2 Understanding Docker: Inside the engine room

This chapter covers

- Docker's architecture
- Tracing the internals of Docker on your host
- Using the Docker Hub to find and download images
- Setting up your own Docker registry
- Getting containers to communicate with each other

Grasping Docker's architecture is key to understanding Docker more fully. In this chapter you'll get an overview of Docker's major components on your machine and on the network, and you'll learn some techniques that will develop this understanding.

In the process, you'll learn some nifty tricks that will help you use Docker (and Linux) more effectively. Many of the later and more advanced techniques in this book will be based on what you see here, so pay special attention to what follows.

2.1 Docker's architecture

Figure 2.1 lays out Docker's architecture, and that will be the centerpiece of this chapter. We're going to start with a high-level look and then focus on each part, with techniques designed to cement your understanding.

Docker on your host machine is (at the time of writing) split into two parts—a daemon with a RESTful API and a client that talks to the daemon. Figure 2.1 shows your host machine running the Docker client and daemon.

TIP A RESTful API uses standard HTTP request types such as `GET`, `POST`, `DELETE`, and others to represent resources and operations on them. In this case images, containers, volumes, and the like are the represented resources.

You invoke the Docker client to get information from or give instructions to the daemon; the daemon is a server that receives requests and returns responses from the client using the HTTP protocol. In turn, it will make requests to other services to send and receive images, also using the HTTP protocol. The server will accept requests from the command-line client or anyone else authorized to connect. The daemon is also

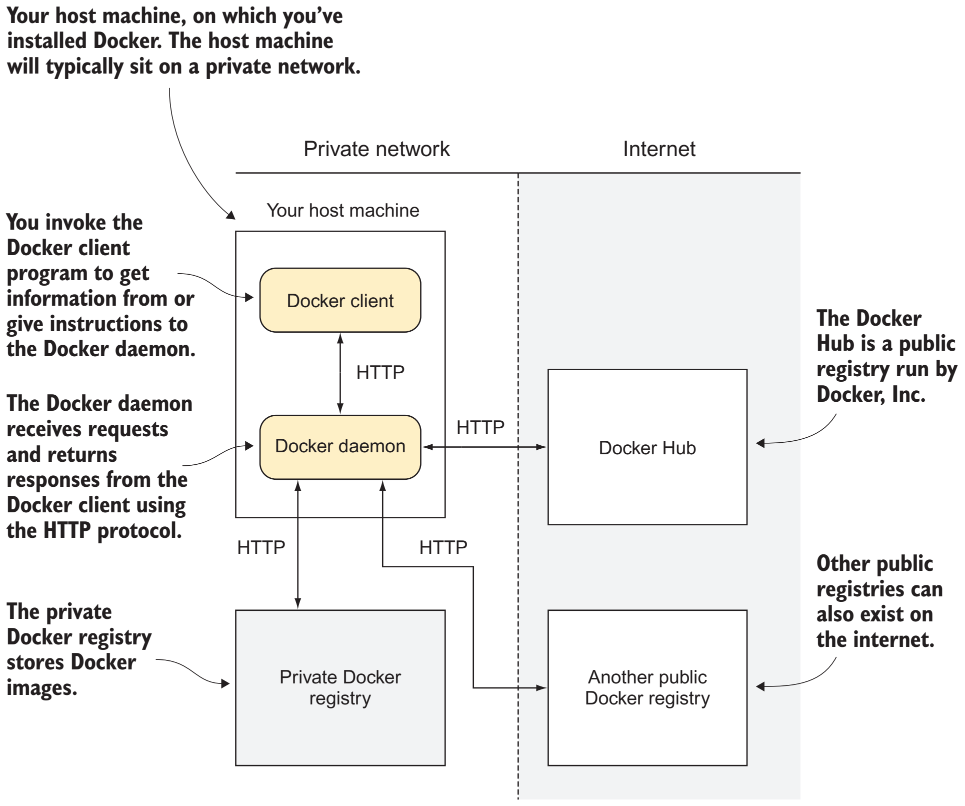

Your host machine, on which you've installed Docker. The host machine will typically sit on a private network.
Private network
Internet
Your host machine
You invoke the Docker client program to get information from or give instructions to the Docker daemon.
Docker client
HTTP
The Docker daemon receives requests and returns responses from the Docker client using the HTTP protocol.
Docker daemon
HTTP
Docker Hub
The Docker Hub is a public registry run by Docker, Inc.
HTTP
HTTP
The private Docker registry stores Docker images.
Private Docker registry
Another public Docker registry
Other public registries can also exist on the internet.

Figure 2.1 Overview of Docker's architecture

responsible for taking care of your images and containers behind the scenes, whereas the client acts as the intermediary between you and the RESTful API.

The private Docker registry is a service that stores Docker images. These can be requested from any Docker daemon that has the relevant access. This registry is on an internal network and isn't publicly accessible, so it's considered private.

Your host machine will typically sit on a private network. The Docker daemon will call out to the internet to retrieve images if requested.

The Docker Hub is a public registry run by Docker Inc. Other public registries can also exist on the internet, and your Docker daemon can interact with them.

In the first chapter we said that Docker containers can be shipped to anywhere you can run Docker—this isn't strictly true. In fact, containers will run on the machine only if the *daemon* can be installed.

The key point to take from figure 2.1 is that when you run Docker on your machine, you may be interacting with other processes on your machine, or even services running on your network or the internet.

Now that you have a picture of how Docker is laid out, we'll introduce various techniques relating to the different parts of the figure.

2.2 The Docker daemon

The Docker daemon (see figure 2.2) is the hub of your interactions with Docker, and as such it's the best place to start gaining an understanding of all the relevant pieces. It controls access to Docker on your machine, manages the state of your containers and images, and brokers interactions with the outside world.

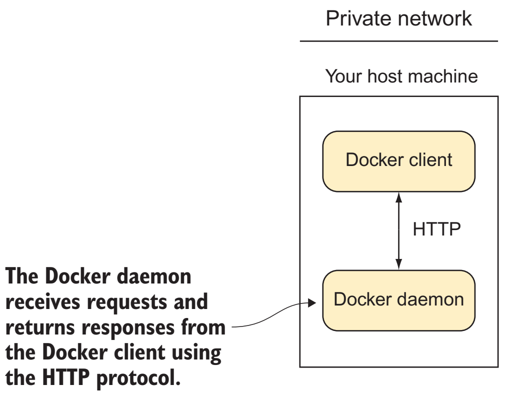

Private network
Your host machine
Docker client
HTTP
Docker daemon
The Docker daemon receives requests and returns responses from the Docker client using the HTTP protocol.

Figure 2.2 The Docker daemon

TIP A *daemon* is a process that runs in the background rather than under the direct control of the user. A *server* is a process that takes requests from a client and performs the actions required to fulfill the requests. Daemons are frequently also servers that accept requests from clients to perform actions for them. The `docker` command is a client, and the Docker daemon acts as the server doing the processing on your Docker containers and images.

Let's look at a couple of techniques that illustrate how Docker effectively runs as a daemon, and how your interactions with it using the `docker` command are limited to simple requests to perform actions, much like interactions with a web server. The first technique allows others to connect to your Docker daemon and perform the same actions you might on your host machine, and the second illustrates that Docker containers are managed by the daemon, not your shell session.

TECHNIQUE 1 Open your Docker daemon to the world

Although by default your Docker daemon is accessible only on your host, there can be good reason to allow others to access it. You might have a problem that you want someone to debug remotely, or you may want to allow another part of your DevOps workflow to kick off a process on a host machine.

WARNING Although this can be a powerful and useful technique, it's considered insecure. A Docker socket can be exploited by anyone with access (including containers with a mounted Docker socket) to get root privileges.

PROBLEM

You want to open your Docker server up for others to access.

SOLUTION

Start the Docker daemon with an open TCP address.

Figure 2.3 gives an overview of how this technique works.

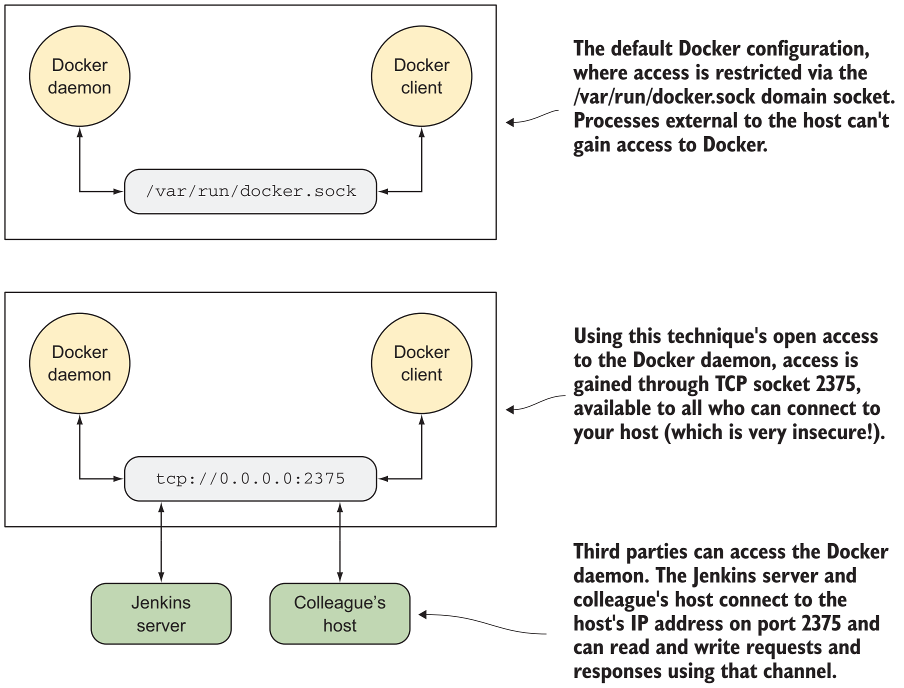

Docker daemon
Docker client
/var/run/docker.sock
The default Docker configuration, where access is restricted via the /var/run/docker.sock domain socket. Processes external to the host can't gain access to Docker.
Docker daemon
Docker client
tcp://0.0.0.0:2375
Using this technique's open access to the Docker daemon, access is gained through TCP socket 2375, available to all who can connect to your host (which is very insecure!).
Jenkins server
Colleague's host
Third parties can access the Docker daemon. The Jenkins server and colleague's host connect to the host's IP address on port 2375 and can read and write requests and responses using that channel.

Figure 2.3 Docker accessibility: normal and opened up

Before you open up the Docker daemon, you must first shut the running one down. How you do this will vary depending on your operating system (non-Linux users should see appendix A). If you're not sure how to do this, start with this command:

```
$ sudo service docker stop
```

If you get a message that looks like this,

```
The service command supports only basic LSB actions (start, stop, restart,
try-restart, reload, force-reload, status). For other actions, please try
to use systemctl.
```

then you have a systemctl-based startup system. Try this command:

```
$ systemctl stop docker
```

If that works, you shouldn't see any output from this command:

```
$ ps -ef | grep -E 'docker(d| -d| daemon)\b' | grep -v grep
```

Once the Docker daemon has been stopped, you can restart it manually and open it up to outside users with the following command:

```
$ sudo docker daemon -H tcp://0.0.0.0:2375
```

This command starts as a daemon (`docker daemon`), defines the host server with the `-H` flag, uses the TCP protocol, opens up all IP interfaces (with `0.0.0.0`), and opens up the standard Docker server port (`2375`).

You can connect from outside with the following command:

```
$ docker -H tcp://<your host's ip>:2375 <subcommand>
```

Or you can export the `DOCKER_HOST` environment variable (this won't work if you have to use `sudo` to run Docker—see technique 41 to find out how to remove this requirement):

```
$ export DOCKER_HOST=tcp://<your host's ip>:2375
$ docker <subcommand>
```

Note that you'll also need to do one of these from inside your local machine as well, because Docker is no longer listening in the default location.

If you want to make this change permanent on your host, you'll need to configure your startup system. See appendix B for information on how to do this.

WARNING If you use this technique to make your Docker daemon listen on a port, be aware that specifying the IP as `0.0.0.0` gives access to users on all network interfaces (both public and private), which is generally considered insecure.

DISCUSSION

This is a great technique if you have a powerful machine dedicated to Docker inside a secure private local network, because everyone on the network can easily point Docker tools to the right place—`DOCKER_HOST` is a well-known environment variable that will inform most programs that access Docker where to look.

As an alternative to the somewhat cumbersome process of stopping the Docker service and running it manually, you could combine mounting the Docker socket as a volume (from technique 45) with using the socat tool to forward traffic from an external port—simply run `docker run -p 2375:2375 -v /var/run/docker.sock:/var/run/docker.sock sequenceid/socat`.

You'll see a specific example of something this technique allows later in the chapter, in technique 5.

TECHNIQUE 2 Running containers as daemons

As you get familiar with Docker (and if you're anything like us), you'll start to think of other use cases for Docker, and one of the first of these is to run Docker containers as background services.

Running Docker containers as services with predictable behavior through software isolation is one of the principal use cases for Docker. This technique will allow you to manage services in a way that works for your operation.

PROBLEM

You want to run a Docker container in the background as a service.

SOLUTION

Use the `-d` flag to the `docker run` command, and use related container-management flags to define the service characteristics.

Docker containers—like most processes—will run by default in the foreground. The most obvious way to run a Docker container in the background is to use the standard `&` control operator. Although this works, you can run into problems if you log out of your terminal session, necessitating that you use the `nohup` flag, which creates a file in your local directory with output that you have to manage… You get the idea: it's far neater to use the Docker daemon's functionality for this.

To do this, you use the `-d` flag.

```
$ docker run -d -i -p 1234:1234 --name daemon ubuntu:14.04 nc -l 1234
```

The `-d` flag, when used with `docker run`, runs the container as a daemon. The `-i` flag gives this container the ability to interact with your Telnet session. With `-p` you publish the 1234 port from the container to the host. The `--name` flag lets you give the container a name so you can refer to it later. Finally, you run a simple listening echo server on port 1234 with netcat (`nc`).

You can now connect to it and send messages with Telnet. You'll see that the container has received the message by using the `docker logs` command, as shown in the following listing.

Listing 2.1 Connecting to the container netcat server with Telnet

```
$ telnet localhost 1234
 Trying ::1...
Connected to localhost.
Escape character is '^]'.
hello daemon
 ^]

telnet> q
 Connection closed.
$ docker logs daemon
 hello daemon
$ docker rm daemon
 daemon
$
```

Connects to the container's netcat server with the telnet command

Inputs a line of text to send to the netcat server

Presses Ctrl-] followed by the Return key to quit the Telnet session

Types q and then the Return key to quit the Telnet program

Runs the docker logs command to see the container's output

Cleans up the container with the rm command

You can see that running a container as a daemon is simple enough, but operationally some questions remain to be answered:

- What happens to the service if it fails?
- What happens to the service when it terminates?
- What happens if the service keeps failing over and over?

Fortunately Docker provides flags for each of these questions!

NOTE Although restart flags are used most often with the daemon flag (`-d`), it's not a requirement to run these flags with `-d`.

The `docker  run--restart` flag allows you to apply a set of rules to be followed (a so-called "restart policy") when the container terminates (see table 2.1).

Table 2.1 Docker restart flag options

Policy	Description
`no`	Don't restart when the container exits
`always`	Always restart when the container exits
`unless-stopped`	Always restart, but remember explicitly stopping
`on-failure[:max-retry]`	Restart only on failure

The `no` policy is simple: when the container exits, it isn't restarted. This is the default.

The `always` policy is also simple, but it's worth discussing briefly:

```
$ docker run -d --restart=always ubuntu echo done
```

This command runs the container as a daemon (`-d`) and always restarts the container on termination (`--restart=always`). It issues a simple `echo` command that completes quickly, exiting the container.

If you run the preceding command and then run a `docker ps` command, you'll see output similar to this:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
➥       STATUS                            PORTS               NAMES
69828b118ec3        ubuntu:14.04        "echo done"         4 seconds ago
➥        Restarting  (0) Less than a second ago            sick_brattain
```

The `docker ps` command lists all the running containers and information about them, including the following:

- When the container was created (`CREATED`).
- The current status of the container—usually this will be `Restarting`, because it will only run for a short time (`STATUS`).
- The exit code of the container's previous run (also under `STATUS`). `0` means the run was successful.
- The container name. By default, Docker names containers by concatenating two random words. Sometimes this produces odd results (which is why we typically recommend giving them a meaningful name).

Note that the `STATUS` column also informed us that the container exited less than a second ago and is restarting. This is because the `echo done` command exits immediately, and Docker must continually restart the container.

It's important to note that Docker reuses the container ID. It doesn't change on restart, and there will only ever be one entry in the `ps` table for this Docker invocation.

Specifying `unless-stopped` is almost the same as `always`—both will cease restarting if you run `docker stop` on a container, but `unless-stopped` will make sure that this stopped status is remembered if the daemon restarts (perhaps if you restart your computer), whereas `always` will bring the container back up again.

Finally, the `on-failure` policy restarts only when the container returns a non-zero exit code (which normally means failing) from its main process:

```
$ docker run -d --restart=on-failure:10 ubuntu /bin/false
```

This command runs the container as a daemon (`-d`) and sets a limit on the number of restart attempts (`--restart=on-failure:10`), exiting if this is exceeded. It runs a simple command (`/bin/false`) that completes quickly and will definitely fail.

If you run the preceding command and wait a minute, and then run `docker ps -a`, you'll see output similar to this:

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED
➥      STATUS                         PORTS               NAMES
b0f40c410fe3        ubuntu:14.04        "/bin/false"        2 minutes ago
➥      Exited (1) 25 seconds ago                          loving_rosalind
```

DISCUSSION

Creating services to run in the background often comes with the difficulty of making sure the background service won't crash in unusual environments. Because it's not immediately visible, users may not notice that something isn't working correctly.

This technique lets you stop thinking about the incidental complexity of your service, caused by the environment and handling restarts. You can focus your thinking on the core functionality.

As a concrete example, you and your team might use this technique to run multiple databases on the same machine and avoid having to write instructions for setting them up or to have terminals open to keep them running.

TECHNIQUE 3 Moving Docker to a different partition

Docker stores all the data relating to your containers and images under a folder. Because it can store a potentially large number of different images, this folder can get big fast!

If your host machine has different partitions (as is common in enterprise Linux workstations), you may encounter space limitations more quickly. In these cases, you may want to move the directory from which Docker operates.

PROBLEM

You want to move where Docker stores its data.

SOLUTION

Stop and start the Docker daemon, specifying the new location with the `-g` flag.

Imagine you want to run Docker from /home/dockeruser/mydocker. First, stop your Docker daemon (see appendix B for a discussion on how to do this). Then run this command:

```
$ dockerd -g /home/dockeruser/mydocker
```

A new set of folders and files will be created in this directory. These folders are internal to Docker, so play with them at your peril (as we've discovered!).

You should be aware that this command will appear to wipe the containers and images from your previous Docker daemon. Don't despair. If you kill the Docker process you just ran, and restart your Docker service, your Docker client will be pointed back at its original location, and your containers and images will be returned to you. If you want to make this move permanent, you'll need to configure your host system's startup process accordingly.

DISCUSSION

Aside from the obvious use case for this (reclaiming space on disks with limited disk space), you could also use this technique if you wanted to strictly partition sets of images and containers. For example, if you have access to multiple private Docker registries with different owners, it might be worth the extra effort to make sure you don't accidentally give private data to the wrong person.

2.3 The Docker client

The Docker client (see figure 2.4) is the simplest component in the Docker architecture. It's what you run when you type commands like `docker run` or `docker pull` on your machine. Its job is to communicate with the Docker daemon via HTTP requests.

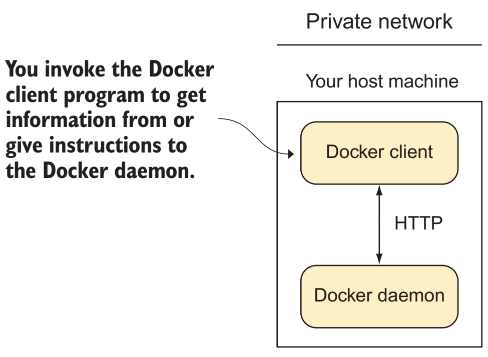

Private network
Your host machine
You invoke the Docker client program to get information from or give instructions to the Docker daemon.
Docker client
HTTP
Docker daemon

Figure 2.4 The Docker client

In this section you're going to see how you can snoop on messages between the Docker client and server. You'll also see a way of using your browser as a Docker client and a few basic techniques related to port mapping that represent baby steps toward orchestration, discussed in part 4 of this book.

TECHNIQUE 4 Using socat to monitor Docker API traffic

Occasionally the `docker` command may not work as you expect. Most often, some aspect of the command-line arguments hasn't been understood, but occasionally there are more serious setup problems, such as the Docker binary being out of date. In order to diagnose the problem, it can be useful to view the flow of data to and from the Docker daemon you're communicating with.

NOTE Don't panic! The presence of this technique doesn't indicate that Docker needs to be debugged often, or that it's in any way unstable! This technique is here primarily as a tool for understanding Docker's architecture, and also to introduce you to socat, a powerful tool. If, like us, you use Docker in a lot of different locations, there will be differences in the Docker versions you use. As with any software, different versions will have different features and flags, which can catch you out.

PROBLEM

You want to debug a problem with a Docker command.

SOLUTION

Use a traffic snooper to inspect the API calls and craft your own.

In this technique you'll insert a proxy Unix domain socket between your request and the server's socket to see what passes through it (as shown in figure 2.5). Note that you'll need root or sudo privileges to make this work.

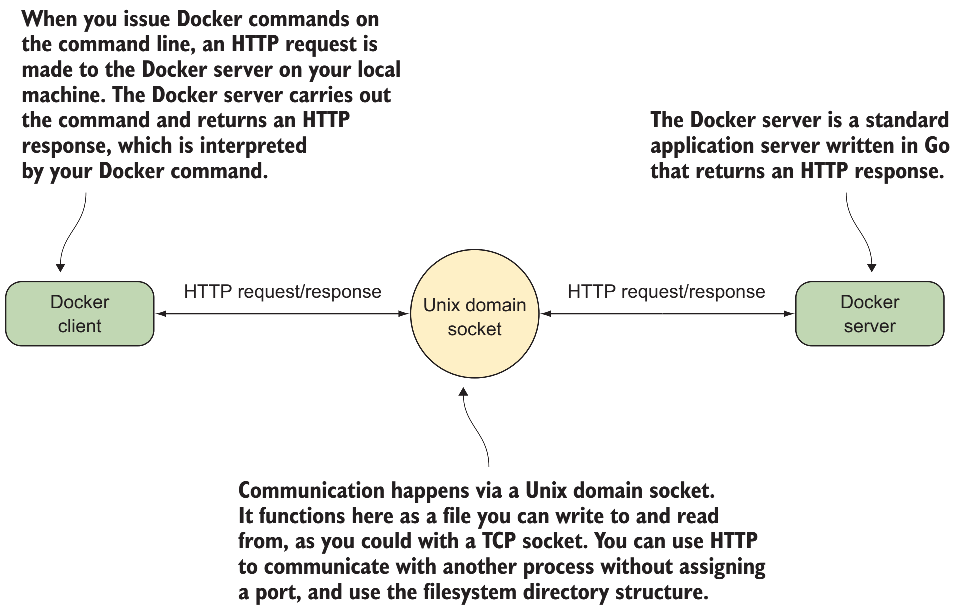

When you issue Docker commands on the command line, an HTTP request is made to the Docker server on your local machine. The Docker server carries out the command and returns an HTTP response, which is interpreted by your Docker command.
The Docker server is a standard application server written in Go that returns an HTTP response.
Docker client
HTTP request/response
Unix domain socket
HTTP request/response
Docker server
Communication happens via a Unix domain socket. It functions here as a file you can write to and read from, as you could with a TCP socket. You can use HTTP to communicate with another process without assigning a port, and use the filesystem directory structure.

Figure 2.5 Docker's client/server architecture on your host

To create this proxy, you'll use socat.

> **TIP** `socat` is a powerful command that allows you to relay data between two data channels of almost any type. If you're familiar with netcat, you can think of it as netcat on steroids. To install it, use the standard package manager for your system.

```
$ socat -v UNIX-LISTEN:/tmp/dockerapi.sock,fork \
  UNIX-CONNECT:/var/run/docker.sock &
```

In this command, `-v` makes the output readable, with indications of the flow of data. The `UNIX-LISTEN` part tells `socat` to listen on a Unix socket, `fork` ensures that `socat` doesn't exit after the first request, and `UNIX-CONNECT` tells `socat` to connect to Docker's Unix socket. The `&` specifies that the command runs in the background. If you usually run the Docker client with sudo, you'll need to do the same thing here as well.

The new route that your requests to the daemon will travel can be seen in figure 2.6. All traffic traveling in each direction will be seen by `socat` and logged to your terminal, in addition to any output that the Docker client provides.

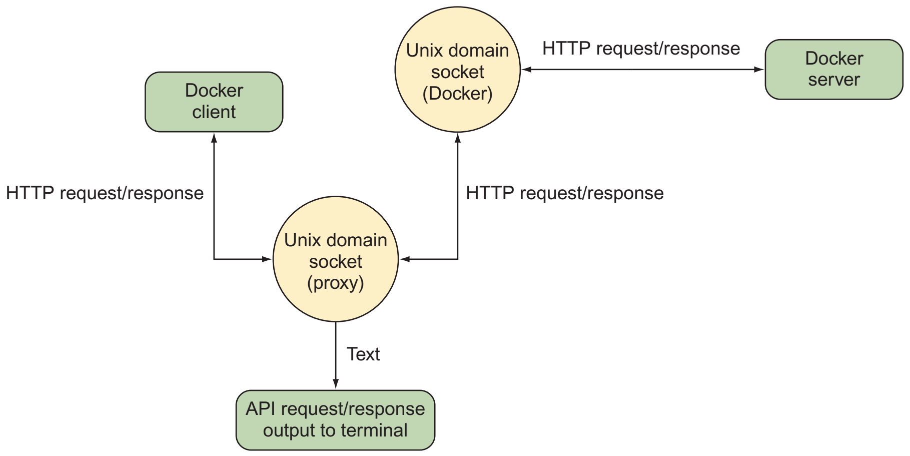


Figure 2.6 Docker client and server with socat inserted as a proxy

The output of a simple `docker` command will now look similar to this:

```
$ docker -H unix:///tmp/dockerapi.sock ps -a
 > 2017/05/15 16:01:51.163427  length=83 from=0 to=82
GET /_ping HTTP/1.1\r
Host: docker\r
User-Agent: Docker-Client/17.04.0-ce (linux)\r
\r
< 2017/05/15 16:01:51.164132  length=215 from=0 to=214
HTTP/1.1 200 OK\r
Api-Version: 1.28\r
Docker-Experimental: false\r
Ostype: linux\r
Server: Docker/17.04.0-ce (linux)\r
Date: Mon, 15 May 2017 15:01:51 GMT\r
Content-Length: 2\r
Content-Type: text/plain; charset=utf-8\r
\r
OK> 2017/05/15 16:01:51.165175  length=105 from=83 to=187
 GET /v1.28/containers/json?all=1 HTTP/1.1\r
Host: docker\r
User-Agent: Docker-Client/17.04.0-ce (linux)\r
\r
< 2017/05/15 16:01:51.165819  length=886 from=215 to=1100
 HTTP/1.1 200 OK\r
Api-Version: 1.28\r
Content-Type: application/json\r
Docker-Experimental: false\r
Ostype: linux\r
Server: Docker/17.04.0-ce (linux)\r
Date: Mon, 15 May 2017 15:01:51 GMT\r
Content-Length: 680\r
\r
[{"Id":"1d0d5b5a7b506417949653a59deac030ccbcbb816842a63ba68401708d55383e",
➥ "Names":["/example1"],"Image":"todoapp","ImageID":
```

The command you issue to see the request and response

The HTTP request begins here, with the right angle bracket on the left.

The HTTP response begins here, with the left angle bracket on the left.

```
➥ "sha256:ccdda5b6b021f7d12bd2c16dbcd2f195ff20d10a660921db0ac5bff5ecd92bc2",
➥ "Command":"npm start","Created":1494857777,"Ports":[],"Labels":{},
➥ "State":"exited","Status":"Exited (0) 45 minutes ago","HostConfig":
➥ {"NetworkMode":"default"},"NetworkSettings":{"Networks":{"bridge":
➥ {"IPAMConfig":null,"Links":null,"Aliases":null,"NetworkID":
➥ "6f327d67a38b57379afa7525ea63829797fd31a948b316fdf2ae0365faeed632",
➥ "EndpointID":"","Gateway":"","IPAddress":"","IPPrefixLen":0,
➥ "IPv6Gateway":"","GlobalIPv6Address":"","GlobalIPv6PrefixLen":0,
➥ "MacAddress":""}}},"Mounts":[]}]
CONTAINER ID        IMAGE               COMMAND             CREATED
➥      STATUS                      PORTS               NAMES
1d0d5b5a7b50        todoapp             "npm start"         45 minutes ago
➥      Exited (0) 45 minutes ago                       example1
```

The JSON content of the response from the Docker server

The output as normally seen by the user, interpreted by the Docker client from the preceding JSON

The specifics of the preceding output will change with the Docker API's growth and development. When you run the preceding command, you'll see later version numbers and different JSON output. You can check your client and server versions of the API by running the `docker version` command.

WARNING If you ran socat as root in the previous example, you'll need to use sudo to run the `docker -H` command. This is because the dockerapi.sock file is owned by root.

Using socat is a powerful way to debug not only Docker, but any other network services you might come across in the course of your work.

DISCUSSION

It's possible to come up with a number of other use cases for this technique:

- Socat is quite a Swiss Army knife and can handle quite a number of different protocols. The preceding example shows it listening on a Unix socket, but you could also make it listen on an external port with `TCP-LISTEN:2375,fork` instead of the `UNIX-LISTEN:…` argument. This acts as a simpler version of technique 1. With this approach there's no need to restart the Docker daemon (which would kill any running containers). You can just bring the socat listener up and down as you desire.
- Because the preceding point is so simple to set up and is temporary, you can use it in combination with technique 47 to join a colleague's running container remotely and help them debug an issue. You can also employ the little-used `docker attach` command to join the same terminal they started with `docker run`, allowing you to collaborate directly.
- If you have a shared Docker server (perhaps set up with technique 1) you could use the ability to expose externals and set up socat as the broker between the outside world and the Docker socket to make it act as a primitive audit log, recording where all requests are coming from and what they're doing.

TECHNIQUE 5 Using Docker in your browser

It can be difficult to sell new technologies, so simple and effective demonstrations are invaluable. Making the demo hands-on is even better, which is why we've found that creating a web page that allows users to interact with a container in a browser is a great technique. It gives newcomers their first taste of Docker in an easily accessible way, allowing them to create a container, use a container terminal in a browser, attach to someone else's terminal, and share control. The significant "wow" factor doesn't hurt either!

PROBLEM

You want to demonstrate the power of Docker without requiring users to install it themselves or run commands they don't understand.

SOLUTION

Start the Docker daemon with an open port and Cross-Origin-Resource Sharing (CORS) enabled, and then serve the docker-terminal repository in your web server of choice.

The most common use of a REST API is to expose it on a server and use JavaScript on a web page to make calls to it. Because Docker happens to perform all interaction via a REST API, you should be able to control Docker in the same way. Although it may initially seem surprising, this control extends all the way to being able to interact with a container via a terminal in your browser.

We've already discussed how to start the daemon on port 2375 in technique 1, so we won't go into any detail on that. Additionally, CORS is too much to go into here. If you're unfamiliar with it, refer to *CORS in Action* by Monsur Hossain (Manning, 2014). In short, CORS is a mechanism that carefully bypasses the usual restriction of JavaScript that limits you to only accessing the current domain. In this case, CORS allows the daemon to listen on a different port from where you serve your Docker terminal page. To enable CORS, you need to start the Docker daemon with the option `--api-enable-cors` alongside the option to make it listen on a port.

Now that the prerequisites are sorted, let's get this running. First, you need to get the code:

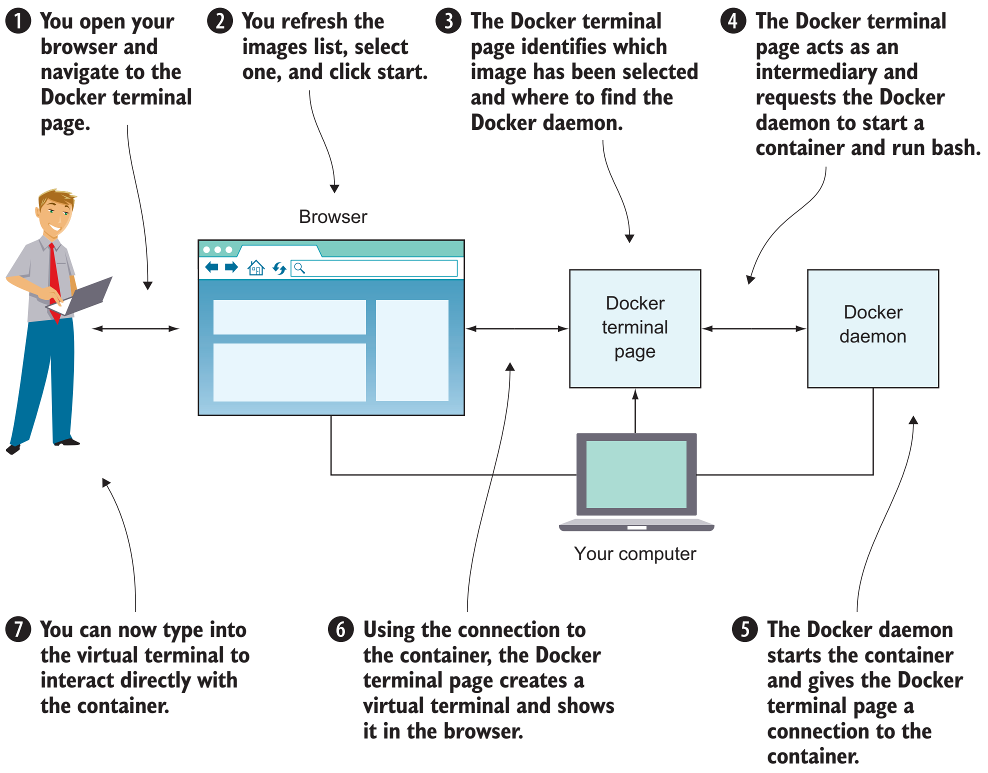
```
git clone https://github.com/aidanhs/Docker-Terminal.git
cd Docker-Terminal
```

Then you need to serve the files:

```
python2 -m SimpleHTTPServer 8000
```

The preceding command uses a module built into Python to serve static files from a directory. Feel free to use any equivalent you prefer. Now visit http://localhost:8000 in your browser and start a container.

Figure 2.7 shows how the Docker terminal connects up. The page is hosted on your local computer and connects to the Docker daemon on your local computer to perform any operations.



1 You open your browser and navigate to the Docker terminal page.
2 You refresh the images list, select one, and click start.
3 The Docker terminal page identifies which image has been selected and where to find the Docker daemon.
4 The Docker terminal page acts as an intermediary and requests the Docker daemon to start a container and run bash.
Browser
Docker terminal page
Docker daemon
Your computer
7 You can now type into the virtual terminal to interact directly with the container.
6 Using the connection to the container, the Docker terminal page creates a virtual terminal and shows it in the browser.
5 The Docker daemon starts the container and gives the Docker terminal page a connection to the container.

Figure 2.7 How the Docker terminal works

It's worth being aware of the following points if you want to give this link to other people:

- The other person must not be using a proxy of any kind. This is the most common source of errors we've seen—Docker terminal uses WebSockets, which don't currently work through proxies.
- Giving a link to `localhost` obviously won't work—you'll need to give out the external IP address.
- Docker terminal needs to know where to find the Docker API—it should do this automatically based on the address you're visiting in the browser, but it's something to be aware of.

TIP If you're more experienced with Docker, you might wonder why we didn't use a Docker image in this technique. The reason is that we're still introducing Docker and didn't want to add to the complexity for readers new to Docker. Dockerizing this technique is left as an exercise for the reader.

DISCUSSION

Although we originally used this technique as an exciting demo for Docker (having multiple people easily share a terminal on disposable machines can be difficult to set up, even with terminal multiplexers), we've found some interesting applications in some unrelated areas. One example is using it to monitor a small group of trainees in some task at the command line. There's no need for you or them to install anything; just open your browser and you can connect to their terminal to jump in and give them a hand at any time!

On a similar note, this has some strengths in collaboration. In the past, when we've wanted to share a bug with a coworker, we've reproduced the bug in a Docker container so we can track it down together. With this technique, there's no need to go through a possible "but why do I want Docker?" discussion beforehand.

TECHNIQUE 6 Using ports to connect to containers

Docker containers have been designed from the outset to run services. In the majority of cases, these will be HTTP services of one kind or another. A significant proportion of these will be web services accessible through the browser.

This leads to a problem. If you have multiple Docker containers running on port 80 in their internal environment, they can't all be accessible on port 80 on your host machine. This technique shows how you can manage this common scenario by exposing and mapping a port from your container.

PROBLEM

You want to make multiple Docker container services available on a port from your host machine.

SOLUTION

Use Docker's `-p` flag to map a container's port to your host machine.

In this example we're going to use the tutum-wordpress image. Let's say you want to run two of these on your host machine to serve different blogs.

Because a number of people have wanted to do this before, someone has prepared an image that anyone can acquire and start up. To obtain images from external locations, you can use the `docker pull` command. By default, images will be downloaded from the Docker Hub:

```
$ docker pull tutum/wordpress
```

Images will also be retrieved automatically when you try to run them if they're not already present on your machine.

To run the first blog, use the following command:

```
$ docker run -d -p 10001:80 --name blog1 tutum/wordpress
```

This `docker run` command runs the container as a daemon (`-d`) with the publish flag (`-p`). It identifies the host port (`10001`) to map to the container port (`80`) and gives the container a name to identify it (`--name blog1 tutum/wordpress`).

You can do the same for the second blog:

```
$ docker run -d -p 10002:80 --name blog2 tutum/wordpress
```

If you now run this command,

```
$ docker ps | grep blog
```

you'll see the two blog containers listed, with their port mappings, looking something like this:

```
$ docker ps | grep blog
9afb95ad3617  tutum/wordpress:latest  "/run.sh" 9 seconds ago
➥ Up 9 seconds   3306/tcp, 0.0.0.0:10001->80/tcp  blog1
31ddc8a7a2fd  tutum/wordpress:latest  "/run.sh" 17 seconds ago
➥ Up 16 seconds  3306/tcp, 0.0.0.0:10002->80/tcp  blog2
```

You'll now be able to access your containers by navigating to http://localhost:10001 and http://localhost:10002.

To remove the containers when you're finished (assuming you don't want to keep them—we'll make use of them in the next technique), run this command:

```
$ docker rm -f blog1 blog2
```

You should now be able to run multiple identical images and services on your host by managing the port allocations yourself, if necessary.

TIP It can be easy to forget which port is the host's and which port is the container's when using the `-p` flag. We think of it as being like reading a sentence from left to right. The user connects to the host (`-p`) and that host port is passed to the container port (`host_port:container_port`). It's also the same format as SSH's port-forwarding commands, if you're familiar with them.

DISCUSSION

Exposing ports is an incredibly important part of many use cases of Docker, and you'll come across it a number of times throughout this book, especially in part 4, where containers talking to each other is part of everyday life.

In technique 80 we'll introduce you to virtual networks and explain what they do behind the scenes and how they direct the host ports to the right container.

TECHNIQUE 7 Allowing container communication

The last technique showed how you can open up your containers to the host network by exposing ports. You won't always want to expose your services to the host machine or the outside world, but you will want to connect containers to one another.

This technique shows how you can achieve this with Docker's user-defined networks feature, ensuring outsiders can't access your internal services.

PROBLEM

You want to allow communication between containers for internal purposes.

SOLUTION

Employ user-defined networks to enable containers to communicate with each other.

User-defined networks are simple and flexible. We have a couple of WordPress blogs running in containers from the previous technique, so let's take a look at how we can reach them from another container (rather than from the outside world, which you've seen already).

First you'll need to create a user-defined network:

```
$ docker network create my_network
0c3386c9db5bb1d457c8af79a62808f78b42b3a8178e75cc8a252fac6fdc09e4
```

This command creates a new virtual network living on your machine that you can use to manage container communication. By default, all containers that you connect to this network will be able to see each other by their names.

Next, assuming that you still have the `blog1` and `blog2` containers running from the previous technique, you can connect one of them to your new network on the fly.

```
$ docker network connect my_network blog1
```

Finally, you can start up a new container, explicitly specifying the network, and see if you can retrieve the first five lines of HTML from the landing page of the blog.

```
$ docker run -it --network my_network ubuntu:16.04 bash
root@06d6282d32a5:/# apt update && apt install -y curl
[...]
root@06d6282d32a5:/# curl -sSL blog1 | head -n5
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US" xml:lang="en-US">
<head>
        <meta name="viewport" content="width=device-width" />
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
root@06d6282d32a5:/# curl -sSL blog2
curl: (6) Could not resolve host: blog2
```

TIP Giving containers names is very useful for assigning memorable hostnames you can later refer to, but it's not strictly necessary—if connections are only outgoing, then you probably won't need to look up the container. If you find you *do* want to look up the host and haven't assigned a name, you can resort to using the short image ID as listed in the terminal prompt (unless it has been overridden with a hostname) or in the `docker ps` output.

Our new container was successfully able to access the blog we connected to `my_network`, displaying some of the HTML of the page we'd see if we visited it in the browser. On the other hand, our new container couldn't see the second blog. Because we never connected it to `my_network`, this makes sense.

DISCUSSION

You can use this technique to set up any number of containers in a cluster on their own private network, only requiring that the containers have some way of discovering each other's names. In technique 80 you'll see a method of doing this that integrates well with Docker networks. Meanwhile, the next technique will start much smaller, demonstrating some benefits of being able to make an explicit connection between a single container and the service it provides.

One additional point of note is the interesting final state of the `blog1` container. All containers are connected to the Docker `bridge` network by default, so when we asked for it to join `my_network`, it did so *in addition to* the network it was already on. In technique 80 we'll look at this in more detail to see how *network straddling* can be used as a model for some real-world situations.

TECHNIQUE 8 Linking containers for port isolation

In the previous technique you saw how to get containers to communicate with user-defined networks. But there's an older method of declaring container communication—Docker's link flag. This isn't the recommended way of working anymore, but it has been part of Docker for a long time, and it's worth being aware of in case you encounter it in the wild.

PROBLEM

You want to allow communication between containers without using user-defined networks.

SOLUTION

Use Docker's linking functionality to allow the containers to communicate with each other.

Taking up the torch of the WordPress example, we're going to separate the MySQL database tier from the WordPress container, and link these to each other without port configuration or creating a network. Figure 2.8 gives an overview of the final state.

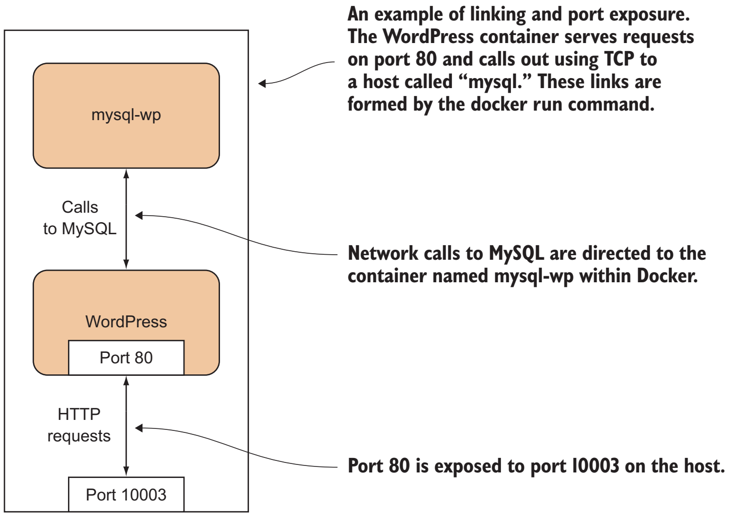

An example of linking and port exposure. The WordPress container serves requests on port 80 and calls out using TCP to a host called "mysql." These links are formed by the docker run command.
mysql-wp
Calls to MySQL
Network calls to MySQL are directed to the container named mysql-wp within Docker.
WordPress
Port 80
HTTP requests
Port 80 is exposed to port 10003 on the host.
Port 10003

Figure 2.8 WordPress setup with linked containers

NOTE Why bother with linking if you can already expose ports to the host and use that? Linking allows you to encapsulate and define the relationships between containers without exposing services to the host's network (and potentially, to the outside world). You might want to do this for security reasons, for example.

Run your containers in the following order, pausing for about a minute between the first and second commands:

```
$ docker run --name wp-mysql \
  -e MYSQL_ROOT_PASSWORD=yoursecretpassword -d mysql
$ docker run --name wordpress \
  --link wp-mysql:mysql -p 10003:80 -d  wordpress
```

First you give the MySQL container the name `wp-mysql` so you can refer to it later. You also must supply an environment variable so the MySQL container can initialize the database (`-e MYSQL_ROOT_PASSWORD=yoursecretpassword`). You run both containers as daemons (`-d`) and use the Docker Hub reference for the official MySQL image.

In the second command you give the WordPress image the name `wordpress` in case you want to refer to it later. You also link the wp-mysql container to the WordPress container (`--link wp-mysql:mysql`). References to a `mysql` server within the WordPress container will be sent to the container named `wp-mysql`. You also use a local port mapping (`-p 10003:80`), as discussed in technique 6, and add the Docker Hub reference for the official WordPress image (`wordpress`). Be aware that links won't wait for services in linked containers to start; hence the instruction to pause between commands. A more precise way of doing this is to look for "mysqid: ready for connections" in the output of `docker logs wp-mysql` before running the WordPress container.

If you now navigate to http://localhost:10003, you'll see the introductory WordPress screen and you can set up your WordPress instance.

The meat of this example is the `--link` flag in the second command. This flag sets up the container's host file so that the WordPress container can refer to a MySQL server, and this will be routed to whatever container has the name `wp-mysql`. This has the significant benefit that different MySQL containers can be swapped in without requiring any change at all to the WordPress container, making configuration management of these different services much easier.

NOTE The containers must be started up in the correct order so that the mapping can take place on container names that are already in existence. Dynamic resolution of links is not (at the time of writing) a feature of Docker.

In order for containers to be linked in this way, their ports must be specified as exposed when building the images. This is achieved using the `EXPOSE` command within the image build's Dockerfile. The ports listed in `EXPOSE` directives in Dockerfiles are also used when using the `-P` flag ("publish all ports." rather than `-p`, which publishes as a specific port) to the `docker run` command.

By starting up different containers in a specific order, you've seen a simple example of Docker orchestration. Docker orchestration is any process that coordinates the running of Docker containers. It's a large and important subject that we'll cover in depth in part 4 of this book.

By splitting your workload into separate containers, you've taken a step toward creating a microservices architecture for your application. In this case you could perform work on the MySQL container while leaving the WordPress container untouched, or vice versa. This fine-grained control over running services is one of the key operational benefits of a microservices architecture.

DISCUSSION

This kind of precise control over a set of containers is not often needed, but it can be useful as a very straightforward and easy-to-reason-about way to swap out containers. Using the example from this technique, you might want to test a different MySQL version—the WordPress image doesn't need to know anything about this, because it just looks for the `mysql` link.

2.4 Docker registries

Once you've created your images, you may want to share them with other users. This is where the concept of the *Docker registry* comes in.

The three registries in figure 2.9 differ in their accessibility. One is on a private network, one is open on a public network, and another is public but accessible only to those registered with Docker. They all perform the same function with the same API, and this is how the Docker daemon knows how to communicate with them interchangeably.

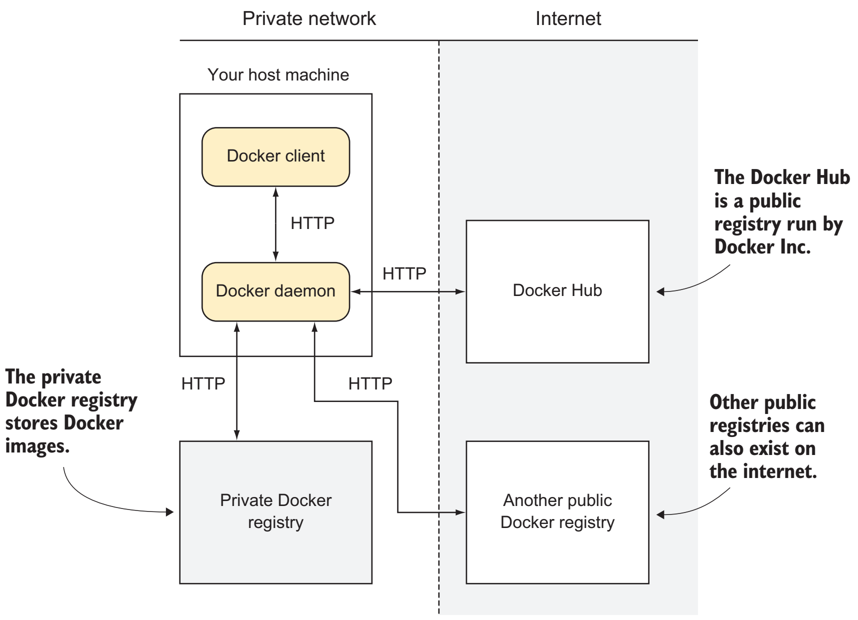

Private network
Internet
Your host machine
Docker client
HTTP
Docker daemon
HTTP
Docker Hub
The Docker Hub is a public registry run by Docker Inc.
HTTP
HTTP
The private Docker registry stores Docker images.
Private Docker registry
Another public Docker registry
Other public registries can also exist on the internet.

Figure 2.9 A Docker registry

A Docker registry allows multiple users to push and pull images from a central store using a RESTful API.

The registry code is, like Docker itself, open source. Many companies (such as ours) set up private registries to store and share their proprietary images internally. This is what we'll discuss before we look more closely at Docker Inc.'s registry.

TECHNIQUE 9 Setting up a local Docker registry

You've seen that Docker Inc. has a service where people can share their images publicly (and you can pay if you want to do it privately). But there are a number of reasons you may want to share images without going via the Hub—some businesses like to keep as much in-house as possible, maybe your images are large and transferring them over the internet will be too slow, or perhaps you want to keep your images private while you experiment and don't want to commit to paying. Whatever the reason, there's a simple solution.

PROBLEM

You want a way to host your images locally.

SOLUTION

Set up a registry server on your local network. Simply issue the following command on a machine with plenty of disk space:

```
$ docker run -d -p 5000:5000 -v $HOME/registry:/var/lib/registry registry:2
```

This command makes the registry available on port 5000 of the Docker host (`-p 5000:5000`). With the `-v` flag, it makes the registry folder on your host (/var/lib/registry) available in the container as `$HOME/registry`. The registry's files will therefore be stored on the host in the /var/lib/registry folder.

On all of the machines that you want to access this registry, add the following to your daemon options (where `HOSTNAME` is the hostname or IP address of your new registry server): `--insecure-registry HOSTNAME` (see appendix B for details on how to do this). You can now issue the following command: `docker push HOSTNAME:5000/image:tag`.

As you can see, the most basic level of configuration for a local registry, with all data stored in the $HOME/registry directory, is simple. If you wanted to scale up or make it more robust, the repository on GitHub (https://github.com/docker/distribution/blob/v2.2.1/docs/storagedrivers.md) outlines some options, like storing data in Amazon S3.

You may be wondering about the `--insecure-registry` option. In order to help users remain secure, Docker will only allow you to pull from registries with a signed HTTPS certificate. We've overridden this because we're fairly comfortable that we can trust our local network. It goes without saying, though, that you should be much more cautious about doing this over the internet.

DISCUSSION

With registries being so easy to set up, a number of possibilities arise. If your company has a number of teams, you might suggest that everyone start up and maintain a registry on a spare machine to permit some fluidity in storing images and moving them around.

This works particularly well if you have an internal IP address range—the `--insecure-registry` command will accept CIDR notation, like `10.1.0.0/16`, for specifying a range of IP addresses that are permitted to be insecure. If you're not familiar with this, we highly recommend you get in touch with your network administrator.

2.5 The Docker Hub

The Docker Hub (see figure 2.10) is a registry maintained by Docker Inc. It has tens of thousands of images on it ready to download and run. Any Docker user can set up a free account and store public Docker images there. In addition to user-supplied images, official images are maintained for reference purposes.

Your images are protected by user authentication, and there's a starring system for popularity, similar to GitHub's. The official images can be representations of Linux distributions like Ubuntu or CentOS, preinstalled software packages like Node.js, or whole software stacks like WordPress.

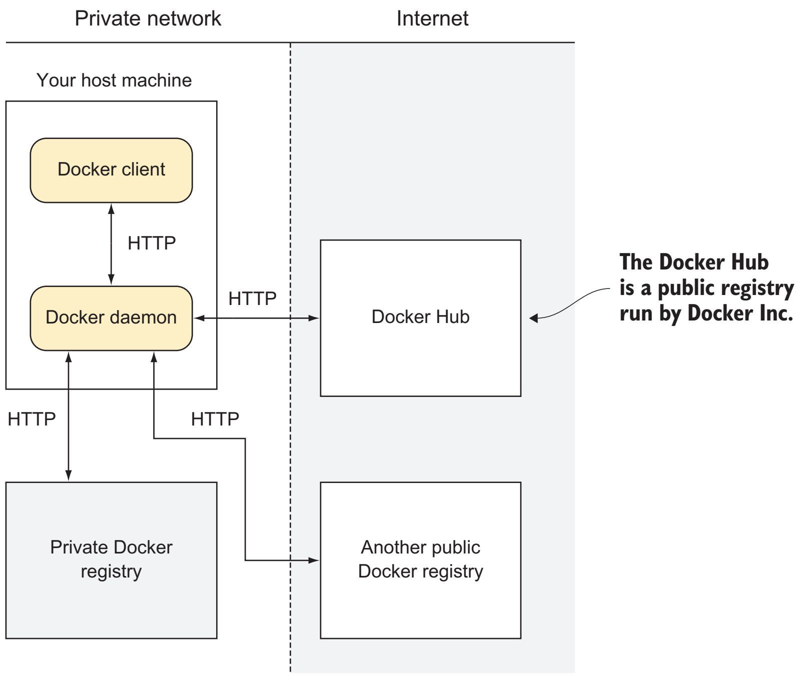


Figure 2.10 The Docker Hub

TECHNIQUE 10 Finding and running a Docker image

Docker registries enable a social coding culture similar to GitHub's. If you're interested in trying out a new software application, or looking for a new one that serves a particular purpose, Docker images can be an easy way to experiment without interfering with your host machine, provisioning a VM, or having to worry about installation steps.

PROBLEM

You want to find an application or tool as a Docker image and try it out.

SOLUTION

Use the `docker search` command to find the image to pull, and then run it.

Let's say you're interested in playing with Node.js. In the following example we searched for images matching "node" with the `docker search` command:

The description is the uploader's explanation of the purpose of the image.

The output of docker search is ordered by the number of stars.

Official images are those trusted by Docker Hub.

Automated images are those built using Docker Hub's automated build feature.

```
$ docker search node
NAME                       DESCRIPTION
➥ STARS     OFFICIAL   AUTOMATED
node                       Node.js is a JavaScript-based platform for...
➥   3935       [OK]
 nodered/node-red-docker     Node-RED Docker images.
➥   57                     [OK]
 strongloop/node             StrongLoop, Node.js, and tools.
➥   38                     [OK]
 kkarczmarczyk/node-yarn     Node docker image with yarn package manage...
➥   25                     [OK]
 bitnami/node                Bitnami Node.js Docker Image
➥   19                     [OK]
siomiz/node-opencv         _/node + node-opencv
➥   10                     [OK]
dahlb/alpine-node          small node for gitlab ci runner
➥   8                      [OK]
cusspvz/node               Super small Node.js container (~15MB) ba...
➥   7                      [OK]
anigeo/node-forever        Daily build node.js with forever
➥   4                      [OK]
seegno/node                A node docker base image.
➥   3                      [OK]
starefossen/ruby-node      Docker Image with Ruby and Node.js installed
➥   3                      [OK]
urbanmassage/node          Some handy (read, better) docker node images
➥   1                      [OK]
xataz/node                 very light node image
➥   1                      [OK]
centralping/node           Bare bones CentOS 7 NodeJS container.
➥   1                      [OK]
joxit/node                 Slim node docker with some utils for dev
➥   1                      [OK]
bigtruedata/node           Docker image providing Node.js & NPM
➥   1                      [OK]
```

```
1science/node                Node.js Docker images based on Alpine Linux
➥    1                      [OK]
domandtom/node               Docker image for Node.js including Yarn an...
➥    0                      [OK]
makeomatic/node              various alpine + node based containers
➥    0                      [OK]
c4tech/node                  NodeJS images, aimed at generated single-p...
➥    0                      [OK]
instructure/node             Instructure node images
➥    0                      [OK]
octoblu/node                 Docker images for node
➥    0                      [OK]
edvisor/node                 Automated build of Node.js with commonly u...
➥    0                      [OK]
watsco/node                  node:7
➥    0                      [OK]
codexsystems/node            Node.js for Development and Production
➥    0                      [OK]
```

Once you've chosen an image, you can download it by performing a `docker pull` command on the name:

```
$ docker pull node
 Using default tag: latest
latest: Pulling from library/node
5040bd298390: Already exists
fce5728aad85: Pull complete
76610ec20bf5: Pull complete
9c1bc3c30371: Pull complete
33d67d70af20: Pull complete
da053401c2b1: Pull complete
05b24114aa8d: Pull complete
Digest:
➥ sha256:ea65cf88ed7d97f0b43bcc5deed67cfd13c70e20a66f8b2b4fd4b7955de92297
Status: Downloaded newer image for node:latest
```

Pulls the image named `node` from the Docker Hub

This message is seen if Docker has pulled a new image (as opposed to identifying that there's no newer image than the one you already have). Your output may be different.

Then you can run it interactively using the `-t` and `-i` flags. The `-t` flag creates a TTY device (a terminal) for you, and the `-i` flag specifies that this Docker session is interactive:

```
$ docker run -t -i node /bin/bash
root@c267ae999646:/# node
> process.version
'v7.6.0'
>
```

TIP You can save keystrokes by replacing `-t -i` with `-ti` or `-it` in the preceding call to `docker run`. You'll see this throughout the book from here on.

Often there will be specific advice from the image maintainers about how the image should be run. Searching for the image on the http://hub.docker.com website will take you to the page for the image. The Description tab may give you more information.

WARNING If you download an image and run it, you're running code that you may not be able to fully verify. Although there is relative safety in using trusted images, nothing can guarantee 100% security when downloading and running software over the internet.

Armed with this knowledge and experience, you can now tap the enormous resources available on Docker Hub. With literally tens of thousands of images to try out, there's much to learn. Enjoy!

DISCUSSION

Docker Hub is an excellent resource, but sometimes it can be slow—it's worth pausing to decide how to best construct your Docker search command to get the best results. The ability to do searches without opening your browser offers you quick insight into possible items of interest in the ecosystem, so you can better target the documentation for images that are likely to fulfill your needs.

When you're rebuilding images, it can also be good to run a search every now and again to see if a stars count suggests that the Docker community has begun to gather around a different image than the one you're currently using.

Summary

- You can open up the Docker daemon API to outsiders, and all they need is some way to make a HTTP request—a web browser is enough.
- Containers don't have to take over your terminal. You can start them up in the background and come back to them later.
- You can make containers communicate, either with user-defined networks (the recommended approach), or with links to very explicitly control inter-container communication.
- Because the Docker daemon API is over HTTP, it's fairly easy to debug it with network monitoring tools if you're having issues.
- One particularly useful tool for debugging and tracing network calls is `socat`.
- Setting up registries isn't just the domain of Docker Inc.; you can set up your own on a local network for free private image storage.
- Docker Hub is a great place to go to find and download ready-made images, particularly ones provided officially by Docker Inc.