 libattr1:amd64=1:2.4.47-1ubuntu1 \
libgcc1:amd64=1:4.9.1-0ubuntu1 libgcc1:amd64=1:4.9.1-0ubuntu1 \
libpython2.7-stdlib:amd64=2.7.6-8 zlib1g:amd64=1:1.2.8.dfsg-1ubuntu1 \
libpcre3:amd64=1:8.31-2ubuntu2 gcc-4.9-base:amd64=4.9.1-0ubuntu1 \
gcc-4.9-base:amd64=4.9.1-0ubuntu1 libpython2.7-minimal:amd64=2.7.6-8 \
mime-support=3.54ubuntu1.1 mime-support=3.54ubuntu1.1 \
libbz2-1.0:amd64=1.0.6-5 libdb5.3:amd64=5.3.28-3ubuntu3 \
libexpat1:amd64=2.1.0-4ubuntu1 libffi6:amd64=3.1~rc1+r3.0.13-12 \
libncursesw5:amd64=5.9+20140118-1ubuntu1 libreadline6:amd64=6.3-4ubuntu2 \
libsqlite3-0:amd64=3.8.2-1ubuntu2 libssl1.0.0:amd64=1.0.1f-1ubuntu2.8 \
libssl1.0.0:amd64=1.0.1f-1ubuntu2.8 readline-common=6.3-4ubuntu2 \
debconf=1.5.51ubuntu2 dpkg=1.17.5ubuntu5.3 dpkg=1.17.5ubuntu5.3 \
libnewt0.52:amd64=0.52.15-2ubuntu5 libslang2:amd64=2.2.4-15ubuntu1 \
vim=2:7.4.052-1ubuntu3
```

At some point your build will fail because a version is no longer available. When this happens, you'll be able to see which package has changed and review the change to determine whether it's OK for your particular image's needs.

This example assumes that you're using ubuntu:14.04. If you're using a different flavor of Debian, fork the repo and change the Dockerfile's `FROM` instruction and build it. The repo is available here: https://github.com/docker-in-practice/get-versions.git.

Although this technique can help you with the stability of your build, it does nothing in terms of security, because you're still downloading packages from a repository you have no direct control over.

DISCUSSION

This technique may seem like a lot of effort to ensure that a text editor is exactly as you expect it to be. In the field, though, package drift can result in bugs that are incredibly difficult to pin down. Libraries and applications can move in subtle ways in builds from one day to the next, and figuring out what happened can ruin your day.

By pinning down the versions as tightly as possible within your Dockerfile, you ensure that one of two things happens. Either the build succeeds and your software will behave the same way as it did yesterday, or it fails to build because a piece of software has changed, and you'll need to retest your development pipeline. In the second case, you're aware of what's changed, and you can narrow down any failures that ensue to that specific change.

The point is that when you're doing continuing builds and integrations, reducing the number of variables that change reduces the time spent debugging. That translates to money for your business.

TECHNIQUE 51 Replacing text with perl -p -i -e

It's not uncommon when building images with Dockerfiles that you'll need to replace specific items of text across multiple files. Numerous solutions for this exist, but we'll introduce a somewhat unusual favorite that's particularly handy in Dockerfiles.

PROBLEM

You want to alter specific lines in files during a build.

SOLUTION

Use the `perl -p -i -e` command.

We recommend this command for a few reasons:

- Unlike `sed -i` (a command with a similar syntax and effect), this command works on multiple files out of the box, even if it encounters a problem with one of the files. This means you can run it across a directory with a `'*'` glob pattern without fear that it will suddenly break when a directory is added in a later revision of the package.
- As with `sed`, you can replace the forward slashes in the search and replace commands with other characters.
- It's easy to remember (we refer to it as the "perl pie" command).

NOTE This technique assumes an understanding of regular expressions. If you're not familiar with regular expressions, there are plenty of websites available to help you.

Here's a typical example of this command's use:

```
perl -p -i -e 's/127\.0\.0\.1/0.0.0.0/g' *
```

In this command, the `-p` flag asks Perl to assume a loop while it processes all the lines seen. The `-i` flag asks Perl to update the matched lines in place, and the `-e` flag asks Perl to treat the supplied string as a Perl program. The `s` is an instruction to Perl to search and replace strings as they're matched in the input. Here `127.0.0.1` is replaced with `0.0.0.0`. The `g` modifier ensures that all matches are updated, not just the first on any given line. Finally, the asterisk (`*`) applies the update to all files in this directory.

The preceding command performs a fairly common action for Docker containers. It replaces the standard localhost IP address (`127.0.0.1`) with one that indicates "any" IPv4 address (`0.0.0.0`) when used as an address to listen on. Many applications restrict access to the localhost IP by only listening on that address, and frequently you'll want to change this in their config files to the "any" address because you'll be accessing the application from your host, which appears to the container to be an external machine.

> **TIP** If an application within a Docker container appears not to be accessible to you from the host machine, despite the port being open, it can be worth trying to update the addresses to listen on to 0.0.0.0 in the application config file and restarting. It may be that the application is rejecting you because you're not coming from its localhost. Using `--net=host` (covered later in technique 109) when running your image can help confirm this hypothesis.

Another nice feature of `perl -p -i -e` (and `sed`) is that you can use other characters to replace the forward slashes if escaping the slashes gets awkward. Here's a real-world example from one of our scripts that adds some directives to the default Apache site file. This awkward command,

```
perl -p -i -e 's/\/usr\/share\/www/\/var\/www\/html/g' /etc/apache2/*
```

becomes this:

```
perl -p -i -e 's@/usr/share/www@/var/www/html/@g' /etc/apache2/*
```

In the rare cases that you want to match or replace both the `/` and `@` characters, you can try other characters such as `|` or `#`.

DISCUSSION

This is one of those tips that applies beyond the world of Docker as much as within it. It's a useful tool to have in your armory.

We find this technique particularly useful because of its broad application beyond use in Dockerfiles, combined with the ease with which it's remembered: it's "easy as pie," if you'll forgive the pun.

TECHNIQUE 52 Flattening images

A consequence of the design of Dockerfiles and their production of Docker images is that the final image contains the data state at each step in the Dockerfile. In the course of building your images, secrets may need to be copied in to ensure the build can work. These secrets may be SSH keys, certificates, or password files. Deleting these secrets before committing your image doesn't provide you with any real protection, as they'll be present in higher layers of the final image. A malicious user could easily extract them from the image.

One way of handling this problem is to flatten the resulting image.

PROBLEM

You want to remove secret information from the layer history of your image.

SOLUTION

Instantiate a container with the image, export it, import it, and then tag it with the original image ID.

To demonstrate a scenario where this could be useful, let's create a simple Dockerfile in a new directory that contains a Big Secret. Run `mkdir secrets && cd secrets` and then create a Dockerfile in that folder with the following contents.

Listing 7.3 A Dockerfile that copies in and deletes a secret

```
FROM debian
RUN echo "My Big Secret" >> /tmp/secret_key
 RUN cat /tmp/secret_key
 RUN rm /tmp/secret_key
```

Place a file with some secret information within your build.

Do something with the secret file. This Dockerfile only cats the file, but yours might SSH to another server or encrypt that secret within the image.

Remove the secret file.

Now run `docker build -t mysecret .` to build and tag that Dockerfile.

Once it's built, you can examine the layers of the resulting Docker image with the `docker history` command:

```
$ docker history mysecret
 IMAGE        CREATED        CREATED BY                                      SIZE
55f3c131a35d 3 days ago /bin/sh -c rm /tmp/secret_key                        0 B
 5b376ff3d7cd 3 days ago /bin/sh -c cat /tmp/secret_key                      0 B
5e39caf7560f 3 days ago /bin/sh -c echo "My Big Secret" >> /tmp/se 14 B
 c90d655b99b2 6 days ago /bin/sh -c #(nop) CMD [/bin/bash]                   0 B
30d39e59ffe2 6 days ago /bin/sh -c #(nop) ADD file:3f1a40df75bc567 85.01 MB
 511136ea3c5a 20 months ago                                                  0 B
```

Runs the docker history command against the name of the image you created

The layer where you removed the secret key

The layer where you added the secret key

The scratch (empty) layer

The layer that added the Debian filesystem. Note that this layer is the largest one in the history.

Now imagine that you've downloaded this image from a public registry. You could inspect the layer history and then run the following command to reveal the secret information:

```
$ docker run 5b376ff3d7cd cat /tmp/secret_key
My Big Secret
```

Here we've run a specific layer and instructed it to `cat` the secret key we removed at a higher layer. As you can see, the file is accessible.

Now you have a "dangerous" container with a secret inside that you've seen can be hacked to reveal its secrets. To make this image safe, you'll need to *flatten* it. This

means you'll keep the same data in the image but remove the intermediate layering information. To achieve this, you need to export the image as a trivially run container and then re-import and tag the resulting image:

```
$ docker run -d mysecret /bin/true
 28cde380f0195b24b33e19e132e81a4f58d2f055a42fa8406e755b2ef283630f
$ docker export 28cde380f | docker import - mysecret
 $ docker history mysecret
IMAGE             CREATED          CREATED BY   SIZE
fdbeae08751b  13 seconds ago                    85.01 MB
```

Runs a trivial command to allow the container to exit quickly, because you don't need it to be running

Runs docker export, taking a container ID as an argument and outputting a TAR file of the filesystem contents. This is piped to docker import, which takes a TAR file and creates an image from the contents.

The docker history output now shows only one layer with the final set of files.

The - argument to the `docker import` command indicates that you wish to read the TAR file from the command's standard input. The final argument to `docker import` indicates how the imported image should be tagged. In this case you're overwriting the previous tag.

Because there's now only one layer in the image, there's no record of the layers that contained the secrets. No secrets can now be extracted from the image.

DISCUSSION

This technique is useful enough to be re-used at various points throughout this book, such as in section 7.3.

One point to consider if you're thinking of using this technique is that the benefits of multilayered images on layer caching and download times can be lost. If your organization plans carefully around this, this technique can play a role in these images' real-world use.

TECHNIQUE 53 Managing foreign packages with Alien

Although most Dockerfile examples in this book (and on the internet) use a Debian-based image, the reality of software development means that many people won't be dealing with them exclusively.

Fortunately tools exist to help you with this.

PROBLEM

You want to install a package from a foreign distribution.

SOLUTION

Use a tool called Alien to convert the package. Alien is embedded into a Docker image we'll use as part of the technique.

Alien is a command-line utility designed to convert package files between the various formats listed in table 7.1. On more than one occasion, we've been required to make packages from foreign package management systems work, such as .deb files in CentOS, and .rpm files in non-Red Hat-based systems.

Table 7.1 Package formats supported by Alien

Extension	Description
.deb	Debian package
.rpm	Red Hat package management
.tgz	Slackware gzipped TAR file
.pkg	Solaris PKG package
.slp	Stampede package

NOTE For the purposes of this technique, Solaris and Stampede packages aren't fully covered. Solaris requires software peculiar to Solaris, and Stampede is an abandoned project.

Researching this book, we discovered that it could be a little fiddly to install Alien on non-Debian-based distributions. This being a Docker book, we've naturally decided to provide a conversion tool in the format of a Docker image. As a bonus, this tool uses the `ENTRYPOINT` command from technique 49 to make using the tools simpler.

As an example, let's download and convert (with Alien) the eatmydata package, which will be used in technique 62.

Retrieves the package files you want to convert

Creates an empty directory to work in

Runs the dockerinpractice/alienate image, mounting the current directory to the container's /io path. The container will examine that directory and try to convert any valid files it finds.

```
$ mkdir tmp && cd tmp
 $ wget \
http://mirrors.kernel.org/ubuntu/pool/main/libe/libeatmydata
➥ /eatmydata_26-2_i386.deb
 $ docker run -v $(pwd):/io dockerinpractice/alienate
 Examining eatmydata_26-2_i386.deb from /io
eatmydata_26-2_i386.deb appears to be a Debian package
 eatmydata-26-3.i386.rpm generated
eatmydata-26.slp generated
eatmydata-26.tgz generated
================================================
/io now contains:
eatmydata-26-3.i386.rpm
eatmydata-26.slp
eatmydata-26.tgz
eatmydata_26-2_i386.deb
================================================
$ ls -1
 eatmydata_26-2_i386.deb
eatmydata-26-3.i386.rpm
eatmydata-26.slp
eatmydata-26.tgz
```

The container informs you of its actions as it runs its Alien wrapper script.

The files have been converted to RPM, Slackware TGZ, and Stampede files.

Alternatively, you can pass the URL of a package to be downloaded and converted directly to the `docker run` command:

```
$ mkdir tmp && cd tmp
$ docker run -v $(pwd):/io dockerinpractice/alienate \
http://mirrors.kernel.org/ubuntu/pool/main/libe/libeatmydata
➥ /eatmydata_26-2_i386.deb
wgetting http://mirrors.kernel.org/ubuntu/pool/main/libe/libeatmydata
➥ /eatmydata_26-2_i386.deb
--2015-02-26 10:57:28--  http://mirrors.kernel.org/ubuntu/pool/main/libe
➥ /libeatmydata/eatmydata_26-2_i386.deb
Resolving mirrors.kernel.org (mirrors.kernel.org)... 198.145.20.143,
➥ 149.20.37.36, 2001:4f8:4:6f:0:1994:3:14, ...
Connecting to mirrors.kernel.org (mirrors.kernel.org)|198.145.20.143|:80...
➥ connected.
HTTP request sent, awaiting response... 200 OK
Length: 7782 (7.6K) [application/octet-stream]
Saving to: 'eatmydata_26-2_i386.deb'

     0K .......                               100% 2.58M=0.003s

2015-02-26 10:57:28 (2.58 MB/s) - 'eatmydata_26-2_i386.deb' saved
➥ [7782/7782]

Examining eatmydata_26-2_i386.deb from /io
eatmydata_26-2_i386.deb appears to be a Debian package
eatmydata-26-3.i386.rpm generated
eatmydata-26.slp generated
eatmydata-26.tgz generated
============================================================
/io now contains:
eatmydata-26-3.i386.rpm
eatmydata-26.slp
eatmydata-26.tgz
eatmydata_26-2_i386.deb
============================================================
$ ls -1
eatmydata_26-2_i386.deb
eatmydata-26-3.i386.rpm
eatmydata-26.slp
eatmydata-26.tgz
```

If you want to run Alien in a container yourself, you can start up the container with this:

```
docker run -ti --entrypoint /bin/bash dockerinpractice/alienate
```

WARNING Alien is a best-effort tool, and it's not guaranteed to work with the packages you give it.

DISCUSSION

Docker use has brought into sharp focus the "distro wars" that lay dormant for some time. Most organizations had settled into simply being Red Hat or Debian shops that didn't need to be concerned with other packaging systems. Now, it's not uncommon to receive requests for the introduction of Docker images that are based on "alien" distributions within an organization.

That's where this technique can help, as "foreign" packages can be converted to a more friendly format. This topic will be revisited in chapter 14, where we'll discuss security.

7.2 Traditional configuration management tools with Docker

Now we'll move on to how Dockerfiles can work alongside more traditional configuration management tools.

We'll look here at traditional configuration management with `make`, show you how you can use your existing Chef scripts to provision your images with Chef Solo, and look at a shell script framework built to help non-Docker experts build images.

TECHNIQUE 54 Traditional: Using make with Docker

At some point you might find that having a bunch of Dockerfiles is limiting your build process. For example, it's impossible to produce any output *files* if you limit yourself to running `docker build`, and there's no way to have variables in Dockerfiles.

This requirement for additional tooling can be addressed by a number of tools (including plain shell scripts). In this technique we'll look at how you can twist the venerable `make` tool to work with Docker.

PROBLEM

You want to add additional tasks around `docker build` execution.

SOLUTION

Use an ancient (in computing terms) tool called `make`.

In case you haven't used it before, `make` is a tool that takes one or more input files and produces an output file, but it can also be used as a task runner. Here's a simple example (note that all indents must be tabs):

Listing 7.4 A simple Makefile

By default, make assumes that all targets are filenames that will be created by the task. The .PHONY indicates for which task names this is not true.

By convention, the first target in a Makefile is "default". When run without an explicit target, make will choose the first in the file. You can see that "default" will execute "createfile" as its only dependency.

createfile is a phony task that depends on the x.y.z task.

catfile is a phony task that runs a single command.

x.y.z is a file task that runs two commands and creates the target x.y.z file.

```
.PHONY: default createfile catfile

default: createfile

createfile: x.y.z

catfile:
	cat x.y.z

x.y.z:
	echo "About to create the file x.y.z"
	echo abc > x.y.z
```

WARNING All indents in a Makefile must be tabs, and each command in a target is run in a different shell (so environment variables won't be carried across).

Once you have the preceding content in a file called Makefile, you can invoke any target with a command like `make createfile`.

Now let's look at some useful patterns in a Makefile—the rest of the targets we'll talk about will be phony, as it's difficult (although possible) to use file-change tracking to trigger Docker builds automatically. Dockerfiles use a cache of layers, so builds tend to be fast.

The first step is to run a Dockerfile. Because a Makefile consists of shell commands, this is easy.

Listing 7.5 Makefile for building an image

```
base:
    docker build -t corp/base .
```

Normal variations of this work as you'd expect (such as piping the file to `docker build` to remove the context, or using `-f` to use a differently named Dockerfile), and you can use the dependencies feature of `make` to automatically build base images (used in `FROM`) where necessary. For example, if you checked out a number of repositories into a subdirectory called repos (also easily doable with `make`), you could add a target, as in the following listing.

Listing 7.6 Makefile for building an image in a subdirectory

```
app1: base
    cd repos/app1 && docker build -t corp/app1 .
```

The downside of this is that every time your base image needs rebuilding, Docker will upload a build context that includes all of your repos. You can fix this by explicitly passing a build context TAR file to Docker.

Listing 7.7 Makefile for building an image with a specific set of files

```
base:
    tar -cvf - file1 file2 Dockerfile | docker build -t corp/base -
```

This explicit statement of dependencies will provide a significant speed increase if your directory contains a large number of files that are irrelevant to the build. You can slightly modify this target if you want to keep all your build dependencies in a different directory.

Listing 7.8 Makefile for building an image with a specific set of files with renamed paths

```
base:
    tar --transform 's/^deps\///' -cf - deps/* Dockerfile | \
        docker build -t corp/base -
```

Here you add everything in the deps directory to the build context, and use the `--transform` option of `tar` (available in recent `tar` versions on Linux) to strip any leading "deps/" from filenames. In this particular case, a better approach would have been to put the deps and Dockerfile in a directory of their own to permit a normal `docker build`, but it's useful to be aware of this advanced use as it can come in handy in the most unlikely places. Always think carefully before using it, though, as it adds complexity to your build process.

Simple variable substitution is a relatively simple matter, but (as with `--transform` previously) think carefully before you use it—Dockerfiles deliberately don't support variables in order to keep builds easily reproducible.

Here we're going to use variables passed to `make` and substitute using `sed`, but you can pass and substitute however you like.

Listing 7.9 Makefile for building an image with basic Dockerfile variable substitution

```
VAR1 ?= defaultvalue
base:
    cp Dockerfile.in Dockerfile
    sed -i 's/{VAR1}/$(VAR1)/' Dockerfile
    docker build -t corp/base .
```

The Dockerfile will be regenerated every time the base target is run, and you can add more variable substitutions by adding more `sed -i` lines. To override the default value of `VAR1`, you run `make VAR1=newvalue base`. If your variables include slashes, you may need to choose a different `sed` separator, like `sed -i 's#{VAR1}#$(VAR1)#' Dockerfile`.

Finally, if you've been using Docker as a build tool, you need to know how to get files back out of Docker. We'll present a couple of different possibilities, depending on your use case.

Listing 7.10 Makefile for copying files out of an image

```
singlefile: base
    docker run --rm corp/base cat /path/to/myfile > outfile
multifile: base
    docker run --rm -v $(pwd)/outdir:/out corp/base sh \
        -c "cp -r /path/to/dir/* /out/"
```

Here, `singlefile` runs `cat` on a file and pipes the output to a new file. This approach has the advantage of automatically setting the correct owner of the file, but it becomes cumbersome for more than one file. The `multifile` approach mounts a volume in the container and copies all files from a directory to the volume. You can follow this up with a `chown` command to set the correct owner on the files, but bear in mind that you'll probably need to invoke it with `sudo`.

The Docker project itself uses the volume-mounting approach when building Docker from source.

DISCUSSION

It might seem a little odd for a tool as old as `make` to appear within a book about a relatively new technology like Docker. Why not use a newer build technology like Ant, or Maven, or any of the many other general build tools available.

The answer is that, for all its faults, `make` is a tool that is

- Unlikely to go away any time soon
- Well-documented
- Highly flexible
- Widely available

Having spent many hours fighting bugs or poorly documented (or undocumented) limitations of new build technologies, or trying to install dependencies of these systems, `make`'s features have saved us many times. It's also more likely that `make` will be available in five years' time when other tools are gone, or have fallen out of maintenance by their owners.

TECHNIQUE 55 Building images with Chef Solo

One of the things that confuses newcomers to Docker is whether Dockerfiles are the only supported configuration management tool, and whether existing configuration management tools should be ported to Dockerfiles. Neither of these is true.

Although Dockerfiles are designed to be a simple and portable means of provisioning images, they're also flexible enough to allow any other configuration management tool to take over. In short, if you can run it in a terminal, you can run it in a Dockerfile.

As a demonstration of this, we'll show you how to get up and running with Chef, arguably the most established configuration management tool, in a Dockerfile. Using a tool like Chef can reduce the amount of work required for you to configure images.

> **NOTE** Although familiarity with Chef isn't required to follow this technique, some familiarity will be required to follow it the first time with ease. Covering a whole configuration management tool is a book in itself. With careful study and some research, this technique can be used to get a good understanding of Chef basics.

PROBLEM

You want to reduce configuration effort by using Chef.

SOLUTION

Install Chef in your container, and run recipes using Chef Solo within that container to provision it, all within your Dockerfile.

What you're going to provision is a simple Hello World Apache website. This will give you a taste of what Chef can do for your configuration.

Chef Solo requires no external Chef server setup. If you're already familiar with Chef, this example can easily be adapted to enable your pre-existing scripts to contact your Chef server if you wish.

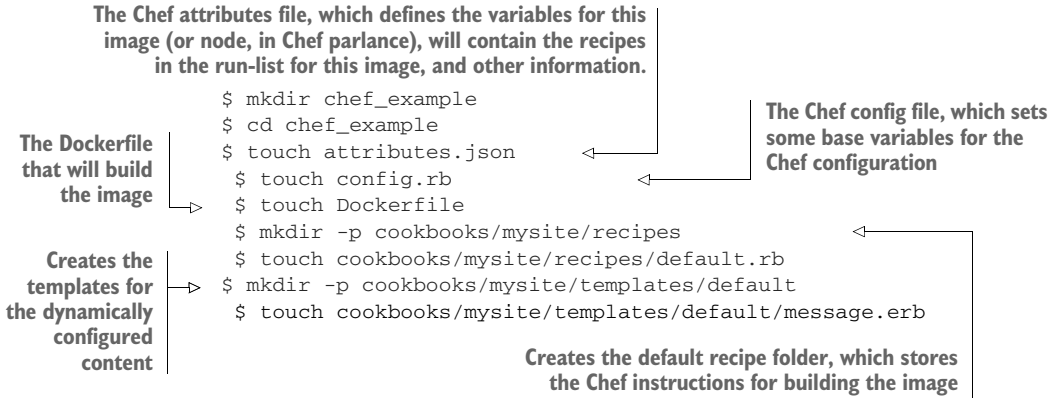
We're going to walk through the creation of this Chef example, but if you want to download the working code, it's available as a Git repository. To download it, run this command:

```
git clone https://github.com/docker-in-practice/docker-chef-solo-example.git
```

We'll begin with the simple aim of setting up a web server with Apache that outputs "Hello World!" (what else?) when you hit it. The site will be served from mysite.com, and a `mysiteuser` user will be set up on the image.

To begin, create a directory and set it up with the files you'll need for Chef configuration.

Listing 7.11 Creating the necessary files for a Chef configuration



First we'll fill out attributes.json.

Listing 7.12 attributes.json

```
{
        "run_list": [
                "recipe[apache2::default]",
                "recipe[mysite::default]"
        ]
}
```

This file sets out the recipes you're going to run. The apache2 recipes will be retrieved from a public repository; the mysite recipes will be written here.

Next, populate your config.rb with some basic information, as shown in the next listing.

Listing 7.13 config.rb

```
base_dir        "/chef/"
file_cache_path base_dir + "cache/"
```

```
cookbook_path   base_dir + "cookbooks/"
verify_api_cert true
```

This file sets up basic information about the location and adds the configuration setting `verify_api_cert` to suppress an irrelevant error.

Now we get to the meat of the work: the image's Chef recipe. Each stanza terminated by an `end` in the code block defines a Chef resource.

Listing 7.14 cookbooks/mysite/recipes/default.rb

```
user "mysiteuser" do
     comment "mysite user"
    home "/home/mysiteuser"
    shell "/bin/bash"
end

directory "/var/www/html/mysite" do
     owner "mysiteuser"
    group "mysiteuser"
    mode 0755
    action :create
end

template "/var/www/html/mysite/index.html" do
     source "message.erb"
    variables(
        :message => "Hello World!"
    )
    user "mysiteuser"
    group "mysiteuser"
    mode 0755
end

web_app "mysite" do
     server_name "mysite.com"
    server_aliases ["www.mysite.com","mysite.com"]
     docroot "/var/www/html/mysite"
    cookbook 'apache2'
end
```

Creates a user

Creates a directory for the web content

Defines a file that will be placed in the web folder. This file will be created from a template defined in the "source" attribute.

Defines a web app for apache2

In a real scenario you'd have to change references from mysite to your website's name. If you're accessing or testing from your host, this doesn't matter.

The content of the website is contained within the template file. It contains one line, which Chef will read, substituting in the "Hello World!" message from config.rb. Chef will then write the substituted file out to the template target (/var/www/html/mysite/index.html). This uses a templating language that we're not going to cover here.

Listing 7.15 cookbooks/mysite/templates/default/message.erb

```
<%= @message %>
```

Finally, you put everything together with the Dockerfile, which sets up the Chef prerequisites and runs Chef to configure the image, as shown in the following listing.

Listing 7.16 Dockerfile

```
FROM ubuntu:14.04

RUN apt-get update && apt-get install -y git curl

RUN curl -L \
https://opscode-omnibus-packages.s3.amazonaws.com/ubuntu/12.04/x86_64
➥ /chefdk_0.3.5-1_amd64.deb \
-o chef.deb
RUN dpkg -i chef.deb && rm chef.deb

COPY . /chef

WORKDIR /chef/cookbooks
 RUN knife cookbook site download apache2
 RUN knife cookbook site download iptables
 RUN knife cookbook site download logrotate

RUN /bin/bash -c 'for f in $(ls *gz); do tar -zxf $f; rm $f; done'

RUN chef-solo -c /chef/config.rb -j /chef/attributes.json

CMD /usr/sbin/service apache2 start && sleep infinity
```

Downloads and installs Chef. If this download doesn't work for you, check the latest code in docker-chef-solo-example mentioned earlier in this discussion, as a later version of Chef may now be required.

Copies the contents of the working folder into the /chef folder on the image

Moves to the cookbooks folder and downloads the apache2 cookbook and its dependencies as tarballs using Chef's knife utility

Extracts the downloaded tarballs and removes them

Runs the chef command to configure your image. Supplies it with the attributes and config files you already created.

Defines the default command for the image. The sleep infinity command ensures that the container doesn't exit as soon as the service command has finished its work.

You're now ready to build and run the \image:

```
docker build -t chef-example .
docker run -ti -p 8080:80 chef-example
```

If you now navigate to http://localhost:8080, you should see your "Hello World!" message.

> **WARNING** If your Chef build takes a long time and you're using the Docker Hub workflow, the build can time out. If this happens, you can perform the build on a machine you control, pay for a supported service, or break the build steps into smaller chunks so that each individual step in the Dockerfile takes less time to return.

Although this is a trivial example, the benefits of using this approach should be clear. With relatively straightforward configuration files, the details of getting the image into

a desired state are taken care of by the configuration management tool. This doesn't mean that you can forget about the details of configuration; changing the values will require you to understand the semantics to ensure you don't break anything. But this approach can save you much time and effort, particularly in projects where you don't need to get into the details too much.

DISCUSSION

The purpose of this technique is to correct a common confusion about the Dockerfile concept, specifically that it's a competitor to other configuration management tools like Chef and Ansible.

What Docker really is (as we say elsewhere in the book) is a *packaging* tool. It allows you to present the results of a build process in a predictable and packaged way. How you choose to build it is up to you. You can use Chef, Puppet, Ansible, Makefiles, shell scripts, or sculpt them by hand.

The reason most people don't use Chef, Puppet, and the like to build images is primarily because Docker images tend to be built as single-purpose and single-process tools. But if you already have configuration scripts to hand, why not re-use them?

7.3 Small is beautiful

If you're creating lots of images and sending them hither and thither, the issue of image size will be more likely to arise. Although Docker's use of image layering can help with this, you may have such a panoply of images on your estate that this isn't practical to manage.

In these cases, it can be helpful to have some best practices in your organization related to reducing images to as small a size as possible. In this section we'll show you some of these, and even how a standard utility image can be reduced by an order of magnitude—a much smaller object to fling around your network.

TECHNIQUE 56 Tricks for making an image smaller

Let's say you've been given an image by a third party, and you want to make the image smaller. The simplest approach is to start with an image that works and remove the unnecessary files.

Classic configuration management tools tend not to remove things unless explicitly instructed to do so—instead, they start from a non-working state and add new configurations and files. This leads to *snowflake* systems crafted for a particular purpose, which may look very different from what you'd get if you ran your configuration management tool against a fresh server, especially if the configuration has evolved over time. Courtesy of layering and lightweight images in Docker, you can perform the reverse of this process and experiment with removing things.

PROBLEM

You want to make your images smaller.

SOLUTION

Follow these steps to reduce the size of an image by removing unnecessary packages and doc files:

1. Run the image.
2. Enter the container.
3. Remove unnecessary files.
4. Commit the container as a new image (see technique 15).
5. Flatten the image (see technique 52).

The last two steps have been covered earlier in the book, so we're only going to cover the first three here.

To illustrate how to do this, we're going to take the image created in technique 49 and try to make that image smaller.

First, run up the image as a container:

```
docker run -ti --name smaller --entrypoint /bin/bash \
dockerinpractice/log-cleaner
```

Because this is a Debian-based image, you can start by seeing which packages you might not need and removing them. Run `dpkg -l | awk '{print $2}'` and you'll get a list of installed packages on the system.

You can then go through those packages running `apt-get purge -y package_name` on them. If there's a scary message warning you that "You are about to do something potentially harmful," press Return to continue.

Once you've removed all the packages that can safely be removed, you can run these commands to clean up the apt cache:

```
apt-get autoremove
apt-get clean
```

This is a relatively safe way to reduce space in your images.

Further significant savings can be made by removing docs. For example, running `rm -rf /usr/share/doc/* /usr/share/man/* /usr/share/info/*` will often remove sizable files you'll probably never need. You can take this to the next level by manually running `rm` on binaries and libraries you don't need.

Another area for rich pickings is the /var folder, which should contain temporary data, or data not essential to the running of programs.

This command will get rid of all files with the .log suffix:

```
find /var | grep '\.log$' | xargs rm -v
```

Now you'll have a much smaller image than you previously had, ready to commit.

DISCUSSION

Using this somewhat manual process, you can get the original dockerinpractice/log-cleaner image down to a few dozen MB quite easily, and even make it smaller if you have the motivation. Remember that due to Docker's layering, you'll need to export and import the image as explained in technique 52; otherwise the image's overall size will include the deleted files.

Technique 59 will show you a much more effective (but risky) way to significantly reduce the size of your images.

TIP An example of the commands described here is maintained at https://github.com/docker-in-practice/log-cleaner-purged, and it can be pulled with Docker from dockerinpractice/log-cleaner-purged.

TECHNIQUE 57 Tiny Docker images with BusyBox and Alpine

Small, usable OSs that can be embedded onto a low-power or cheap computer have existed since Linux began. Fortunately, the efforts of these projects have been repurposed to produce small Docker images for use where image size is important.

PROBLEM

You want a small, functional image.

SOLUTION

Use a small base image, like BusyBox or Alpine, when building your own images.

This is another area where the state of the art is fast changing. The two popular choices for minimal Linux base images are BusyBox and Alpine, and each has different characteristics.

If lean but useful is your aim, BusyBox may fit the bill. If you start up a BusyBox image with the following command, something surprising happens:

```
$ docker run -ti busybox /bin/bash
exec: "/bin/bash": stat /bin/bash: no such file or directory2015/02/23 >
09:55:38 Error response from daemon: Cannot start container >
73f45e34145647cd1996ae29d8028e7b06d514d0d32dec9a68ce9428446faa19: exec: >
"/bin/bash": stat /bin/bash: no such file or directory
```

BusyBox is so lean it has no bash! Instead it uses ash, which is a posix-compliant shell—effectively a limited version of more advanced shells such as bash and ksh.

```
$ docker run -ti busybox /bin/ash
/ #
```

As the result of many decisions like this, the BusyBox image weighs in at under 2.5 MB.

WARNING BusyBox can contain some other nasty surprises. The `tar` version, for example, will have difficulty untarring TAR files created with GNU `tar`.

This is great if you want to write a small script that only requires simple tools, but if you want to run anything else you'll have to install it yourself. BusyBox comes with no package management.

Other maintainers have added package management functionality to BusyBox. For example, progrium/busybox might not be the smallest BusyBox container (it's currently a little under 5 MB), but it has opkg, which means you can easily install other common packages while keeping the image size to an absolute minimum. If you're missing bash, for example, you can install it like this:

```
$ docker run -ti progrium/busybox /bin/ash
/ # opkg-install bash > /dev/null
/ # bash
bash-4.3#
```

When committed, this results in a 6 MB image.

Another interesting Docker image (which has become a Docker standard for small images) is gliderlabs/alpine. It's similar to BusyBox but has a more extensive range of packages that you can browse at https://pkgs.alpinelinux.org/packages.

The packages are designed to be lean on install. To take a concrete example, here's a Dockerfile that results in an image that's just over a quarter of a gigabyte.

Listing 7.17 Ubuntu plus mysql-client

```
FROM ubuntu:14.04
RUN apt-get update -q \
&& DEBIAN_FRONTEND=noninteractive apt-get install -qy mysql-client \
&& apt-get clean && rm -rf /var/lib/apt
ENTRYPOINT ["mysql"]
```

> **TIP** The `DEBIAN_FRONTEND=noninteractive` before the `apt-get install` ensures that the install doesn't prompt for any input during the install. As you can't easily engineer responses to questions when running commands, this is often useful in Dockerfiles.

By contrast, the following listing results in an image that's a little over 36 MB.

Listing 7.18 Alpine plus `mysql-client`

```
FROM gliderlabs/alpine:3.6
RUN apk-install mysql-client
ENTRYPOINT ["mysql"]
```

DISCUSSION

This is an area in which there's been much development over the past couple of years. The Alpine base image has edged out BusyBox to become something of a Docker standard, with the help of some backing from Docker Inc. itself.

In addition, the other more "standard" base images have been on diets themselves. The Debian image stands at roughly 100 MB as we prepare the second edition of this book—much less than it originally was.

One point worth referring to here is that there's a lot of discussion around reducing the size of images, or using smaller base images, when this is not something that needs to be addressed. Remember that it's often best to spend time and effort overcoming existing bottlenecks than achieving theoretical benefits that may turn out to give little bang for your buck.

TECHNIQUE 58 The Go model of minimal containers

Although it can be illuminating to winnow down your working containers by removing redundant files, there's another option—compiling minimal binaries without dependencies.

Doing this radically simplifies the task of configuration management—if there's only one file to deploy and no packages are required, a significant amount of configuration management tooling becomes redundant.

PROBLEM

You want to build binary Docker images with no external dependencies.

SOLUTION

Build a statically linked binary—one that will not try to load any system libraries when it starts running.

To demonstrate how this can be useful, we'll first create a small Hello World image with a small C program. Then we'll go on to show you how to do something equivalent for a more useful application.

A MINIMAL HELLO WORLD BINARY

To create the minimal Hello World binary, first create a new directory and a Dockerfile, as shown in the following listing.

Listing 7.19 Hello Dockerfile

```
FROM gcc
 RUN echo 'int main() { puts("Hello world!"); }' > hi.c
 RUN gcc -static hi.c -w -o hi
```

The gcc image is an image designed for compiling.

Creates a simple one-line C program

Compiles the program with the -static flag, and suppresses warnings with -w

The preceding Dockerfile compiles a simple Hello World program without dependencies. You can now build it and extract that binary from the container, as shown in the next listing.

Listing 7.20 Extracting the binary from the image

```
$ docker build -t hello_build .
 $ docker run --name hello hello_build /bin/true
 $ docker cp hello:/hi hi
 $ docker rm hello
 hello
$ docker rmi hello_build
 Deleted: 6afcbf3a650d9d3a67c8d67c05a383e7602baecc9986854ef3e5b9c0069ae9f2
$ mkdir -p new_folder
```

Copies the "hi" binary using the docker cp command

Builds the image containing the statically linked "hi" binary

Runs the image with a trivial command in order to copy out the binary

Cleanup: you don't need these anymore

Makes a new folder called "new_folder"

```
$ mv hi new_folder
$ cd new_folder
```

Moves the "hi" binary into this folder

Changes directory into this new folder

You now have a statically built binary in a fresh directory and have moved into it.

Now create another Dockerfile, as shown in the next listing.

Listing 7.21 Minimal Hello Dockerfile

```
FROM scratch
 ADD hi /hi
 CMD ["/hi"]
```

Uses the zero-byte scratch image

Adds the "hi" binary to the image

Defaults the image to run the "hi" binary

Build and run it as shown in the following listing.

Listing 7.22 Creating the minimal container

```
$ docker build -t hello_world .
Sending build context to Docker daemon 931.3 kB
Sending build context to Docker daemon
Step 0 : FROM scratch
 --->
Step 1 : ADD hi /hi
 ---> 2fe834f724f8
Removing intermediate container 01f73ea277fb
Step 2 : ENTRYPOINT /hi
 ---> Running in 045e32673c7f
 ---> 5f8802ae5443
Removing intermediate container 045e32673c7f
Successfully built 5f8802ae5443
$ docker run hello_world
Hello world!
$ docker images | grep hello_world
hello_world      latest    5f8802ae5443   24 seconds ago  928.3 kB
```

The image builds, runs, and weighs in at under 1 MB.

A MINIMAL GO WEB SERVER IMAGE

That was a relatively trivial example, but you can apply the same principle to programs built in Go. An interesting feature of the Go language is that it's relatively easy to build such static binaries.

To demonstrate this ability, we created a simple web server in Go whose code is available at https://github.com/docker-in-practice/go-web-server.

The Dockerfile for building this simple web server is shown in the following listing.

Listing 7.23 Dockerfile to statically compile a Go web server

```
FROM golang:1.4.2
 RUN CGO_ENABLED=0 go get \
 -a -ldflags '-s' -installsuffix cgo \
 github.com/docker-in-practice/go-web-server
 CMD ["cat","/go/bin/go-web-server"]
```

This build is known to work against this version number of the golang image; if the build fails, it may be that this version is no longer available.

"go get" retrieves the source code from the URL provided and compiles it locally. The CGO_ENABLED environment variable is set to 0 to prevent cross-compilation.

Sets a number of miscellaneous flags to the Go compiler to ensure static compilation and reduce size

The Go web server source code repository

Defaults the resulting image to output the executable

If you save this Dockerfile into an empty directory and build it, you'll now have an image containing the program. Because you specified the default command of the image to output the executable content, you now just need to run the image and send the output to a file on your host, as the following listing shows.

Listing 7.24 Getting the Go web server from the image

```
$ docker build -t go-web-server .
 $ mkdir -p go-web-server && cd go-web-server
 $ docker run go-web-server > go-web-server
 $ chmod +x go-web-server
 $ echo Hi > page.html
```

Builds and tags the image

Makes and moves into a fresh directory to deposit the binary

Runs the image and redirects the binary output to a file

Makes the binary executable

Creates a web page for the server to serve

Now, as with the "hi" binary, you have a binary with no library dependencies or need to access the filesystem. We're therefore going to create a Dockerfile from the zero-byte scratch image and add the binary to it, as before.

Listing 7.25 Go web server Dockerfile

```
FROM scratch
ADD go-web-server /go-web-server
 ADD page.html /page.html
 ENTRYPOINT ["/go-web-server"]
```

Adds the static binary to the image

Adds a page to serve from the web server

Makes the binary the default program run by the image

Now build it and run the image. The resulting image is a little over 4 MB in size.

Listing 7.26 Building and running the Go web server image

```
$ docker build -t go-web-server .
$ docker images | grep go-web-server
go-web-server    latest    de1187ee87f3  3 seconds ago   4.156 MB
$ docker run -p 8080:8080 go-web-server -port 8080
```

You can access it on http://localhost:8080. If the port is already in use, you can replace the 8080s in the preceding code with a port of your choice.

DISCUSSION

If you can bundle applications into one binary, why bother with Docker at all? You can move the binary around, run multiple copies, and so on.

You can do so if you want, but you'd lose the following:

- All the container management tools in the Docker ecosystem
- The metadata within the Docker images that document significant application information, such as ports, volumes, and labels
- The isolation that gives Docker its operational power

As a concrete example, etcd is a static binary by default, but when we examine it in technique 74 we'll demonstrate it inside a container to make it easier to see how the same process would work across multiple machines and ease deployment.

TECHNIQUE 59 Using inotifywait to slim containers

We're now going to take slimming our containers to the next level by using a nifty tool that tells us what files are being referenced when we run a container.

This could be called the nuclear option, as it can be quite risky to implement on production. But it can be an instructive means of learning about your system, even if you don't follow through with using it for real—a crucial part of configuration management is understanding what your application requires to operate correctly.

PROBLEM

You want to reduce your container to the smallest possible set of files and permissions.

SOLUTION

Monitor which files are accessed by your program with inotify, and then remove any that seem unused.

At a high level, you need to know what files are being accessed when you run a command in a container. If you remove all the other files on the container filesystem, you'll theoretically still have everything you need.

In this walkthrough, we're going to use the log-cleaner-purged image from technique 56. You'll install inotify-tools, and then run `inotifywait` to get a report on which files were accessed. You'll then run a simulation of the image's entrypoint (the log_clean script). Then, using the file report generated, you'll remove any file that hasn't been accessed.

Listing 7.27 Performing manual install steps while monitoring with inotifywait

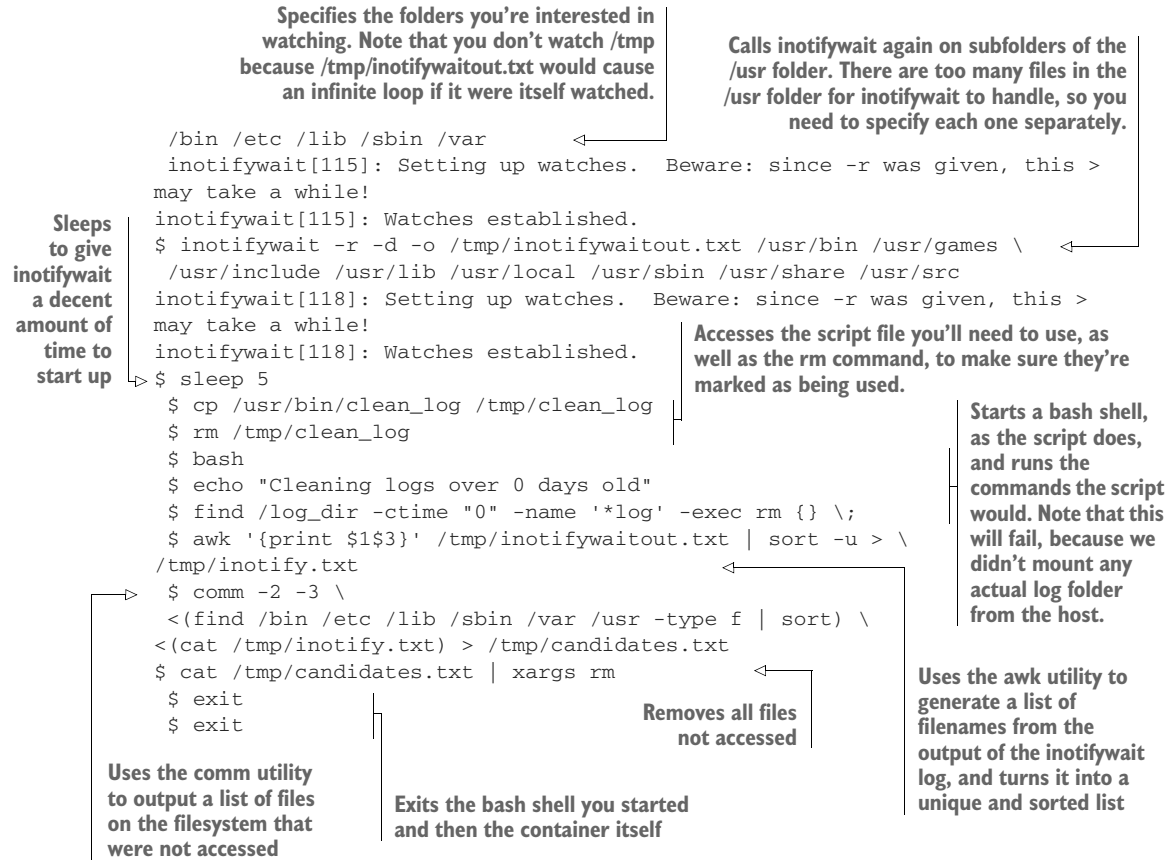
```
[host]$ docker run -ti --entrypoint /bin/bash \
 --name reduce dockerinpractice/log-cleaner-purged
 $ apt-get update && apt-get install -y inotify-tools
 $ inotifywait -r -d -o /tmp/inotifywaitout.txt \
```

Overrides the default entrypoint for this image

Gives the container a name you can refer to later

Runs inotifywait in recursive (-r) and daemon (-d) modes to get a list of accessed files in the outfile (specified with the -o flag)

Installs the inotify-tools package

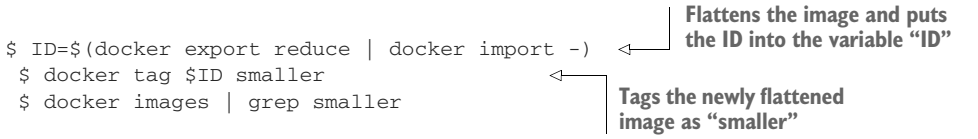


At this point you've

- Placed a watch on files to see what files are being accessed
- Run all the commands to simulate the running of the script
- Run commands to ensure you access the script you'll definitely need, and the rm utility
- Gained a list of all files not accessed during the run
- Removed all the non-accessed files

Now you can flatten this container (see technique 52) to create a new image and test that it still works.

Listing 7.28 Flattening the image and running it



```
smaller  latest  2af3bde3836a  18 minutes ago  6.378 MB
 $ mkdir -p /tmp/tmp
 $ touch /tmp/tmp/a.log
 $ docker run -v /tmp/tmp:/log_dir smaller \
/usr/bin/clean_log 0
Cleaning logs over 0 days old
$ ls /tmp/tmp/a.log
 ls: cannot access /tmp/tmp/a.log: No such file or directory
```

The image is now less than 10% of its previous size.

Creates a new folder and file to simulate a log directory for testing

Runs the newly created image over the test directory and checks that the file created has been removed

We reduced the size of this image from 96 MB to around 6.5 MB, and it still appears to work. Quite a saving!

> **WARNING** This technique, like overclocking your CPU, is not an optimization for the unwary. This particular example works well because it's an application that's quite limited in scope, but your mission-critical business application is likely to be more complex and dynamic in how it accesses files. You could easily remove a file that wasn't accessed on your run, but that is needed at some other point.

If you're a little nervous of potentially breaking your image by removing files you'll need later, you can use the /tmp/candidates.txt file to get a list of the biggest files that were untouched, like this:

```
cat /tmp/candidates.txt | xargs wc -c | sort -n | tail
```

You can then remove the larger files that you're sure won't be needed by your application. There can be big wins here too.

DISCUSSION

Although this technique has been presented as a Docker technique, it falls into the category of "generally useful" techniques that can be applied in other contexts. It's particularly useful when debugging processes where you don't know quite what's happening, and you want to see which files are being referenced. `strace` is another option for doing this, but `inotifywait` is in some ways an easier tool to use for this purpose.

This general approach is also used as one avenue of attack in technique 97 in the context of reducing the attack surface of a container.

TECHNIQUE 60 Big can be beautiful

Although this section is about keeping images small, it's worth remembering that small is not necessarily better. As we'll discuss, a relatively large monolithic image can be more efficient than a small one.

PROBLEM

You want to reduce disk space use and network bandwidth due to Docker images.

SOLUTION

Create a universal, large, monolithic base image for your organization.

It's paradoxical, but a large monolithic image could save you disk space and network bandwidth.

Recall that Docker uses a copy-on-write mechanism when its containers are running. This means that you could have hundreds of Ubuntu containers running, but only a small amount of additional disk space is used for each container started.

If you have lots of different, smaller images on your Docker server, as in figure 7.1, more disk space may be used than if you have one larger monolithic image with everything you need in it.

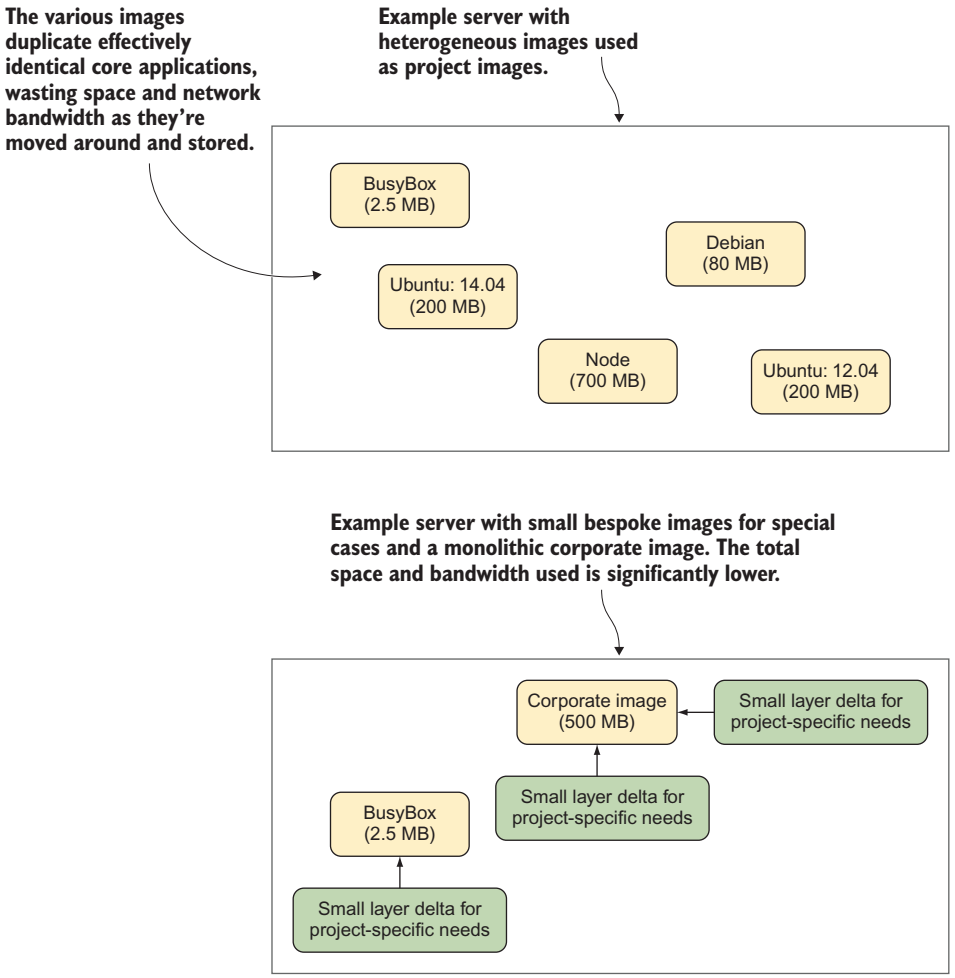


Figure 7.1 Many small base images vs. fewer large base images

You may be reminded of the principle of a shared library. A shared library can be loaded by multiple applications at once, reducing the amount of disk and memory needed to run the required programs. In the same way, a shared base image for your organization can save space, as it only needs to be downloaded once and should contain everything you need. Programs and libraries previously required in multiple images are now only required once.

In addition, there can be other benefits of sharing a monolithic, centrally managed image across teams. The maintenance of this image can be centralized, improvements can be shared, and issues with the build need only be solved once.

If you're going to adopt this technique, here are some things to watch out for:

- The base image should be reliable first. If it doesn't behave consistently, the users will avoid using it.
- Changes to the base image must be tracked somewhere that's visible so that users can debug problems themselves.
- Regression tests are essential to reduce confusion when updating the vanilla image.
- Be careful about what you add to the base—once it's in the base image, it's hard to remove, and the image can bloat fast.

DISCUSSION

We used this technique to great effect in our 600-strong development company. A monthly build of core applications was bundled into a large image and published on the internal Docker registry. Teams would build on the so-called "vanilla" corporate image by default, and create bespoke layers if necessary on top of that.

It's worth taking a look at technique 12 for some additional details on monolithic containers—particularly for the mention of the phusion/base image Docker image, an image designed with running multiple processes in mind.

Summary

- `ENTRYPOINT` is another way to start Docker containers that allows runtime parameters to be configured.
- Flattening images can be used to prevent the leakage of secrets from your build via your image's layers.
- Packages foreign to your chosen base image's distribution can be integrated using Alien.
- Traditional build tools such as `make`, as well as modern ones like Chef, still have their place in the Docker world.
- Docker images can be reduced using smaller base images, using languages appropriate to the task, or removing unnecessary files.
- It's worth considering whether the size of your image is the most important challenge to tackle.