

1

Discovering Docker

This chapter covers

- What Docker is
- The uses of Docker and how it can save you time and money
- The differences between containers and images
- Docker's layering feature
- Building and running a to-do application using Docker

Docker is a platform that allows you to “build, ship, and run any app, anywhere.” It has come a long way in an incredibly short time and is now considered a standard way of solving one of the costliest aspects of software: deployment.

Before Docker came along, the development pipeline typically involved combinations of various technologies for managing the movement of software, such as virtual machines, configuration management tools, package management systems, and complex webs of library dependencies. All these tools needed to be managed and maintained by specialist engineers, and most had their own unique ways of being configured.

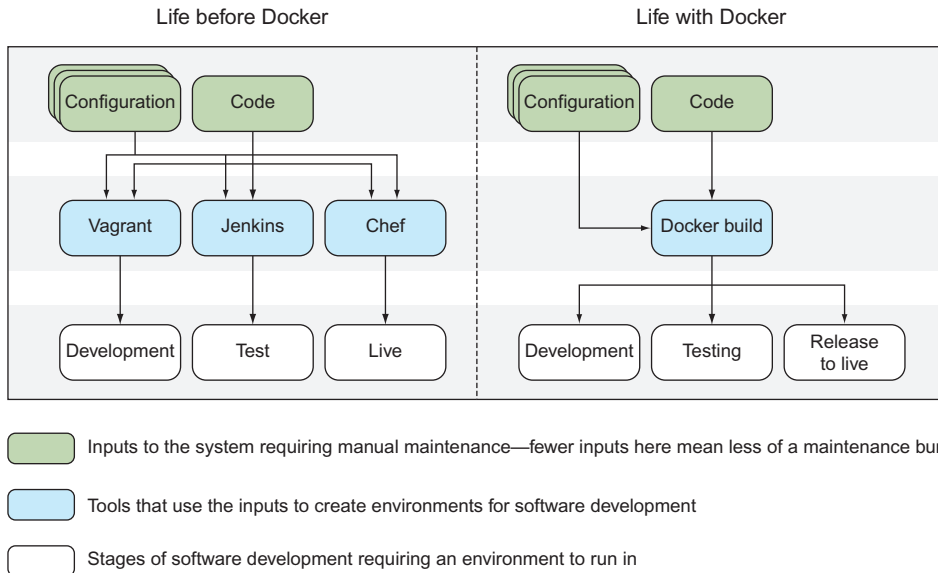


Figure 1.1 How Docker has eased the tool maintenance burden

Docker has changed all of this, allowing different engineers involved in this process to effectively speak one language, making working together a breeze. Everything goes through a common pipeline to a single output that can be used on any target—there’s no need to continue maintaining a bewildering array of tool configurations, as shown in figure 1.1.

At the same time, there’s no need to throw away your existing software stack if it works for you—you can package it up in a Docker container as-is, for others to consume. As a bonus, you can see how these containers were built, so if you need to dig into the details, you can.

This book is aimed at intermediate developers with some knowledge of Docker. If you’re OK with the basics, feel free to skip to the later chapters. The goal of this book is to expose the real-world challenges that Docker brings and to show how they can be overcome. But first we’re going to provide a quick refresher on Docker itself. If you want a more thorough treatment of Docker’s basics, take a look at *Docker in Action* by Jeff Nickoloff (Manning, 2016).

In chapter 2 you’ll be introduced to Docker’s architecture more deeply, with the aid of some techniques that demonstrate its power. In this chapter you’re going to learn what Docker is, see why it’s important, and start using it.

1.1 *The what and why of Docker*

Before we get our hands dirty, we’ll discuss Docker a little so that you understand its context, where the name “Docker” came from, and why we’re using it at all!

1.1.1 What is Docker?

To get an understanding of what Docker is, it's easier to start with a metaphor than a technical explanation, and the Docker metaphor is a powerful one. A *docker* was a laborer who moved commercial goods into and out of ships when they docked at ports. There were boxes and items of differing sizes and shapes, and experienced dockers were prized for their ability to fit goods into ships by hand in cost-effective ways (see figure 1.2). Hiring people to move stuff around wasn't cheap, but there was no alternative.

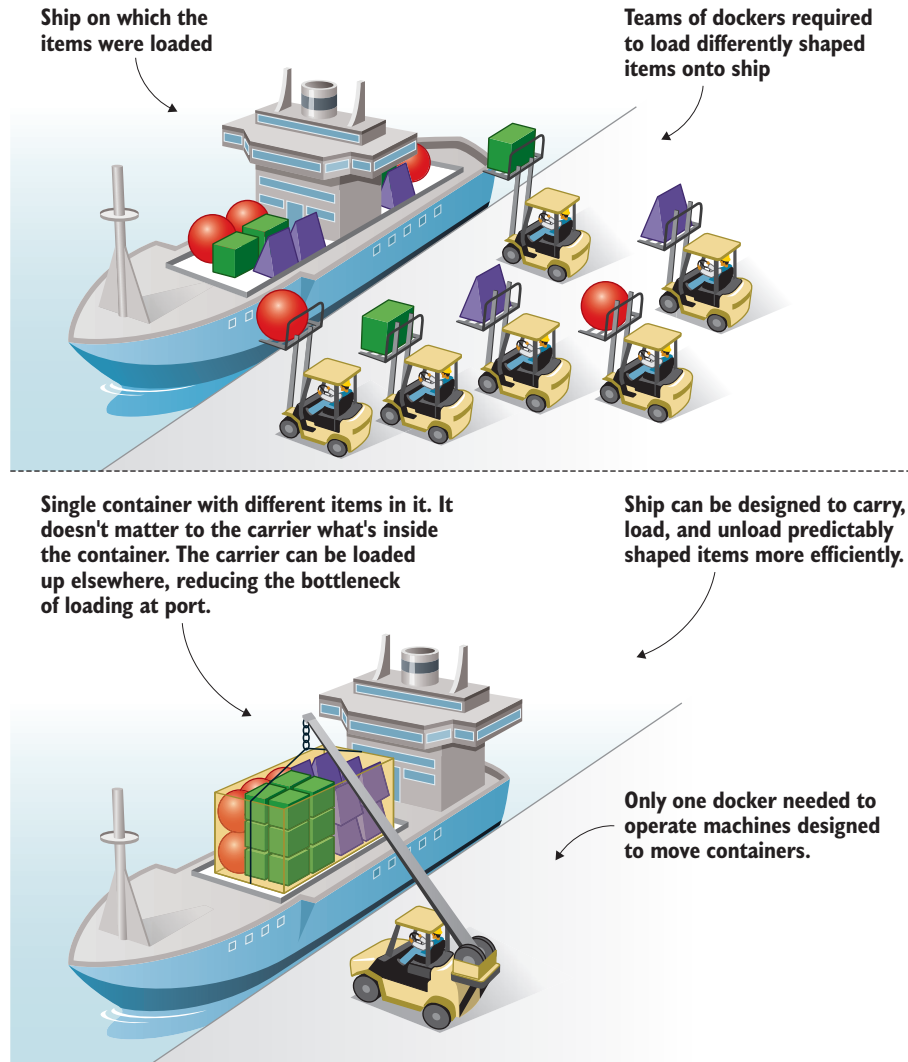


Figure 1.2 Shipping before and after standardized containers

This should sound familiar to anyone working in software. Much time and intellectual energy is spent getting metaphorically odd-shaped software into differently-sized metaphorical ships full of other odd-shaped software, so they can be sold to users or businesses elsewhere.

Figure 1.3 shows how time and money can be saved with the Docker concept. Before Docker, deploying software to different environments required significant effort. Even if you weren't hand-running scripts to provision software on different machines (and plenty of people do exactly that), you'd still have to wrestle with configuration management tools that manage state on what are increasingly fast-moving environments starved of resources. Even when these efforts were encapsulated in VMs, a lot of time was spent managing the deployment of these VMs, waiting for them to boot, and managing the overhead of resource use they created.

With Docker, the configuration effort is separated from the resource management, and the deployment effort is trivial: run `docker run`, and the environment's image is pulled down and ready to run, consuming fewer resources and contained so that it doesn't interfere with other environments.

You don't need to worry about whether your container is going to be shipped to a Red Hat machine, an Ubuntu machine, or a CentOS VM image; as long as it has Docker on it, it'll be good to go.

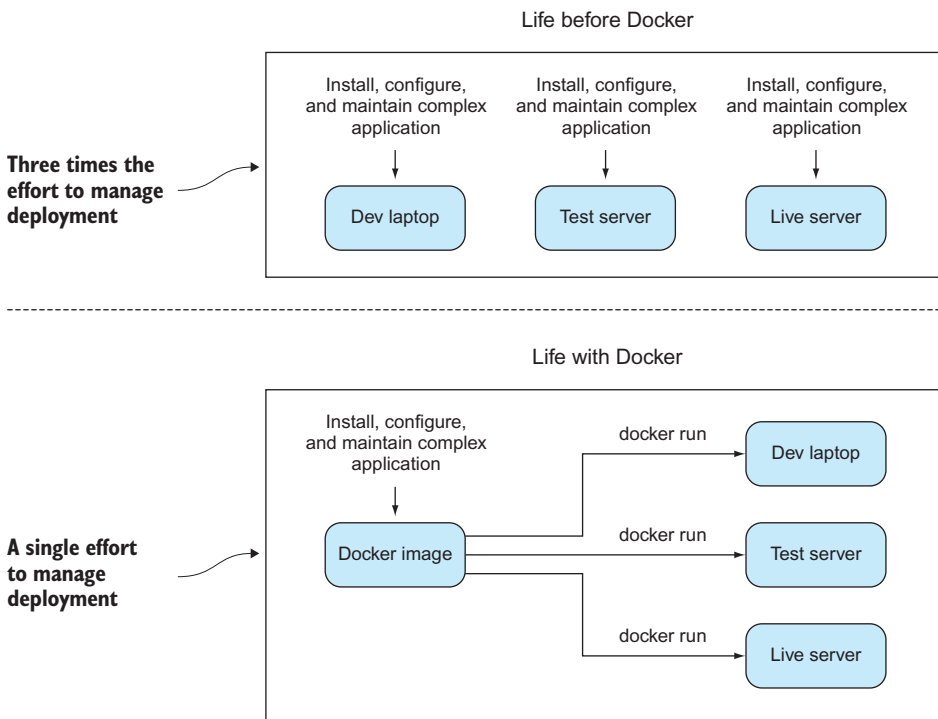


Figure 1.3 Software delivery before and after Docker

1.1.2 What is Docker good for?

Some crucial practical questions arise: why would you use Docker, and for what? The short answer to the “why” is that for a modicum of effort, Docker can save your business a lot of money quickly. Some of these ways (and by no means all) are discussed in the following subsections. We’ve seen all of these benefits firsthand in real working contexts.

REPLACING VIRTUAL MACHINES (VMs)

Docker can be used to replace VMs in many situations. If you only care about the application, not the operating system, Docker can replace the VM, and you can leave worrying about the OS to someone else. Not only is Docker quicker than a VM to spin up, it’s more lightweight to move around, and due to its layered filesystem, you can more easily and quickly share changes with others. It’s also firmly rooted in the command line and is eminently scriptable.

PROTOTYPING SOFTWARE

If you want to quickly experiment with software without either disrupting your existing setup or going through the hassle of provisioning a VM, Docker can give you a sandbox environment in milliseconds. The liberating effect of this is difficult to grasp until you experience it for yourself.

PACKAGING SOFTWARE

Because a Docker image has effectively no dependencies for a Linux user, it’s a great way to package software. You can build your image and be sure that it can run on any modern Linux machine—think Java, without the need for a JVM.

ENABLING A MICROSERVICES ARCHITECTURE

Docker facilitates the decomposition of a complex system to a series of composable parts, which allows you to reason about your services in a more discrete way. This can allow you to restructure your software to make its parts more manageable and pluggable without affecting the whole.

MODELING NETWORKS

Because you can spin up hundreds (even thousands) of isolated containers on one machine, modeling a network is a breeze. This can be great for testing real-world scenarios without breaking the bank.

ENABLING FULL-STACK PRODUCTIVITY WHEN OFFLINE

Because you can bundle all the parts of your system into Docker containers, you can orchestrate these to run on your laptop and work on the move, even when offline.

REDUCING DEBUGGING OVERHEAD

Complex negotiations between different teams about software delivered is a commonplace within the industry. We’ve personally experienced countless discussions about broken libraries; problematic dependencies; updates applied wrongly, or in the wrong order, or even not performed at all; unreproducible bugs; and so on. It’s likely you have too. Docker allows you to state clearly (even in script form) the steps for debugging a problem on a system with known properties, making bug and environment reproduction a much simpler affair, and one normally separated from the host environment provided.

DOCUMENTING SOFTWARE DEPENDENCIES AND TOUCHPOINTS

By building your images in a structured way, ready to be moved to different environments, Docker forces you to document your software dependencies explicitly from a base starting point. Even if you decide not to use Docker everywhere, this documentation can help you install your software in other places.

ENABLING CONTINUOUS DELIVERY

Continuous delivery (CD) is a paradigm for software delivery based on a pipeline that rebuilds the system on every change and then delivers to production (or “live”) through an automated (or partly automated) process.

Because you can control the build environment’s state more exactly, Docker builds are more reproducible and replicable than traditional software building methods. This makes implementing CD much easier. Standard CD techniques, such as Blue/Green deployment (where “live” and “last” deployments are maintained on live) and Phoenix deployment (where whole systems are rebuilt on each release), are made trivial by implementing a reproducible Docker-centric build process.

You now know a bit about how Docker can help you. Before we dive into a real example, let’s go over a couple of core concepts.

1.1.3 Key concepts

In this section we’re going to cover some key Docker concepts, which are illustrated in figure 1.4.

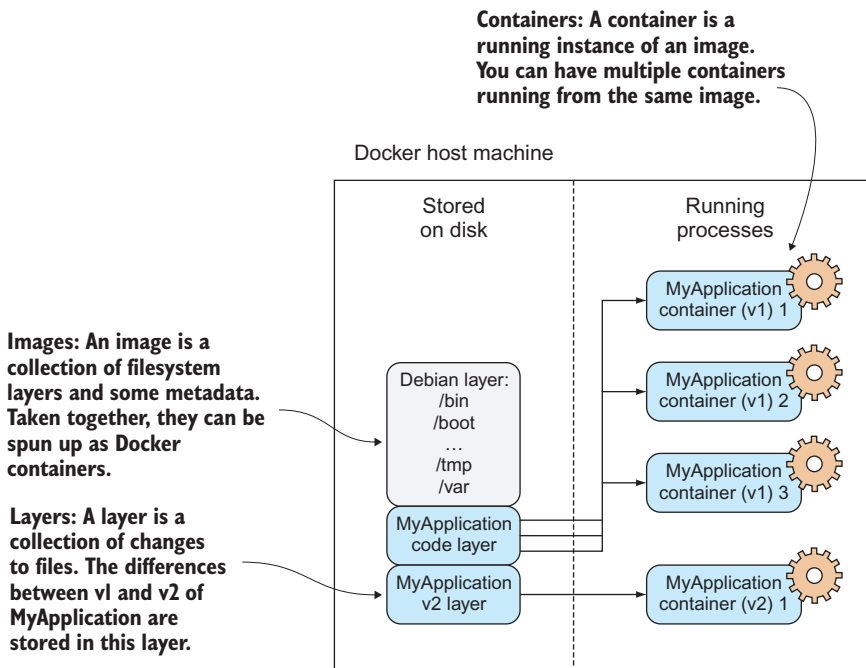


Figure 1.4 Core Docker concepts

It's most useful to get the concepts of images, containers, and layers clear in your mind before you start running Docker commands. In short, *containers* are running systems defined by *images*. These images are made up of one or more *layers* (or sets of diffs) plus some metadata for Docker.

Let's look at some of the core Docker commands. We'll turn images into containers, change them, and add layers to new images that we'll commit. Don't worry if all of this sounds confusing. By the end of the chapter it will all be much clearer.

KEY DOCKER COMMANDS

Docker's central function is to build, ship, and run software in any location that has Docker. To the end user, Docker is a command-line program that they run. Like git (or any source control tool), this program has subcommands that perform different operations. The principal Docker subcommands you'll use on your host are listed in table 1.1.

Table 1.1 Docker subcommands

Command	Purpose
<code>docker build</code>	Build a Docker image
<code>docker run</code>	Run a Docker image as a container
<code>docker commit</code>	Commit a Docker container as an image
<code>docker tag</code>	Tag a Docker image

IMAGES AND CONTAINERS

If you're unfamiliar with Docker, this may be the first time you've come across the words "container" and "image" in this context. They're probably the most important concepts in Docker, so it's worth spending a bit of time to make sure the difference is clear. In figure 1.5 you'll see an illustration of these concepts, with three containers started up from one base image.

One way to look at images and containers is to see them as analogous to programs and processes. In the same way a process can be seen as an "application being executed," a Docker container can be viewed as a Docker image in execution.

If you're familiar with object-oriented principles, another way to look at images and containers is to view images as classes and containers as objects. In the same way that objects are concrete instances of classes, containers are instances of images. You can create multiple containers from a single image, and they are all isolated from one another in the same way objects are. Whatever you change in the object, it won't affect the class definition—they're fundamentally different things.

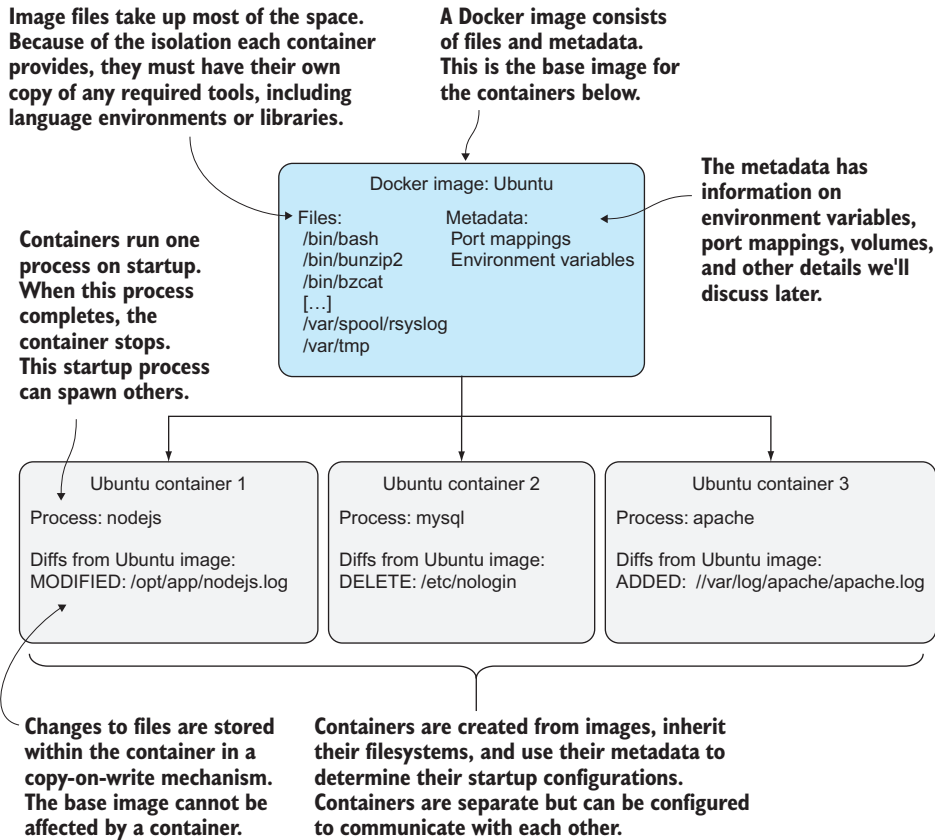


Figure 1.5 Docker images and containers

1.2 Building a Docker application

We're going to get our hands dirty now by building a simple to-do application (todoapp) image with Docker. In the process, you'll see some key Docker features like Dockerfiles, image re-use, port exposure, and build automation. Here's what you'll learn in the next 10 minutes:

- How to create a Docker image using a Dockerfile
- How to tag a Docker image for easy reference
- How to run your new Docker image

A to-do app is one that helps you keep track of things you want to get done. The app we build will store and display short strings of information that can be marked as done, presented in a simple web interface. Figure 1.6 shows what we'll achieve by doing this.

The details of the application are unimportant. We're going to demonstrate that from the single short Dockerfile we're about to give you, you can reliably build, run,

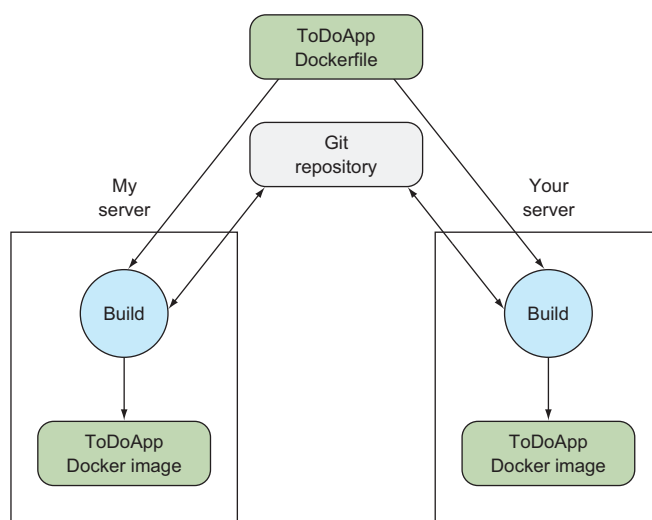


Figure 1.6 Building a Docker application

stop, and start an application in the same way on both your host and ours without needing to worry about application installations or dependencies. This is a key part of what Docker offers—reliably reproduced and easily managed and shared development environments. This means no more complex or ambiguous installation instructions to follow and potentially get lost in.

NOTE This to-do application will be used a few times throughout the book, and it's quite a useful one to play with and demonstrate, so it's worth familiarizing yourself with it.

1.2.1 Ways to create a new Docker image

There are four standard ways to create Docker images. Table 1.2 itemizes these methods.

Table 1.2 Options for creating Docker images

Method	Description	See technique
Docker commands / “By hand”	Fire up a container with <code>docker run</code> and input the commands to create your image on the command line. Create a new image with <code>docker commit</code> .	See technique 15
Dockerfile	Build from a known base image, and specify the build with a limited set of simple commands.	Discussed shortly
Dockerfile and configuration management (CM) tool	Same as Dockerfile, but you hand over control of the build to a more sophisticated CM tool.	See technique 55
Scratch image and import a set of files	From an empty image, import a TAR file with the required files.	See technique 11

The first “by hand” option is fine if you’re doing proofs of concept to see whether your installation process works. At the same time, you should keep notes about the steps you’re taking so that you can return to the same point if you need to.

At some point you’re going to want to define the steps for creating your image. This is the Dockerfile option (and the one we’ll use here).

For more complex builds, you may want to go for the third option, particularly when the Dockerfile features aren’t sophisticated enough for your image’s needs.

The final option builds from a null image by overlaying the set of files required to run the image. This is useful if you want to import a set of self-contained files created elsewhere, but this method is rarely seen in mainstream use.

We’ll look at the Dockerfile method now; the other methods will be covered later in the book.

1.2.2 Writing a Dockerfile

A Dockerfile is a text file with a series of commands in it. Listing 1.1 is the Dockerfile we’ll use for this example. Create a new folder, move into it, and create a file called “Dockerfile” with these contents.

Listing 1.1 The `todoapp` Dockerfile

```
FROM node
LABEL maintainer ian.miell@gmail.com
RUN git clone -q https://github.com/docker-in-practice/todo.git
WORKDIR todo
RUN npm install > /dev/null
EXPOSE 8000
CMD ["npm", "start"]
```

Defines the base image (points to `FROM node`)

Declares the maintainer (points to `LABEL maintainer ian.miell@gmail.com`)

Clones the todoapp code (points to `RUN git clone -q https://github.com/docker-in-practice/todo.git`)

Moves to the new cloned directory (points to `WORKDIR todo`)

Runs the node package manager's install command (npm) (points to `RUN npm install > /dev/null`)

Specifies which command will be run on startup (points to `CMD ["npm", "start"]`)

Specifies that containers from the built image should listen on this port (points to `EXPOSE 8000`)

You begin the Dockerfile by defining the base image with the `FROM` command. This example uses a Node.js image so you have access to the Node.js binaries. The official Node.js image is called `node`.

Next, you declare the maintainer with the `LABEL` command. In this case, we’re using one of our email addresses, but you can replace this with your own reference because it’s your Dockerfile now. This line isn’t required to make a working Docker image, but it’s good practice to include it. At this point, the build has inherited the state of the node container, and you’re ready to work on top of it.

Next, you clone the `todoapp` code with a `RUN` command. This uses the specified command to retrieve the code for the application, running `git` within the container. Git is installed inside the base node image in this case, but you can’t take this kind of thing for granted.

Now you move to the new cloned directory with a `WORKDIR` command. Not only does this change directories within the build context, but the last `WORKDIR` command determines which directory you're in by default when you start up your container from your built image.

Next, you run the node package manager's install command (`npm`). This will set up the dependencies for your application. You aren't interested in the output in this example, so you redirect it to `/dev/null`.

Because port 8000 is used by the application, you use the `EXPOSE` command to tell Docker that containers from the built image should listen on this port.

Finally, you use the `CMD` command to tell Docker which command will be run when the container is started up.

This simple example illustrates several key features of Docker and Dockerfiles. A Dockerfile is a simple sequence of a limited set of commands run in strict order. It affects the files and metadata of the resulting image. Here the `RUN` command affects the filesystem by checking out and installing applications, and the `EXPOSE`, `CMD`, and `WORKDIR` commands affect the metadata of the image.

1.2.3 Building a Docker image

You've defined your Dockerfile's build steps. Now you're going to build the Docker image from it by typing the command in figure 1.7. The output will look similar to this:

The docker command
↓
`docker build .`
↑
The docker build subcommand

Figure 1.7 Docker build command

Each command results in a new image being created, and the image ID is output.

Docker uploads the files and directories under the path supplied to the docker build command.

Each build step is numbered sequentially from 1 and is output with the command.

Sending build context to Docker daemon 2.048kB

Step 1/7 : FROM node

---> 2ca756a6578b

Step 2/7 : LABEL maintainer ian.miell@gmail.com

---> Running in bf73f87c88d6

---> 5383857304fc

Removing intermediate container bf73f87c88d6

Step 3/7 : RUN git clone -q https://github.com/docker-in-practice/todo.git

---> Running in 761baf524cc1

---> 4350cb1c977c

Removing intermediate container 761baf524cc1

Step 4/7 : WORKDIR todo

---> a1b24710f458

Removing intermediate container 0f8cd22f8e83

Step 5/7 : RUN npm install > /dev/null

---> Running in 92a8f9ba530a

npm info it worked if it ends with ok

[...]

npm info ok

To save space, each intermediate container is removed before continuing.

Debug of the build is output here (and edited out of this listing).

```

---> 6ee4d7bba544
Removing intermediate container 92a8f9ba530a
Step 6/7 : EXPOSE 8000
---> Running in 8e33c1ded161
---> 3ea44544f13c
Removing intermediate container 8e33c1ded161
Step 7/7 : CMD npm start
---> Running in ccc076ee38fe
---> 66c76cea05bb
Removing intermediate container ccc076ee38fe
Successfully built 66c76cea05bb

```

← The final image ID for this build, ready to tag

You now have a Docker image with an image ID (“66c76cea05bb” in the preceding example, but your ID will be different). It can be cumbersome to keep referring to this ID, so you can tag it for easier reference, as shown in figure 1.8.

Type the preceding command, replacing the 66c76cea05bb with whatever image ID was generated for you.

You can now build your own copy of a Docker image from a Dockerfile, reproducing an environment defined by someone else!

```

The docker command
    |
    v
docker tag 66c76cea05bb todoapp
    ^           ^           ^
The docker tag The image Tag
subcommand    ID to tag  name
               for image

```

Figure 1.8 Docker tag subcommand

1.2.4 Running a Docker container

You’ve built and tagged your Docker image. Now you can run it as a container:

Listing 1.2 `docker run` output for `todoapp`

```

$ docker run -i -t -p 8000:8000 --name example1 todoapp
npm install
npm info it worked if it ends with ok
npm info using npm@2.14.4
npm info using node@v4.1.1
npm info prestart todomvc-swarm@0.0.1

```

← The docker run subcommand starts the container, -p maps the container’s port 8000 to the port 8000 on the host machine, --name gives the container a unique name, and the last argument is the image.

```

> todomvc-swarm@0.0.1 prestart /todo
> make all

```

← The output of the container’s starting process is sent to the terminal.

```

npm install
npm info it worked if it ends with ok
npm info using npm@2.14.4
npm info using node@v4.1.1
npm WARN package.json todomvc-swarm@0.0.1 No repository field.
npm WARN package.json todomvc-swarm@0.0.1 license should be a valid SPDX
➡ license expression
npm info preinstall todomvc-swarm@0.0.1
npm info package.json statics@0.1.0 license should be a valid SPDX license
➡ expression

```

```

npm info package.json react-tools@0.11.2 No license field.
npm info package.json react@0.11.2 No license field.
npm info package.json node-
  jsx@0.11.0 license should be a valid SPDX license expression
npm info package.json ws@0.4.32 No license field.
npm info build /todo
npm info linkStuff todomvc-swarm@0.0.1
npm info install todomvc-swarm@0.0.1
npm info postinstall todomvc-swarm@0.0.1
npm info prepublish todomvc-swarm@0.0.1
npm info ok
if [ ! -e dist/ ]; then mkdir dist; fi
cp node_modules/react/dist/react.min.js dist/react.min.js

```

```

LocalTodoApp.js:9:    // TODO: default english version
LocalTodoApp.js:84:      fwdList = this.host.get('/TodoList#' + listId);
    // TODO fn+id sig
TodoApp.js:117:      // TODO scroll into view
TodoApp.js:176:      if (i >= list.length()) { i = list.length() - 1; } // TODO
➡ .length
local.html:30:    <!-- TODO 2-split, 3-split -->
model/TodoList.js:29:    // TODO one op - repeated spec? long spec?
view/Footer.jsx:61:    // TODO: show the entry's metadata
view/Footer.jsx:80:      todoList.addObject(new TodoItem()); // TODO
➡ create default
view/Header.jsx:25:    // TODO list some meaningful header (apart from the
➡ id)

```

```

npm info start todomvc-swarm@0.0.1

> todomvc-swarm@0.0.1 start /todo
> node TodoAppServer.js

```

Press Ctrl-C here to terminate the process and the container.

```

Swarm server started port 8000
^Cshutting down http-server...
  closing swarm host...
swarm host closed
npm info lifecycle todomvc-swarm@0.0.1~poststart: todomvc-swarm@0.0.1
npm info ok
$ docker ps -a

```

Run this command to see containers that have been started and removed, along with an ID and status (like a process).

```

CONTAINER ID  IMAGE  COMMAND  CREATED  STATUS  PORTS  NAMES
b9db5ada0461  todoapp  "npm start"  2 minutes ago  Exited (0)  2 minutes ago
➡ example1

```

```

$ docker start example1
example1

```

Restart the container, this time in the background.

```

$ docker ps

```

```

CONTAINER ID  IMAGE  COMMAND  CREATED  STATUS
➡ PORTS  NAMES
b9db5ada0461  todoapp  "npm start"  8 minutes ago  Up 10 seconds
➡ 0.0.0.0:8000->8000/tcp  example1

```

Run the ps command again to see the changed status.

```

$ docker diff example1
C /root
C /root/.npm
C /root/.npm/_locks
C /root/.npm/anonymous-cli-metrics.json

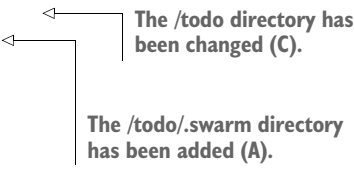
```

The docker diff subcommand shows you what files have been affected since the image was instantiated as a container.

```

C /todo
A /todo/.swarm
A /todo/.swarm/_log
A /todo/dist
A /todo/dist/LocalTodoApp.app.js
A /todo/dist/ToDoApp.app.js
A /todo/dist/react.min.js
C /todo/node_modules

```



The /todo directory has been changed (C).

The /todo/.swarm directory has been added (A).

The `docker run` subcommand starts up the container. The `-p` flag maps the container’s port 8000 to the port 8000 on the host machine, so you should now be able to navigate with your browser to `http://localhost:8000` to view the application. The `--name` flag gives the container a unique name you can refer to later for convenience. The last argument is the image name.

Once the container has been started, you can press `Ctrl-C` to terminate the process and the container. You can run the `ps` command to see the containers that have been started but not removed. Note that each container has its own container ID and status, analogous to a process. Its status is `Exited`, but you can restart it. After you do, notice how the status has changed to `Up` and the port mapping from container to host machine is now displayed.

The `docker diff` subcommand shows you which files have been affected since the image was instantiated as a container. In this case, the `todo` directory has been changed (C), and the other listed files have been added (A). No files have been deleted (D), which is the other possibility.

As you can see, the fact that Docker “contains” your environment means that you can treat it as an entity on which actions can be predictably performed. This gives Docker its breadth of power—you can affect the software lifecycle from development to production and maintenance. These changes are what this book will cover, showing you in practical terms what can be done with Docker.

Next you’re going to learn about layering, another key concept in Docker.

1.2.5 Docker layering

Docker layering helps you manage a big problem that arises when you use containers at scale. Imagine what would happen if you started up hundreds—or even thousands—of the to-do app, and each of those required a copy of the files to be stored somewhere.

As you can imagine, disk space would run out pretty quickly! By default, Docker internally uses a copy-on-write mechanism to reduce the amount of disk space required (see figure 1.9). Whenever a running container needs to write to a file, it records the change by copying the item to a new area of disk. When a Docker commit is performed, this new area of disk is frozen and recorded as a layer with its own identifier.

This partly explains how Docker containers can start up so quickly—they have nothing to copy because all the data has already been stored as the image.

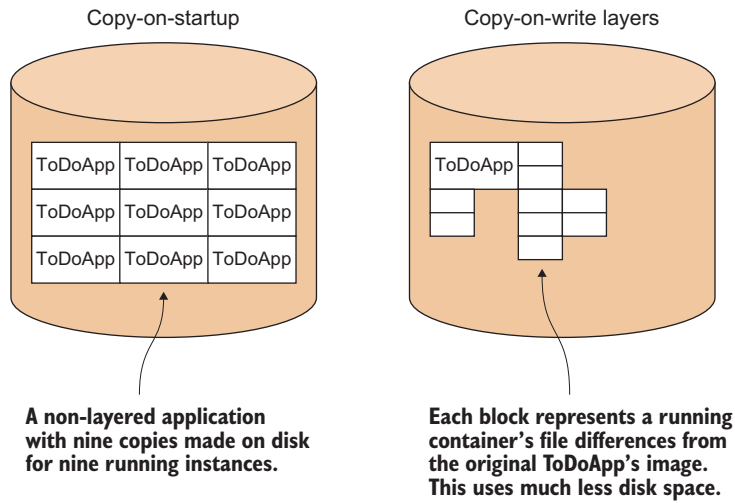


Figure 1.9 Docker filesystem layers

TIP Copy-on-write is a standard optimization strategy used in computing. When you create a new object (of any type) from a template, rather than copying the entire set of data required, you only copy data over when it's changed. Depending on the use case, this can save considerable resources.

Figure 1.10 illustrates that the to-do app you've built has three layers you're interested in. The layers are static, so if you need to change anything in a higher layer, you can just build on top of the image you wish to take as a reference. In the to-do app, you built from the publicly available node image and layered changes on top.

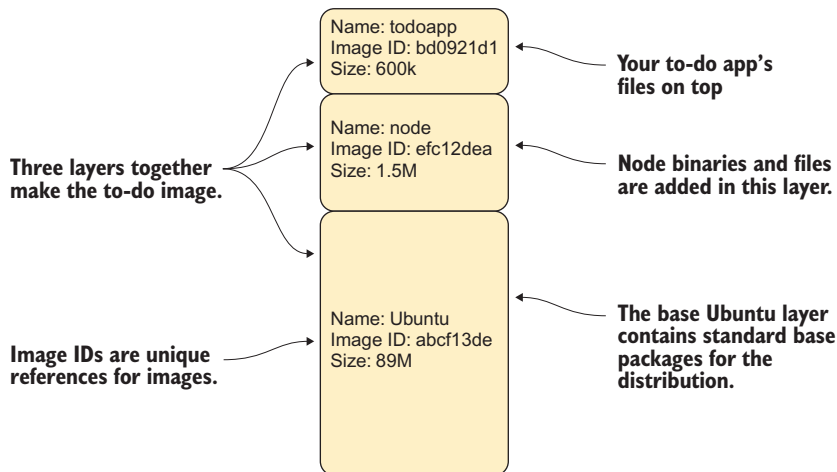


Figure 1.10 The todoapp's filesystem layering in Docker

All three layers can be shared across multiple running containers, much as a shared library can be shared in memory across multiple running processes. This is a vital feature for operations, allowing you to run numerous containers based on different images on host machines without running out of disk space.

Imagine that you're running the to-do app as a live service for paying customers. You can scale up your offering to a large number of users. If you're developing, you can spin up many different environments on your local machine at once. If you're moving through test, you can run many more tests simultaneously, and far more quickly than before. All these things are made possible by layering.

By building and running an application with Docker, you've begun to see the power that Docker can bring to your workflow. Reproducing and sharing specific environments and being able to land these in various places gives you both flexibility and control over development.

Summary

- Docker is an attempt to do for software what containerization did for the shipping industry: reducing the cost of local differences through standardization.
- Some of the uses for Docker include prototyping software, packaging software, reducing the cost of testing and debugging environments, and enabling DevOps practices such as continuous delivery (CD).
- You can build and run a Docker application from a Dockerfile using the commands `docker build` and `docker run`.
- A Docker image is the template for a running container. This is similar to the difference between a program executable and a running process.
- Changes to running containers can be committed and tagged as new images.
- Images are created from a layered filesystem, which reduces the space used by Docker images on your host.