4 Building images

This chapter covers

- Some basics of image creation
- Manipulating the Docker build cache for fast and reliable builds
- Configuring timezones as part of an image build
- Running commands directly on your containers from the host
- Drilling down into the layers created by an image build
- Using the more advanced ONBUILD feature when building and using images

To get beyond the basics of using Docker, you'll want to start creating your own building blocks (images) to pull together in interesting ways. This chapter will cover some of the important parts of image creation, looking at practicalities that you might otherwise stumble over.

4.1 Building images

Although the simplicity of Dockerfiles makes them a powerful time-saving tool, there are some subtleties that can cause confusion. We'll take you over a few time-saving features and their details, starting with the ADD instruction. Then we'll cover the Docker build cache, how it can let you down, and how you can manipulate it to your advantage.

Remember to refer to the official Docker documentation for complete Dockerfile instructions at https://docs.docker.com.

TECHNIQUE 20 Injecting files into your image using ADD

Although it's possible to add files within a Dockerfile using the RUN command and basic shell primitives, this can quickly become unmanageable. The ADD command was added to the list of Dockerfile commands to address the need to put large numbers of files into an image without fuss.

PROBLEM

You want to download and unpack a tarball into your image in a concise way.

SOLUTION

Tar and compress your files, and use the ADD directive in your Dockerfile.

Create a fresh environment for this Docker build with mkdir add_example && cd add_example. Then retrieve a tarball and give it a name you can reference later.

Listing 4.1 Downloading a TAR file

```
$ curl \
https://www.flamingspork.com/projects/libeatmydata/
➥ libeatmydata-105.tar.gz > my.tar.gz
```

In this case we've used a TAR file from another technique, but it could be any tarball you like.

Listing 4.2 Adding a TAR file to an image

```
FROM debian
RUN mkdir -p /opt/libeatmydata
ADD my.tar.gz /opt/libeatmydata/
RUN ls -lRt /opt/libeatmydata
```

Build this Dockerfile with docker build --no-cache . and the output should look like this:

Listing 4.3 Building an image with a TAR file

```
$ docker build --no-cache .
Sending build context to Docker daemon 422.9 kB
Sending build context to Docker daemon
Step 0 : FROM debian
 ---> c90d655b99b2
Step 1 : RUN mkdir -p /opt/libeatmydata
 ---> Running in fe04bac7df74
 ---> c0ab8c88bb46
Removing intermediate container fe04bac7df74
```

```
Step 2 : ADD my.tar.gz /opt/libeatmydata/
 ---> 06dcd7a88eb7
Removing intermediate container 3f093a1f9e33
Step 3 : RUN ls -lRt /opt/libeatmydata
 ---> Running in e3283848ad65
/opt/libeatmydata:
total 4
drwxr-xr-x 7 1000 1000 4096 Oct 29 23:02 libeatmydata-105

/opt/libeatmydata/libeatmydata-105:
total 880
drwxr-xr-x 2 1000 1000   4096 Oct 29 23:02 config
drwxr-xr-x 3 1000 1000   4096 Oct 29 23:02 debian
drwxr-xr-x 2 1000 1000   4096 Oct 29 23:02 docs
drwxr-xr-x 3 1000 1000   4096 Oct 29 23:02 libeatmydata
drwxr-xr-x 2 1000 1000   4096 Oct 29 23:02 m4
-rw-r--r-- 1 1000 1000   9803 Oct 29 23:01 config.h.in
[...edited...]
-rw-r--r-- 1 1000 1000   1824 Jun 18  2012 pandora_have_better_malloc.m4
-rw-r--r-- 1 1000 1000    742 Jun 18  2012 pandora_header_assert.m4
-rw-r--r-- 1 1000 1000    431 Jun 18  2012 pandora_version.m4
 ---> 2ee9b4c8059f
Removing intermediate container e3283848ad65
Successfully built 2ee9b4c8059f
```

You can see from this output that the tarball has been unpacked into the target directory by the Docker daemon (the extended output of all the files has been edited). Docker will unpack tarfiles of most standard types (.gz, .bz2, .xz, .tar).

It's worth observing that although you can download tarballs from URLs, they'll only be unpacked automatically if they're stored in the local filesystem. This can lead to confusion.

If you repeat the preceding process with the following Dockerfile, you'll notice that the file is downloaded but not unpacked.

Listing 4.4 Directly adding of the TAR file from the URL

```
FROM debian
RUN mkdir -p /opt/libeatmydata
ADD \
 https://www.flamingspork.com/projects/libeatmydata/libeatmydata-105.tar.gz \
/opt/libeatmydata/
 RUN ls -lRt /opt/libeatmydata
```

The file is retrieved from the internet using a URL.

The destination directory is indicated by the directory name and a trailing slash. Without the trailing slash, the argument is treated as a filename for the downloaded file.

Here's the resulting build output:

```
Sending build context to Docker daemon 422.9 kB
Sending build context to Docker daemon
Step 0 : FROM debian
 ---> c90d655b99b2
```

```
Step 1 : RUN mkdir -p /opt/libeatmydata
 ---> Running in 6ac454c52962
 ---> bdd948e413c1
Removing intermediate container 6ac454c52962
Step 2 : ADD \
https://www.flamingspork.com/projects/libeatmydata/libeatmydata-105.tar.gz
➥ /opt/libeatmydata/
Downloading [=================================================>] \
419.4 kB/419.4 kB
 ---> 9d8758e90b64
Removing intermediate container 02545663f13f
Step 3 : RUN ls -lRt /opt/libeatmydata
 ---> Running in a947eaa04b8e
/opt/libeatmydata:
total 412
-rw------- 1 root root 419427 Jan  1  1970 \
libeatmydata-105.tar.gz
  ---> f18886c2418a
Removing intermediate container a947eaa04b8e
Successfully built f18886c2418a
```

The libeatmydata-105.tar.gz file has been downloaded and placed in the /opt/libeatmydata directory without being unpacked.

Note that without the trailing slash in the ADD line in the previous Dockerfile, the file would be downloaded and saved with that filename. The trailing slash indicates that the file should be downloaded and placed in the directory specified.

All new files and directories are owned by root (or whoever has group or user IDs of 0 within the container).

Whitespace in filenames

If your filenames have whitespace in them, you'll need to use the quoted form of ADD (or COPY):

```
ADD "space file.txt" "/tmp/space file.txt"
```

DISCUSSION

The ADD Dockerfile instruction is quite a workhorse, with a number of different pieces of functionality you can take advantage of. If you're going to write more than a couple of Dockerfiles (which you likely will as you go through this book), it's worth reading the official Dockerfile instructions documentation—there aren't many (18 instructions are listed in the documentation at the time of writing at https://docs.docker.com/engine/reference/builder) and you'll only use a few of them regularly.

People often ask about adding compressed files without extracting them. For this you should use the COPY command, which looks exactly like the ADD command but doesn't unpack any files and won't download over the internet.

TECHNIQUE 21 Rebuilding without the cache

Building with Dockerfiles takes advantage of a useful caching feature: steps that have already been built are only rebuilt if the commands have changed. The next listing shows the output of a rebuild of the to-do app from chapter 1.

Listing 4.5 Rebuilding with the cache

```
$ docker build .
Sending build context to Docker daemon  2.56 kB
Sending build context to Docker daemon
Step 0 : FROM node
 ---> 91cbcf796c2c
Step 1 : MAINTAINER ian.miell@gmail.com
 ---> Using cache
  ---> 8f5a8a3d9240
 Step 2 : RUN git clone -q https://github.com/docker-in-practice/todo.git
 ---> Using cache
 ---> 48db97331aa2
Step 3 : WORKDIR todo
 ---> Using cache
 ---> c5c85db751d6
Step 4 : RUN npm install > /dev/null
 ---> Using cache
 ---> be943c45c55b
Step 5 : EXPOSE 8000
 ---> Using cache
 ---> 805b18d28a65
Step 6 : CMD npm start
 ---> Using cache
 ---> 19525d4ec794
Successfully built 19525d4ec794
```

Indicates you're using the cache

Specifies the cached image/layer ID

The final image is "rebuilt," but in reality nothing has changed.

As useful and time-saving as this is, it's not always the behavior you want.

Taking the preceding Dockerfile as an example, imagine you'd changed your source code and pushed it to the Git repository. The new code wouldn't be checked out, because the `git clone` command hasn't changed. As far as the Docker build is concerned, it's the same, so the cached image can be reused.

In these cases, you'll want to rebuild your image without using the cache.

PROBLEM

You want to rebuild your Dockerfile without using the cache.

SOLUTION

To force a rebuild without using the image cache, run your `docker build` with the `--no-cache` flag. The following listing runs the previous build with `--no-cache`.

Listing 4.6 Forcing a rebuild without using the cache

```
$ docker build --no-cache .
 Sending build context to Docker daemon  2.56 kB
Sending build context to Docker daemon
```

Rebuilds the Docker image, ignoring cached layers with the --no-cache flag

```
Step 0 : FROM node
 ---> 91cbcf796c2c
Step 1 : MAINTAINER ian.miell@gmail.com
 ---> Running in ca243b77f6a1
  ---> 602f1294d7f1
 Removing intermediate container ca243b77f6a1
Step 2 : RUN git clone -q https://github.com/docker-in-practice/todo.git
 ---> Running in f2c0ac021247
 ---> 04ee24faaf18
Removing intermediate container f2c0ac021247
Step 3 : WORKDIR todo
 ---> Running in c2d9cd32c182
 ---> 4e0029de9074
Removing intermediate container c2d9cd32c182
Step 4 : RUN npm install > /dev/null
 ---> Running in 79122dbf9e52
npm WARN package.json todomvc-swarm@0.0.1 No repository field.
 ---> 9b6531f2036a
Removing intermediate container 79122dbf9e52
Step 5 : EXPOSE 8000
 ---> Running in d1d58e1c4b15
 ---> f7c1b9151108
Removing intermediate container d1d58e1c4b15
Step 6 : CMD npm start
 ---> Running in 697713ebb185
 ---> 74f9ad384859
Removing intermediate container 697713ebb185
Successfully built 74f9ad384859
```

No mention of caching this time

Intervening images have a different ID than in the previous listing.

A new image is built.

The output shows no mention of caching, and each intervening layer ID is different from the output in listing 4.5.

Similar problems can occur in other situations. We were flummoxed early on using Dockerfiles when a network blip meant that a command didn't retrieve something properly from the network, but the command didn't error. We kept calling `docker build`, but the resulting bug wouldn't go away! This was because a "bad" image had found its way into the cache, and we didn't understand the way Docker caching worked. Eventually we figured it out.

DISCUSSION

Removing caching can be a useful sanity check once you've got your final Dockerfile, to make sure it works from top to bottom, particularly when you're using internal web resources in your company that you may have changed while iterating on the Dockerfile. This situation doesn't occur if you're using `ADD`, because Docker will download the file every time to check if it has changed, but that behavior can be tiresome if you're pretty sure it's going to stay the same and you just want to get going with writing the rest of the Dockerfile.

TECHNIQUE 22 Busting the cache

Using the `--no-cache` flag is often enough to get around any problems with the cache, but sometimes you'll want a more fine-grained solution. If you have a build that takes a long time, for example, you may want to use the cache up to a certain point, and then invalidate it to rerun a command and create a new image.

PROBLEM

You want to invalidate the Docker build cache from a specific point in the Dockerfile build.

SOLUTION

Add a benign comment after the command to invalidate the cache.

Starting with the Dockerfile in https://github.com/docker-in-practice/todo (which corresponds to the `Step` lines in the following output), we've done a build and then added a comment in the Dockerfile on the line with `CMD`. You can see the output of doing `docker build` again here:

```
$ docker build .
 Sending build context to Docker daemon  2.56 kB
Sending build context to Docker daemon
Step 0 : FROM node
 ---> 91cbcf796c2c
Step 1 : MAINTAINER ian.miell@gmail.com
 ---> Using cache
 ---> 8f5a8a3d9240
Step 2 : RUN git clone -q https://github.com/docker-in-practice/todo.git
 ---> Using cache
 ---> 48db97331aa2
Step 3 : WORKDIR todo
 ---> Using cache
 ---> c5c85db751d6
Step 4 : RUN npm install
 ---> Using cache
 ---> be943c45c55b
Step 5 : EXPOSE 8000
 ---> Using cache
  ---> 805b18d28a65
Step 6 : CMD ["npm","start"] #bust the cache
  ---> Running in fc6c4cd487ce
 ---> d66d9572115e
 Removing intermediate container fc6c4cd487ce
Successfully built d66d9572115e
```

A "normal" docker build

Cache is used up to here.

Cache has been invalidated, but the command is effectively unchanged.

A new image has been created.

The reason this trick works is because Docker treats the non-whitespace change to the line as though it were a new command, so the cached layer is not re-used.

You may be wondering (as we did when we first looked at Docker) whether you can move Docker layers from image to image, merging them at will as though they were change sets in Git. This isn't possible at present within Docker. A layer is defined as a change set from a given image only. Because of this, once the cache has been broken, it can't be re-applied for commands re-used later in the build. For this reason, you're advised to put commands that are less likely to change nearer the top of the Dockerfile if possible.

DISCUSSION

For the initial iteration on a Dockerfile, splitting up every single command into a separate layer is excellent for speed of iteration, because you can selectively rerun parts of the process, as shown in the previous listing, but it's not so great for producing a small final image. It's not unheard-of for builds with a reasonable amount of complexity to approach the hard limit of 42 layers. To mitigate this, once you have a working build you're happy with, you should look at the steps in technique 56 for creating a production-ready image.

TECHNIQUE 23 Intelligent cache-busting using build-args

In the previous technique you saw how the cache can be busted mid-build by changing the relevant line.

In this technique we're going to take things a step further by controlling whether or not the cache is busted from the build command.

PROBLEM

You want to bust the cache on demand when performing a build, without editing the Dockerfile.

SOLUTION

Use the ARG directive in your Dockerfile to enable surgical cache-busting.

To demonstrate this, you're again going to use the Dockerfile at https://github.com/docker-in-practice/todo, but make a minor change to it.

What you want to do is control the busting of the cache before the npm install. Why would you want to do this? As you've learned, by default Docker will only break the cache if the command in the Dockerfile changes. But let's imagine that updated npm packages are available, and you want to make sure you get them. One option is to manually change the line (as you saw in the previous technique), but a more elegant way to achieve the same thing involves using the Docker ARGS directive and a bash trick.

Add the ARG line to the Dockerfile as follows.

Listing 4.7 Simple Dockerfile with bustable cache

```
WORKDIR todo
ARG CACHEBUST=no
 RUN npm install
```

The ARG directive sets an environment variable for the build.

In this example, you use the ARG directive to set the CACHEBUST environment variable and default it to no if it's not set by the docker build command.

Now build that Dockerfile "normally":

```
$ docker build .
Sending build context to Docker daemon   2.56kB
Step 1/7 : FROM node
latest: Pulling from library/node
aa18ad1a0d33: Pull complete
```

```
15a33158a136: Pull complete
f67323742a64: Pull complete
c4b45e832c38: Pull complete
f83e14495c19: Pull complete
41fea39113bf: Pull complete
f617216d7379: Pull complete
cbb91377826f: Pull complete
Digest: sha256:
➥ a8918e06476bef51ab83991aea7c199bb50bfb131668c9739e6aa7984da1c1f6
Status: Downloaded newer image for node:latest
 ---> 9ea1c3e33a0b
Step 2/7 : MAINTAINER ian.miell@gmail.com
 ---> Running in 03dba6770157
 ---> a5b55873d2d8
Removing intermediate container 03dba6770157
Step 3/7 : RUN git clone https://github.com/docker-in-practice/todo.git
 ---> Running in 23336fd5991f
Cloning into 'todo'...
 ---> 8ba06824d184
Removing intermediate container 23336fd5991f
Step 4/7 : WORKDIR todo
 ---> f322e2dbeb85
Removing intermediate container 2aa5ae19fa63
Step 5/7 : ARG CACHEBUST=no
 ---> Running in 9b4917f2e38b
 ---> f7e86497dd72
Removing intermediate container 9b4917f2e38b
Step 6/7 : RUN npm install
 ---> Running in a48e38987b04
npm info it worked if it ends with ok
[...]
added 249 packages in 49.418s
npm info ok
 ---> 324ba92563fd
Removing intermediate container a48e38987b04
Step 7/7 : CMD npm start
 ---> Running in ae76fa693697
 ---> b84dbc4bf5f1
Removing intermediate container ae76fa693697
Successfully built b84dbc4bf5f1
```

If you build it again with exactly the same `docker build` command, you'll observe that the Docker build cache is used, and no changes are made to the resulting image.

```
$ docker build .
Sending build context to Docker daemon   2.56kB
Step 1/7 : FROM node
 ---> 9ea1c3e33a0b
Step 2/7 : MAINTAINER ian.miell@gmail.com
 ---> Using cache
 ---> a5b55873d2d8
Step 3/7 : RUN git clone https://github.com/docker-in-practice/todo.git
 ---> Using cache
 ---> 8ba06824d184
```

```
Step 4/7 : WORKDIR todo
 ---> Using cache
 ---> f322e2dbeb85
Step 5/7 : ARG CACHEBUST=no
 ---> Using cache
 ---> f7e86497dd72
Step 6/7 : RUN npm install
 ---> Using cache
 ---> 324ba92563fd
Step 7/7 : CMD npm start
 ---> Using cache
 ---> b84dbc4bf5f1
Successfully built b84dbc4bf5f1
```

At this point you decide that you want to force the npm packages to be rebuilt. Perhaps a bug has been fixed, or you want to be sure you're up to date. This is where the ARG variable you added to the Dockerfile in listing 4.7 comes in. If this ARG variable is set to a value never used before on your host, the cache will be busted from that point.

This is where you use the build-arg flag to docker build, along with a bash trick to force a fresh value:

```
$ docker build --build-arg CACHEBUST=${RANDOM} .
 Sending build context to Docker daemon 4.096 kB
Step 1/9 : FROM node
 ---> 53d4d5f3b46e
Step 2/9 : MAINTAINER ian.miell@gmail.com
 ---> Using cache
 ---> 3a252318543d
Step 3/9 : RUN git clone https://github.com/docker-in-practice/todo.git
 ---> Using cache
 ---> c0f682653a4a
Step 4/9 : WORKDIR todo
 ---> Using cache
 ---> bd54f5d70700
Step 5/9 : ARG CACHEBUST=no
  ---> Using cache
 ---> 3229d52b7c33
Step 6/9 : RUN npm install
  ---> Running in 42f9b1f37a50
npm info it worked if it ends with ok
npm info using npm@4.1.2
npm info using node@v7.7.2
npm info attempt registry request try #1 at 11:25:55 AM
npm http request GET https://registry.npmjs.org/compression
npm info attempt registry request try #1 at 11:25:55 AM
[...]
Step 9/9 : CMD npm start
 ---> Running in 19219fe5307b
 ---> 129bab5e908a
Removing intermediate container 19219fe5307b
Successfully built 129bab5e908a
```

Run docker build with the build-arg flag, setting the CACHEBUST argument to a pseudo-random value generated by bash

Because the ARG CACHEBUST=no line itself has not changed, the cache is used here.

Because the CACHEBUST arg was set to a previously unset value, the cache is busted, and the npm install command is run again.

Note that the cache is busted on the line *following* the ARG line, not the ARG line itself. This can be a little confusing. The key thing to look out for is the “Running in” phrase—this means that a new container has been created to run the build line in.

The use of the ${RANDOM} argument is worth explaining. Bash provides you with this reserved variable name to give you an easy way of getting a value between one and five digits long:

```
$ echo ${RANDOM}
19856
$ echo ${RANDOM}
26429
$ echo ${RANDOM}
2856
```

This can come in handy, such as when you want a probably unique value to create files just for a specific run of a script.

You can even produce a much longer random number if you’re concerned about clashes:

```
$ echo ${RANDOM}${RANDOM}
434320509
$ echo ${RANDOM}${RANDOM}
1327340
```

Note that if you’re not using bash (or a shell that has this RANDOM variable available), this technique won’t work. In that case, you could use the date command instead to produce a fresh value:

```
$ docker build --build-arg CACHEBUST=$(date +%s) .
```

DISCUSSION

This technique has demonstrated a few things that will come in handy when using Docker. You’ve learned about using the --build-args flag to pass in a value to the Dockerfile and bust the cache on demand, creating a fresh build without changing the Dockerfile.

If you use bash, you’ve also learned about the RANDOM variable, and how it can be useful in other contexts than just Docker builds.

TECHNIQUE 24 Intelligent cache-busting using the ADD directive

In the previous technique you saw how the cache could be busted mid-build at a time of your choosing, which was itself a level up from using the --no-cache flag to ignore the cache completely.

Now you’re going to take it to the next level, so that you can automatically bust the cache only when it’s necessary to. This can save you a lot of time and compute—and therefore money!

PROBLEM

You want to bust the cache when a remote resource has changed.

SOLUTION

Use the Dockerfile `ADD` directive to only bust the cache when the response from a URL changes.

One of the early criticisms of Dockerfiles was that their claim of producing reliable build results was misleading. Indeed, we took this very subject up with the creator of Docker back in 2013 (http://mng.bz/B8E4).

Specifically, if you make a call to the network with a directive in your Dockerfile like this,

```
RUN git clone https://github.com/nodejs/node
```

then by default the Docker build will perform this once per Docker daemon. The code on GitHub could change substantially, but as far as your Docker daemon is concerned, the build is up to date. Years could pass, and the same Docker daemon will still be using the cache.

This may sound like a theoretical concern, but it's a very real one for many users. We've seen this happen many times at work, causing confusion. You've already seen some solutions to this, but for many complex or large builds, those solutions are not granular enough.

THE SMART CACHE-BUSTING PATTERN

Imagine you have a Dockerfile that looks like the following listing (note that it won't work! It's just a Dockerfile pattern to show the principle).

Listing 4.8 An example Dockerfile

Installs a series of packages as a prerequisite

```
FROM ubuntu:16.04
RUN apt-get install -y git and many other packages
 RUN git clone https://github.com/nodejs/node
 WORKDIR node
RUN make && make install
```

Clones a regularly changing repository (nodejs is just an example)

Runs a make and install command, which builds the project

This Dockerfile presents some challenges to creating an efficient build process. If you want to build everything from scratch each time, the solution is simple: use the `--no-cache` argument to `docker build`. The problem with this is that each time you run a build you're repeating the package installation in the second line, which is (mostly) unnecessary.

This challenge can be solved by busting the cache just before the `git clone` (demonstrated in the last technique). This raises another challenge, however: what if the Git repository hasn't changed? Then you're doing a potentially costly network transfer, followed by a potentially costly `make` command. Network, compute, and disk resources are all being used unnecessarily.

One way to get around this is to use technique 23, where you pass in a build argument with a new value every time you know that the remote repository has changed. But this still requires manual investigation to determine whether there has been a change, and intervention.

What you need is a command that can determine whether the resource has changed since the last build, and only then bust the cache.

THE ADD DIRECTIVE—UNEXPECTED BENEFITS

Enter the `ADD` directive!

You're already familiar with `ADD`, as it's a basic Dockerfile directive. Normally it's used to add a file to the resulting image, but there are two useful features of `ADD` that you can use to your advantage in this context: it caches the contents of the file it refers to, and it can take a network resource as an argument. This means that you can bust the cache whenever the output of a web request changes.

How can you take advantage of this when cloning a repository? Well, that depends on the nature of the resource you're referencing over the network. Many resources will have a page that changes when the repository itself changes, but these will vary from resource type to resource type. Here we'll focus on GitHub repos, because that's a common use case.

The GitHub API provides a useful resource that can help here. It has URLs for each repository that return JSON for the most recent commits. When a new commit is made, the content of the response changes.

Listing 4.9 Using ADD to trigger a cache bust

It doesn't matter where the output of the file goes, so we send it to /dev/null.

```
FROM ubuntu:16.04
ADD https://api.github.com/repos/nodejs/node/commits
➥ /dev/null
 RUN git clone https://github.com/nodejs/node
 [...]
```

The URL that changes when a new commit is made

The git clone will take place only when a change is made.

The result of the preceding listing is that the cache is busted only when a commit has been made to the repo since the last build. No human intervention is required, and no manual checking.

If you want to test this mechanism with a frequently changing repo, try using the Linux kernel.

Listing 4.10 Adding the Linux kernel code to an image

```
FROM ubuntu:16.04
ADD https://api.github.com/repos/torvalds/linux/commits /dev/null
 RUN echo "Built at: $(date)" >> /build_time
```

The ADD command, this time using the Linux repository

Outputs the system date into the built image, which will show when the last cache-busting build took place

If you create a folder and put the preceding code into a Dockerfile, and then run the following command regularly (every hour, for example), the output date will change only when the Linux Git repo changes.

Listing 4.11 Building a Linux code image

```
$ docker build -t linux_last_updated .
 $ docker run linux_last_updated cat /build_time
```

Builds the image and gives it the name linux_last_updated

Outputs the contents of the /build_time file from the resulting image

DISCUSSION

This technique demonstrated a valuable automated technique for ensuring builds only take place when necessary.

It also demonstrated some of the details of how the ADD command works. You saw that the "file" could be a network resource, and that if the contents of the file (or network resource) change from a previous build, a cache bust takes place.

In addition, you also saw that network resources have related resources that can indicate whether the resource you're referencing has changed. Although you could, for example, reference the main GitHub page to see if there are any changes there, it's likely that the page changes more frequently than the last commit (such as if the time of the web response is buried in the page source, or if there's a unique reference string in each response).

In the case of GitHub, you can reference the API, as you saw. Other services, such as BitBucket, offer similar resources. The Kubernetes project, for example, offers this URL to indicate which release is stable: https://storage.googleapis.com/kubernetes-release/release/stable.txt. If you were building a Kubernetes-based project, you might put an ADD line in your Dockerfile to bust the cache whenever this response changes.

TECHNIQUE 25 Setting the right time zone in your containers

If you've ever installed a full operating system, you'll know that setting the time zone is part of the process of setting it up. Even though a container isn't an operating system (or a virtual machine), it contains the files that tell programs how to interpret the time for the configured timezone.

PROBLEM

You want to set the time zone correctly for your containers.

SOLUTION

Replace the container's localtime file with a link to the time zone you want.

The following listing demonstrates the problem. It doesn't matter where in the world you run this, the container will show the same time zone.

Listing 4.12 Container starting with wrong time zone

```
$ date +%Z
 GMT
 $ docker run centos:7 date +%Z
 UTC
```

The time zone on the host is GMT.

Runs a command to display the time zone on the host

Runs a container and outputs the date from within that

The time zone in the container is GMT.

The container contains the files that determine which time zone is used by the container to interpret the time value it gets. The actual time used is, of course, tracked by the host operating system.

The next listing shows how you can set the time zone to the one you want.

Listing 4.13 Dockerfile for replacing the centos:7 default time zone

```
FROM centos:7
 RUN rm -rf /etc/localtime
 RUN ln -s /usr/share/zoneinfo/GMT /etc/localtime
 CMD date +%Z
```

Removes the existing localtime symlink file

Starts from the centos image we just looked at

Replaces the /etc/localtime link with a link to the time zone you want

Shows the time zone of your container as the default command to run

In listing 4.13 the key file is /etc/localtime. This points to the file that tells the container which time zone to use when it's asked for the time. The default time given is in the UTC time standard, which is used if the file doesn't exist (the minimal BusyBox image, for example, doesn't have it).

The following listing shows the output of building the preceding Dockerfile.

Listing 4.14 Building a time-zone-replacing Dockerfile

```
$ docker build -t timezone_change .
 Sending build context to Docker daemon 62.98 kB
Step 1 : FROM centos:7
7: Pulling from library/centos
45a2e645736c: Pull complete
Digest: sha256:
➥ c577af3197aacedf79c5a204cd7f493c8e07ffbce7f88f7600bf19c688c38799
Status: Downloaded newer image for centos:7
 ---> 67591570dd29
Step 2 : RUN rm -rf /etc/localtime
 ---> Running in fb52293849db
 ---> 0deda41be8e3
Removing intermediate container fb52293849db
Step 3 : RUN ln -s /usr/share/zoneinfo/GMT /etc/localtime
 ---> Running in 47bf21053b53
```

Builds the container

```
 ---> 5b5cb1197183
Removing intermediate container 47bf21053b53
Step 4 : CMD date +%Z
 ---> Running in 1e481eda8579
 ---> 9477cdaa73ac
Removing intermediate container 1e481eda8579
Successfully built 9477cdaa73ac
$ docker run timezone_change
 GMT
```

Runs the container

Outputs the specified time zone

In this way you can specify the time zone to use within—and only within—your container. Many applications depend on this setting, so it comes up not infrequently if you're running a Docker service.

There's another problem that this container-level time granularity can solve. If you're working for a multinational organization and run many different applications on servers based in data centers around the world, the ability to change the time zone in your image and trust that it will report the right time wherever it lands is a useful trick to have to hand.

DISCUSSION

Because the point of Docker images is explicitly to provide a consistent experience no matter where you run your container, there are a number of things you may stumble across if you do want varied results depending on where an image is deployed.

As an example, if you're automatically producing CSV spreadsheets of data for users in different locations, they may have certain expectations of the data format. American users might expect dates in the mm/dd format, whereas Europeans might expect dates in dd/mm format, and Chinese users might expect dates in their own character set.

In the next technique we'll consider locale settings, which affect how dates and times are printed in the `local` format, among other things.

TECHNIQUE 26 Locale management

In addition to time zones, locales are another aspect of Docker images that can be relevant when building images or running containers.

> **NOTE** A locale defines which language and country settings your programs should use. Typically a locale will be set in the environment through the `LANG`, `LANGUAGE`, and `locale-gen` variables, and through variables beginning with `LC_`, such as `LC_TIME`, whose setting determines how the time is displayed to the user.

> **NOTE** An encoding (in this context) is the means by which text is stored as bytes on a computer. A good introduction to this subject is available from W3C here: https://www.w3.org/International/questions/qa-what-is-encoding. It's worth taking the time to understand this subject, as it comes up in all sorts of contexts.

PROBLEM

You're seeing encoding errors in your application builds or deployments.

SOLUTION

Ensure the langugage-specific environment variables are correctly set in your Dockerfile.

Encoding issues aren't always obvious to all users, but they can be fatal when building applications.

Here are a couple of examples of typical encoding errors when building applications in Docker.

Listing 4.15 Typical encoding errors

```
MyFileDialog:66: error: unmappable character for encoding ASCII

UnicodeEncodeError: 'ascii' codec can't encode character u'\xa0' in
➥ position 20: ordinal not in range(128)
```

These errors can kill a build or an application stone dead.

> **TIP** A non-exhaustive list of key words to look out for in the error are "encoding," "ascii," "unicode," "UTF-8," "character," and "codec." If you see these words, chances are you're dealing with an encoding issue.

WHAT DOES THIS HAVE TO DO WITH DOCKER?

When you set up a full-blown operating system, you're typically guided through a setup process that asks you to confirm your preferred time zone, language, keyboard layout, and so on.

Docker containers, as you know by now, aren't full-blown operating systems set up for general use. Rather they're (increasingly) minimal environments for running applications. By default, therefore, they may not come with all the setup you're used to with an operating system.

In particular, Debian removed their dependency on the locales package in 2011, which means that, by default, there's no locale setup in a container based on a Debian image. For example, the following listing shows a Debian-derived Ubuntu image's default environment.

Listing 4.16 Default environment on an Ubuntu container

```
$  docker run -ti ubuntu bash
root@d17673300830:/# env
HOSTNAME=d17673300830
TERM=xterm
LS_COLORS=rs=0 [...]
HIST_FILE=/root/.bash_history
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
SHLVL=1
HOME=/root
_=/usr/bin/envj
```

There are no LANG or similar LC_ settings available in the image by default.

Our Docker host is shown in the next listing.

Listing 4.17 LANG setting on Docker host OS

```
$ env | grep LANG
LANG=en_GB.UTF-8
```

There's a LANG setting in our shell that informs applications that the preferred encoding in our terminal is British English, with text encoded in UTF-8.

To demonstrate an encoding issue, we'll create a file locally that contains a UTF-8-encoded UK currency symbol (the UK's pound sign), and then show how the interpretation of that file changes depending on the terminal's encoding.

Listing 4.18 Creating and showing a UTF-8-encoded UK currency symbol

```
$ env | grep LANG
LANG=en_GB.UTF-8
$ echo -e "\xc2\xa3" > /tmp/encoding_demo
 $ cat /tmp/encoding_demo
 £
```

Uses echo with the -e flag to output two bytes into a file, which represent a UK pound sign

Cats the file; we'll see a pound sign.

In UTF-8, a pound sign is represented by two bytes. We output these two bytes using `echo -e` and the `\x` notation and redirect output into a file. When we `cat` the file, the terminal reads the two bytes and knows to interpret the output as a pound sign.

Now if we change our terminal's encoding to use the Western (ISO Latin 1) encoding (which sets up our local LANG also) and output the file, it looks quite different:

Listing 4.19 Demonstrating the encoding problem with the UK currency symbol

```
$ env | grep LANG
 LANG=en_GB.ISO8859-1
 $ cat /tmp/encoding_demo
 Â£
```

The LANG environment variable is now set to Western (ISO Latin 1), which is set by the terminal.

The two bytes are interpreted differently, as two separate characters that are displayed to us.

The `\xc2` byte is interpreted as a capital *A* with a circumflex on top, and the `\xa3` byte is interpreted as a UK pound sign!

> **NOTE** We say "we" rather than "you" above deliberately! Debugging and controlling encodings is a tricky affair, which can depend on a combination of the running application's state, the environment variables you have set up, the running application, and all of the preceding factors that create the data you're examining!

As you've seen, encodings can be affected by the encoding set in the terminal. Getting back to Docker, we noted that no encoding environment variables were set by default in our Ubuntu container. Because of this, you can get different results when running the same commands on your host or in a container. If you see errors that seem to relate to encodings, you may need to set them in your Dockerfile.

Setting up encodings in a Dockerfile

We'll now look at how you can control the encoding of a Debian-based image. We've chosen this image because it's likely to be one of the more common contexts. This example will set up a simple image that just outputs its default environment variables.

Listing 4.20 Setting up a Dockerfile example

```
FROM ubuntu:16.04
RUN apt-get update && apt-get install -y locales
RUN locale-gen en_US.UTF-8
ENV LANG en_US.UTF-8
ENV LANGUAGE en_US:en
CMD env
```

Uses a Debian-derived base image

Updates the package index and installs the locales package

Generates the locale for US English, encoded with UTF-8

Sets the LANG environment variable

Sets the LANGUAGE environment variable

The default command env will display the environment setup for the container.

You may be wondering what the differences between the LANG and LANGUAGE variables are. Briefly, LANG is the default setting for the preferred language and encoding settings. It also provides a default when applications look for the more specific LC_* settings. LANGUAGE is used to provide an ordered list of languages preferred by applications if the principal one isn't available. More information can be found by running man locale.

Now you can build the image, and run it to see what's changed.

Listing 4.21 Building and running the encoding image

```
$ docker build -t encoding .
 [...]
$ docker run encoding
 no_proxy=*.local, 169.254/16
LANGUAGE=en_US:en
 HOSTNAME=aa9d6c8a3ff5
HOME=/root
HIST_FILE=/root/.bash_history
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
LANG=en_US.UTF-8
 PWD=/
```

Builds the encoding Docker image

Runs the built Docker image

The LANGUAGE variable is set in the environment

The LANG variable is set in the environment

DISCUSSION

Like the previous time zone technique, this technique illustrates an issue that catches people out on a regular basis. Like many of the more irritating issues we come across, these don't always make themselves obvious when the image is being built, which makes the time wasted debugging these issues very frustrating. For this reason, it's worth keeping these settings in mind when supporting others who are using Docker images.

TECHNIQUE 27 Stepping through layers with the image-stepper

If you've built an image that has a number of steps, you can often find yourself in the position of wanting to know where a particular file was introduced, or what state it was in at a particular point in the build. Combing through each image layer can be laborious, because you'll have to determine the order of layers, retrieve each ID, and start each one up using that ID.

This technique shows you a one-liner that tags each layer of the build in order, meaning you only have to increment a number to work through the images and find out whatever it is you need to know.

PROBLEM

You want to easily refer to each step of your build.

SOLUTION

Use the docker-in-practice/image-stepper image to order the tags for your image.

To illustrate this technique, we'll first show you a script that achieves this result so you understand how it works. Then we'll give you a constructed image to make achieving the result easier.

Here's a simple script that tags every layer in a given image (myimage) in the order of creation.

The Dockerfile for myimage follows.

Listing 4.22 Dockerfile for image with multiple layers

```
FROM debian
 RUN touch /file1
 RUN touch /file2
 RUN touch /file3
 RUN touch /file4
 RUN touch /file5
 RUN touch /file6
 RUN touch /file7
 RUN touch /file8
 RUN touch /file9
 RUN touch /file10
 CMD ["cat","/file1"]
```

Uses debian as a base image

Creates 10 files in separate layers

Runs a bespoke command that cats the first file

This is a simple enough Dockerfile, but it's one where it will be clear which stage you are at in the build.

Build this docker image with the following command.

Listing 4.23 Building the myimage image

```
$ docker build -t myimage -q .
 sha256:b21d1e1da994952d8e309281d6a3e3d14c376f9a02b0dd2ecbe6cabffea95288
```

Builds the image with the quiet (-q) flag, tagging it as myimage

The image identifier is the only output.

Once the image is built, you can run the following script.

Listing 4.24 Tagging each layer of myimage in numerical order

```
#!/bin/bash
x=1
 for id in $(docker history -q "myimage:latest" |
➥ grep -vw missing
➥ | tac)
 do
    docker tag "${id}" "myimage:latest_step_${x}"
     ((x++))
 done
```

Doesn't consider the remotely built layers, which are marked as missing (see the note below)

Initializes the counter variable (x) to I

Runs a for loop to retrieve the history of the image

Uses the tac utility to reverse the order of image IDs the docker history command outputs

In each iteration of the loop, tags the image appropriately with the incrementing number

Increments the step counter

If you save the preceding file as tag.sh and run it, the image will be tagged in layer order.

NOTE This tagging method technique will only work on images built locally. See the note in technique 16 for more information.

Listing 4.25 Tagging and showing the layers

```
$ ./tag.sh
 $ docker images | grep latest_step
 myimage    latest_step_12   1bfca0ef799d   3 minutes ago   123.1 MB
 myimage    latest_step_11   4d7f66939a4c   3 minutes ago   123.1 MB
 myimage    latest_step_10   78d31766b5cb   3 minutes ago   123.1 MB
 myimage    latest_step_9    f7b4dcbdd74f   3 minutes ago   123.1 MB
 myimage    latest_step_8    69b2fa0ce520   3 minutes ago   123.1 MB
 myimage    latest_step_7    b949d71fb58a   3 minutes ago   123.1 MB
 myimage    latest_step_6    8af3bbf1e7a8   3 minutes ago   123.1 MB
 myimage    latest_step_5    ce3dfbdfed74   3 minutes ago   123.1 MB
 myimage    latest_step_4    598ed62cabb9   3 minutes ago   123.1 MB
 myimage    latest_step_3    6b290f68d4d5   3 minutes ago   123.1 MB
 myimage    latest_step_2    586da987f40f   3 minutes ago   123.1 MB
 myimage    latest_step_1    19134a8202e7   7 days ago      123.1 MB
```

Runs the script from listing 4.24

Runs a docker images command with a simple grep to see the tagged layers

The steps to build the myimage image

The original (and older) base image has also been tagged as latest_step_l.

Now that you've seen the principle, we'll demonstrate how to dockerize this one-off script and make it work for the general case.

NOTE The code for this technique is available at https://github.com/docker-in-practice/image-stepper.

First, turn the previous script into a script that can take arguments.

Listing 4.26 Generic tagging script for the image-stepper image

```
#!/bin/bash
 IMAGE_NAME=$1
 IMAGE_TAG=$2
 if [[ $IMAGE_NAME = '' ]]
 then
     echo "Usage: $0 IMAGE_NAME [ TAG ]"
     exit 1
 fi
 if [[ $IMAGE_TAG = '' ]]
 then
     IMAGE_TAG=latest
 fi
 x=1
 for id in $(docker history -q "${IMAGE_NAME}:${IMAGE_TAG}" |
➥ grep -vw missing | tac)
 do
     docker tag "${id}" "${IMAGE_NAME}:${IMAGE_TAG}_step_$x"
     ((x++))
 done
```

Defines a bash script that can take two arguments: the image name to process, and the tag you want to step up to

The script from listing 4.24, with the arguments substituted in

You can then embed the script in listing 4.26 into a Docker image that you place into a Dockerfile and run as the default `ENTRYPOINT`.

Listing 4.27 Dockerfile for image-stepper image

```
FROM ubuntu:16.04
 RUN apt-get update -y && apt-get install -y docker.io
 ADD image_stepper /usr/local/bin/image_stepper
 ENTRYPOINT ["/usr/local/bin/image_stepper"]
```

Uses Ubuntu as a base layer

Installs docker.io to get the Docker client binary

Adds the script from listing 4.26 to the image

Runs the image_stepper script by default

The Dockerfile in listing 4.27 creates an image that runs the script in listing 4.26. The command in listing 4.28 runs this image, giving `myimage` as an argument.

This image, when run against another Docker image built on your host, will then create the tags for each step, allowing you to easily look at the layers in order.

The version of the client binary installed by the docker.io package must be compatible with the version of the Docker daemon on your host machine, typically meaning the client must not be newer.

Listing 4.28 Running image-stepper against another image

```
$ docker run --rm
➥ -v /var/run/docker.sock:/var/run/docker.sock
➥ dockerinpractice/image-stepper

➥ myimage
 Unable to find image 'dockerinpractice/image-stepper:latest' locally
 latest: Pulling from dockerinpractice/image-stepper
 b3e1c725a85f: Pull complete
 4daad8bdde31: Pull complete
 63fe8c0068a8: Pull complete
 4a70713c436f: Pull complete
 bd842a2105a8: Pull complete
 1a3a96204b4b: Pull complete
 d3959cd7b55e: Pull complete
 Digest: sha256:
➥ 65e22f8a82f2221c846c92f72923927402766b3c1f7d0ca851ad418fb998a753
 Status: Downloaded newer image for dockerinpractice/image-stepper:latest
 $ docker images | grep myimage

 myimage    latest          2c182dabe85c    24 minutes ago   123 MB
 myimage    latest_step_12  2c182dabe85c    24 minutes ago   123 MB
 myimage    latest_step_11  e0ff97533768    24 minutes ago   123 MB
 myimage    latest_step_10  f46947065166    24 minutes ago   123 MB
 myimage    latest_step_9   8a9805a19984    24 minutes ago   123 MB
 myimage    latest_step_8   88e42bed92ce    24 minutes ago   123 MB
 myimage    latest_step_7   5e638f955e4a    24 minutes ago   123 MB
 myimage    latest_step_6   f66b1d9e9cbd    24 minutes ago   123 MB
 myimage    latest_step_5   bd07d425bd0d    24 minutes ago   123 MB
 myimage    latest_step_4   ba913e75a0b1    24 minutes ago   123 MB
 myimage    latest_step_3   2ebcda8cd503    24 minutes ago   123 MB
 myimage    latest_step_2   58f4ed4fe9dd    24 minutes ago   123 MB
 myimage    latest_step_1   19134a8202e7    2 weeks ago      123 MB
 $ docker run myimage:latest_step_8 ls / | grep file
 file1
 file2
 file3
 file4
 file5
 file6
 file7
```

Runs the image-stepper image as a container, and removes the container when done

Mounts the host's docker socket, so you can use the Docker client installed in listing 4.27

Downloads the image-stepper image from the Docker Hub

Tags the myimage created previously

The output of the docker run command

Runs docker images and greps out the images you've just tagged

The images are tagged.

Picks a step at random and lists the files in the root directory, grep-ing out the ones created in the Dockerfile from listing 4.27

The files shown are those created up to that step.

NOTE On non-Linux OSs (such as Mac and Windows) you may need to specify the folder in which Docker runs in your Docker preferences as a file sharing setting.

This technique is useful for seeing where a particular file was added within a build, or what state a file was in at a particular point in the build. When debugging a build, this can be invaluable!

DISCUSSION

This technique is used in technique 52 to demonstrate that a deleted secret is accessible within a layer within an image.

TECHNIQUE 28 Onbuild and golang

The `ONBUILD` directive can cause a lot of confusion for new Docker users. This technique demonstrates its use in a real-world context by building and running a Go application with a two-line Dockerfile.

PROBLEM

You want to reduce the steps in building an image required for an application.

SOLUTION

Use the `ONBUILD` command to automate and encapsulate the building of an image.

First you'll run the process through, and then we'll explain what's going on. The example we'll use is the outyet project, which is an example in the golang GitHub repository. All it does is set up a web service that returns a page telling you whether Go 1.4 is available yet.

Build the image as follows.

Listing 4.29 Building the outyet image

```
$ git clone https://github.com/golang/example
$ cd example/outyet
$ docker build -t outyet .
```

Clones the Git repository

Navigates to the outyet folder

Builds the outyet image

Run a container from the resulting image, and retrieve the served web page.

Listing 4.30 Running and validating the outyet image

```
$ docker run
➥ --publish 8080:8080
➥ --name outyet1 -d outyet
$ curl localhost:8080
<!DOCTYPE html><html><body><center>
    <h2>Is Go 1.4 out yet?</h2>
    <h1>
        <a href="https://go.googlesource.com/go/+/go1.4">YES!</a>
    </h1>
</center></body></html>
```

The --publish flag tells Docker to publish the container's port 8080 on the external port 6060.

The --name flag gives your container a predictable name to make it easier to work with.

Runs the container in the background

Curls the output container's port

The webpage that the container serves

That's it—a simple application that returns a web page that tells you whether Go 1.4 is out yet or not.

If you look around the cloned repository, you'll see the Dockerfile is just two lines!

Listing 4.31 The onyet Dockerfile

```
FROM golang:onbuild
 EXPOSE 8080
```

Starts the build from the golang:onbuild image

Exposes port 8080

Confused yet? OK, it may make more sense when you look at the Dockerfile for the golang:onbuild image.

Listing 4.32 The golang:onbuild Dockerfile

```
FROM golang:1.7
 RUN mkdir -p /go/src/app
 WORKDIR /go/src/app
 CMD ["go-wrapper", "run"]
 ONBUILD COPY . /go/src/app
 ONBUILD RUN go-wrapper download
 ONBUILD RUN go-wrapper install
```

Uses the golang:l.7 image as a base

Makes a folder to store the application in

Moves into that folder

Sets the resulting image's command to call the go-wrapper to run the go app

The first ONBUILD command copies the code in the context of the Dockerfile into the image.

The second ONBUILD command downloads any dependencies, again using the go-wrapper command.

The third ONBUILD

The golang:onbuild image defines what happens when the image is used in the FROM directive in any other Dockerfile. The result is that when a Dockerfile uses this image as a base, the ONBUILD commands will fire as soon as the FROM image is downloaded, and (if not overridden) the CMD will be run when the resulting image is run as a container.

Now the output of the docker build command in the next listing may make more sense.

```
Step 1 : FROM golang:onbuild
 onbuild: Pulling from library/golang
 6d827a3ef358: Pull complete
 2726297beaf1: Pull complete
 7d27bd3d7fec: Pull complete
 62ace0d726fe: Pull complete
 af8d7704cf0d: Pull complete
 6d8851391f39: Pull complete
 988b98d9451c: Pull complete
 5bbc96f59ddc: Pull complete
 Digest: sha256:
➥ 886a63b8de95d5767e779dee4ce5ce3c0437fa48524aedd93199fb12526f15e0
 Status: Downloaded newer image for golang:onbuild
 # Executing 3 build triggers...
 Step 1 : COPY . /go/src/app
 Step 1 : RUN go-wrapper download
  ---> Running in c51f9b0c4da8
```

The FROM directive is run, and the golang:onbuild image is

The Docker build signals its intention to run the ONBUILD directives.

The first ONBUILD directive copies the Go code in the Dockerfile's context into the build.

The second ONBUILD directive is fired, which downloads.

```
+ exec go get -v -d
 Step 1 : RUN go-wrapper install
  ---> Running in adaa8f561320
+ exec go install -v
 app
 ---> 6bdbbeb8360f
Removing intermediate container 47c446aa70e3
 Removing intermediate container c51f9b0c4da8
 Removing intermediate container adaa8f561320
 Step 2 : EXPOSE 8080
  ---> Running in 564d4a34a34b
 ---> 5bf1767318e5
Removing intermediate container 564d4a34a34b
Successfully built 5bf1767318e5
```

The go-wrapper call triggers a shell call to go get.

The third ONBUILD directive is fired, which installs the application.

The go-wrapper call triggers a shell call to go install.

The EXPOSE directive in the second line of the Dockerfile is run.

The three containers built as a result of the ONBUILD command are removed.

The result of this technique is that you have an easy way to build an image that only contains the code required to run it, and no more. Leaving the build tools lying around in the image not only makes it larger than it needs to be, but also increases the security attack surface of the running container.

DISCUSSION

Because Docker and Go are fashionable technologies currently often seen together, we've used this to demonstrate how ONBUILD can be used to build a Go binary.

Other examples of ONBUILD images exist. There are node:onbuild and python:onbuild images available on Docker Hub.

It's hoped that this might inspire you to construct your own ONBUILD image that could help your organization with common patterns of building. This standardization can help reduce impedance mismatch between different teams even further.

Summary

- You can insert files from your local machine and from the internet into images.
- The cache is a crucial part of building images, but it can be a fickle friend and occasionally needs prompting to do what you want.
- You can "bust" the cache using build arguments or using the ADD directive, or you can ignore the cache completely with the no-cache option.
- The ADD directive is generally used to inject local files and folders into the built image.
- System configuration may still be relevant inside Docker, and image build time is a great time to do it.
- You can debug your build process using the "image-stepper" technique (technique 27), which tags each stage of the build for you.
- The time zone setting is the most common "gotcha" when configuring containers, especially when you are a non-U.S. or multinational company.
- Images with ONBUILD are very easy to use, because you might not need to customize the build at all.