

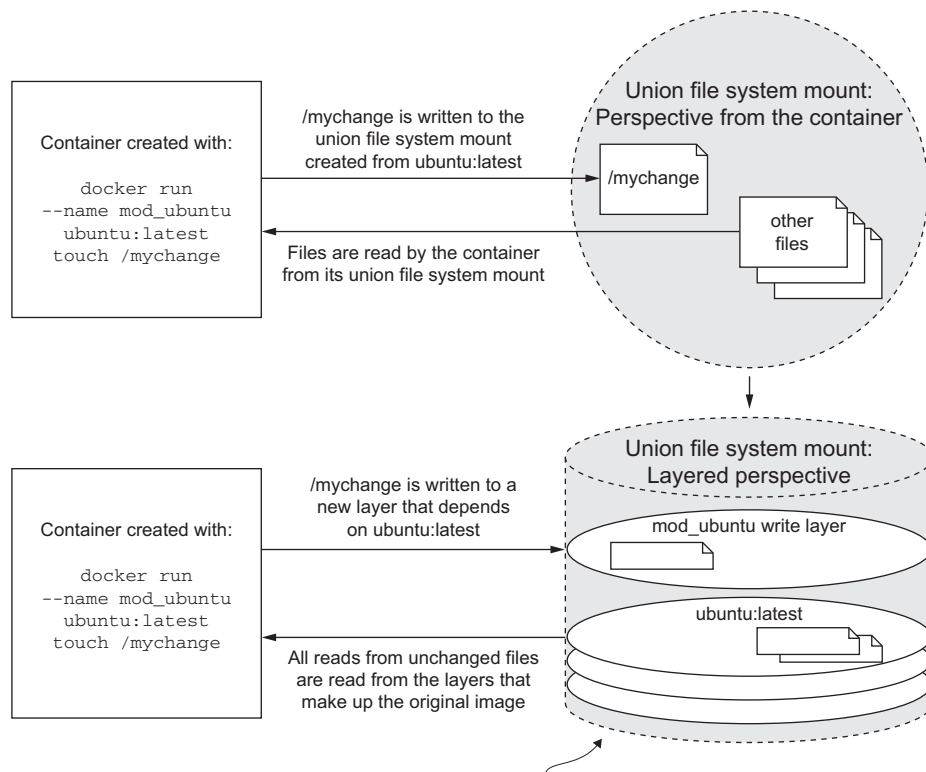
- Authors need to know the impact that adding, changing, and deleting files have on resulting images.
- Authors need have a solid understanding of the relationship between layers and how layers relate to images, repositories, and tags.

Start by considering a simple example. Suppose you want to make a single change to an existing image. In this case the image is `ubuntu:latest`, and you want to add a file named `mychange` to the root directory. You should use the following command to do this:

```
docker run --name mod_ubuntu ubuntu:latest touch /mychange
```

The resulting container (named `mod_ubuntu`) will be stopped but will have written that single change to its file system. As discussed in chapters 3 and 4, the root file system is provided by the image that the container was started from. That file system is implemented with something called a union file system.

A union file system is made up of layers. Each time a change is made to a union file system, that change is recorded on a new layer on top of all of the others. The “union” of all of those layers, or top-down view, is what the container (and user) sees when accessing the file system. Figure 7.2 illustrates the two perspectives for this example.



By looking at the union file system from the side—the perspective of its layers—you can begin to understand the relationship between different images and how file changes impact image size.

**Figure 7.2** A simple file write example on a union file system from two perspectives

When you read a file from a union file system, that file will be read from the top-most layer where it exists. If a file was not created or changed on the top layer, the read will fall through the layers until it reaches a layer where that file does exist. This is illustrated in figure 7.3.

All this layer functionality is hidden by the union file system. No special actions need to be taken by the software running in a container to take advantage of these features. Understanding layers where files were added covers one of three types of file system writes. The other two are deletions and file changes.

Like additions, both file changes and deletions work by modifying the top layer. When a file is deleted, a delete record is written to the top layer, which overshadows any versions of that file on lower layers. When a file is changed, that change is written to the top layer, which again shadows any versions of that file on lower layers. The changes made to the file system of a container are listed with the `docker diff` command you used earlier in the chapter:

```
docker diff mod_ubuntu
```

This command will produce the output:

```
A /mychange
```

The **A** in this case indicates that the file was added. Run the next two commands to see how a file deletion is recorded:

```
docker run --name mod_busybox_delete busybox:latest rm /etc/profile
docker diff mod_busybox_delete
```

This time the output will have two rows:

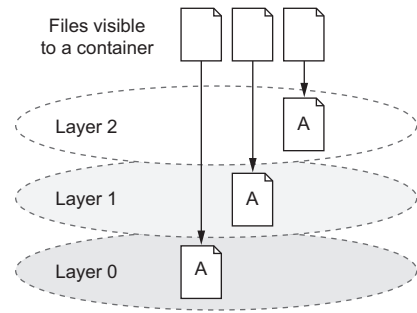
```
C /etc
D /etc/profile
```

The **D** indicates a deletion, but this time the parent folder of the file was also included. The **C** indicates that it was changed. The next two commands demonstrate a file change:

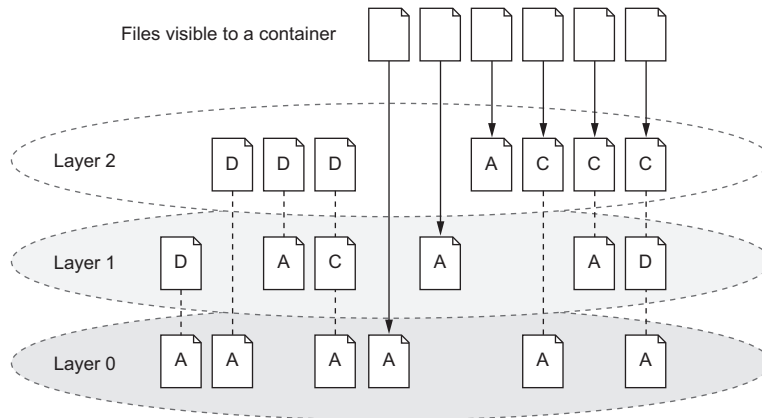
```
docker run --name mod_busybox_change busybox:latest touch /etc/profile
docker diff mod_busybox_change
```

The `diff` subcommand will show two changes:

```
C /etc
C /etc/profile
```



**Figure 7.3** Reading files that are located on different layers



**Figure 7.4** Various file addition, change, and deletion combinations over a three-layered image

Again, the **C** indicates a change, and the two items are the file and the folder where it's located. If a file nested five levels deep were changed, there would be a line for each level of the tree. File-change mechanics are the most important thing to understand about union file systems.

Most union file systems use something called copy-on-write, which is easier to understand if you think of it as copy-on-change. When a file in a read-only layer (not the top layer) is modified, the whole file is first copied from the read-only layer into the writable layer before the change is made. This has a negative impact on runtime performance and image size. Section 7.2.3 covers the way this should influence your image design.

Take a moment to solidify your understanding of the system by examining how the more comprehensive set of scenarios is illustrated in figure 7.4. In this illustration files are added, changed, deleted, and added again over a range of three layers.

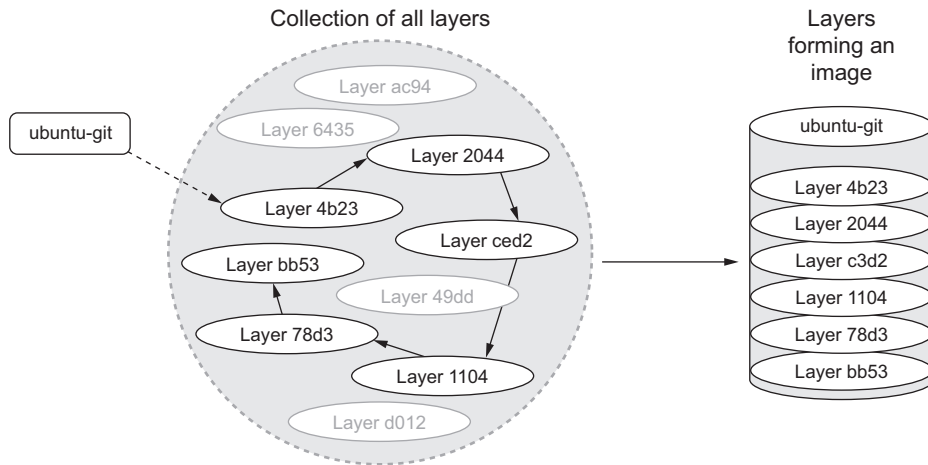
Knowing how file system changes are recorded, you can begin to understand what happens when you use the `docker commit` command to create a new image.

### 7.2.2 Reintroducing images, layers, repositories, and tags

You've created an image using the `docker commit` command, and you understand that it commits the top-layer changes to an image. But we've yet to define *commit*.

Remember, a union file system is made up of a stack of layers where new layers are added to the top of the stack. Those layers are stored separately as collections of the changes made in that layer and metadata for that layer. When you commit a container's changes to its file system, you're saving a copy of that top layer in an identifiable way.

When you commit the layer, a new ID is generated for it, and copies of all the file changes are saved. Exactly how this happens depends on the storage engine that's



**Figure 7.5** An image is the collection of layers produced by traversing the parent graph from a top layer.

being used on your system. It's less important for you to understand the details than it is for you to understand the general approach. The metadata for a layer includes that generated identifier, the identifier of the layer below it (parent), and the execution context of the container that the layer was created from. Layer identities and metadata form the graph that Docker and the UFS use to construct images.

An image is the stack of layers that you get by starting with a given top layer and then following all the links defined by the parent ID in each layer's metadata, as shown in figure 7.5.

Images are stacks of layers constructed by traversing the layer dependency graph from some starting layer. The layer that the traversal starts from is the top of the stack. This means that a layer's ID is also the ID of the image that it and its dependencies form. Take a moment to see this in action by committing the `mod_ubuntu` container you created earlier:

```
docker commit mod_ubuntu
```

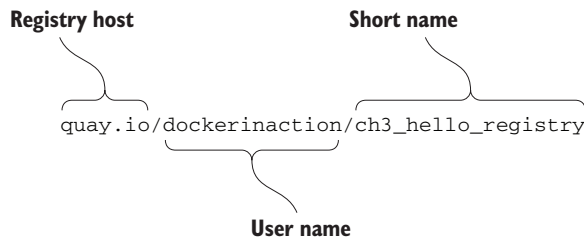
That commit subcommand will generate output that includes a new image ID like this:

```
6528255cda2f9774a11a6b82be46c86a66b5feff913f5bb3e09536a54b08234d
```

You can create a new container from this image using the image ID as it's presented to you. Like containers, layer IDs are large hexadecimal numbers that can be difficult for a person to work with directly. For that reason, Docker provides repositories.

In chapter 3, a *repository* is roughly defined as a named bucket of images. More specifically, repositories are location/name pairs that point to a set of specific layer

identifiers. Each repository contains at least one tag that points to a specific layer identifier and thus the image definition. Let's revisit the example used in chapter 3:



This repository is located in the registry hosted at `quay.io`. It's named for the user (`dockerinaction`) and a unique short name (`ch3_hello_registry`). Pulling this repository would pull all the images defined for each tag in the repository. In this example, there's only one tag, `latest`. That tag points to a layer with the short form ID `07c0f84777ef`, as illustrated in figure 7.6.

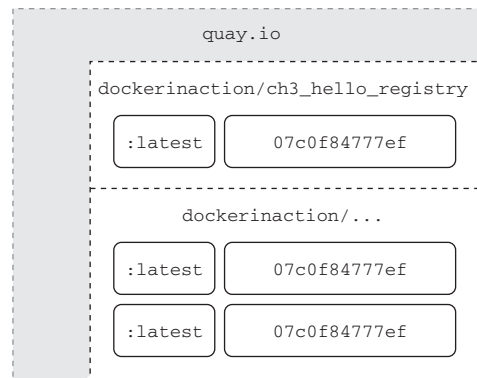
Repositories and tags are created with the `docker tag`, `docker commit`, or `docker build` commands. Revisit the `mod_ubuntu` container again and put it into a repository with a tag:

```
docker commit mod_ubuntu myuser/myfirstrepo:mytag
# Outputs:
# 82ec7d2c57952bf57ab1ffdf40d5374c4c68228e3e923633734e68a11f9a2b59
```

The generated ID that's displayed will be different because another copy of the layer was created. With this new friendly name, creating containers from your images requires little effort. If you want to copy an image, you only need to create a new tag or repository from the existing one. You can do that with the `docker tag` command. Every repository contains a "latest" tag by default. That will be used if the tag is omitted like in the previous command:

```
docker tag myuser/myfirstrepo:mytag myuser/mod_ubuntu
```

By this point you should have a strong understanding of basic UFS fundamentals as well as how Docker creates and manages layers, images, and repositories. With these in mind, let's consider how they might impact image design.



**Figure 7.6** A visual representation of repositories

All layers below the writable layer created for a container are immutable, meaning they can never be modified. This property makes it possible to share access to images instead of creating independent copies for every container. It also makes individual layers highly reusable. The other side of this property is that anytime you make changes to an image, you need to add a new layer, and old layers are never removed. Knowing that images will inevitably need to change, you need to be aware of any image limitations and keep in mind how changes impact image size.

### 7.2.3 *Managing image size and layer limits*

If images evolved in the same way that most people manage their file systems, Docker images would quickly become unusable. For example, suppose you wanted to make a different version of the ubuntu-git image you created earlier in this chapter. It may seem natural to modify that ubuntu-git image. Before you do, create a new tag for your ubuntu-git image. You'll be reassigning the latest tag:

```
docker tag ubuntu-git:latest ubuntu-git:1.9
```

← Create new tag: 1.9

The first thing you'll do in building your new image is remove the version of Git you installed:

```
docker run --name image-dev2 \
  --entrypoint /bin/bash \
  ubuntu-git:latest -c "apt-get remove -y git"
```

← Execute bash command

← Remove Git

Commit image → 

```
docker commit image-dev2 ubuntu-git:removed
```

```
docker tag -f ubuntu-git:removed ubuntu-git:latest
```

 ← Reassign latest tag

```
docker images
```

 ← Examine image sizes

The image list and sizes reported will look something like the following:

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
ubuntu-git	latest	826c66145a59	10 seconds ago	226.6 MB
ubuntu-git	removed	826c66145a59	10 seconds ago	226.6 MB
ubuntu-git	1.9	3e356394c14e	41 hours ago	226 MB
...				

Notice that even though you removed Git, the image actually increased in size. Although you could examine the specific changes with `docker diff`, you should be quick to realize that the reason for the increase has to do with the union file system.

Remember, UFS will mark a file as deleted by actually adding a file to the top layer. The original file and any copies that existed in other layers will still be present in the image. It's important to minimize image size for the sake of the people and systems that will be consuming your images. If you can avoid causing long download times and significant disk usage with smart image creation, then your consumers will benefit. There's also another risk with this approach that you should be aware of.

The union file system on your computer may have a layer count limit. These limits vary, but a limit of 42 layers is common on computers that use the AUFS system. This number may seem high, but it's not unreachable. You can examine all the layers in an image using the `docker history` command. It will display the following:

- Abbreviated layer ID
- Age of the layer
- Initial command of the creating container
- Total file size of that layer

By examining the history of the `ubuntu-git:removed` image, you can see that three layers have already been added on the top of the original `ubuntu:latest` image:

```
docker history ubuntu-git:removed
```

Outputs are something like:

IMAGE	CREATED	CREATED BY	SIZE
826c66145a59	24 minutes ago	/bin/bash -c apt-get remove	662 kB
3e356394c14e	42 hours ago	git	0 B
bbf1d5d430cd	42 hours ago	/bin/bash	37.68 MB
b39b81afc8ca	3 months ago	/bin/sh -c #(nop) CMD [/bin	0 B
615c102e2290	3 months ago	/bin/sh -c sed -i 's/^#\s*\	1.895 kB
837339b91538	3 months ago	/bin/sh -c echo '#!/bin/sh'	194.5 kB
53f858aaaf03	3 months ago	/bin/sh -c #(nop) ADD file:	188.1 MB
511136ea3c5a	22 months ago		0 B

You can flatten images if you export them and then reimport them with Docker. But that's a bad idea because you lose the change history as well as any savings customers might get when they download images with the same lower levels. Flattening images defeats the purpose. The smarter thing to do in this case is to create a branch.

Instead of fighting the layer system, you can solve both the size and layer growth problems by using the layer system to create branches. The layer system makes it trivial to go back in the history of an image and make a new branch. You are potentially creating a new branch every time you create a container from the same image.

In reconsidering your strategy for your new `ubuntu-git` image, you should simply start from `ubuntu:latest` again. With a fresh container from `ubuntu:latest`, you could install whatever version of Git you want. The result would be that both the original `ubuntu-git` image you created and the new one would share the same parent, and the new image wouldn't have any of the baggage of unrelated changes.

Branching increases the likelihood that you'll need to repeat steps that were accomplished in peer branches. Doing that work by hand is prone to error. Automating image builds with Dockerfiles is a better idea.

Occasionally the need arises to build a full image from scratch. This practice can be beneficial if your goal is to keep images small and if you're working with technologies that have few dependencies. Other times you may want to flatten an image to trim an image's history. In either case, you need a way to import and export full file systems.

### 7.3 **Exporting and importing flat file systems**

On some occasions it's advantageous to build images by working with the files destined for an image outside the context of the union file system or a container. To fill this need, Docker provides two commands for exporting and importing archives of files.

The `docker export` command will stream the full contents of the flattened union file system to stdout or an output file as a tarball. The result is a tarball that contains all the files from the container perspective. This can be useful if you need to use the file system that was shipped with an image outside the context of a container. You can use the `docker cp` command for this purpose, but if you need several files, exporting the full file system may be more direct.

Create a new container and use the `export` subcommand to get a flattened copy of its filesystem:

```
docker run --name export-test \
  dockerinaction/ch7_packed:latest ./echo For Export
docker export --output contents.tar export-test
docker rm export-test
tar -tf contents.tar
```



This will produce a file in the current directory named `contents.tar`. That file should contain two files. At this point you could extract, examine, or change those files to whatever end. If you had omitted the `--output` (or `-o` for short), then the contents of the file system would be streamed in tarball format to stdout. Streaming the contents to stdout makes the `export` command useful for chaining with other shell programs that work with tarballs.

The `docker import` command will stream the content of a tarball into a new image. The `import` command recognizes several compressed and uncompressed forms of tarballs. An optional Dockerfile instruction can also be applied during file-system import. Importing file systems is a simple way to get a complete minimum set of files into an image.

To see how useful this is, consider a statically linked Go version of Hello World. Create an empty folder and copy the following code into a new file named `hello-world.go`:

```
package main
import "fmt"
func main() {
    fmt.Println("hello, world!")
}
```

You may not have Go installed on your computer, but that's no problem for a Docker user. By running the next command, Docker will pull an image containing the Go compiler, compile and statically link the code (which means it can run all by itself), and place that program back into your folder:

```
docker run --rm -v "$(pwd)":/usr/src/hello \
  -w /usr/src/hello golang:1.3 go build -v
```



If everything works correctly, you should have an executable program (binary file) in the same folder, named `hello`. Statically linked programs have no external file dependencies at runtime. That means this statically linked version of Hello World can run in a container with no other files. The next step is to put that program in a tarball:

```
tar -cf static_hello.tar hello
```

Now that the program has been packaged in a tarball, you can import it using the `docker import` command:

```
docker import -c "ENTRYPOINT [\"/hello\"]" - \
  dockerinaction/ch7_static < static_hello.tar
```

← Tar file streamed via UNIX pipe

In this command you use the `-c` flag to specify a Dockerfile command. The command you use sets the entrypoint for the new image. The exact syntax of the Dockerfile command is covered in chapter 8. The more interesting argument on this command is the hyphen (`-`) at the end of the first line. This hyphen indicates that the contents of the tarball will be streamed through stdin. You can specify a URL at this position if you're fetching the file from a remote web server instead of from your local file system.

You tagged the resulting image as the `dockerinaction/ch7_static` repository. Take a moment to explore the results:

```
docker run dockerinaction/ch7_static
docker history dockerinaction/ch7_static
```

← Outputs: hello, world!

You'll notice that the history for this image has only a single entry (and layer):

IMAGE	CREATED	CREATED BY	SIZE
edafbd4a0ac5	11 minutes ago		1.824 MB

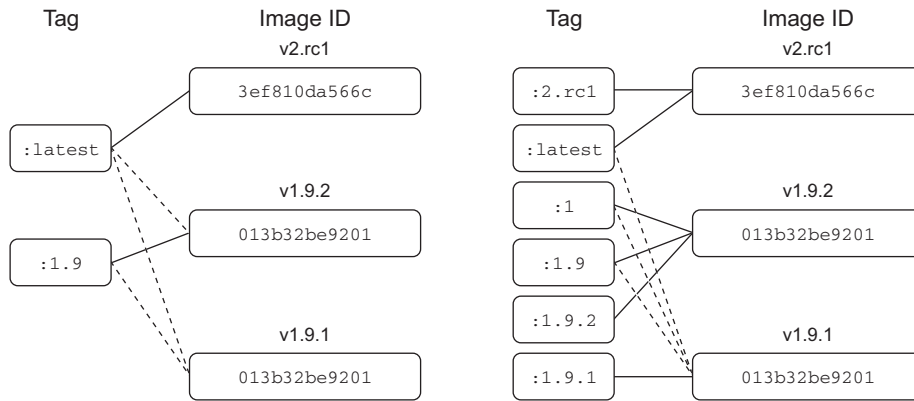
In this case, the image we produced was small for two reasons. First, the program we produced was only just over 1.8 MB, and we included no operating system files or support programs. This is a minimalistic image. Second, there's only one layer. There are no deleted or unused files carried with the image in lower layers. The downside to using single-layer (or flat) images is that your system won't benefit from layer reuse. That might not be a problem if all your images are small enough. But the overhead may be significant if you use larger stacks or languages that don't offer static linking.

There are trade-offs to every image design decision, including whether or not to use flat images. Regardless of the mechanism you use to build images, your users need a consistent and predictable way to identify different versions.

## 7.4 Versioning best practices

Pragmatic versioning practices help users make the best use of images. The goal of an effective versioning scheme is to communicate clearly and provide adoption flexibility.

It's generally insufficient to build or maintain only a single version of your software unless it's your first. If you're releasing the first version of your software, you should be mindful of your users' adoption experience immediately. The reason why versions are



**Figure 7.7** Two different tagging schemes (left and right) for the same repository with three images. Dotted lines represent old relationships between a tag and an image.

important is that they identify contracts that your adopters depend on. Unexpected software changes cause problems.

With Docker, the key to maintaining multiple versions of the same software is proper repository tagging. The understanding that every repository contains multiple tags and that multiple tags can reference the same image is at the core of a pragmatic tagging scheme.

The `docker tag` command is unlike the other two commands that can be used to create tags. It's the only one that's applied to existing images. To understand how to use tags and how they impact the user adoption experience, consider the two tagging schemes for a repository shown in figure 7.7.

There are two problems with the tagging scheme on the left side of figure 7.7. First, it provides poor adoption flexibility. A user can choose to declare a dependency on `1.9` or `latest`. When a user adopts version 1.9 and that implementation is actually 1.9.1, they may develop dependencies on behavior defined by that build version. Without a way to explicitly depend on that build version, they will experience pain when 1.9 is updated to point to 1.9.2.

The best way to eliminate this problem is to define and tag versions at a level where users can depend on consistent contracts. This is not advocating a three-tiered versioning system. It means only that the smallest unit of the versioning system you use captures the smallest unit of contract iteration. By providing multiple tags at this level, you can let users decide how much version drift they want to accept.

Consider the right side of figure 7.7. A user who adopts version 1 will always use the highest minor and build version under that major version. Adopting 1.9 will always use the highest build version for that minor version. Adopters who need to carefully migrate between versions of their dependencies can do so with control and at times of their choosing.

The second problem is related to the `latest` tag. On the left, `latest` currently points to an image that's not otherwise tagged, and so an adopter has no way of knowing what version of the software that is. In this case, it's referring to a release candidate for the next major version of the software. An unsuspecting user may adopt the `latest` tag with the impression that it's referring to the latest build of an otherwise tagged version.

There are other problems with the `latest` tag. It's adopted more frequently than it should be. This happens because it's the default tag, and Docker has a young community. The impact is that a responsible repository maintainer should always make sure that its repository's `latest` refers to the latest stable build of its software instead of the true latest.

The last thing to keep in mind is that in the context of containers, you're versioning not only your software but also a snapshot of all of your software's packaged dependencies. For example, if you package software with a particular distribution of Linux, like Debian, then those additional packages become part of your image's interface contract. Your users will build tooling around your images and in some cases may come to depend on the presence of a particular shell or script in your image. If you suddenly rebase your software on something like CentOS but leave your software otherwise unchanged, your users will experience pain.

In situations where the software dependencies change, or the software needs to be distributed on top of multiple bases, then those dependencies should be included with your tagging scheme.

The Docker official repositories are ideal examples to follow. Consider this tag list for the official golang repository, where each row represents a distinct image:

1.3.3,	1.3		
1.3.3-onbuild,	1.3-onbuild		
1.3.3-cross,	1.3-cross		
1.3.3-wheezy,	1.3-wheezy		
1.4.2,	1.4,	1,	latest
1.4.2-onbuild,	1.4-onbuild,	1-onbuild,	onbuild
1.4.2-cross,	1.4-cross,	1-cross,	cross
1.4.2-wheezy,	1.4-wheezy,	1-wheezy,	wheezy

The columns are neatly organized by their scope of version creep with build-level tags on the left and major versions to the right. Each build in this case has an additional base image component, which is annotated in the tag.

Users know that the latest version is actually version 1.4.2. If an adopter needs the latest image built on the `debian:wheezy` platform, they can use the `wheezy` tag. Those who need a 1.4 image with `ONBUILD` triggers can adopt `1.4-onbuild`. This scheme puts the control and responsibility for upgrades in the hands of your adopters.

## 7.5 Summary

This is the first chapter to cover the creation of Docker images, tag management, and other distribution concerns such as image size. Learning this material will help you

build images and become a better consumer of images. The following are the key points in the chapter:

- New images are created when changes to a container are committed using the `docker commit` command.
- When a container is committed, the configuration it was started with will be encoded into the configuration for the resulting image.
- An image is a stack of layers that's identified by its top layer.
- An image's size on disk is the sum of the sizes of its component layers.
- Images can be exported to and imported from a flat tarball representation using the `docker export` and `docker import` commands.
- The `docker tag` command can be used to assign several tags to a single repository.
- Repository maintainers should keep pragmatic tags to ease user adoption and migration control.
- Tag your latest stable build with the `latest` tag.
- Provide fine-grained and overlapping tags so that adopters have control of the scope of their dependency version creep.

# *Build automation and advanced image considerations*

---

## ***This chapter covers***

- Automated packaging with Dockerfile
- Metadata instructions
- File system instructions
- Packaging for multiprocess and durable containers
- Trusted base images
- Working with users
- Reducing the image attack surface

A Dockerfile is a file that contains instructions for building an image. The instructions are followed by the Docker image builder from top to bottom and can be used to change anything about an image. Building images from Dockerfiles makes tasks like adding files to a container from your computer simple one-line instructions. This section covers the basics of working with Dockerfile builds and the best reasons to use them, a lean overview of the instructions, and how to add future build behavior. We'll get started with a familiar example.

## 8.1 Packaging Git with a Dockerfile

Let's start by revisiting the Git on Ubuntu example. Having previously built a similar image by hand, you should recognize many of the details and advantages of working with a Dockerfile.

First, create a new directory and from that directory create a new file with your favorite text editor. Name the new file `Dockerfile`. Write the following five lines and then save the file:

```
# An example Dockerfile for installing Git on Ubuntu
FROM ubuntu:latest
MAINTAINER "dockerinaction@allingeek.com"
RUN apt-get install -y git
ENTRYPOINT ["git"]
```

Before dissecting this example, build a new image from it with the `docker build` command from the same directory containing the `Dockerfile`. Tag the new image with `auto`:

```
docker build --tag ubuntu-git:auto .
```

Outputs several lines about steps and output from `apt-get` and will finally display a message like this:

```
Successfully built 0bca8436849b
```

Running this command starts the build process. When it's completed, you should have a brand-new image that you can test. View the list of all your `ubuntu-git` images and test the newest one with this command:

```
docker images
```

The new build tagged “auto” should now appear in the list:

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
ubuntu-git	auto	0bca8436849b	10 seconds ago	225.9 MB
ubuntu-git	latest	826c66145a59	10 minutes ago	226.6 MB
ubuntu-git	removed	826c66145a59	10 minutes ago	226.6 MB
ubuntu-git	1.9	3e356394c14e	41 hours ago	226 MB
...				

Now you can run a Git command using the new image:

```
docker run --rm ubuntu-git:auto
```

These commands demonstrate that the image you built with the `Dockerfile` works and is functionally equivalent to the one you built by hand. Examine what you did to accomplish this:

First, you created a `Dockerfile` with four instructions:

- `FROM ubuntu:latest`—Tells Docker to start from the latest Ubuntu image just as you did when creating the image manually.

- **MAINTAINER**—Sets the maintainer name and email for the image. Providing this information helps people know whom to contact if there's a problem with the image. This was accomplished earlier when you invoked `commit`.
- **RUN** `apt-get install -y git`—Tells the builder to run the provided command to install Git.
- **ENTRYPOINT** `["git"]`—Sets the entrypoint for the image to `git`.

Dockerfiles, like most scripts, can include comments. Any line beginning with a `#` will be ignored by the builder. It's important for Dockerfiles of any complexity to be well documented. In addition to improving Dockerfile maintainability, comments help people audit images that they're considering for adoption and spread best practices.

The only special rule about Dockerfiles is that the first instruction must be **FROM**. If you're starting from an empty image and your software has no dependencies, or you'll provide all the dependencies, then you can start from a special empty repository named `scratch`.

After you saved the Dockerfile, you started the build process by invoking the `docker build` command. The command had one flag set and one argument. The `--tag` flag (or `-t` for short) specifies the full repository designation that you want to use for the resulting image. In this case you used `ubuntu-git:auto`. The argument that you included at the end was a single period. That argument told the builder the location of the Dockerfile. The period told it to look for the file in the current directory.

The `docker build` command has another flag, `--file` (or `-f` for short), that lets you set the name of the Dockerfile. `Dockerfile` is the default, but with this flag you could tell the builder to look for a file named `BuildScript`. This flag sets only the name of the file, not the location of the file. That must always be specified in the location argument.

The builder works by automating the same tasks that you'd use to create images by hand. Each instruction triggers the creation of a new container with the specified modification. After the modification has been made, the builder commits the layer and moves on to the next instruction and container created from the fresh layer.

The builder validated that the image specified by the **FROM** instruction was installed as the first step of the build. If it were not, Docker would have automatically tried to pull the image. Take a look at the output from the `build` command that you ran:

```
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:latest
----> b39b81afc8ca
```

You can see that in this case the base image specified by the **FROM** instruction is `ubuntu:latest`, which should have already been installed on your machine. The abbreviated image ID of the base image is included in the output.

The next instruction sets the maintainer information on the image. This creates a new container and then commits the resulting layer. You can see the result of this operation in the output for step 1:

```
Step 1 : MAINTAINER "dockerinaction@allingeek.com"
---> Running in 938ff06bf8f4
---> 80a695671201
Removing intermediate container 938ff06bf8f4
```

The output includes the ID of the container that was created and the ID of the committed layer. That layer will be used as the top of the image for the next instruction, `RUN`. The output for the `RUN` instruction was clouded with all the output for the command `apt-get install -y git`. If you're not interested in this output, you can invoke the `docker build` command with the `--quiet` or `-q` flag. Running in quiet mode will suppress output from the intermediate containers. Without the container output, the `RUN` step produces output that looks like this:

```
Step 2 : RUN apt-get install -y git
---> Running in 4438c3b2c049
---> 1c20f8970532
Removing intermediate container 4438c3b2c049
```

Although this step usually takes much longer to complete, you can see the instruction and input as well as the ID of the container where the command was run and the ID of the resulting layer. Finally, the `ENTRYPOINT` instruction performs all the same steps, and the output is similarly unsurprising:

```
Step 3 : ENTRYPOINT git
---> Running in c9b24b0f035c
---> 89d726cf3514
Removing intermediate container c9b24b0f035c
```

A new layer is being added to the resulting image after each step in the build. Although this means you could potentially branch on any of these steps, the more important implication is that the builder can aggressively cache the results of each step. If a problem with the build script occurs after several other steps, the builder can restart from the same position after the problem has been fixed. You can see this in action by breaking your Dockerfile.

Add this line to the end of your Dockerfile:

```
RUN This will not work
```

Then run the build again:

```
docker build --tag ubuntu-git:auto .
```

The output will show which steps the builder was able to skip in favor of cached results:

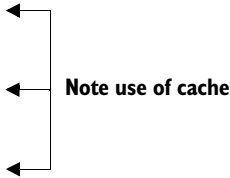
```
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
```



```

Step 0 : FROM ubuntu:latest
---> b39b81afc8ca
Step 1 : MAINTAINER "dockerinaction@allingeek.com"
---> Using cache
---> 80a695671201
Step 2 : RUN apt-get install -y git
---> Using cache
---> 1c20f8970532
Step 3 : ENTRYPOINT git
---> Using cache
---> 89d726cf3514
Step 4 : RUN This will not work
---> Running in f68f0e0418b5
/bin/sh: 1: This: not found
INFO[0001] The command [/bin/sh -c This will not work] returned a non-zero
code: 127

```



Note use of cache

Steps 1 through 3 were skipped because they were already built during your last build. Step 4 failed because there's no program with the name `This` in the container. The container output was valuable in this case because the error message informs you about the specific problem with the Dockerfile. If you fix the problem, the same steps will be skipped again, and the build will succeed, resulting in output like `Successfully built d7a8ee0cebd4`.

The use of caching during the build can save time if the build includes downloading material, compiling programs, or anything else that is time-intensive. If you need a full rebuild, you can use the `--no-cache` flag on `docker build` to disable the use of the cache. But make sure you're disabling the cache only when absolutely required.

This short example uses 4 of the 14 Dockerfile instructions. The example was limited in that all the files that were added to the image were downloaded from the network; it modified the environment in a very limited way and provided a very general tool. The next example with a more specific purpose and local code will provide a more complete Dockerfile primer.

## 8.2 A Dockerfile primer

Dockerfiles are expressive and easy to understand due to their terse syntax that allows for comments. You can keep track of changes to Dockerfiles with any version-control system. Maintaining multiple versions of an image is as simple as maintaining multiple Dockerfiles. The Dockerfile build process itself uses extensive caching to aid rapid development and iteration. The builds are traceable and reproducible. They integrate easily with existing build systems and many continuous build and integration tools. With all these reasons to prefer Dockerfile builds to hand-made images, it's important to learn how to write them.

The examples in this section cover each of the Dockerfile instructions except for one. The `ONBUILD` instruction has a specific use case and is covered in the next section. Every instruction is covered here at an introductory level. For deep coverage of each instruction, the best reference will always be the Docker documentation

online at <https://docs.docker.com/reference/builder/>. Docker also provides a best practices section in its documentation: <http://docs.docker.com/reference/builder>.

### 8.2.1 *Metadata instructions*

The first example builds a base image and two other images with distinct versions of the mailer program you used in chapter 2. The purpose of the program is to listen for messages on a TCP port and then send those messages to their intended recipients. The first version of the mailer will listen for messages but only log those messages. The second will send the message as an [HTTP POST](#) to the defined URL.

One of the best reasons to use Dockerfile builds is that they simplify copying files from your computer into an image. But it's not always appropriate for certain files to be copied to images. The first thing to do when starting a new project is to define which files should never be copied into any images. You can do this in a file called `.dockerignore`. In this example you'll be creating three Dockerfiles, and none needs to be copied into the resulting images.

Use your favorite text editor to create a new file named `.dockerignore` and copy in the following lines:

```
.dockerignore
mailer-base.df
mailer-logging.df
mailer-live.df
```

Save and close the file when you've finished. This will prevent the `.dockerignore` file, or files named `mailer-base.df`, `mailer-log.df`, or `mailer-live.df`, from ever being copied into an image during a build. With that bit of accounting finished, you can begin working on the base image.

Building a base image helps create common layers. Each of the different versions of the mailer will be built on top of an image called `mailer-base`. When you create a Dockerfile, you need to keep in mind that each Dockerfile instruction will result in a new layer being created. Instructions should be combined whenever possible because the builder won't perform any optimization. Putting this in practice, create a new file named `mailer-base.df` and add the following lines:

```
FROM debian:wheezy
MAINTAINER Jeff Nickoloff "dia@allingeek.com"
RUN groupadd -r -g 2200 example && \
    useradd -rM -g example -u 2200 example
ENV APPROOT="/app" \
    APP="mailer.sh" \
    VERSION="0.6"
LABEL base.name="Mailer Archetype" \
    base.version="${VERSION}"
WORKDIR $APPROOT
ADD . $APPROOT
ENTRYPOINT ["/app/mailer.sh"]
EXPOSE 33333
# Do not set the default user in the base otherwise
```

← This file does not exist yet

```
# implementations will not be able to update the image
# USER example:example
```

Put it all together by running the `docker build` command from the directory where the mailer-base file is located. The `-f` flag tells the builder which filename to use as input:

```
docker build -t dockerinaction/mailer-base:0.6 -f mailer-base.df .
```

Five new instructions are introduced in this Dockerfile. The first new instruction is `ENV`. `ENV` sets environment variables for an image similar to the `--env` flag on `docker run` or `docker create`. In this case, a single `ENV` instruction is used to set three distinct environment variables. That could have been accomplished with three subsequent `ENV` instructions, but doing so would result in the creation of three layers. You can keep things looking well structured by using a backslash to escape the newline character (just like shell scripting):

```
Step 3 : ENV APPROOT "/app" APP "mailer.sh" VERSION "0.6"
---> Running in 05cb87a03b1b
---> 054f1747aa8d
Removing intermediate container 05cb87a03b1b
```

Environment variables declared in the Dockerfile are made available to the resulting image but can be used in other Dockerfile instructions as substitutions. In this Dockerfile the environment variable `VERSION` was used as a substitution in the next new instruction, `LABEL`:

```
Step 4 : LABEL base.name "Mailer Archetype" base.version "${VERSION}"
---> Running in 0473087065c4
---> ab76b163e1d7
Removing intermediate container 0473087065c4
```

The `LABEL` instruction is used to define key/value pairs that are recorded as additional metadata for an image or container. This mirrors the `--label` flag on `docker run` and `docker create`. Like the `ENV` instruction before it, multiple labels can and should be set with a single instruction. In this case, the value of the `VERSION` environment variable was substituted for the value of the `base.version` label. By using an environment variable in this way, the value of `VERSION` will be available to processes running inside a container as well as recorded to an appropriate label. This increases maintainability of the Dockerfile because it's more difficult to make inconsistent changes when the value is set in a single location.

The next two instructions are `WORKDIR` and `EXPOSE`. These are similar in operation to their corresponding flags on the `docker run` and `docker create` commands. An environment variable was substituted for the argument to the `WORKDIR` command:

```
Step 5 : WORKDIR $APPROOT
---> Running in 073583e0d554
---> 363129ccda97
Removing intermediate container 073583e0d554
```

The result of the `WORKDIR` instruction will be an image with the default working directory set to `/app`. Setting `WORKDIR` to a location that doesn't exist will create that location just like the command-line option. Last, the `EXPOSE` command creates a layer that opens TCP port 33333:

```
Step 7 : EXPOSE 33333
---> Running in a6c4f54b2907
---> 86e0b43f234a
Removing intermediate container a6c4f54b2907
```

The parts of this Dockerfile that you should recognize are the `FROM`, `MAINTAINER`, and `ENTRYPOINT` instructions. In brief, the `FROM` instruction sets the layer stack to start from the `debian:wheezy` image. Any new layers built will be placed on top of that image. The `MAINTAINER` instruction sets the `Author` value in the image metadata. The `ENTRYPOINT` instruction sets the executable to be run at container startup. Here, it's setting the instruction to `exec ./mailer.sh` and using the shell form of the instruction.

The `ENTRYPOINT` instruction has two forms: the shell form and an `exec` form. The shell form looks like a shell command with whitespace-delimited arguments. The `exec` form is a string array where the first value is the command to execute and the remaining values are arguments. A command specified using the shell form would be executed as an argument to the default shell. Specifically, the command used in this Dockerfile will be executed as `/bin/sh -c 'exec ./mailer.sh'` at runtime. Most importantly, if the shell form is used for `ENTRYPOINT`, then all other arguments provided by the `CMD` instruction or at runtime as extra arguments to `docker run` will be ignored. This makes the shell form of `ENTRYPOINT` less flexible.

You can see from the build output that the `ENV` and `LABEL` instructions each resulted in a single step and layer. But the output doesn't show that the environment variable values were substituted correctly. To verify that, you'll need to inspect the image:

```
docker inspect dockerinaction/mailer-base:0.6
```

**TIP** Remember, the `docker inspect` command can be used to view the metadata of either a container or an image. In this case, you used it to inspect an image.

The relevant lines are these:

```
"Env": [
  "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
  "APPROOT=/app",
  "APP=mailer.sh",
  "VERSION=0.6"
],
...
"Labels": {
  "base.name": "Mailer Archetype",
  "base.version": "0.6"
},
...
"WorkingDir": "/app"
```

The metadata makes it clear that the environment variable substitution works. You can use this form of substitution in the `ENV`, `ADD`, `COPY`, `WORKDIR`, `VOLUME`, `EXPOSE`, and `USER` instructions.

The last commented line is a metadata instruction `USER`. It sets the user and group for all further build steps and containers created from the image. In this case, setting it in a base image would prevent any downstream Dockerfiles from installing software. That would mean that those Dockerfiles would need to flip the default back and forth for permission. Doing so would create at least two additional layers. The better approach would be to set up the user and group accounts in the base image and let the implementations set the default user when they've finished building.

The most curious thing about this Dockerfile is that the `ENTRYPOINT` is set to a file that doesn't exist. The entrypoint will fail when you try to run a container from this base image. But now that the entrypoint is set in the base image, that's one less layer that will need to be duplicated for specific implementations of the mailer. The next two Dockerfiles build `mailer.sh` different implementations.

### 8.2.2 File system instructions

Images that include custom functionality will need to modify the file system. A Dockerfile defines three instructions that modify the file system: `COPY`, `VOLUME`, and `ADD`. The Dockerfile for the first implementation should be placed in a file named `mailer-logging.df`:

```
FROM dockerinaction/mailer-base:0.6
COPY ["/log-impl", "${APPROOT}"]
RUN chmod a+x ${APPROOT}/${APP} && \
    chown example:example /var/log
USER example:example
VOLUME ["/var/log"]
CMD ["/var/log/mailer.log"]
```

In this Dockerfile you used the image generated from `mailer-base` as the starting point. The three new instructions are `COPY`, `VOLUME`, and `CMD`. The `COPY` instruction will copy files from the file system where the image is being built into the build container. The `COPY` instruction takes at least two arguments. The last argument is the destination, and all other arguments are source files. This instruction has only one unexpected feature: any files copied will be copied with file ownership set to root. This is the case regardless of how the default user is set before the `COPY` instruction. It's better to delay any `RUN` instructions to change file ownership until all the files that you need to update have been copied into the image.

The `COPY` instruction will honor both shell style and exec style arguments, just like `ENTRYPOINT` and other instructions. But if any of the arguments contains whitespace, then you'll need to use the exec form.

**TIP** Using the exec (or string array) form wherever possible is the best practice. At a minimum, a Dockerfile should be consistent and avoid mixing styles.

This will make your Dockerfiles more readable and ensure that instructions behave as you'd expect without detailed understanding of their nuances.

The second new instruction is `VOLUME`. This behaves exactly as you'd expect if you understand what the `--volume` flag does on a call to `docker run` or `docker create`. Each value in the string array argument will be created as a new volume definition in the resulting layer. Defining volumes at image build time is more limiting than at run-time. You have no way to specify a bind-mount volume or read-only volume at image build time. This instruction will only create the defined location in the file system and then add a volume definition to the image metadata.

The last instruction in this Dockerfile is `CMD`. `CMD` is closely related to the `ENTRYPOINT` instruction. They both take either shell or exec forms and are both used to start a process within a container. But there are a few important differences.

The `CMD` command represents an argument list for the entrypoint. The default entrypoint for a container is `/bin/sh`. If no entrypoint is set for a container, then the values are passed, because the command will be wrapped by the default entrypoint. But if the entrypoint is set and is declared using the exec form, then you use `CMD` to set default arguments. This base for this Dockerfile defines the `ENTRYPOINT` as the `mailer` command. This Dockerfile injects an implementation of `mailer.sh` and defines a default argument. The argument used is the location that should be used for the log file.

Before building the image, you'll need to create the logging version of the `mailer` program. Create a directory at `./log-impl`. Inside that directory create a file named `mailer.sh` and copy the following script into the file:

```
#!/bin/sh
printf "Logging Mailer has started.\n"
while true
do
    MESSAGE=$(nc -l -p 33333)
    printf "[Message]: %s\n" "$MESSAGE" > $1
    sleep 1
done
```

The structural specifics of this script are unimportant. All you need to know is that this script will start a `mailer` daemon on port 33333 and write each message that it receives to the file specified in the first argument to the program. Use the following command to build the `mailer-logging` image from the directory containing `mailer-logging.df`:

```
docker build -t dockerinaction/mailer-logging -f mailer-logging.df .
```

The results of this image build should be anti-climactic. Go ahead and start up a named container from this new image:

```
docker run -d --name logging-mailer dockerinaction/mailer-logging
```

The logging mailer should now be built and running. Containers that link to this implementation will have their messages logged to `/var/log/mailer.log`. That's not

very interesting or useful in a real-world situation, but it might be handy for testing. An implementation that sends email would be better for operational monitoring.

The next implementation example uses the Simple Email Service provided by Amazon Web Services to send email. Get started with another Dockerfile. Name this file `mailer-live.df`:

```
FROM dockerinaction/mailer-base:0.6
ADD ["/live-impl", "${APPROOT}"]
RUN apt-get update && \
    apt-get install -y curl python && \
    curl "https://bootstrap.pypa.io/get-pip.py" -o "get-pip.py" && \
    python get-pip.py && \
    pip install awscli && \
    rm get-pip.py && \
    chmod a+x "${APPROOT}/${APP}"
RUN apt-get install -y netcat
USER example:example
CMD ["mailer@dockerinaction.com", "pager@dockerinaction.com"]
```

This Dockerfile includes one new instruction, `ADD`. The `ADD` instruction operates similarly to the `COPY` instruction with two important differences. The `ADD` instruction will

- Fetch remote source files if a URL is specified
- Extract the files of any source determined to be an archive file

The auto-extraction of archive files is the more useful of the two. Using the remote fetch feature of the `ADD` instruction isn't good practice. The reason is that although the feature is convenient, it provides no mechanism for cleaning up unused files and results in additional layers. Instead, you should use a chained `RUN` instruction like the third instruction of `mailer-live.df`.

The other instruction to note in this Dockerfile is the `CMD` instruction, where two arguments are passed. Here you're specifying the From and To fields on any emails that are sent. This differs from `mailer-logging.df`, which specifies only one argument.

Next, create a new subdirectory named `live-impl` under the location containing `mailer-live.df`. Add the following script to a file in that directory named `mailer.sh`:

```
#!/bin/sh
printf "Live Mailer has started.\n"
while true
do
    MESSAGE=$(nc -l -p 33333)
    aws ses send-email --from $1 \
        --destination {"ToAddresses":["${2}"]} \
        --message {"Subject":{"Data":{"Mailer Alert"}},\
            {"Body":{"Text":{"Data":{"$MESSAGE}}}}"}
    sleep 1
done
```

The key takeaway from this script is that, like the other mailer implementation, it will wait for connections on port 33333, take action on any received messages, and then

sleep for a moment before waiting for another message. This time, though, the script will send an email using the Simple Email Service command-line tool. Build and start a container with these two commands:

```
docker build -t dockerinaction/mailler-live -f mailler-live.df .
docker run -d --name live-mailer dockerinaction/mailler-live
```

If you link a watcher to these, you'll find that the logging mailer works as advertised. But the live mailer seems to be having difficulty connecting to the Simple Email Service to send the message. With a bit of investigation, you'll eventually realize that the container is misconfigured. The `aws` program requires certain environment variables to be set.

You'll need to set `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_DEFAULT_REGION` in order to get this example working. Discovering execution preconditions this way can be frustrating for users. Section 8.4.1 details an image design pattern that reduces this friction and helps adopters.

Before you get to design patterns, you need to learn about the final Dockerfile instruction. Remember, not all images contain applications. Some are built as platforms for downstream images. Those cases specifically benefit from the ability to inject downstream build-time behavior.

### 8.3 *Injecting downstream build-time behavior*

Only one Dockerfile instruction isn't covered in the primer. That instruction is `ONBUILD`. The `ONBUILD` instruction defines instructions to execute if the resulting image is used as a base for another build. For example, you could use `ONBUILD` instructions to compile a program that's provided by a downstream layer. The upstream Dockerfile copies the contents of the build directory into a known location and then compiles the code at that location. The upstream Dockerfile would use a set of instructions like this:

```
ONBUILD COPY [".", "/var/myapp"]
ONBUILD RUN go build /var/myapp
```

The instructions following `ONBUILD` instructions aren't executed when their containing Dockerfile is built. Instead, those instructions are recorded in the resulting image's metadata under `ContainerConfig.OnBuild`. The previous instructions would result in the following metadata inclusions:

```
...
"ContainerConfig": {
  ...
  "OnBuild": [
    "COPY [\".\", \"/var/myapp\"]",
    "RUN go build /var/myapp"
  ],
  ...
}
```

This metadata is carried forward until the resulting image is used as the base for another Dockerfile build. When a downstream Dockerfile uses the upstream image



(the one with the `ONBUILD` instructions) in a `FROM` instruction, those `ONBUILD` instructions are executed after the `FROM` instruction and before the next instruction in a Dockerfile.

Consider the following example to see exactly when `ONBUILD` steps are injected into a build. You need to create two Dockerfiles and execute two build commands to get the full experience. First, create an upstream Dockerfile that defines the `ONBUILD` instructions. Name the file `base.df` and add the following instructions:

```
FROM busybox:latest
WORKDIR /app
RUN touch /app/base-evidence
ONBUILD RUN ls -al /app
```

You can see that the image resulting from building `base.df` will add an empty file named `base-evidence` to the `/app` directory. The `ONBUILD` instruction will list the contents of the `/app` directory at build time, so it's important that you not run the build in quiet mode if you want to see exactly when changes are made to the file system.

The next file to create is the downstream Dockerfile. When this is built, you will be able to see exactly when the changes are made to the resulting image. Name the file `downstream.df` and include the following contents:

```
FROM dockerinaction/ch8_onbuild
RUN touch downstream-evidence
RUN ls -al .
```

This Dockerfile will use an image named `dockerinaction/ch8_onbuild` as a base, so that's the repository name you'll want to use when you build the base. Then you can see that the downstream build will create a second file and then list the contents of `/app` again.

With these two files in place, you're ready to start building. Run the following to create the upstream image:

```
docker build -t dockerinaction/ch8_onbuild -f base.df .
```

The output of the build should look like this:

```
Sending build context to Docker daemon 3.072 kB
Sending build context to Docker daemon
Step 0 : FROM busybox:latest
---> e72ac664f4f0
Step 1 : WORKDIR /app
---> Running in 4e9a3df4cf17
---> a552ff53eedc
Removing intermediate container 4e9a3df4cf17
Step 2 : RUN touch /app/base-evidence
---> Running in 352819bec296
---> bf38c3e396b2
Removing intermediate container 352819bec296
Step 3 : ONBUILD run ls -al /app
---> Running in fd70cef7e6ca
---> 6a53dbe28364
```

```
Removing intermediate container fd70cef7e6ca
Successfully built 6a53dbe28364
```

Then build the downstream image with this command:

```
docker build -t dockerinaction/ch8_onbuild_down -f downstream.df .
```

The results clearly show when the `ONBUILD` instruction (from the base image) is executed:

```
Sending build context to Docker daemon 3.072 kB
Sending build context to Docker daemon
Step 0 : FROM dockerinaction/ch8_onbuild
# Executing 1 build triggers
Trigger 0, RUN ls -al /app
Step 0 : RUN ls -al /app
----> Running in dd33ddea1fd4
total 8
drwxr-xr-x    2 root    root    4096 Apr 20 23:08 .
drwxr-xr-x   30 root    root    4096 Apr 20 23:08 ..
-rw-r--r--    1 root    root      0 Apr 20 23:08 base-evidence
----> 92782cc4e1f6
Removing intermediate container dd33ddea1fd4
Step 1 : RUN touch downstream-evidence
----> Running in 076b7e110b6a
----> 92cc1250b23c
Removing intermediate container 076b7e110b6a
Step 2 : RUN ls -al .
----> Running in b3fe2daac529
total 8
drwxr-xr-x    2 root    root    4096 Apr 20 23:08 .
drwxr-xr-x   31 root    root    4096 Apr 20 23:08 ..
-rw-r--r--    1 root    root      0 Apr 20 23:08 base-evidence
-rw-r--r--    1 root    root      0 Apr 20 23:08 downstream-evidence
----> 55202310df7b
Removing intermediate container b3fe2daac529
Successfully built 55202310df7b
```

You can see the builder registering the `ONBUILD` instruction with the container metadata in step 3 of the base build. Later, the output of the downstream image build shows which triggers (`ONBUILD` instructions) it has inherited from the base image. The builder discovers and processes the trigger immediately after step 0, the `FROM` instruction. The output then includes the result of the `RUN` instruction specified by the trigger. The output shows that only evidence of the base build is present. Later, when the builder moves on to instructions from the downstream Dockerfile, it lists the contents of the `/app` directory again. The evidence of both changes is listed.

That example is more illustrative than it is useful. You should consider browsing Docker Hub and looking for images tagged with `onbuild` suffixes to get an idea about how this is used in the wild. Here are a few of my favorites:

- [https://registry.hub.docker.com/\\_/python/](https://registry.hub.docker.com/_/python/)
- [https://registry.hub.docker.com/\\_/golang/](https://registry.hub.docker.com/_/golang/)
- [https://registry.hub.docker.com/\\_/node/](https://registry.hub.docker.com/_/node/)

## 8.4 Using startup scripts and multiprocess containers

Whatever tooling you choose to use, you'll always need to consider a few image design aspects. You'll need to ask yourself whether the software running in your container requires any startup assistance, supervision, monitoring, or coordination with other in-container processes. If so, then you'll need to include a startup script or initialization program with the image and install it as the entrypoint.

### 8.4.1 Environmental preconditions validation

Failure modes are difficult to communicate and can catch someone off guard if they occur at arbitrary times. If container configuration problems always cause failures at startup time for an image, users can be confident that a started container will keep running.

In software design, failing fast and precondition validation are best practices. It makes sense that the same should hold true for image design. The preconditions that should be evaluated are assumptions about the context.

Docker containers have no control over the environment where they're created. They do, however, have control of their own execution. An image author can solidify the user experience of their image by introducing environment and dependency validation prior to execution of the main task. A container user will be better informed about the requirements of an image if containers built from that image fail fast and display descriptive error messages.

For example, WordPress requires certain environment variables to be set or container links to be defined. Without that context, WordPress would be unable to connect to the database where the blog data is stored. It would make no sense to start WordPress in a container without access to the data it's supposed to serve. WordPress images use a script as the container entrypoint. That script validates that the container context is set in a way that's compatible with the contained version of WordPress. If any required condition is unmet (a link is undefined or a variable is unset), then the script will exit before starting WordPress, and the container will stop unexpectedly.

This type of startup script is generally use-case specific. If you're packaging a specific piece of software in an image, you'll need to write the script yourself. Your script should validate as much of the assumed context as possible. This should include the following:

- Presumed links (and aliases)
- Environment variables
- Network access
- Network port availability
- Root file system mount parameters (read-write or read-only)
- Volumes
- Current user

You can use whatever scripting or programming language you want to accomplish the task. In the spirit of building minimal images, it's a good idea to use a language or

scripting tool that's already included with the image. Most base images ship with a shell like `/bin/sh` or `/bin/bash`. Shell scripts are the most common for that reason.

Consider the following shell script that might accompany a program that depends on a web server. At container startup, this script enforces that either another container has been linked to the web alias and has exposed port 80 or the `WEB_HOST` environment variable has been defined:

```
#!/bin/bash
set -e

if [ -n "$WEB_PORT_80_TCP" ]; then
    if [ -z "$WEB_HOST" ]; then
        WEB_HOST='web'
    else
        echo >&2 '[WARN]: Linked container, "web" overridden by $WEB_HOST.'
        echo >&2 "===> Connecting to WEB_HOST ($WEB_HOST)"
    fi
fi

if [ -z "$WEB_HOST" ]; then
    echo >&2 '[ERROR]: specify a linked container, "web" or WEB_HOST environ-
    ment variable'
    exit 1
fi

exec "$@" # run the default command
```

If you're unfamiliar with shell scripting, this is an appropriate time to learn it. The topic is approachable, and there are several excellent resources for self-directed learning. This specific script uses a pattern where both an environment variable and a container link are tested. If the environment variable is set, the container link will be ignored. Finally, the default command is executed.

Images that use a startup script to validate configuration should fail fast if someone uses them incorrectly, but those same containers may fail later for other reasons. You can combine startup scripts with container restart policies to make reliable containers. But container restart policies are not perfect solutions. Containers that have failed and are waiting to be restarted aren't running. This means that an operator won't be able to execute another process within a container that's in the middle of a backoff window. The solution to this problem involves making sure the container never stops.

### 8.4.2 Initialization processes

UNIX-based computers usually start an initialization (`init`) process first. That `init` process is responsible for starting all the other system services, keeping them running, and shutting them down. It's often appropriate to use an `init`-style system to launch, manage, restart, and shut down container processes with a similar tool.

`Init` processes typically use a file or set of files to describe the ideal state of the initialized system. These files describe what programs to start, when to start them, and what actions to take when they stop. Using an `init` process is the best way to launch

multiple programs, clean up orphaned processes, monitor processes, and automatically restart any failed processes.

If you decide to adopt this pattern, you should use the init process as the entry-point of your application-oriented Docker container. Depending on the init program you use, you may need to prepare the environment beforehand with a startup script.

For example, the runit program doesn't pass environment variables to the programs it launches. If your service uses a startup script to validate the environment, it won't have access to the environment variables it needs. The best way to fix that problem might be to use a startup script for the runit program. That script might write the environment variables to some file so the startup script for your application can access them.

There are several open source init programs. Full-featured Linux distributions ship with heavyweight and full-featured init systems like SysV, Upstart, and systemd. Linux Docker images like Ubuntu, Debian, and CentOS typically have their init programs installed but nonfunctioning out of the box. These can be complex to configure and typically have hard dependencies on resources that require root access. For that reason, the community has tended toward the use of lighter-weight init programs.

Popular options include runit, BusyBox init, Supervisord, and DAEMON Tools. These all attempt to solve similar problems, but each has its benefits and costs. Using an init process is a best practice for application containers, but there's no perfect init program for every use case. When evaluating any init program for use in a container, consider these factors:

- Additional dependencies the program will bring into the image
- File sizes
- How the program passes signals to its child processes (or if it does)
- Required user access
- Monitoring and restart functionality (backoff-on-restart features are a bonus)
- Zombie process cleanup features

Whichever init program you decide on, make sure your image uses it to boost adopter confidence in containers created from your image. If the container needs to fail fast to communicate a configuration problem, make sure the init program won't hide that failure.

These are the tools at your disposal to build images that result in durable containers. Durability is not security, and although adopters of your durable images might trust that they will keep running as long as they can, they shouldn't trust your images until they've been hardened.

## 8.5 Building hardened application images

As an image author, it's difficult to anticipate all the scenarios where your work will be used. For that reason, harden the images you produce whenever possible. *Hardening an image* is the process of shaping it in a way that will reduce the attack surface inside any Docker containers based on it.

A general strategy for hardening an application image is to minimize the software included with it. Naturally, including fewer components reduces the number of potential vulnerabilities. Further, building minimal images keeps image download times short and helps adopters deploy and build containers more rapidly.

There are three things that you can do to harden an image beyond that general strategy. First, you can enforce that your images are built from a specific image. Second, you can make sure that regardless of how containers are built from your image, they will have a sensible default user. Last, you should eliminate a common path for root user escalation.

### 8.5.1 **Content addressable image identifiers**

The image identifiers discussed so far in this book are all designed to allow an author to update images in a transparent way to adopters. An image author chooses what image their work will be built on top of, but that layer of transparency makes it difficult to trust that the base hasn't changed since it was vetted for security problems. Since Docker 1.6, the image identifier has included an optional digest component.

An image ID that includes the digest component is called a content addressable image identifier (CAIID). This refers to a specific layer containing specific content, instead of simply referring to a particular and potentially changing layer.

Now image authors can enforce a build from a specific and unchanging starting point as long as that image is in a version 2 repository. Append an @ symbol followed by the digest in place of the standard tag position.

Use `docker pull` and observe the line labeled `digest` in the output to discover the digest of an image from a remote repository. Once you have the digest, you can use it as the identifier to `FROM` instructions in a Dockerfile. For example, consider the following, which uses a specific snapshot of `debian:jessie` as a base:

```
docker pull debian:jessie
# Output:
# ...
# Digest: sha256:d5e87cfcb730...

# Dockerfile:
FROM debian@sha256:d5e87cfcb730...
...
```

Regardless of when or how many times the Dockerfile is used to build an image, they will all use the content identified with that CAIID as their base. This is particularly useful for incorporating known updates to a base into your images and identifying the exact build of the software running on your computer.

Although this doesn't directly limit the attack surface of your images, using CAIIDs will prevent it from changing without your knowledge. The next two practices do address the attack surface of an image.

### 8.5.2 User permissions

The known container breakout tactics all rely on having system administrator privileges inside the container. Chapter 6 covers the tools used to harden containers. That chapter includes a deep dive into user management and a brief discussion of the UNIX Linux namespace. This section covers standard practices for establishing reasonable user defaults for images.

First, please understand that a Docker user can always override image defaults when they create a container. For that reason, there's no way for an image to prevent containers from running as the root user. The best things an image author can do are create other non-root users and establish a non-root default user and group.

Dockerfile includes a `USER` instruction that sets the user and group in the same way you would with the `docker run` or `docker create` command. The instruction itself was covered in the Dockerfile primer. This section is about considerations and best practices.

The best practice and general guidance is to drop privileges as soon as possible. You can do this with the `USER` instruction before any containers are ever created or with a startup script that's run at container boot time. The challenge for an image author is to determine the earliest appropriate time.

If you drop privileges too early, the active user may not have permission to complete the instructions in a Dockerfile. For example, this Dockerfile won't build correctly:

```
FROM busybox:latest
USER 1000:1000
RUN touch /bin/busybox
```

Building that Dockerfile would result in step 2 failing with a message like `touch: /bin/busybox: Permission denied`. File access is obviously impacted by user changes. In this case UID 1000 doesn't have permission to change the ownership of the file `/bin/busybox`. That file is currently owned by root. Reversing the second and third lines would fix the build.

The second timing consideration is the permissions and capabilities needed at runtime. If the image starts a process that requires administrative access at runtime, then it would make no sense to drop user access to a non-root user before that point. For example, any process that needs access to the system port range (1–1024) will need to be started by a user with administrative (at the very least `CAP_NET_ADMIN`) privileges. Consider what happens when you try to bind to port 80 as a non-root user with Netcat. Place the following Dockerfile in a file named `UserPermissionDenied.df`:

```
FROM busybox:latest
USER 1000:1000
ENTRYPOINT ["nc"]
CMD ["-l", "-p", "80", "0.0.0.0"]
```

Build the Dockerfile and run the resulting image in a container. In this case the user (UID 1000) will lack the required privileges, and the command will fail:

```
docker build \
  -t dockerinaction/ch8_perm_denied \
  -f UserPermissionDenied.df \
  .
docker run dockerinaction/ch8_perm_denied
# Output:
# nc: bind: Permission denied
```

In cases like these, you may see no benefit in changing the default user. Instead, any startup scripts that you build should take on the responsibility of dropping permissions as soon as possible. The last question is which user should be dropped into?

Docker currently lacks support for the Linux USR namespace. This means that UID 1000 in the container is UID 1000 on the host machine. All other aspects apart from the UID and GID are segregated, just as they would be between computers. For example, UID 1000 on your laptop might be your username, but the username associated with UID 1000 inside a BusyBox container is default.

Ultimately, until Docker adopts the USR namespace, it will be difficult for image authors to know which UID/GID is appropriate to use. The only thing we can be sure of is that it's inappropriate to use common or system-level UID/GIDs where doing so can be avoided. With that in mind, it's still burdensome to use raw UID/GID numbers. Doing so makes scripts and Dockerfiles less readable. For that reason, it's typical for image authors to include `RUN` instructions that create users and groups used by the image. The following is the second instruction in a Postgres Dockerfile:

```
# add our user and group first to make sure their IDs get assigned
# consistently, regardless of whatever dependencies get added
RUN groupadd -r postgres && useradd -r -g postgres postgres
```

This instruction simply creates a postgres user and group with automatically assigned UID and GID. The instruction is placed early in the Dockerfile so that it will always be cached between rebuilds, and the IDs remain consistent regardless of other users that are added as part of the build. This user and group could then be used in a `USER` instruction. That would make for a safer default. But Postgres containers require elevated privileges during startup. Instead, this particular image uses a `su` or `sudo`-like program called `gosu` to start the Postgres process as the postgres user. Doing so makes sure that the process runs without administrative access in the container.

User permissions are one of the more nuanced aspects of building Docker images. The general rule you should follow is that if the image you're building is designed to run some specific application code, then the default execution should drop user permissions as soon as possible.

A properly functioning system should be reasonably secure with reasonable defaults in place. Remember, though, an application or arbitrary code is rarely perfect and could be intentionally malicious. For that reason, you should take additional steps to reduce the attack surface of your images.



### 8.5.3 SUID and SGID permissions

The last hardening action to cover is the mitigation of SUID or SGID permissions. The well-known file system permissions (read, write, execute) are only a portion of the set defined by Linux. In addition to those, two are of particular interest: SUID and SGID.

These two are similar in nature. An executable file with the SUID bit set will always execute as its owner. Consider a program like `/usr/bin/passwd`, which is owned by the root user and has the SUID permission set. If a non-root user like bob executes `passwd`, he will execute that program as the root user. You can see this in action by building an image from the following Dockerfile:

```
FROM ubuntu:latest
# Set the SUID bit on whoami
RUN chmod u+s /usr/bin/whoami
# Create an example user and set it as the default
RUN adduser --system --no-create-home --disabled-password --disabled-login \
    --shell /bin/sh example
USER example
# Set the default to compare the container user and
# the effective user for whoami
CMD printf "Container running as:          %s\n" $(id -u -n) && \
    printf "Effectively running whoami as: %s\n" $(whoami)
```

Once you've created the Dockerfile, you need to build an image and run the default command in a container:

```
docker build -t dockerinaction/ch8_whoami .
docker run dockerinaction/ch8_whoami
```

Doing so prints results like these to the terminal:

```
Container running as:          example
Effectively running whoami as: root
```

The output of the default command shows that even though you've executed the `whoami` command as the example user, it's running from the context of the root user. The SGID works similarly. The difference is that the execution will be from the owning group's context, not the owning user.

Running a quick search on your base image will give you an idea of how many and which files have these permissions:

```
docker run --rm debian:wheezy find / -perm +6000 -type f
```

It will display a list like this:

```
/sbin/unix_chkpwd
/bin/ping6
/bin/su
/bin/ping
/bin/umount
/bin/mount
/usr/bin/chage
```

```

/usr/bin/passwd
/usr/bin/gpasswd
/usr/bin/chfn
/usr/bin/newgrp
/usr/bin/wall
/usr/bin/expiry
/usr/bin/chsh
/usr/lib/pt_chown

```

This command will find all of the SGID files:

```
docker run --rm debian:wheezy find / -perm +2000 -type f
```

The resulting list is much shorter:

```

/sbin/unix_chkpwd
/usr/bin/chage
/usr/bin/wall
/usr/bin/expiry

```

Each of the listed files in this particular image has the SUID or SGID permission, and a bug in any of them could be used to compromise the root account inside a container. The good news is that files that have either of these permissions set are typically useful during image builds but rarely required for application use cases. If your image is going to be running software that's arbitrary or externally sourced, it's a best practice to mitigate this risk of escalation.

Fix this problem and either delete all these files or unset their SUID and SGID permissions. Taking either action would reduce the image's attack surface. The following Dockerfile instruction will unset the SUID and GUID permissions on all files currently in the image:

```

RUN for i in $(find / -type f \( -perm +6000 -o -perm +2000 \)); \
do chmod ug-s $i; done

```

Hardening images will help users build hardened containers. Although it's true that no hardening measures will protect users from intentionally building weak containers, those measures will help the more unsuspecting and most common type of user.

## 8.6 **Summary**

Most Docker images are built automatically from Dockerfiles. This chapter covers the build automation provided by Docker and Dockerfile best practices. Before moving on, make sure that you've understood these key points:

- Docker provides an automated image builder that reads instructions from Dockerfiles.
- Each Dockerfile instruction results in the creation of a single image layer.
- Merge instructions whenever possible to minimize the size of images and layer count.

- Dockerfiles include instructions to set image metadata like the default user, exposed ports, default command, and entrypoint.
- Other Dockerfile instructions copy files from the local file system or remote location into the produced images.
- Downstream builds inherit build triggers that are set with `ONBUILD` instructions in an upstream Dockerfile.
- Startup scripts should be used to validate the execution context of a container before launching the primary application.
- A valid execution context should have appropriate environment variables set, network dependencies available, and an appropriate user configuration.
- Init programs can be used to launch multiple processes, monitor those processes, reap orphaned child processes, and forward signals to child processes.
- Images should be hardened by building from content addressable image identifiers, creating a non-root default user, and disabling or removing any executable with SUID or SGID permissions.



# *Public and private software distribution*

---

## ***This chapter covers***

- Choosing a project distribution method
- Using hosted infrastructure
- Running and using your own registry
- Understanding manual image distribution workflows
- Distributing image sources

You have your own images from software you've written, customized, or just pulled from the internet. But what good is an image if nobody can install it? Docker is different from other container management tools because it provides image distribution features.

There are several ways to get your images out to the world. This chapter explores those distribution paradigms and provides a framework for making or choosing one or more for your own projects.

Hosted registries offer both public and private repositories with automated build tools. Running a private registry lets you hide and customize your image distribution infrastructure. Heavier customization of a distribution workflow might

This chapter will teach you how to select and use a method for distributing your images to the world or just at work.

The most difficult thing about choosing a distribution method is choosing the appropriate method for your situation. To help with this problem, each method presented in this chapter is examined on the same set of selection criteria.

### 9.1.1 A distribution spectrum

The methods included in the spectrum range from hosted registries like Docker Hub to totally custom distribution architectures or source-distribution methods. Some of these subjects will be covered in more detail than others. Particular focus is placed on private registries because they provide the most balance between the two concerns.

The diagram illustrates the distribution spectrum of container registries, ranging from simple/restrictive to complicated/flexible. It lists five types of registries with their examples and characteristics.

Distribution spectrum	
Simple/restrictive	Complicated/flexible
Hosted registry with public repositories	Image source distributions
Examples: - Docker Hub - Quay.io	Example: - Include a Dockerfile with your project source
Hosted registry with private repositories	Custom image distribution infrastructure
Examples: - Docker Hub - Quay.io - Tutum.co - qcr.io	Examples: - SFTP - HTTP downloads - Configuration management tools
Private registries	
Using registry software: - Local private network - Corporate network - Private cloud infrastructure	

**Figure 9.1 The image distribution spectrum**

### 9.1.2 Selection criteria

Choosing the best distribution method for your needs may seem daunting with this many options. In situations like these you should take the time to understand the options, identify criteria for making a selection, and avoid the urge to make a quick decision or settle.

The following identified selection criteria are based on differences across the spectrum and on common business concerns. When making a decision, consider how important each of these is in your situation:

- Cost
- Visibility
- Transport speed or bandwidth overhead
- Longevity control
- Availability control
- Access control
- Artifact integrity
- Artifact confidentiality
- Requisite expertise

How each distribution method stacks up against these criteria is covered in the relevant sections over the rest of this chapter.

#### **COST**

Cost is the most obvious criterion, and the distribution spectrum ranges in cost from free to very expensive and “it’s complicated.” Lower cost is generally better, but cost is typically the most flexible criterion. For example, most people will trade cost for artifact confidentiality if the situation calls for it.

#### **VISIBILITY**

Visibility is the next most obvious criterion for a distribution method. Secret projects or internal tools should be difficult if not impossible for unauthorized people to discover. In another case, public works or open source projects should be as visible as possible to promote adoption.

#### **TRANSPORTATION**

Transportation speed and bandwidth overhead are the next most flexible criteria. File sizes and image installation speed will vary between methods that leverage image layers, concurrent downloads, and prebuilt images and those that use flat image files or rely on deployment time image builds. High transportation speeds or low installation latency is critical for systems that use just-in-time deployment to service synchronous requests. The opposite is true in development environments or asynchronous processing systems.

#### **LONGEVITY**

Longevity control is a business concern more than a technical concern. Hosted distribution methods are subject to other people’s or companies’ business concerns. An executive faced with the option of using a hosted registry might ask, “What happens if

they go out of business or pivot away from repository hosting?” The question reduces to, “Will the business needs of the third party change before ours?” If this is a concern for you, then longevity control is important. Docker makes it simple to switch between methods, and other criteria like requisite expertise or cost may actually trump this concern. For those reasons, longevity control is another of the more flexible criteria.

#### **AVAILABILITY**

Availability control is the ability to control the resolution of availability issues with your repositories. Hosted solutions provide no availability control. Businesses typically provide some service-level agreement on availability if you’re a paying customer, but there’s nothing you can do to directly resolve an issue. On the other end of the spectrum, private registries or custom solutions put both the control and responsibility in your hands.

#### **ACCESS CONTROL**

Access control protects your images from modification or access by unauthorized parties. There are varying degrees of access control. Some systems provide only access control of modifications to a specific repository, whereas others provide course control of entire registries. Still other systems may include pay walls or digital rights management controls. Projects typically have specific access control needs dictated by the product or business. This makes access control requirements one of the least flexible and most important to consider.

#### **INTEGRITY**

Artifact integrity and confidentiality both fall in the less-flexible and more-technical end of the spectrum. Artifact integrity is trustworthiness and consistency of your files and images. Violations of integrity may include man-in-the-middle attacks, where an attacker intercepts your image downloads and replaces the content with their own. They might also include malicious or hacked registries that lie about the payloads they return.

#### **CONFIDENTIALITY**

Artifact confidentiality is a common requirement for companies developing trade secrets or proprietary software. For example, if you use Docker to distribute cryptographic material, then confidentiality will be a major concern. Artifact integrity and confidentiality features vary across the spectrum. Overall, the out-of-the-box distribution security features won’t provide the tightest confidentiality or integrity. If that’s one of your needs, an information security professional will need to implement and review a solution.

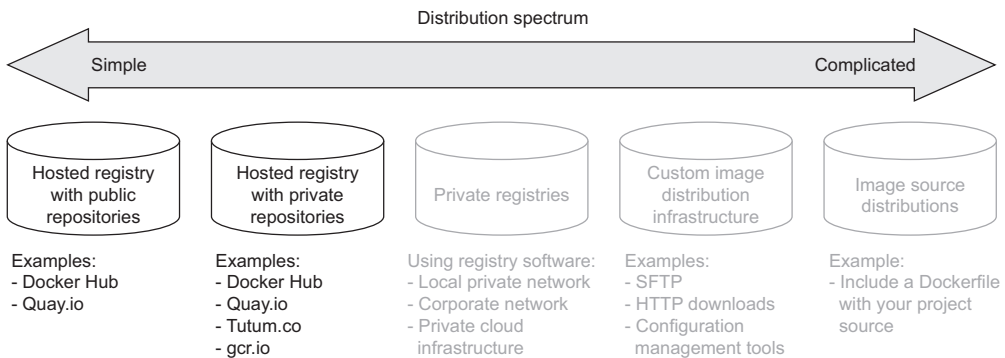
The last thing to consider when choosing a distribution method is the level of expertise required. Using hosted methods can be very simple and requires little more than a mechanical understanding of the tools. Building custom image or image source distribution pipelines requires expertise with a suite of related technologies. If you don’t have that expertise or don’t have access to someone who does, using more complicated solutions will be a challenge. In that case, you may be able to reconcile the gap at additional cost.

With this strong set of selection criteria, you can begin learning about and evaluating different distribution methods. The best place to start is on the far left of the spectrum with hosted registries.

## 9.2 *Publishing with hosted registries*

As a reminder, Docker registries are services that make repositories accessible to Docker pull commands. A registry hosts repositories. The simplest way to distribute your images is by using hosted registries.

A hosted registry is a Docker registry service that's owned and operated by a third-party vendor. Docker Hub, Quay.io, Tutum.co, and Google Container Registry are all examples of hosted registry providers. By default, Docker publishes to Docker Hub. Docker Hub and most other hosted registries provide both public and private registries, as shown in figure 9.2.



**Figure 9.2** The simplest side of the distribution spectrum and the topic of this section

The example images used in this book are distributed with public repositories hosted on Docker Hub and Quay.io. By the end of this section you'll understand how to publish your own images using hosted registries and how hosted registries measure up to the selection criteria.

### 9.2.1 *Publishing with public repositories: Hello World via Docker Hub*

The simplest way to get started with public repositories on hosted registries is to push a repository that you own to Docker Hub. To do so, all you need is a Docker Hub account and an image to publish. If you haven't done so already, sign up for a Docker Hub account now.

Once you have your account, you need to create an image to publish. Create a new Dockerfile named `HelloWorld.df` and add the following instructions:

```
FROM busybox:latest
CMD echo Hello World
```

← **From HelloWorld.df**



Chapter 8 covers Dockerfile instructions. As a reminder, the `FROM` instruction tells the Docker image builder which existing image to start the new image from. The `CMD` instruction sets the default command for the new image. Containers created from this image will display “Hello World” and exit. Build your new image with the following command:

```
docker build \
  -t <insert Docker Hub username>/hello-dockerfile \
  -f HelloWorld.df \
  .
```

← Insert your username

Be sure to substitute your Docker Hub username in that command. Authorization to access and modify repositories is based on the username portion of the repository name on Docker Hub. If you create a repository with a username other than your own, you won’t be able to publish it.

Publishing images on Docker Hub with the `docker` command-line tool requires that you establish an authenticated session with that client. You can do that with the `login` command:

```
docker login
```

This command will prompt you for your username, email address, and password. Each of those can be passed to the command as arguments using the `--username`, `--email`, and `--password` flags. When you log in, the `docker` client maintains a map of your credentials for the different registries that you authenticate with in a file. It will specifically store your username and an authentication token, not your password.

You will be able to push your repository to the hosted registry once you’ve logged in. Use the `docker push` command to do so:

```
docker push <insert Docker Hub username>/hello-dockerfile
```

← Insert your username

Running that command should create output like the following:

```
The push refers to a repository
[dockerinaction/hello-dockerfile] (len: 1)
7f6d4eb1f937: Image already exists
8c2e06607696: Image successfully pushed
6ce2e90b0bc7: Image successfully pushed
cf2616975b4a: Image successfully pushed
Digest:
sha256:ef18de4b0ddf9ebd1cf5805fae1743181cbf3642f942cae8de7c5d4e375b1f20
```

The command output includes upload statuses and the resulting repository content digest. The push operation will create the repository on the remote registry, upload each of the new layers, and then create the appropriate tags.

Your public repository will be available to the world as soon as the push operation is completed. Verify that this is the case by searching for your username and your new repository. For example, use the following command to find the example owned by the `dockerinaction` user:

```
docker search dockerinaction/hello-dockerfile
```

Replace the `dockerinaction` username with your own to find your new repository on Docker Hub. You can also log in to the Docker Hub website and view your repositories to find and modify your new repository.

Having distributed your first image with Docker Hub, you should consider how this method measures up to the selection criteria; see table 9.1.

**Table 9.1 Performance of public hosted repositories**

Criteria	Rating	Notes
Cost	Best	Public repositories on hosted registries are almost always free. That price is difficult to beat. These are especially helpful when you're getting started with Docker or publishing open source software.
Visibility	Best	Hosted registries are well-known hubs for software distribution. A public repository on a hosted registry is an obvious distribution choice if you want your project to be well known and visible to the public.
Transport speed/size	Better	Hosted registries like Docker Hub are layer-aware and will work with Docker clients to transfer only the layers that the client doesn't already have. Further, pull operations that require multiple repositories to be transferred will perform those transfers in parallel. For those reasons, distributing an image from a hosted repository is fast, and the payloads are minimal.
Availability control	Worst	You have no availability control over hosted registries.
Longevity control	Good	You have no longevity control over hosted registries. But registries will all conform to the Docker registry API, and migrating from one host to another should be a low-cost exercise.
Access control	Better	Public repositories are open to the public for read access. Write access is still controlled by whatever mechanisms the host has put in place. Write access to public repositories on Docker Hub is controlled two ways. First, repositories owned by an individual may be written to only by that individual account. Second, repositories owned by organizations may be written to by any user who is part of that organization.
Artifact integrity	Best	The most recent version of the Docker registry API provides content-addressable images. These let you request an image with a specific cryptographic signature. The Docker client will validate the integrity of the returned image by recalculating the signature and comparing it to the one requested. Older versions of Docker that are unaware of the V2 registry API don't support this feature. In those cases and for other cases where signatures are unknown, a high degree of trust is put into the authorization and at-rest security features provided by the host.
Secrecy	Worst	Hosted registries and public repositories are never appropriate for storing and distributing cleartext secrets or sensitive code. Anyone can access these secrets.
Requisite experience	Best	Using public repositories on hosted registries requires only that you be minimally familiar with Docker and capable of setting up an account through a website. This solution is within reach for any Docker user.

Public repositories on hosted registries are the best choice for owners of open source projects or people who are just getting started with Docker. People should still be skeptical of software that they download and run from the internet, and so public repositories that don't expose their sources can be difficult for some users to trust. Hosted (trusted) builds solve this problem to a certain extent.

### 9.2.2 Publishing public projects with automated builds

A few different hosted registries offer automated builds. Automated builds are images that are built by the registry provider using image sources that you've made available. Image consumers have a higher degree of trust for these builds because the registry owner is building the images from source that can be reviewed.

Distributing your work with automated builds requires two components: a hosted image repository and a hosted Git repository where your image sources are published. Git is a popular distributed version-control system. A Git repository stores the change history for your project. Although distributed version-control systems like Git don't have architectural centralization, a few popular companies provide Git repository hosting. Docker Hub integrates with both Github.com and Bitbucket.org for automated builds.

Both of these hosted Git repository tools provide something called webhooks. In this context, a *webhook* is a way for your Git repository to notify your image repository that a change has been made to the source. When Docker Hub receives a webhook for your Git repository, it will start an automated build for your Docker Hub repository. This automation is shown in figure 9.3.

The automated build process pulls the sources for your project including a Dockerfile from your registered Git repository. The Docker Hub build fleet will use a `docker build` command to build a new image from those sources, tag it in accordance with the repository configuration, and then push it into your Docker Hub repository.

#### CREATING A DOCKER HUB AUTOMATED BUILD

The following example will walk you through the steps required to set up your own Docker Hub repository as an automated build. This example uses Git. Whole books

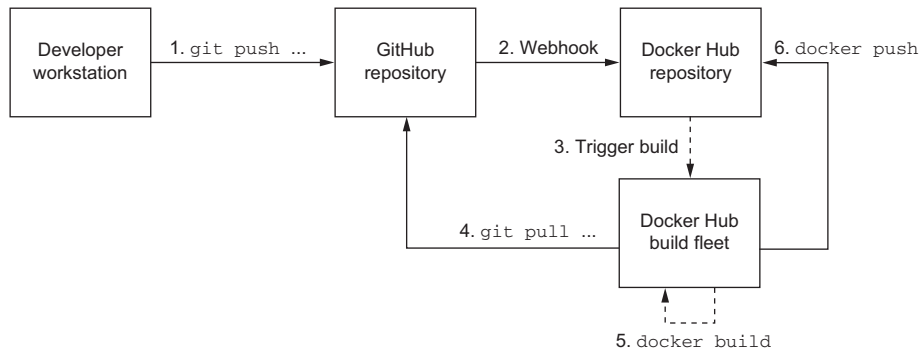


Figure 9.3 The Docker Hub automated build workflow

have been written about Git, and so we can't cover it in detail here. Git ships with several operating systems today, but if it isn't installed on your computer or you need general help, check the website at <https://git-scm.com>. For the purposes of this example, you need accounts on both Docker Hub and Github.com.

Log in to your Github.com account and create a new repository. Name it hello-docker and make sure that the repository is public. Don't initialize the repository with a license or a .gitignore file. Once the repository has been created on GitHub, go back to your terminal and create a new working directory named hello-docker.

Create a new file named Dockerfile and include the following lines:

```
FROM busybox:latest
CMD echo Hello World
```

This Dockerfile will produce a simple Hello World image. The first thing you need to do to get this built into a new repository at Docker Hub is add it to your Git repository. The following Git commands will create a local repository, add the Dockerfile to the repository, commit the change, and push your changes to your repository on GitHub. Be sure to replace `<your username>` with your GitHub username:

```
git init
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
git remote add origin \
    https://github.com/<your username>/hello-docker.git
```

Use your full name →

← Use your email address

← Use your GitHub username

Don't add or commit your files to your repository yet. Before you push your work to GitHub, you should create a new automated build and repository on Docker Hub. You must perform this step through the website at <https://hub.docker.com>. Once you log in, click the Create button in the header and select Automated Build from the dropdown menu. The website will walk you through setting up the automated build.

The steps include authenticating with GitHub and granting Docker Hub limited access to your account. That access is required so that Docker Hub can find your repositories and register appropriate webhooks for you. Next, you'll be prompted for the GitHub repository that you'd like to use for the automated build. Select the hello-docker repository that you just created. Once you complete the creation wizard, you should be directed to your repository page. Now go back to your terminal to add and push your work to your GitHub repository.

```
git add Dockerfile
git commit -m "first commit"
git push -u origin master
```

When you execute the last command, you may be prompted for your Github.com login credentials. After you present them, your work will be uploaded to GitHub, and you can view your Dockerfile online. Now that your image source is available online at GitHub, a build should have been triggered for your Docker Hub repository. Head back to the repository page and click the Build Details tab. You should see a build

listed that was triggered from your latest push to the GitHub repository. Once that is complete, head back to the command line to search for your repository:

```
docker search <your username>/hello-docker
```

← Insert your Docker Hub username

Automated builds are preferred by image consumers and simplify image maintenance for most cases. There will be times when you don't want to make your source available to the general public. The good news is that most hosted repository providers offer private repositories.

### 9.2.3 Private hosted repositories

Private repositories are similar to public repositories from an operational and product perspective. Most registry providers offer both options, and any differences in provisioning through their websites will be minimal. Because the Docker registry API makes no distinction between the two types of repositories, registry providers that offer both generally require you to provision private registries through their website, app, or API.

The tools for working with private repositories are identical to those for working with public repositories, with one exception. Before you can use `docker pull` or `docker run` to install an image from a private repository, you need to have authenticated with the registry where the repository is hosted. To do so, you will use the `docker login` command just as you would if you were using `docker push` to upload an image.

The following commands prompt you to authenticate with the registries provided by Docker Hub, quay.io, and tutum.co. After creating accounts and authenticating, you'll have full access to your public and private repositories on all three registries. The `login` subcommand takes an optional server argument:

```
docker login
# Username: dockerinaction
# Password:
# Email: book@dockerinaction.com
# WARNING: login credentials saved in /Users/xxx/.dockercfg.
# Login Succeeded

docker login tutum.co
# Username: dockerinaction
# Password:
# Email: book@dockerinaction.com
# WARNING: login credentials saved in /Users/xxx/.dockercfg.
# Login Succeeded

docker login quay.io
# Username: dockerinaction
# Password:
# Email: book@dockerinaction.com
# WARNING: login credentials saved in /Users/xxx/.dockercfg.
# Login Succeeded
```

Before you decide that private hosted repositories are the distribution solution for you, consider how they might fulfill your selection criteria; see table 9.2

**Table 9.2 Performance of private hosted repositories**

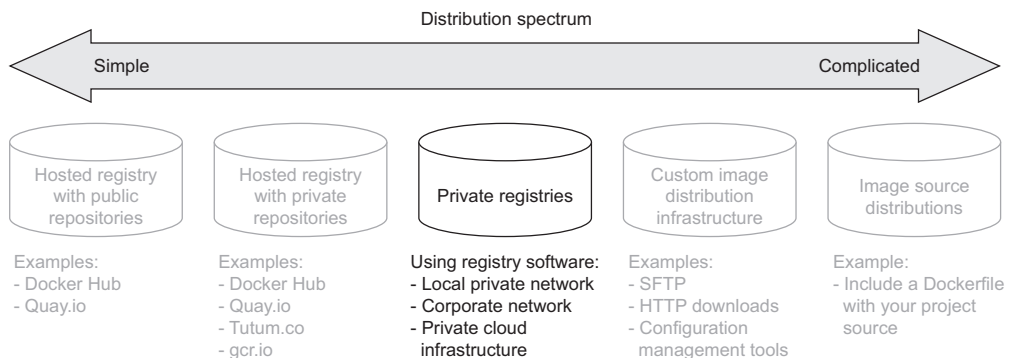
Criteria	Rating	Notes
Cost	Best	The cost of private repositories typically scales with the number of repositories that you need. Plans usually range from a few dollars per month for 5 repositories up to around \$50 for 50 repositories. Price pressure of storage and monthly virtual server hosting is a driving factor here. Users or organizations that require more than 50 repositories may find it more appropriate to run their own private registry.
Visibility	Best	Private repositories are by definition private. These are typically excluded from indexes and should require authentication before a registry acknowledges the repository's existence. Private repositories are poor candidates for publicizing availability of some software or distributing open source images. Instead they're great tools for small private projects or organizations that don't want to incur the overhead associated with running their own registry.
Transport speed/size	Better	Any hosted registry like Docker Hub will minimize the bandwidth used to transfer an image and enable clients to transfer an image's layers in parallel. Ignoring potential latency introduced by transferring files over the internet, hosted registries should always perform well against other non-registry solutions.
Availability control	Worst	No hosted registry provides any availability control. Unlike public repositories, however, using private repositories will make you a paying customer. Paying customers may have stronger SLA guarantees or access to support personnel.
Longevity control	Good	You have no longevity control over hosted registries. But registries will all conform to the Docker registry API, and migrating from one host to another should be a low-cost exercise.
Access control	Better	Both read and write access to private repositories is restricted to users with authorization.
Artifact integrity	Best	It's reasonable to expect all hosted registries to support the V2 registry API and content-addressable images.
Secrecy	Worst	Despite the privacy provided by these repositories, these are never suitable for storing clear-text secrets or trade-secret code. Although the registries require user authentication and authorization to requested resources, there are several potential problems with these mechanisms. The provider may use weak credential storage, have weak or lost certificates, or leave your artifacts unencrypted at rest. Finally, your secret material should not be accessible to employees of the registry provider.
Requisite experience	Best	Just like public repositories, using private repositories on hosted registries requires only that you be minimally familiar with Docker and capable of setting up an account through a website. This solution is within reach for any Docker user.

Individuals and small teams will find the most utility in private hosted repositories. Their low cost and basic authorization features are friendly to low-budget projects or private projects with minimal security requirements. Large companies or projects that need a higher degree of secrecy and have a suitable budget may find their needs better met by running their own private registry.

### 9.3 Introducing private registries

When you have a hard requirement on availability control, longevity control, or secrecy, then running a private registry may be your best option. In doing so, you gain control without sacrificing interoperability with Docker pull and push mechanisms or adding to the learning curve for your environment. People can interact with a private registry exactly as they would with a hosted registry.

The Docker registry software (called Distribution) is open source software and distributed under the Apache 2 license. The availability of this software and permissive license keep the engineering cost of running your own registry low. It's available through Docker Hub and is simple to use for non-production purposes. Figure 9.4 illustrates that private registries fall in the middle of the distribution spectrum.



**Figure 9.4** Private registries in the image distribution spectrum

Running a private registry is a great distribution method if you have special infrastructure use cases like the following:

- Regional image caches
- Team-specific image distribution for locality or visibility
- Environment or deployment stage-specific image pools
- Corporate processes for approving images
- Longevity control of external images

Before deciding that this is the best choice for you, consider the costs detailed in the selection criteria, shown in table 9.3.

**Table 9.3 Performance of private registries**

Criteria	Rating	Notes
Cost	Good	At a minimum, a private registry adds to hardware overhead (virtual or otherwise), support expense, and risk of failure. But the community has already invested the bulk of the engineering effort required to deploy a private registry by building the open source software. Cost will scale on different dimensions than hosted registries. Whereas the cost of hosted repositories scales with raw repository count, the cost of private registries scales with transaction rates and storage usage. If you build a system with high transaction rates, you'll need to scale up the number of registry hosts so that you can handle the demand. Likewise, registries that serve some number of small images will have lower storage costs than those serving the same number of large images.
Visibility	Good	Private registries are as visible as you decide to make them. But even a registry that you own and open up to the world will be less visible than advertised popular registries like Docker Hub.
Transport speed/size	Best	Latency between any client and any registry will vary based on the distance between those two nodes on the network, the speed of the network, and the congestion on the registry. Private registries may be faster or slower than hosted registries due to variance in any of those variables. But private registries will appeal most to people and organizations that are doing so for internal infrastructure. Eliminating a dependency on the internet or inter-datacenter networking will have a proportional improvement on latency. Because this solution is using a Docker registry, it will share the same parallelism gains as hosted registry solutions.
Availability control	Best	You have full control over availability as the registry owner.
Longevity control	Best	You have full control over solution longevity as the registry owner.
Access control	Good	The registry software doesn't include any authentication or authorization features out of the box. But implementing those features can be achieved with a minimal engineering exercise.
Artifact integrity	Best	Version 2 of the registry API supports content-addressable images, and the open source software supports a pluggable storage back end. For additional integrity protections, you can force the use of TLS over the network and use back-end storage with encryption at rest.
Secrecy	Good	Private registries are the first solution on the spectrum appropriate for storage of trade secrets or secret material. You control the authentication and authorization mechanisms. You also control the network and in-transit security mechanisms. Most importantly, you control the at-rest storage. It's in your power to ensure that the system is configured in such a way that your secrets stay secret.
Requisite experience	Good	Getting started and running a local registry requires only basic Docker experience. But running and maintaining a highly available production private registry requires experience with several technologies. The specific set depends on what features you want to take advantage of. Generally, you'll want to be familiar with NGINX to build a proxy, LDAP or Kerberos to provide authentication, and Redis for caching.



The biggest trade-off going from hosted registries to private registries is gaining flexibility and control while requiring greater depth and breadth of engineering experience to build and maintain the solution. The remainder of this section covers what you need to implement all but the most complicated registry deployment designs and highlights opportunities for customization in your environment.

### 9.3.1 Using the registry image

Whatever your reasons for doing so, getting started with the Docker registry software is easy. The Distribution software is available on Docker Hub in a repository named `registry`. Starting a local registry in a container can be done with a single command:

```
docker run -d -p 5000:5000 \
  -v "$(pwd)"/data:/tmp/registry-dev \
  --restart=always --name local-registry registry:2
```

The image that's distributed through Docker Hub is configured for insecure access from the machine running a client's Docker daemon. When you've started the registry, you can use it like any other registry with `docker pull`, `run`, `tag`, and `push` commands. In this case, the registry location is `localhost:5000`. The architecture of your system should now match that described in figure 9.5.

Companies that want tight version control on their external image dependencies will pull images from external sources like Docker Hub and copy them into their own

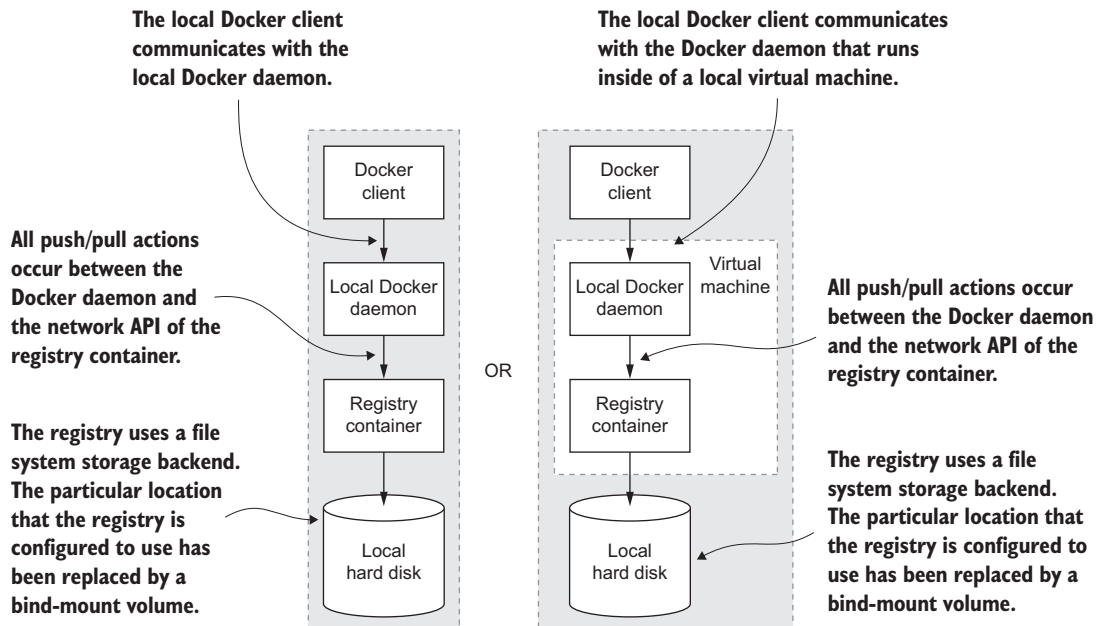


Figure 9.5 Interactions between the `docker` client, daemon, local registry container, and local storage

registry. To get an idea of what it's like working with your registry, consider a workflow for copying images from Docker Hub into your new registry:

```

Pull demo image from Docker Hub → docker pull dockerinaction/ch9_registry_bound

docker images -f "label=dia_exercise=ch9_registry_bound" ← Verify image is discoverable
                                                             with label filter

docker tag dockerinaction/ch9_registry_bound \
    localhost:5000/dockerinaction/ch9_registry_bound ← Push demo image into
docker push localhost:5000/dockerinaction/ch9_registry_bound your private registry

```

In running these four commands, you copy an example repository from Docker Hub into your local repository. If you execute these commands from the same location as where you started the registry, you'll find that the newly created data subdirectory contains new registry data.

### 9.3.2 *Consuming images from your registry*

The tight integration you get with the Docker ecosystem can make it feel like you're working with software that's already installed on your computer. When internet latency has been eliminated, such as when you're working with a local registry, it can feel even less like you're working with distributed components. For that reason, the exercise of pushing data into a local repository isn't very exciting on its own.

The next set of commands should impress on you that you're working with a real registry. These commands will remove the example repositories from the local cache for your Docker daemon, demonstrate that they're gone, and then reinstall them from your personal registry:

```

docker rmi \
    dockerinaction/ch9_registry_bound \
    localhost:5000/dockerinaction/ch9_registry_bound ← Remove tagged
                                                         reference

Pull from registry again → docker images -f "label=dia_exercise=ch9_registry_bound"

docker pull localhost:5000/dockerinaction/ch9_registry_bound

docker images -f "label=dia_exercise=ch9_registry_bound" ← Demonstrate that
                                                         image is back

docker rm -vf local-registry ← Clean up local registry

```

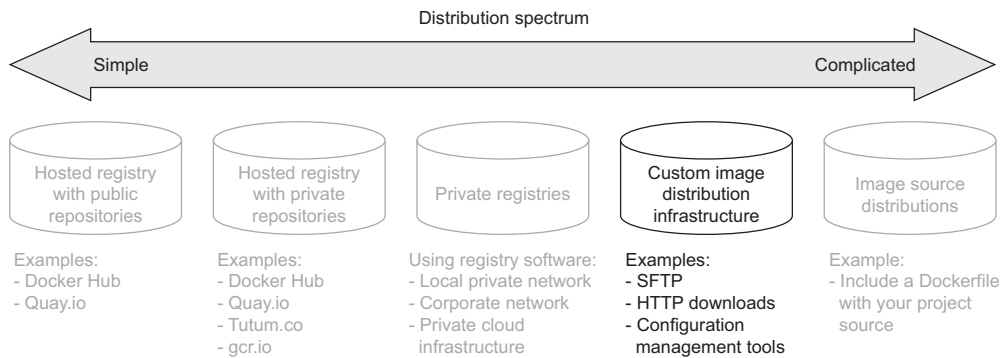
You can work with this registry locally as much as you want, but the insecure default configuration will prevent remote Docker clients from using your registry (unless they specifically allow insecure access). This is one of the few issues that you'll need to address before deploying a registry in a production environment. Chapter 10 covers the registry software in depth.

This is the most flexible distribution method that involves Docker registries. If you need greater control over the transport, storage, and artifact management, you should consider working directly with images in a manual distribution system.

## 9.4 Manual image publishing and distribution

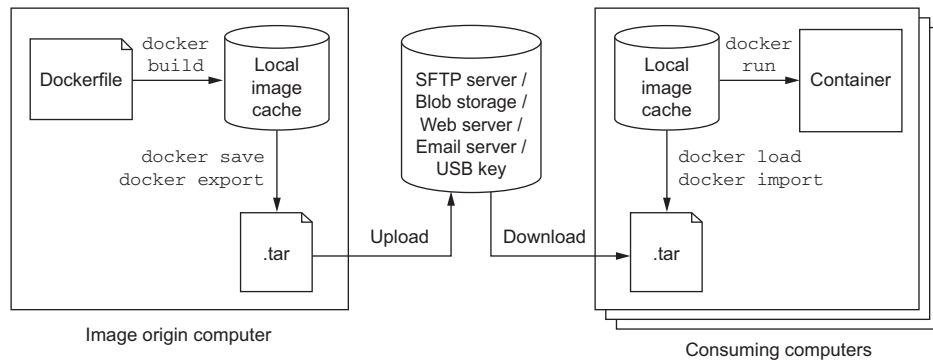
Images are files, and you can distribute them as you would any other file. It's common to see software available for download on websites, File Transport Protocol (FTP) servers, corporate storage networks, or via peer-to-peer networks. You could use any of these distribution channels for image distribution. You can even use email or USB keys in cases where you know your image recipients.

When you work with images as files, you use Docker only to manage local images and create files. All other concerns are left for you to implement. That void of functionality makes manual image publishing and distribution the second-most flexible but complicated distribution method. This section covers custom image distribution infrastructure, shown on the spectrum in figure 9.6.



**Figure 9.6** Docker image distribution over custom infrastructure

We've already covered all the methods for working with images as files. Chapter 3 covers loading images into Docker and saving images to your hard drive. Chapter 7 covers exporting and importing full file systems as flattened images. These techniques are the foundation for building distribution workflows like the one shown in figure 9.7.



**Figure 9.7** A typical manual distribution workflow with producer, transport, and consumers