

docker0 interface like your home router. Each of the virtual interfaces created for containers is linked to docker0, and together they form a network. This bridge interface is attached to the network where the host is attached.

Using the `docker` command-line tool, you can customize the IP addresses used, the host interface that docker0 is connected to, and the way containers communicate with each other. The connections between interfaces describe how exposed or isolated any specific network container is from the rest of the network. Docker uses kernel namespaces to create those private virtual interfaces, but the namespace itself doesn't provide the network isolation. Network exposure or isolation is provided by the host's firewall rules (every modern Linux distribution runs a firewall). With the options provided, there are four archetypes for network containers.

5.2.2 Four network container archetypes

All Docker containers follow one of four archetypes. These archetypes define how a container interacts with other local containers and the host's network. Each serves a different purpose, and you can think of each as having a different level of isolation. When you use Docker to create a container, it's important to carefully consider what you want to accomplish and use the strongest possible container without compromising that goal. Figure 5.5 illustrates each archetype, where the strongest containers (most isolated) are on the left and the weakest are on the right.

The four archetypes are these:

- Closed containers
- Bridged containers
- Joined containers
- Open containers

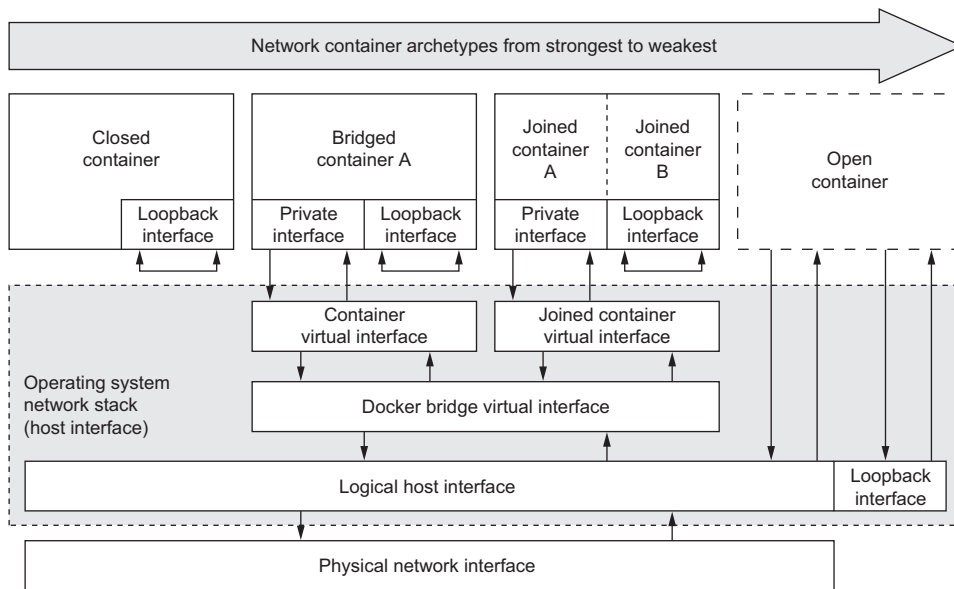


Figure 5.5 Four container network archetypes and their interaction with the Docker network topology

Over the next four subsections I introduce each archetype. Few readers will have an occasion to use all four. In reading about how to build them and when to use them, you'll be able to make that distinction yourself.

5.3 Closed containers

The strongest type of network container is one that doesn't allow any network traffic. These are called closed containers. Processes running in such a container will have access only to a loopback interface. If they need to communicate only with themselves or each other, this will be suitable. But any program that requires access to the network or the internet won't operate correctly in such a container. For example, if the software needs to download updates, it won't be able to because it can't use the network.

Most readers will be coming from a server software or web application background, and in that context it can be difficult to imagine a practical use for a container that has no network access. There are so many ways to use Docker that it's easy to forget about volume containers, backup jobs, offline batch processing, or diagnostic tools. The challenge you face is not justifying Docker for each feature but knowing which features best fit the use cases that you might be taking for granted.

Docker builds this type of container by simply skipping the step where an externally accessible network interface is created. As you can see in figure 5.6, the closed archetype has no connection to the Docker bridge interface. Programs in these containers can talk only to themselves.

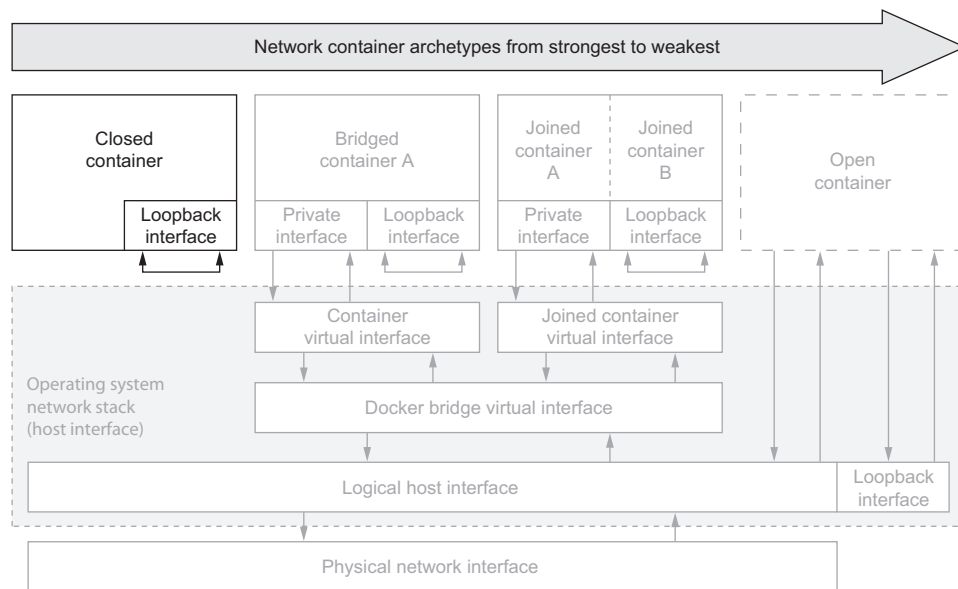


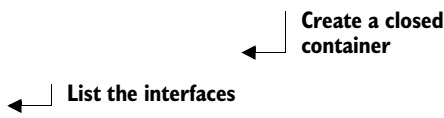
Figure 5.6 The closed container archetype and relevant components

All Docker containers, including closed containers, have access to a private loopback interface. You may have experience working with loopback interfaces already. It's common for people with moderate experience to have used `localhost` or `127.0.0.1` as an address in a URL. In these cases you were telling a program to bind to or contact a service bound to your computer's loopback network interface.

By creating private loopback interfaces for each container, Docker enables programs run inside a container to communicate through the network but without that communication leaving the container.

You can tell Docker to create a closed container by specifying `none` with the `--net` flag as an argument to the `docker run` command:

```
docker run --rm \
  --net none \
  alpine:latest \
  ip addr
```



Annotations for the first command:

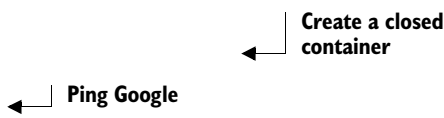
- ← Create a closed container (points to `--net none`)
- ← List the interfaces (points to `ip addr`)

Running this example, you can see that the only network interface available is the loopback interface, bound to the address `127.0.0.1`. This configuration means three things:

- Any program running in the container can connect to or wait for connections on that interface.
- Nothing outside the container can connect to that interface.
- No program running inside that container can reach anything outside the container.

That last point is important and easily demonstrable. If you're connected to the internet, try to reach a popular service that should always be available. In this case, try to reach Google's public DNS service:

```
docker run --rm \
  --net none \
  alpine:latest \
  ping -w 2 8.8.8.8
```



Annotations for the second command:

- ← Create a closed container (points to `--net none`)
- ← Ping Google (points to `ping -w 2 8.8.8.8`)

In this example you create a closed container and try to test the speed between your container and the public DNS server provided by Google. This attempt should fail with a message like “ping: send-to: Network is unreachable.” This makes sense because we know that the container has no route to the larger network.

When to use closed containers

Closed containers should be used when the need for network isolation is the highest or whenever a program doesn't require network access. For example, running a terminal text editor shouldn't require network access. Running a program to generate a random password should be run inside a container without network access to prevent the theft of that number.

There aren't many ways to customize the network configuration for a closed container. Although this type may seem overly limiting, it's the safest of the four options and can be extended to be more accommodating. These are not the default for Docker containers, but as a best practice you should try to justify using anything weaker before doing so. Docker creates bridged containers by default.

5.4 Bridged containers

Bridged containers relax network isolation and in doing so make it simpler to get started. This archetype is the most customizable and should be hardened as a best practice. Bridged containers have a private loopback interface and another private interface that's connected to the rest of the host through a network bridge.

This section is the longest of the chapter. Bridged containers are the most common network container archetype (see figure 5.7), and this section introduces several new options that you can use with other archetypes. Everything covered before section 5.6 is in the context of bridged containers.

All interfaces connected to `docker0` are part of the same virtual subnet. This means they can talk to each other and communicate with the larger network through the `docker0` interface.

5.4.1 Reaching out

The most common reason to choose a bridged container is that the process needs access to the network. To create a bridged container you can either omit the `--net`

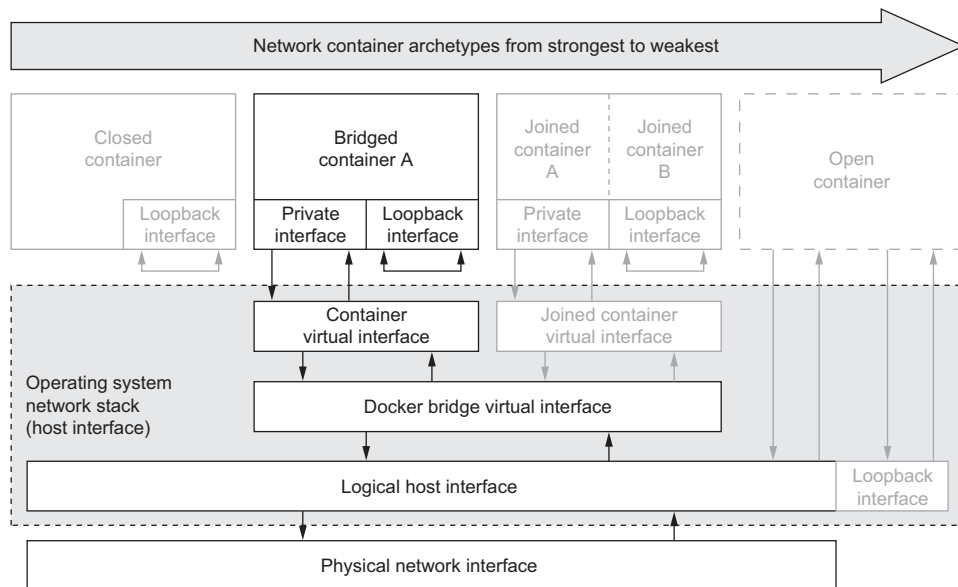


Figure 5.7 The bridged container archetype and relevant components

option to the `docker run` command or you can set its value to `bridge`. I use each form in the following examples:

```
docker run --rm \
  --net bridge \
  alpine:latest \
  ip addr
```

Join the bridge network

List the container interfaces

Just like the first example for closed containers, this command will create a new container from the latest alpine image and list the available network interfaces. This time it will list two interfaces: an Ethernet interface and a local loopback. The output will include details like the IP address and subnet mask of each interface, the maximum packet size (MTU), and various interface metrics.

Now that you've verified that your container has another interface with an IP address, try to access the network again. This time omit the `--net` flag to see that bridge is the default Docker network container type:

```
docker run --rm \
  alpine:latest \
  ping -w 2 8.8.8.8
```

Run ping command against Google

Note omission of the `--net` option

Pinging Google's public DNS server from this bridged container works, and no additional options are required. After running this command you'll see your container run a ping test for two seconds and report on the network statistics gathered.

Now you know that if you have some software that needs to access the internet, or some other computer on a private network, you can use a bridged container.

5.4.2 Custom name resolution

Domain Name System (DNS) is a protocol for mapping host names to IP addresses. This mapping enables clients to decouple from a dependency on a specific host IP and instead depend on whatever host is referred to by a known name. One of the most basic ways to change outbound communications is by creating names for IP addresses.

It is typical for containers on the bridge network and other computers on your network to have IP addresses that aren't publicly routable. This means that unless you're running your own DNS server, you can't refer to them by a name. Docker provides different options for customizing the DNS configuration for a new container.

First, the `docker run` command has a `--hostname` flag that you can use to set the host name of a new container. This flag adds an entry to the DNS override system inside the container. The entry maps the provided host name to the container's bridge IP address:

```
docker run --rm \
  --hostname barker \
  alpine:latest \
  nslookup barker
```

Set the container host name

Resolve the host name to an IP address

This example creates a new container with the host name `barker` and runs a program to look up the IP address for the same name. Running this example will generate output that looks something like the following:

```

Server:      10.0.2.3
Address 1: 10.0.2.3

Name:        barker
Address 1: 172.17.0.22 barker

```

The IP address on the last line is the bridge IP address for the new container. The IP address provided on the line labeled `Server` is the address of the server that provided the mapping.

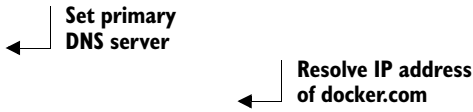
Setting the host name of a container is useful when programs running inside a container need to look up their own IP address or must self-identify. Because other containers don't know this hostname, its uses are limited. But if you use an external DNS server, you can share those hostnames.

The second option for customizing the DNS configuration of a container is the ability to specify one or more DNS servers to use. To demonstrate, the following example creates a new container and sets the DNS server for that container to Google's public DNS service:

```

docker run --rm \
  --dns 8.8.8.8 \
  alpine:latest \
  nslookup docker.com

```



Set primary
DNS server

Resolve IP address
of docker.com

Using a specific DNS server can provide consistency if you're running Docker on a laptop and often move between internet service providers. It's a critical tool for people building services and networks. There are a few important notes on setting your own DNS server:


- *The value must be an IP address.* If you think about it, the reason is obvious; the container needs a DNS server to perform the lookup on a name.
- *The `--dns=[]` flag can be set multiple times to set multiple DNS servers (in case one or more are unreachable).*
- *The `--dns=[]` flag can be set when you start up the Docker daemon that runs in the background.* When you do so, those DNS servers will be set on every container by default. But if you stop the daemon with containers running and change the default when you restart the daemon, the running containers will still have the old DNS settings. You'll need to restart those containers for the change to take effect.

The third DNS-related option, `--dns-search=[]`, allows you to specify a DNS search domain, which is like a default host name suffix. With one set, any host names that don't have a known top-level domain (like `.com` or `.net`) will be searched for with the specified suffix appended.

```

docker run --rm \
  --dns-search docker.com \
  busybox:latest \
  nslookup registry.hub

```



Set search
domain

Look up shortcut for
registry.hub.docker.com

This command will resolve to the IP address of `registry.hub.docker.com` because the DNS search domain provided will complete the host name.

This feature is most often used for trivialities like shortcut names for internal corporate networks. For example, your company might maintain an internal documentation wiki that you can simply reference at `http://wiki/`. But this can be much more powerful.

Suppose you maintain a single DNS server for your development and test environments. Rather than building environment-aware software (with hard-coded environment-specific names like `myservice.dev.mycompany.com`), you might consider using DNS search domains and using environment-unaware names (like `myservice`):

```
docker run --rm \
  --dns-search dev.mycompany \
  busybox:latest \
  nslookup myservice
```

← **Note dev prefix**

← **Resolves to myservice.dev.mycompany**

```
docker run --rm \
  --dns-search test.mycompany \
  busybox:latest \
  nslookup myservice
```

← **Note test prefix**

← **Resolves to myservice.test.mycompany**

Using this pattern, the only change is the context in which the program is running. Like providing custom DNS servers, you can provide several custom search domains for the same container. Simply set the flag as many times as you have search domains. For example:

```
docker run --rm \
  --dns-search mycompany \
  --dns-search myothercompany ...
```

This flag can also be set when you start up the Docker daemon to provide defaults for every container created. Again, remember that these options are only set for a container when it is created. If you change the defaults when a container is running, that container will maintain the old values.

The last DNS feature to consider provides the ability to override the DNS system. This uses the same system that the `--hostname` flag uses. The `--add-host=[]` flag on the `docker run` command lets you provide a custom mapping for an IP address and host name pair:

```
docker run --rm \
  --add-host test:10.10.10.255 \
  alpine:latest \
  nslookup test
```

← **Add host entry**

← **Resolves to 10.10.10.255**

Like `--dns` and `--dns-search`, this option can be specified multiple times. But unlike those other options, this flag can't be set as a default at daemon startup.

This feature is a sort of name resolution scalpel. Providing specific name mappings for individual containers is the most fine-grained customization possible. You can use this to effectively block targeted host names by mapping them to a known IP address like `127.0.0.1`. You could use it to route traffic for a particular destination through a

proxy. This is often used to route unsecure traffic through secure channels like an SSH tunnel. Adding these overrides is a trick that has been used for years by web developers who run their own local copies of a web application. If you spend some time thinking about the interface that name-to-IP address mappings provide, I'm sure you can come up with all sorts of uses.

All the custom mappings live in a file at `/etc/hosts` inside your container. If you want to see what overrides are in place, all you have to do is inspect that file. Rules for editing and parsing this file can be found online and are a bit beyond the scope of this book:

```

docker run --rm \
  --hostname mycontainer \
  --add-host docker.com:127.0.0.1 \
  --add-host test:10.10.10.2 \
  alpine:latest \
  cat /etc/hosts

```

Annotations for the command:

- Create host entry**: Points to `--add-host` options.
- Set host name**: Points to `--hostname mycontainer`.
- Create another host entry**: Points to the second `--add-host` option.
- View all entries**: Points to `cat /etc/hosts`.

This should produce output that looks something like the following:

```

172.17.0.45 mycontainer
127.0.0.1 localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
10.10.10.2 test
127.0.0.1 docker.com

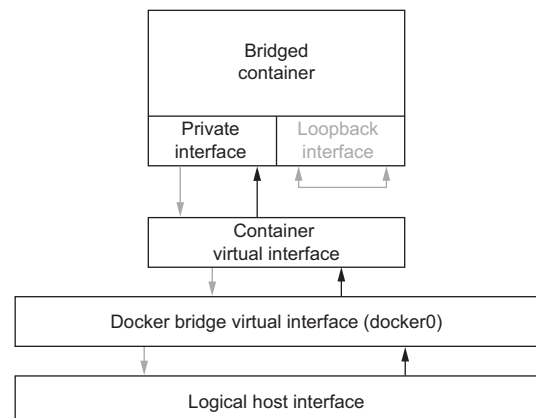
```

DNS is a powerful system for changing behavior. The name-to-IP address map provides a simple interface that people and programs can use to decouple themselves from specific network addresses. If DNS is your best tool for changing outbound traffic behavior, then the firewall and network topology is your best tool for controlling inbound traffic.

5.4.3 Opening inbound communication

Bridged containers aren't accessible from the host network by default. Containers are protected by your host's firewall system. The default network topology provides no route from the host's external interface to a container interface. That means there's just no way to get to a container from outside the host. The flow of inbound network traffic is shown in figure 5.8.

Figure 5.8 An inbound traffic route to a bridged container



Containers wouldn't be very useful if there were no way to get to them through the network. Luckily, that's not the case. The `docker run` command provides a flag, `-p=[]` or `--publish=[]`, that you can use to create a mapping between a port on the host's network stack and the new container's interface. You've used this a few times earlier in this book, but it's worth mentioning again. The format of the mapping can have four forms:

- `<containerPort>`

This form binds the container port to a dynamic port on all of the host's interfaces:

```
docker run -p 3333 ...
```

- `<hostPort>:<containerPort>`

This form binds the specified container port to the specified port on each of the host's interfaces:

```
docker run -p 3333:3333 ...
```

- `<ip>::<containerPort>`

This form binds the container port to a dynamic port on the interface with the specified IP address:

```
docker run -p 192.168.0.32::2222 ...
```

- `<ip>:<hostPort>:<containerPort>`

This form binds the container port to the specified port on the interface with the specified IP address:

```
docker run -p 192.168.0.32:1111:1111 ...
```

These examples assume that your host's IP address is 192.168.0.32. This is arbitrary but useful to demonstrate the feature. Each of the command fragments will create a route from a port on a host interface to a specific port on the container's interface. The different forms offer a range of granularity and control. This flag is another that can be repeated as many times as you need to provide the desired set of mappings.

The `docker run` command provides an alternate way to accomplish opening channels. If you can accept a dynamic or ephemeral port assignment on the host, you can use the `-P`, or `--publish-all`, flag. This flag tells the Docker daemon to create mappings, like the first form of the `-p` option for all ports that an image reports, to `expose`. Images carry a list of ports that are exposed for simplicity and as a hint to users where contained services are listening. For example, if you know that an image like `dockerinaction/ch5_expose` exposes ports 5000, 6000, and 7000, each of the following commands do the same thing:

```
docker run -d --name dawson \
  -p 5000 \
  -p 6000 \
  -p 7000 \
  dockerinaction/ch5_expose
```

Expose all ports

```
docker run -d --name woolery \
  -P \
  dockerinaction/ch5_expose
```

← **Expose relevant ports**

It's easy to see how this can save a user some typing, but it begs two questions. First, how is this used if the image doesn't expose the port you want to use? Second, how do you discover which dynamic ports were assigned?

The `docker run` command provides another flag, `--expose`, that takes a port number that the container should expose. This flag can be set multiple times, once for each port:

```
docker run -d --name philbin \
  --expose 8000 \
  -P \
  dockerinaction/ch5_expose
```

← **Expose another port**

← **Publish all ports**

Using `--expose` in this way will add port 8000 to the list of ports that should be bound to dynamic ports using the `-P` flag. After running the example, you can see what these ports were mapped to by using `docker ps`, `docker inspect`, or a new command, `docker port`. The `port` subcommand takes either the container name or ID as an argument and produces a simple list with one port map entry per line:

```
docker port philbin
```

Running this command should produce a list like the following:

```
5000/tcp -> 0.0.0.0:49164
6000/tcp -> 0.0.0.0:49165
7000/tcp -> 0.0.0.0:49166
8000/tcp -> 0.0.0.0:49163
```

With the tools covered in this section, you should be able to manage routing any inbound traffic to the correct bridged container running on your host. There's one other subtle type of communication: inter-container communication.

5.4.4 Inter-container communication

As a reminder, all the containers covered so far use the Docker bridge network to communicate with each other and the network that the host is on. All local bridged containers are on the same bridge network and can communicate with each other by default. Figure 5.9 illustrates the network relationship between five containers on the same host.

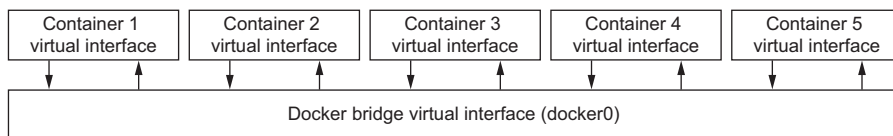


Figure 5.9 Five containers connected to the same Docker bridge (`docker0`)

In order to make sure that you have a full appreciation for this openness, the following command demonstrates how containers can communicate over this network:

```
docker run -it --rm dockerinaction/ch5_nmap -sS -p 3333 172.17.0.0/24
```

This command will run a program called `nmap` to scan all the interfaces attached to the bridge network. In this case it's looking for any interface that's accepting connections on port 3333. If you had such a service running in another container, this command would have discovered it, and you could use another program to connect to it.

Allowing communication in this way makes it simple to build cooperating containers. No additional work needs to be done to build pipes between containers. It's as free as an open network. This may be tolerable but can be risky for users who are unaware. It's common for software to ship with low-security features like default passwords or disabled encryption. Naïve users may expect that the network topology or some local firewall will protect containers from open access. This is true to some extent, but by default any container is fully accessible from any other local container.

When you start the Docker daemon, you can configure it to disallow network connections between containers. Doing so is a best practice in multi-tenant environments. It minimizes the points (called an attack surface) where an attacker might compromise other containers. You can achieve this by setting `--icc=false` when you start the Docker daemon:

```
docker -d --icc=false ...
```

When inter-container communication is disabled, any traffic from one container to another will be blocked by the host's firewall except where explicitly allowed. These exceptions are covered in section 5.4.

Disabling inter-container communication is an important step in any Docker-enabled environment. In doing so, you create an environment where explicit dependencies must be declared in order to work properly. At best, a more promiscuous configuration allows containers to be started when their dependencies aren't ready. At worst, leaving inter-container communication enabled allows compromised programs within containers to attack other local containers.

5.4.5 *Modifying the bridge interface*

Before moving on to the next archetype, this seems like an appropriate time to demonstrate the configuration options that modify the bridge interface. Outside this section, examples will always assume that you're working with the default bridge configuration.

Docker provides three options for customizing the bridge interface that the Docker daemon builds on first startup. These options let the user do the following:

- Define the address and subnet of the bridge
- Define the range of IP addresses that can be assigned to containers
- Define the maximum transmission unit (MTU)

To define the IP address of the bridge and the subnet range, use the `--bip` flag when you start the Docker daemon. There are all sorts of reasons why you might want to use a different IP range for your bridge network. When you encounter one of those situations, making the change is as simple as using one flag.

Using the `--bip` flag (which stands for *bridge IP*), you can set the IP address of the bridge interface that Docker will create and the size of the subnet using a classless inter-domain routing (CIDR) formatted address. CIDR notation provides a way to specify an IP address and its routing prefix. See appendix B for a brief primer on CIDR notation. There are several guides online detailing how to build CIDR formatted addresses, but if you're familiar with bit masking, the following example will be sufficient to get you started.

Suppose you want to set your bridge IP address to 192.168.0.128 and allocate the last 128 addresses in that subnet prefix to the bridge network. In that case, you'd set the value of `--bip` to 192.168.0.128/25. To be explicit, using this value will create the `docker0` interface, set its IP address to 192.168.0.128, and allow IP addresses that range from 192.168.0.128 to 192.168.0.255. The command would be similar to this:

```
docker -d --bip "192.168.0.128" ...
```

With a network defined for the bridge, you can go on to customize which IP addresses in that network can be assigned to new containers. To do so, provide a similar CIDR notation description to the `--fixed-cidr` flag.

Working from the previous situation, if you wanted to reserve only the last 64 addresses of the network assigned to the bridge interface, you would use 192.168.0.192/26. When the Docker daemon is started with this set, new containers will receive an IP address between 192.168.0.192 and 192.168.0.255. The only caveat with this option is that the range specified must be a subnet of the network assigned to the bridge (if you're confused, there's lots of great documentation and tooling on the internet to help):

```
docker -d --fixed-cidr "192.168.0.192/26"
```

I'm not going to spend too much effort on the last setting. Network interfaces have a limit to the maximum size of a packet (a packet is an atomic unit of communication). By protocol, Ethernet interfaces have a maximum packet size of 1500 bytes. This is the configured default. In some specific instances you'll need to change the MTU on the Docker bridge. When you encounter such a scenario, you can use the `--mtu` flag to set the size in bytes:

```
docker -d -mtu 1200
```

Users who are more comfortable with Linux networking primitives may like to know that they can provide their own custom bridge interface instead of using the default bridge. To do so, configure your bridge interface and then tell the Docker daemon to use it instead of `docker0` when you start the daemon. The flag to use is `-b` or `--bridge`.

If you've configured a bridge named `mybridge`, you'd start Docker with a command like the following:

```
docker -d -b mybridge ...
```

```
docker -d --bridge mybridge ...
```

Building custom bridges requires a deeper understanding of Linux kernel tools than is necessary for this book. But you should know that this ability is available if you do the research required.

5.5 *Joined containers*

The next less isolated network container archetype is called a joined container. These containers share a common network stack. In this way there's no isolation between joined containers. This means reduced control and security. Although this isn't the least secure archetype, it's the first one where the walls of a jail have been torn down.

Docker builds this type of container by providing access to the interfaces created for a specific container to another new container. Interfaces are in this way shared like managed volumes. Figure 5.10 shows the network architecture of two joined containers.

The easiest way to see joined containers in action is to use a special case and join it with a new container. The first command starts a server that listens on the loopback interface. The second command lists all the open ports. The second command lists

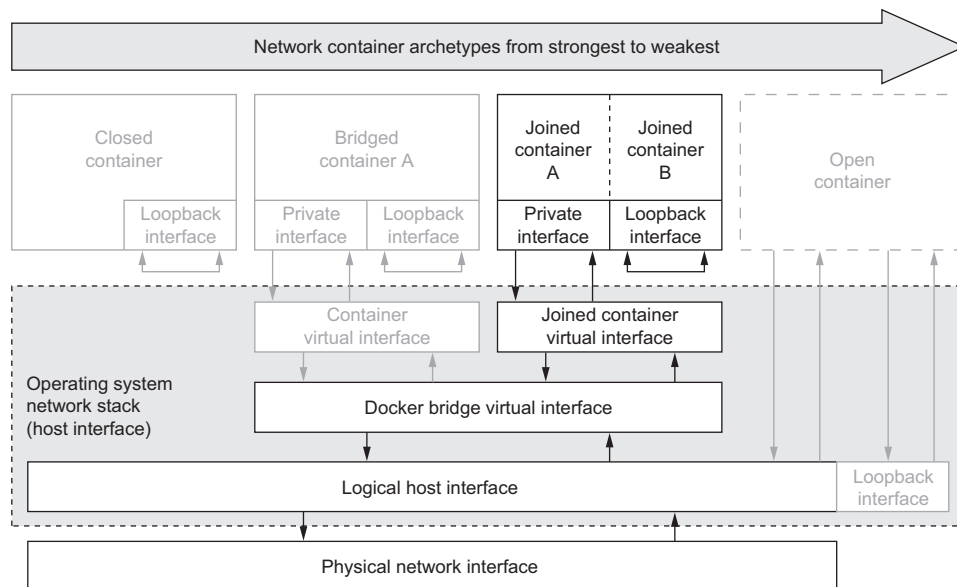


Figure 5.10 Two containers share the same bridge and loopback interface.

the open port created by the first command because both containers share the same network interface:

```
docker run -d --name brady \
  --net none alpine:latest \
  nc -l 127.0.0.1:3333

docker run -it \
  --net container:brady \
  alpine:latest netstat -al
```

By running these two commands you create two containers that share the same network interface. Because the first container is created as a closed container, the two will only share that single loopback interface. The `container` value of the `--net` flag lets you specify the container that the new container should be joined with. Either the container name or its raw ID identifies the container that the new container should reuse.

Containers joined in this way will maintain other forms of isolation. They will maintain different file systems, different memory, and so on. But they will have the exact same network components. That may sound concerning, but this type of container can be useful.

In the last example you joined two containers on a network interface that has no access to the larger network. In doing so, you expanded the usefulness of a closed container. You might use this pattern when two different programs with access to two different pieces of data need to communicate but shouldn't share direct access to the other's data. Alternatively, you might use this pattern when you have network services that need to communicate but network access or service discovery mechanisms like DNS are unavailable.

Setting aside security concerns, using joined containers reintroduces port conflict issues. A user should be aware of this whenever they're joining two containers. It's likely if they're joining containers that run similar services that they will create conflicts. Under those circumstances, the conflicts will need to be resolved using more traditional methods like changing application configuration. These conflicts can occur on any shared interfaces. When programs are run outside a container, they share access to the host's interfaces with every other program running on the computer, so this specific scope increase is still an improvement on today's status quo.

When two containers are joined, all interfaces are shared, and conflicts might happen on any of them. At first it might seem silly to join two containers that need bridge access. After all, they can already communicate over the Docker bridge subnet. But consider situations where one process needs to monitor the other through otherwise protected channels. Communication between containers is subject to firewall rules. If one process needs to communicate with another on an unexposed port, the best thing to do may be to join the containers.

The best reasons to use joined containers

Use joined containers when you want to use a single loopback interface for communication between programs in different containers.

Use joined containers if a program in one container is going to change the joined network stack and another program is going to use that modified network.

Use joined containers when you need to monitor the network traffic for a program in another container.

Before you start talking about how insecure Docker is because it allows any new container to join a running one, keep in mind that issuing any commands to Docker requires privileged access. Attackers with privileged access can do whatever they want, including attacking the code or data running in any container directly. In that context, this kind of network stack manipulation is no big deal.

In contexts where people build multi-tenant systems, it's a huge deal. If you're building or considering using such a service, the first thing you should do is set up multiple accounts and try to gain access to one from the other. If you can, think twice about using the service for anything important. Joining another user's network stack or mounting their volumes is a magnificent problem.

Joined containers are a bit weaker but are not the weakest type of network container. That title belongs to open containers.

5.6 Open containers

Open containers are dangerous. They have no network container and have full access to the host's network. This includes access to critical host services. Open containers provide absolutely no isolation and should be considered only in cases when you have no other option. The only redeeming quality is that unprivileged containers are still unable to actually reconfigure the network stack. Figure 5.11 shows the network architecture of an open container.

This type of container is created when you specify `host` as the value of the `--net` option on the `docker run` command:

```
docker run --rm \
  --net host \
  alpine:latest ip addr
```

← Create an open container

Running this command will create a container from the latest alpine image and without any network jail. When you execute `ip addr` inside that container, you can inspect all the host machine's network interfaces. You should see several interfaces listed, including one named `docker0`. As you may have noticed, this example creates a container that executes a discrete task and then immediately removes the container.

Using this configuration, processes can bind to protected network ports numbered lower than 1024.

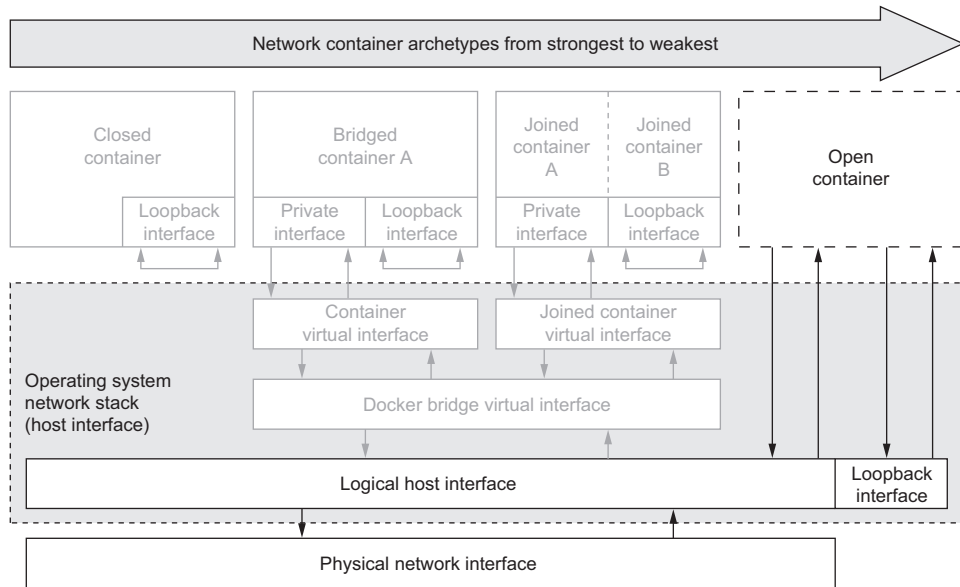


Figure 5.11 An open container is a container with full access to the host networking resources.

5.7 Inter-container dependencies

Now that you’ve learned what kind of network containers you can build with Docker and how those containers interact with the network, you need to learn how to use network software in one container from another. You’ve seen how containers use the bridge network to communicate and may have started thinking about how to piece together a small system. When you consider that the bridge network assigns IP addresses to containers dynamically at creation time, local service discovery can seem complicated.

One way to solve the problem would be to use a local DNS server and a registration hook when containers start. Another would be to write your programs to scan the local network for IP addresses listening on known ports. Both approaches handle dynamic environments but require a non-trivial workload and additional tooling. Each of these approaches will fail if arbitrary inter-container communication has been disabled. You could force all traffic out and back through the host’s interface to known published ports. But there are several occasions when you’ll need privileged access to a network port. Docker provides another tool that you’ve already seen to handle this use case.

5.7.1 Introducing links for local service discovery

When you create a new container, you can tell Docker to link it to any other container. That target container must be running when the new container is created. The reason

is simple. Containers hold their IP address only when they're running. If they're stopped, they lose that lease.

Adding a link on a new container does three things:

- Environment variables describing the target container's end point will be created.
- The link alias will be added to the DNS override list of the new container with the IP address of the target container.
- Most interestingly, if inter-container communication is disabled, Docker will add specific firewall rules to allow communication between linked containers.

The first two features of links are great for basic service discovery, but the third feature enables users to harden their local container networks without sacrificing container-to-container communication.

The ports that are opened for communication are those that have been exposed by the target container. So the `--expose` flag provides a shortcut for only one particular type of container to host port mapping when ICC is enabled. When ICC is disabled, `--expose` becomes a tool for defining firewall rules and explicit declaration of a container's interface on the network. In the same context, links become a more static declaration of local runtime service dependencies. Here's a simple example; these images don't actually exist:

```

docker run -d --name importantData \
  --expose 3306 \
  dockerinaction/mysql_noauth \
  service mysql_noauth start

docker run -d --name importantWebapp \
  --link importantData:db \
  dockerinaction/ch5_web startapp.sh -db tcp://db:3306

docker run -d --name buggyProgram \
  dockerinaction/ch5_buggy

```

Named target of a link

Create link and set alias to db

This container has no route to importantData.

Reading through this example, you can see that I've started some foolishly configured MySQL server (a popular database server). The name implies that the server has disabled any authentication requirements and anyone who can connect to the server can access the data. I then started an important web application that needs access to the data in the importantData container. I added a link from the importantWebapp container to the importantData container. Docker will add information to that new container that will describe how to connect to importantData. This way, when the web application opens a database connection to `tcp://db:3306`, it will connect to the database. Lastly, I started another container that's known to contain buggy code. It's running as a nonprivileged user, but it may be possible for an attacker to inspect the bridge network if the program is compromised.

If I'm running with inter-container communication enabled, attackers could easily steal the data from the database in the importantData container. They would be able to do a simple network scan to identify the open port and then gain access by simply opening a connection. Even a casual traffic observer might think this connection appropriate because no container dependencies have been strongly modeled.

If I were running this example with inter-container communication disabled, an attacker would be unable to reach any other containers from the container running the compromised software.

This is a fairly silly example. Please don't think that simply disabling inter-container communication will protect resources if those resources don't protect themselves. With appropriately configured software, strong network rules, and declared service dependencies, you can build systems that achieve good defense in depth.

5.7.2 Link aliases

Links are one-way network dependencies created when one container is created and specifies a link to another. As mentioned previously, the `--link` flag used for this purpose takes a single argument. That argument is a map from a container name or ID to an alias. The alias can be anything as long as it's unique in the scope of the container being created. So, if three containers named a, b, and c already exist and are running, then I could run the following:

```
docker run --link a:alias-a --link b:alias-b --link c:alias-c ...
```

But if I made a mistake and assigned some or all containers to the same alias, then that alias would only contain connection information for one of the other containers. In this case, the firewall rules would still be created but would be nearly useless without that connection information.

Link aliases create a higher-level issue. Software running inside a container needs to know the alias of the container or host it's connecting to so it can perform the lookup. Similar to host names, link aliases become a symbol that multiple parties must agree on for a system to operate correctly. Link aliases function as a contract.

A developer may build their application to assume that a database will have an alias of "database" and always look for it at `tcp://database:3306` because a DNS override with that host name would exist. This expected host name approach would work as long as the person or process building the container either creates a link aliased to a database or uses `--add-host` to create the host name. Alternatively, the application could always look for connection information from an environment variable named `DATABASE_PORT`. The environment variable approach will work only when a link is created with that alias.

The trouble is that there are no dependency declarations or runtime dependency checks. It's easy for the person building the container to do so without providing the required linkage. Docker users must either rely on documentation to communicate these dependencies or include custom dependency checking and fail-fast behavior on

container startup. I recommend building the dependency-checking code first. For example, the following script is included in [dockerinaction/ch5_ff](#) to validate that a link named “database” has been set at startup:

```
#!/bin/sh

if [ -z ${DATABASE_PORT+x} ]
then
    echo "Link alias 'database' was not set!"
    exit
else
    exec "$@"
fi
```

You can see this script at work by running the following:

```
docker run -d --name mydb --expose 3306 \
    alpine:latest nc -l 0.0.0.0:3306

docker run -it --rm \
    dockerinaction/ch5_ff echo This "shouldn't" work.

docker run -it --rm \
    --link mydb:wrongalias \
    dockerinaction/ch5_ff echo Wrong.

docker run -it --rm \
    --link mydb:database \
    dockerinaction/ch5_ff echo It worked.

docker stop mydb && docker rm mydb
```

Test without link

Create valid link target

Test with incorrect link alias

Test correct alias

Shut down link target container

This example script relies on the environment modifications made by Docker when links are created. You’ll find these very useful when you start building your own images in chapter 7.

5.7.3 *Environment modifications*

I’ve mentioned that creating a link will add connection information to a new container. This connection information is injected in the new container by adding environment variables and a host name mapping in the DNS override system. Let’s start with an example to inspect the link modifications:

```
docker run -d --name mydb \
    --expose 2222 --expose 3333 --expose 4444/udp \
    alpine:latest nc -l 0.0.0.0:2222

docker run -it --rm \
    --link mydb:database \
    dockerinaction/ch5_ff env

docker stop mydb && docker rm mydb
```

Create valid link target

Create link and list environment variables

This should output a block of lines that include the following:

```
DATABASE_PORT=tcp://172.17.0.23:3333
DATABASE_PORT_3333_TCP=tcp://172.17.0.23:3333
DATABASE_PORT_2222_TCP=tcp://172.17.0.23:2222
DATABASE_PORT_4444_UDP=udp://172.17.0.23:4444
DATABASE_PORT_2222_TCP_PORT=2222
DATABASE_PORT_3333_TCP_PORT=3333
DATABASE_PORT_4444_UDP_PORT=4444
DATABASE_PORT_3333_TCP_ADDR=172.17.0.23
DATABASE_PORT_2222_TCP_ADDR=172.17.0.23
DATABASE_PORT_4444_UDP_ADDR=172.17.0.23
DATABASE_PORT_2222_TCP_PROTO=tcp
DATABASE_PORT_3333_TCP_PROTO=tcp
DATABASE_PORT_4444_UDP_PROTO=udp
DATABASE_NAME=/furious_lalande/database
```

These are a sample of environment variables created for a link. All the variables relating to a specific link will use the link alias as a prefix. There will always be a single variable with the `_NAME` suffix that includes the name of the current container, a slash, and the link alias. For each port exposed by the linked container, there will be four individual environment variables with the exposed port in the variable name. The patterns are as follows:

- `<ALIAS>_PORT_<PORT NUMBER>_<PROTOCOL TCP or UDP>_PORT`

This variable will simply contain the port number. That is curious because the value will be contained in the variable name. This could be useful if you're filtering the list of environment variables for those containing the string `TCP_PORT`. Doing so would render the list of ports.

- `<ALIAS>_PORT_<PORT NUMBER>_<PROTOCOL TCP or UDP>_ADDR`

The values of variables with this pattern will be the IP address of the container serving the connection. If the alias is the same, these should all have the same value.

- `<ALIAS>_PORT_<PORT NUMBER>_<PROTOCOL TCP or UDP>_PROTO`

Like variables with the `_PORT` suffix, the values of these variables are actually contained within the variable name. It's important not to assume that the protocol will always be TCP. UDP is also supported.

- `<ALIAS>_PORT_<PORT NUMBER>_<PROTOCOL TCP or UDP>`

Variables of this form contain all the previous information encoded in URL form.

One additional environment variable of the form `<ALIAS>_<PORT>` will be created and will contain connection information for one of the exposed ports in URL form.

These environment variables are available for any need application developers might have in connecting to linked containers. But if developers have the port and protocol predetermined, then all they really need is host name resolution and they can rely on DNS for that purpose.

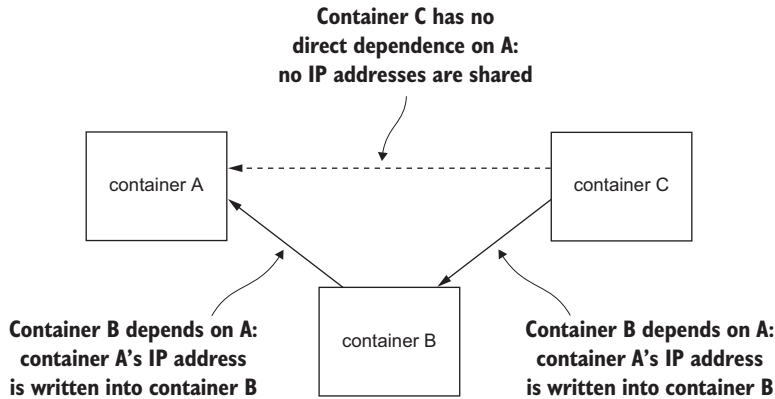


Figure 5.12 Links are not transitive. There's no link from container C to container A.

5.7.4 *Link nature and shortcomings*

The nature of links is such that dependencies are directional, static, and nontransitive. Nontransitive means that linked containers won't inherit links. More explicitly, if I link container B to container A, and then link container C to container B, there will be no link from container C to container A. Consider the container relationship in figure 5.12.

Links work by determining the network information of a container (IP address and exposed ports) and then injecting that into a new container. Because this is done at container creation time, and Docker can't know what a container's IP address will be before that container is running, links can only be built from new containers to existing containers. This is not to say that communication is one way but rather that discovery is one way. This also means that if a dependency stops for some reason, the link will be broken. Remember that containers maintain IP address leases only when they're running. So if a container is stopped or restarted, it will lose its IP lease and any linked containers will have stale data.

This property has caused some to criticize the value of links. The issue is that the deeper a dependency fails, the greater the domino effect of required container restarts. This might be an issue for some, but you must consider the specific impact.

If a critical service like a database fails, an availability event has already occurred. The chosen service discovery method impacts the recovery routine. An unavailable service might recover on the same or different IP address. Links will break only if the IP address changes and will require restarts. This leads some to jump to more dynamic lookup systems like DNS.

But even DNS systems have time-to-live (TTL) values that might slow the propagation of IP address changes. If an IP address changes during recovery, it might feel easier to use DNS, but recovery would only happen more quickly if connections to the

database can fail, reconnect attempts can time out, and the DNS TTL expires in less time than it takes to restart a container. In abandoning container linking, you'll be forced to enable inter-container communication.

If your applications are slow to start and you need to handle IP address changes on service recovery, you may want to consider DNS. Otherwise, consider the static dependency chain that has been modeled using container links. Building a system that restarts appropriate containers on a dependency failure would be an achievable exercise.

This chapter focused on single-host Docker networking, and in that scope links are incredibly useful tools. Most environments span more than one computer. Service portability is the idea that a service could be running on any machine, in any container in a larger environment. It's the idea that a system where any process might run anywhere is more robust than systems with strict locality constraints. I think this is true, but it's important to show how Docker can be used in either situation.

5.8 Summary

Networking is a broad subject that would take several books to properly cover. This chapter should help readers with a basic understanding of network fundamentals adopt the single-host networking facilities provided by Docker. In reading this material, you learned the following:

- Docker provides four network container archetypes: closed containers, bridged containers, joined containers, and open containers.
- Docker creates a bridge network that binds participating containers to each other and to the network that the host is attached to.
- The bridge interface created by Docker can be replaced or customized using `docker` command-line options when the Docker daemon is started.
- Options on the `docker run` command can be used to expose ports on a container's interface, bind ports exposed by a container to the host's network interface, and link containers to each other.
- Disabling arbitrary inter-container communication is simple and builds a system with defense in depth.
- Using links provides a low-overhead local service discovery mechanism and maps specific container dependencies.

Limiting risk with isolation

This chapter covers

- Setting resource limits
- Sharing container memory
- Users, permissions, and administrative privileges
- Granting access to specific Linux features
- Working with enhanced Linux isolation and security tools: SELinux and AppArmor

Containers provide isolated process contexts, not whole system virtualization. The semantic difference may seem subtle, but the impact is drastic. Chapter 1 touches on the differences a bit. Chapters 2 through 5 each cover a different isolation feature set of Docker containers. This chapter covers the remaining four and also includes information about enhancing security on your system.

The features covered in this chapter focus on managing or limiting the risks of running software. You will learn how to give containers resource allowances, open access to shared memory, run programs as specific users, control the type of changes that a container can make to your computer, and integrate with other

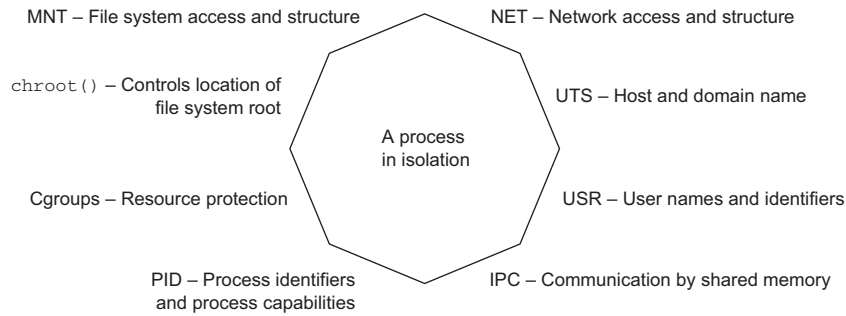


Figure 6.1 Eight-sided containers

Linux isolation tools. Some of these topics involve Linux features that are beyond the scope of this book. In those cases I try to give you an idea about their purpose and some basic usage examples, and you can integrate them with Docker. Figure 6.1 shows the eight namespaces and features that are used to build Docker containers.

One last reminder, Docker and the technology it uses are evolving projects. Once you get the learning tools presented in this chapter, remember to check for developments, enhancements, and new best practices when you go to build something valuable.

6.1 Resource allowances

Physical system resources like memory and time on the CPU are scarce. If the resource consumption of processes on a computer exceeds the available physical resources, the processes will experience performance issues and may stop running. Part of building a system that creates strong isolation includes providing resource allowances on individual containers.

If you want to make sure that a program won't overwhelm others on your computer, the easiest thing to do is set a limit on the resources that it can use. Docker provides three flags on the `docker run` and `docker create` commands for managing three different types of resource allowances that you can set on a container. Those three are memory, CPU, and devices.

6.1.1 Memory limits

Memory limits are the most basic restriction you can place on a container. They restrict the amount of memory that processes inside a container can use. Memory limits are useful for making sure that one container can't overwhelm the others running on a single system. You can put a limit in place by using the `-m` or `--memory` flag on the

`docker run` or `docker create` commands. The flag takes a value and a unit. The format is as follows:

`<number><optional unit>` where unit = b, k, m or g

In the context of these commands, `b` refers to bytes, `k` to kilobytes, `m` to megabytes, and `g` to gigabytes. Put this new knowledge to use and start up a database application that you'll use in other examples:

```
docker run -d --name ch6_mariadb \
  --memory 256m \
  --cpu-shares 1024
  --user nobody \
  --cap-drop all \
  dockerfile/mariadb
```

← **Set a memory constraint**

With this command you install database software called MariaDB and start a container with a memory limit of 256 megabytes. You might have noticed a few extra flags on this command. This chapter covers each of those, but you may already be able to guess what they do. Something else to note is that you don't expose any ports or bind any ports to the host's interfaces. It will be easiest to connect to this database by linking to it from another container. Before we get to that, I want to make sure you have a full understanding of what happens here and how to use memory limits.

The most important thing to understand about memory limits is that they're not reservations. They don't guarantee that the specified amount of memory will be available. They're only a protection from overconsumption.

Before you put a memory allowance in place, you should consider two things. First, can the software you're running operate under the proposed memory allowance? Second, can the system you're running on support the allowance?

The first question is often difficult to answer. It's not common to see minimum requirements published with open source software these days. Even if it were, though, you'd have to understand how the memory requirements of the software scale based on the size of the data you're asking it to handle. For better or worse, people tend to overestimate and adjust based on trial and error. In the case of memory-sensitive tools like databases, skilled professionals such as database administrators can make better-educated estimates and recommendations. Even then, the question is often answered by another: how much memory do you have? And that leads to the second question.

Can the system you're running on support the allowance? It's possible to set a memory allowance that's bigger than the amount of available memory on the system. On hosts that have swap space (virtual memory that extends onto disk), a container may realize the allowance. It's always possible to impose an allowance that's greater than any physical memory resource. In those cases the limitations of the system will always cap the container.

Finally, understand that there are several ways that software can fail if it exhausts the available memory. Some programs may fail with a memory access fault, whereas others may start writing out-of-memory errors to their logging. Docker neither detects this problem nor attempts to mitigate the issue. The best it can do is apply the restart logic you may have specified using the `--restart` flag described in chapter 2.


6.1.2 CPU

Processing time is just as scarce as memory, but the effect of starvation is performance degradation instead of failure. A paused process that is waiting for time on the CPU is still working correctly. But a slow process may be worse than a failing one if it's running some important data-processing program, a revenue-generating web application, or a back-end service for your app. Docker lets you limit a container's CPU resources in two ways.

First, you can specify the relative weight of a container. Linux uses this to determine the percentage of CPU time the container should use relative to other running containers. That percentage is for the sum of the computing cycles of all processors available to the container.

To set the CPU shares of a container and establish its relative weight, both `docker run` and `docker create` offer a `--cpu-shares` flag. The value provided should be an integer (which means you shouldn't quote it). Start another container to see how CPU shares work:

```
docker run -d -P --name ch6_wordpress \
--memory 512m \
--cpu-shares 512 \
--user nobody \
--cap-drop net_raw \
--link ch6_mariadb \
wordpress:4.1
```



Set a relative process weight

This command will download and start WordPress version 4.1. It's written in PHP and is a great example of software that has been challenged by adapting to security risks. Here we've started it with a few extra precautions. If you'd like to see it running on your computer, use `docker port ch6_wordpress` to get the port number (I'll call it `<port>`) that the service is running on and open <http://localhost:<port>> in your web browser. Remember, if you're using Boot2Docker, you'll need to use `boot2docker ip` to determine the IP address of the virtual machine where Docker is running. When you have that, substitute that value for localhost in the preceding URL.

When you started the MariaDB container, you set its relative weight (`cpu-shares`) to 1024, and you set the relative weight of WordPress to 512. These settings create a system where the MariaDB container gets two CPU cycles for every one WordPress cycle. If you started a third container and set its `--cpu-shares` value to 2048, it would get half of the CPU cycles, and MariaDB and WordPress would split the other half at

Total shares: 1536	MariaDB @1024 or ~66%	WordPress @512 or ~33%	
Total shares: 3584	MariaDB @1024 or ~28%	WordPress @512 or ~14%	A third container @2048 or ~57%

Figure 6.2 Relative weight and CPU shares

the same proportions as they were before. Figure 6.2 shows how portions change based on the total weight of the system.

CPU shares differ from memory limits in that they're enforced only when there is contention for time on the CPU. If other processes and containers are idle, then the container may burst well beyond its limits. This is preferable because it makes sure that CPU time is not wasted and that limited processes will yield if another process needs the CPU. The intent of this tool is to prevent one or a set of processes from overwhelming a computer, not to hinder performance of those processes. The defaults won't limit the container, and it will be able to use 100% of the CPU.

The second feature Docker exposes is the ability to assign a container to a specific CPU set. Most modern hardware uses multi-core CPUs. Roughly speaking, a CPU can process as many instructions in parallel as it has cores. This is especially useful when you're running many processes on the same computer.

A context switch is the task of changing from executing one process to executing another. Context switching is expensive and may cause a noticeable impact on the performance of your system. In some cases it may make sense to try to minimize context switching by making sure that some critical processes are never executed on the same set of CPU cores. You can use the `--cpuset-cpus` flag on `docker run` or `docker create` to limit a container to execute only on a specific set of CPU cores.

You can see the CPU set restrictions in action by stressing one of your machine cores and examining your CPU workload:

```
# Start a container limited to a single CPU and run a load generator
docker run -d \
    --cpuset-cpus 0 \
    --name ch6_stresser dockerinaction/ch6_stresser
```

← Restrict to CPU number 0

```
# Start a container to watch the load on the CPU under load
docker run -it --rm dockerinaction/ch6_htop
```

Once you run the second command, you'll see `htop` display the running processes and the workload of the available CPUs. The `ch6_stresser` container will stop running after 30 seconds, so it's important not to delay when you run this experiment.

When you finish with `htop`, press `Q` to quit. Before moving on, remember to shut down and remove the container named `ch6_stresser`:

```
docker rm -vf ch6_stresser
```

I thought this was exciting when I first used it. To get the best appreciation, repeat this experiment a few different times using different values for the `--cpuset-cpus` flag. If you do, you'll see the process assigned to different cores or different sets of cores. The value can be either list or range forms:

- `0,1,2`—A list including the first three cores of the CPU
- `0-2`—A range including the first three cores of the CPU

6.1.3 Access to devices

Devices are the last resource type. This control differs from memory and CPU limits in that access to devices is not a limit. This is more like resource authorization control.

Linux systems have all sorts of devices, including hard drives, optical drives, USB drives, mouse, keyboard, sound devices, and webcams. Containers have access to some of these devices by default, and Docker creates others for each container (like virtual terminals).

On occasion it may be important to share other devices between a host and a specific container. Consider a situation where you're running some computer vision software that requires access to a webcam. In that case you'll need to grant access to the container running your software to the webcam device attached to the system; you can use the `--device` flag to specify a set of devices to mount into the new container. The following example would map your webcam at `/dev/video0` to the same location within a new container. Running this example will work only if you have a webcam at `/dev/video0`:

```
docker -it --rm \
  --device /dev/video0:/dev/video0 \
  ubuntu:latest ls -al /dev
```

← Mount video0

The value provided must be a map between the device file on the host operating system and the location inside the new container. The device flag can be set many times to grant access to different devices.

People in situations with custom hardware or proprietary drivers will find this kind of access to devices useful. It's preferable to resorting to modifying their host operating system.

6.2 Shared memory

Linux provides a few tools for sharing memory between processes running on the same computer. This form of inter-process communication (IPC) performs at memory speeds. It's often used when the latency associated with network or pipe-based IPC drags software performance down below requirements. The best examples of shared

memory-based IPC use are in scientific computing and some popular database technologies like PostgreSQL.

Docker creates a unique IPC namespace for each container by default. The Linux IPC namespace partitions shared memory primitives such as named shared memory blocks and semaphores, as well as message queues. It's okay if you're not sure what these are. Just know that they're tools used by Linux programs to coordinate processing. The IPC namespace prevents processes in one container from accessing the memory on the host or in other containers.

6.2.1 *Sharing IPC primitives between containers*

I've created an image named `dockerinactionch6_ipc` that contains both a producer and consumer. They communicate using shared memory. The following will help you understand the problem with running these in separate containers:

```
docker -d -u nobody --name ch6_ipc_producer \
    dockerinaction/ch6_ipc -producer      ← Start producer
docker -d -u nobody --name ch6_ipc_consumer \
    dockerinaction/ch6_ipc -consumer      ← Start consumer
```

These commands start two containers. The first creates a message queue and begins broadcasting messages on it. The second should pull from the message queue and write the messages to the logs. You can see what each is doing by using the following commands to inspect the logs of each:

```
docker logs ch6_ipc_producer
docker logs ch6_ipc_consumer
```

Notice that something is wrong with the containers you started. The consumer never sees any messages on the queue. Each process used the same key to identify the shared memory resource, but they referred to different memory. The reason is that each container has its own shared memory namespace.

If you need to run programs that communicate with shared memory in different containers, then you'll need to join their IPC namespaces with the `--ipc` flag. The `--ipc` flag has a container mode that will create a new container in the same IPC namespace as another target container. This is just like the `--net` flag covered in chapter 5. Figure 6.3 illustrates the relationship between containers and their namespaced shared memory pools.

Use the following commands to test joined IPC namespaces for yourself:

```
Start new consumer → docker rm -v ch6_ipc_consumer      ← Remove original consumer
                    docker -d --name ch6_ipc_consumer \
                      --ipc container:ch6_ipc_producer \
                      dockerinaction/ch6_ipc -consumer ← Join IPC namespace
```

These commands rebuild the consumer container and reuse the IPC namespace of the `ch6_ipc_producer` container. This time the consumer should be able to access the

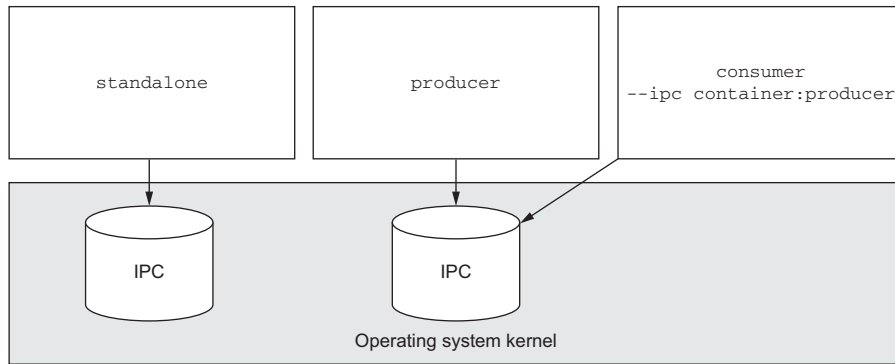


Figure 6.3 Three containers and their shared memory pools. Producer and consumer share a single pool.

same memory location where the server is writing. You can see this working by using the following commands to inspect the logs of each:

```
docker logs ch6_ipc_producer
```

```
docker logs ch6_ipc_consumer
```

Remember to clean up your running containers before moving on:

- The `v` option will clean up volumes.
- The `f` option will kill the container if it is running.
- The `rm` command takes a list of containers.

```
docker rm -vf ch6_ipc_producer ch6_ipc_consumer
```

There are obvious security implications to reusing the shared memory namespaces of containers. But this option is available if you need it. Sharing memory between containers is a safer alternative than sharing memory with the host.

6.2.2 Using an open memory container

Memory isolation is a desirable trait. If you encounter a situation where you need to operate in the same namespace as the rest of the host, you can do so using an open memory container:

```
docker -d --name ch6_ipc_producer \
  --ipc host \
  dockerinaction/ch6_ipc -producer
```

← Start a producer

← Use open memory container

```
docker -d --name ch6_ipc_consumer \
  --ipc host \
  dockerinaction/ch6_ipc -consumer
```

← Start a consumer

← Use open memory container

These containers will be able to communicate with each other and any other processes running on the host computer immediately. As you can see in this example, you

enable this feature by specifying `host` on the `--ipc` flag. You might use this in cases when you need to communicate with a process that must run on the host, but in general you should try to avoid this if possible.

Feel free to check out the source code for this example. It's an ugly but simple C program. You can find it by checking out the source repository linked to from the image's page on Docker Hub.

You can clean up the containers you created in this section using the same cleanup command as in section 6.2.1:

```
docker rm -vf ch6_ipc_producer ch6_ipc_consumer
```

Open memory containers are a risk, but it's a far better idea to use them than to run those processes outside a container.

6.3 *Understanding users*

Docker starts containers as the root user inside that container by default. The root user has almost full privileged access to the state of the container. Any processes running as that user inherit those permissions. It follows that if there's a bug in one of those processes, they might damage the container. There are ways to limit the damage, but the most effective way to prevent these types of issues is not to use the root user.

There are reasonable exceptions when using the root user is the best if not only available option. You use the root user for building images and at runtime when there's no other option. There are other similar situations when you want to run system administration software inside a container. In those cases the process needs privileged access not only to the container but also to the host operating system. This section covers the range of solutions to these problems.

6.3.1 *Introduction to the Linux user namespace*

Linux recently released a new user (USR) namespace that allows users in one namespace to be mapped to users in another. The new namespace operates like the process identifier (PID) namespace.

Docker hasn't yet been integrated with the USR namespace. This means that a container running with a user ID (number, not name) that's the same as a user on the host machine has the same host file permissions as that user. This isn't a problem. The file system available inside a container has been mounted in such a way that changes that are made inside that container will stay inside that container's file system. But this does impact volumes.

When Docker adopts the USR namespace, you'll be able to map user IDs on the host to user IDs in a container namespace. So, I could map user 1000 on my host to user 2 in the container. This is particularly useful for resolving file permissions issues in cases like reading and writing to volumes.

6.3.2 Working with the run-as user

Before you create a container, it would be nice to be able to tell what username (and user ID) is going to be used by default. The default is specified by the image. There's currently no way to examine an image to discover attributes like the default user. This information is not included on Docker Hub. And there's no command to examine image metadata.

The closest feature available is the `docker inspect` command. If you missed it in chapter 2, the `inspect` subcommand displays the metadata of a specific container. Container metadata includes the metadata of the image it was created from. Once you've created a container—let's call it `bob`—you can get the username that the container is using with the following commands:

```
docker create --name bob busybox:latest ping localhost
```

Display all
of bob's
metadata

```
docker inspect bob
```

```
docker inspect --format "{{.Config.User}}" bob
```

Show only run-as user
defined by bob's image

If the result is blank, the container will default to running as the root user. If it isn't blank, either the image author specifically named a default run-as user or you set a specific run-as user when you created the container. The `--format` or `-f` option used in the second command allows you to specify a template to render the output. In this case you've selected the `User` field of the `Config` property of the document. The value can be any valid GoLang template, so if you're feeling up to it, you can get creative with the results.

There are problems with this approach. First, the run-as user might be changed by whatever script the image uses to start up. These are sometimes referred to as boot, or init, scripts. The metadata returned by `docker inspect` includes only the configuration that the container was started with. So if the user changes, it won't be reflected there. Second, you have to create a container from an image in order to get the information. That can be dangerous.

Currently, the only way to fix both problems would be to look inside the image. You could expand the image files after you download them and examine the metadata and init scripts by hand, but doing so is time-consuming and easy to get wrong. For the time being, it may be better to run a simple experiment to determine the default user. This will solve the first problem but not the second:

Outputs: root

```
docker run --rm --entrypoint "" busybox:latest whoami
```

```
docker run --rm --entrypoint "" busybox:latest id
```

Outputs: uid=0(root) gid=0(root)
groups=l0(wheel)

This demonstrates two commands that you might use to determine the default user of an image (in this case, `busybox:latest`). Both the `whoami` and `id` commands are common among Linux distributions, and so they're likely to be available in any given

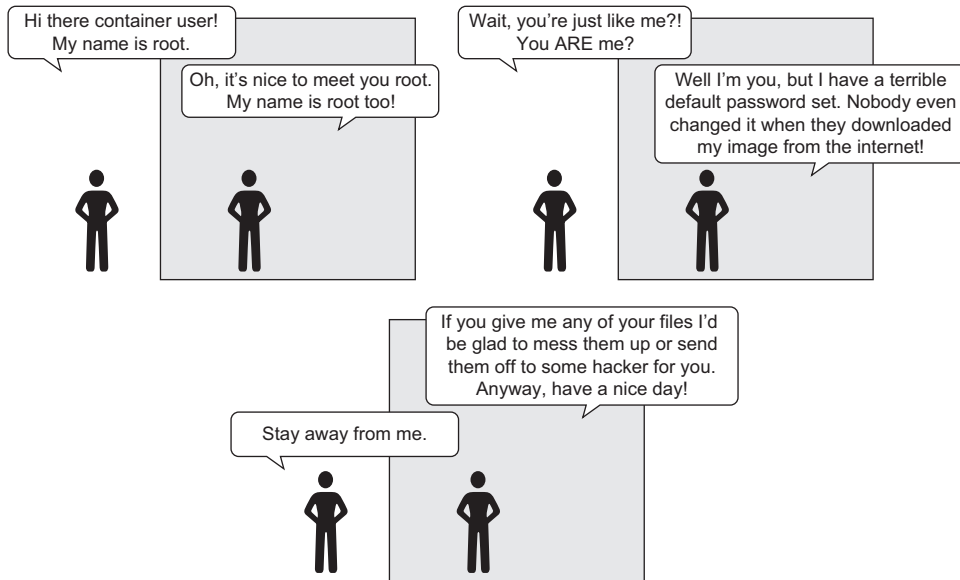


Figure 6.4 Root vs. root—a security drama

image. The second command is superior because it shows both the name and ID details for the run-as user. Both these commands are careful to unset the `entrypoint` of the container. This will make sure that the command specified after the image name is the command that is executed by the container. These are poor substitutes for a first-class image metadata tool, but they get the job done. Consider the brief exchange between two root users in figure 6.4.

You can entirely avoid the default user problem if you change the run-as user when you create the container. The quirk with using this is that the username must exist on the image you're using. Different Linux distributions ship with different users predefined, and some image authors reduce or augment that set. You can get a list of available users in an image with the following command:

```
docker run --rm busybox:latest awk -F: '$0=$1' /etc/passwd
```

I won't go into much detail here, but the Linux user database is stored in a file located at `/etc/passwd`. This command will read that file and pull a list of the usernames. Once you've identified the user you want to use, you can create a new container with a specific run-as user. Docker provides the `--user` or `-u` flag on `docker run` and `docker create` for setting the user. This will set the user to “nobody”:

```
docker run --rm \
  --user nobody \
  busybox:latest id
```

← Set run-as user to nobody

← Outputs: uid=99(nobody)
gid=99(nobody)

This command used the “nobody” user. That user is very common and intended for use in restricted-privileges scenarios like running applications. That was just one

example. You can use any username defined by the image here, including root. This only scratches the surface of what you can do with the `-u` or `--user` flag. The value can accept any user or group pair. It can also accept user and group names or IDs. When you use IDs instead of names, the options start to open up:

```

Set run-as user to nobody and group to default → docker run --rm \
    -u nobody:default \
    busybox:latest id
    ← Outputs: uid=99(nobody) gid=1000(default)

docker run --rm \
    -u 10000:20000 \
    busybox:latest id
    ← Set UID and GID
    ← Outputs: uid=10000 gid=20000
  
```

The second command starts a new container that sets the run-as user and group to a user and group that do not exist in the container. When that happens, the IDs won't resolve to a user or group name, but all file permissions will work as if the user and group did exist. Depending on how the software packaged in the container is configured, changing the run-as user may cause problems. Otherwise, this is a powerful feature that can make file-permission issues simple to resolve.

You should be very careful about which users on your systems are able to control your Docker daemon. If a user can control your Docker daemon, that person can effectively control the root account on your host.

The best way to be confident in your runtime configuration is to pull images from trusted sources or build your own. It's entirely possible to do malicious things like turning a default non-root user into the root user (making your attempt at safety a trap) or opening up access to the root account without authentication. Chapter 7 covers this topic briefly. For now, I'll wrap up with another interesting example:

```

docker run -it --name escalation -u nobody busybox:latest \
    /bin/sh -c "whoami; su -c whoami"
    ← Outputs: "nobody" and then "root"
  
```

That was too easy. The official BusyBox image has no password set for root (or many other accounts). The official Ubuntu image ships with the account locked; you'd have to start the container with root or promote via an SUID binary (more on these in chapter 7). The impact of such weak authentication is that a process running as any user could self-promote to root in the container. The lesson here is that you might consider learning to start setting passwords or disabling the root account like Ubuntu and others. This means modifying images, so even if you're not a software author, you'll benefit from reading part 2 of this book.

6.3.3 Users and volumes

Now that you've learned how users inside containers share the same user ID space as the users on your host system, you need to learn how those two might interact. The main reason for that interaction is the file permissions on files in volumes. For example, if you're running a Linux terminal, you should be able to use these commands

directly; otherwise, you'll need to use the `boot2docker ssh` command to get a shell in your Boot2Docker virtual machine:

```

echo "e=mc^2" > garbage
chmod 600 garbage
sudo chown root:root garbage
docker run --rm -v "$(pwd)"/garbage:/test/garbage \
  -u nobody \
  ubuntu:latest cat /test/garbage
docker run --rm -v "$(pwd)"/garbage:/test/garbage \
  -u root ubuntu:latest cat /test/garbage
# Outputs: "e=mc^2"
# cleanup that garbage
sudo rm -f garbage

```

Make file readable only by its owner →

← **Create new file on your host**

← **Make file owned by root (assuming you have sudo access)**

← **Try to read file as nobody**

← **Try to read file as "container root"**

The second-to-last `docker` command should fail with an error message like “Permission denied.” But the last `docker` command should succeed and show you the contents of the file you created in the first command. This means that file permissions on files in volumes are respected inside the container. But this also reflects that the user ID space is shared. Both root on the host and root in the container have user ID 0. So, although the container’s nobody user with ID 65534 can’t access a file owned by root on the host, the container’s root user can.

Unless you want a file to be accessible to a container, don’t mount it into that container with a volume.

The good news about this example is that you’ve seen how file permissions are respected and can solve some more mundane—but practical—operational issues. For example, how do you handle a log file written to a volume?

The preferred way is with volume containers, as described in chapter 4. But even then you need to consider file ownership and permission issues. If logs are written to a volume by a process running as user 1001 and another container tries to access that file as user 1002, then file permissions might prevent the operation.

One way to overcome this obstacle would be to specifically manage the user ID of the running user. You can either edit the image ahead of time by setting the user ID of the user you’re going to run the container with, or you can use the desired user and group ID:

```

mkdir logFiles
sudo chown 2000:2000 logFiles
docker run --rm -v "$(pwd)"/logFiles:/logFiles \
  -u 2000:2000 ubuntu:latest \
  /bin/bash -c "echo This is important info > /logFiles/important.log"

```

Write important log file →

← **Set ownership of directory to desired user and group**

← **Set UID:GID to 2000:2000**

Append to log
from another
container

```
docker run --rm -v "$(pwd)"/logFiles:/logFiles \
  -u 2000:2000 ubuntu:latest \
  /bin/bash -c "echo More info >> /logFiles/important.log"
```

Also set UID:GID to 2000:2000

```
sudo rm -r logFiles
```

After running this example, you'll see that the file could be written to the directory that's owned by user 2000. Not only that, but any container that uses a user or group with write access to the directory could write a file in that directory or to the same file if the permissions allow. This trick works for reading, writing, and executing files.

6.4 Adjusting OS feature access with capabilities

Docker can adjust the feature authorization of processes within containers. In Linux these feature authorizations are called capabilities, but as native support expands to other operating systems, other back-end implementations would need to be provided. Whenever a process attempts to make a gated system call, the capabilities of that process are checked for the required capability. The call will succeed if the process has the required capability and fail otherwise.

When you create a new container, Docker drops a specific set of capabilities by default. This is done to further isolate the running process from the administrative functions of the operating system. In reading this list of dropped capabilities, you might be able to guess at the reason for their removal. At the time of this writing, this set includes the following:

- *SETPCAP*—Modify process capabilities
- *SYS_MODULE*—Insert/remove kernel modules
- *SYS_RAWIO*—Modify kernel memory
- *SYS_PACCT*—Configure process accounting
- *SYS_NICE*—Modify priority of processes
- *SYS_RESOURCE*—Override resource limits
- *SYS_TIME*—Modify the system clock
- *SYS_TTY_CONFIG*—Configure TTY devices
- *AUDIT_WRITE*—Write the audit log
- *AUDIT_CONTROL*—Configure audit subsystem
- *MAC_OVERRIDE*—Ignore kernel MAC policy
- *MAC_ADMIN*—Configure MAC configuration
- *SYSLOG*—Modify kernel print behavior
- *NET_ADMIN*—Configure the network
- *SYS_ADMIN*—Catchall for administrative functions

The default set of capabilities provided to Docker containers provides a reasonable feature reduction, but there will be times when you need to add or reduce this set further. For example, the capability *NET_RAW* can be dangerous. If you wanted to be a bit more careful than the default configuration, you could drop *NET_RAW* from the list of

capabilities. You can drop capabilities from a container using the `--cap-drop` flag on `docker create` or `docker run`.

```
docker run --rm -u nobody \
  ubuntu:latest \
  /bin/bash -c "capsh --print | grep net_raw"
```

```
docker run --rm -u nobody \
  --cap-drop net_raw \
  ubuntu:latest \
  /bin/bash -c "capsh --print | grep net_raw"
```

← Drop NET_RAW
capability

In Linux documentation you'll often see capabilities named in all uppercase and prefixed with `CAP_`, but that prefix won't work if provided to the capability-management options. Use unprefixed and lowercase names for the best results.

Similar to the `--cap-drop` flag, the `--cap-add` flag will add capabilities. If you needed to add the `SYS_ADMIN` capability for some reason, you'd use a command like the following:

```
docker run --rm -u nobody \
  ubuntu:latest \
  /bin/bash -c "capsh --print | grep sys_admin"
```

← SYS_ADMIN is
not included

```
docker run --rm -u nobody \
  --cap-add sys_admin \
  ubuntu:latest \
  /bin/bash -c "capsh --print | grep sys_admin"
```

← Add SYS_ADMIN

Like other container-creation options, both `--cap-add` and `--cap-drop` can be specified multiple times to add or drop multiple capabilities. These flags can be used to build containers that will let a process perform exactly and only what is required for proper operation.

6.5 *Running a container with full privileges*

In those cases when you need to run a system administration task inside a container, you can grant that container privileged access to your computer. Privileged containers maintain their file system and network isolation but have full access to shared memory and devices and possess full system capabilities. You can perform several interesting tasks, like running Docker inside a container, with privileged containers.

The bulk of the uses for privileged containers is administrative. Take, for example, an environment where the root file system is read-only, or installing software outside a container has been disallowed, or you have no direct access to a shell on the host. If you wanted to run a program to tune the operating system (for something like load balancing) and you had access to run a container on that host, then you could simply run that program in a privileged container.

If you find a situation that can be solved only with the reduced isolation of a privileged container, use the `--privileged` flag on `docker create` or `docker run` to enable this mode:

```
docker run --rm \
  --privileged \
  ubuntu:latest id
```

← Check out our IDs

```
docker run --rm \
  --privileged \
  ubuntu:latest capsh -print
```

← Check out our Linux capabilities

```
docker run --rm \
  --privileged \
  ubuntu:latest ls /dev
```

← Check out list of mounted devices

```
docker run --rm \
  --privileged \
  ubuntu:latest ifconfig
```

← Examine network configuration

Privileged containers are still partially isolated. For example, the network namespace will still be in effect. If you need to tear down that namespace, you'll need to combine this with `--net host` as well.

6.6 Stronger containers with enhanced tools

Docker uses reasonable defaults and a “batteries included” toolset to ease adoption and promote best practices. But you can enhance the containers it builds if you bring additional tools. Tools you can use to harden your containers include AppArmor and SELinux. If you use the LXC container provider, you can also provide custom LXC configuration and get into fine-tuning containers. If you're using LXC, you can even use a Linux feature called `seccomp-bpf` (secure computing with system call filtering).

Whole books have been written about each of these tools. They bring their own nuances, benefits, and required skillsets. Their use is—without question—worth the effort. Support for each varies by Linux distribution, so you may be in for a bit of work. But once you've adjusted your host configuration, the Docker integration is simpler.

Security research

The information security space is very complicated and constantly evolving. It's easy to feel overwhelmed when reading through open conversations between InfoSec professionals. These are often highly skilled people with long memories and very different contexts from developers or general users. If you can take any one thing away from open InfoSec conversations, it is that balancing system security with user needs is complex.

One of the best things you can do if you're new to this space is start with articles, papers, blogs, and books before you jump into conversations. This will give you an opportunity to digest one perspective and gain some deeper insight before switching to thought from a different perspective. When you've had an opportunity to form your own insight and opinions, these conversations become much more valuable.

(continued)

It's very difficult to read one paper or learn one thing and know the best possible way to build a hardened solution. Whatever your situation, the system will evolve to include improvements from several sources. So the best thing you can do is take each tool and learn it by itself. Don't be intimidated by the depth some tools require for a strong understanding. The effort will be worth the result, and you'll understand the systems you use much better for it.

Docker isn't a perfect solution. Some would argue that it's not even a security tool. But what improvements it provides are far better than the alternative where people forego any isolation due to perceived cost. If you've read this far, maybe you'd be willing to go further with these auxiliary topics.

6.6.1 Specifying additional security options

Docker provides a single flag for specifying options for Linux Security Modules (LSM) at container creation or runtime. LSM is a framework that Linux adopted to act as an interface layer between the operating system and security providers.

AppArmor and SELinux are both LSM providers. They both provide mandatory access control (MAC—the system defines access rules) and replace the standard Linux discretionary access control (file owners define access rules).

The flag available on `docker run` and `docker create` is `--security-opt`. This flag can be set multiple times to pass multiple values. The values can currently be one of six formats:

- To set a SELinux user label, use the form `label:user:<USERNAME>` where `<USERNAME>` is the name of the user you want to use for the label.
- To set a SELinux role label, use the form `label:role:<ROLE>` where `<ROLE>` is the name of the role you want to apply to processes in the container.
- To set a SELinux type label, use the form `label:type:<TYPE>` where `<TYPE>` is the type name of the processes in the container.
- To set a SELinux level label, use the form `label:level:<LEVEL>` where `<LEVEL>` is the level where processes in the container should run. Levels are specified as low-high pairs. Where abbreviated to the low level only, SELinux will interpret the range as single level.
- To disable SELinux label confinement for a container, use the form `label:disable`.
- To apply an AppArmor profile on the container, use the form `label:apparmor:<PROFILE>` where `<PROFILE>` is the name of the AppArmor profile to use.

As you can guess from these options, SELinux is a labeling system. A set of labels, called a *context*, is applied to every file and system object. A similar set of labels is applied to every user and process. At runtime when a process attempts to interact

with a file or system resource, the sets of labels are evaluated against a set of allowed rules. The result of that evaluation determines whether the interaction is allowed or blocked.

The last option will set an AppArmor profile. AppArmor is frequently substituted for SELinux because it works with file paths instead of labels and has a training mode that you can use to passively build profiles based on observed application behavior. These differences are often cited as reasons why AppArmor is easier to adopt and maintain.

6.6.2 Fine-tuning with LXC

Docker was originally built to use software called Linux Containers (LXC). LXC is a container runtime provider—a tool that actually works with Linux to create namespaces and all the components that go into building a container.

As Docker matured and portability became a concern, a new container runtime called libcontainer was built, replacing LXC. Docker ships with libcontainer by default, but Docker uses an interface layer so that users can change the container execution provider. LXC is a more mature library than libcontainer and provides many additional features that diverge from the goals of Docker. If you're running a system where you can and want to use LXC, you can change the container provider and take advantage of those additional features. Before investing too heavily, know that some of those additional features will greatly reduce the portability of your containers.

To use LXC, you need to install it and make sure the Docker daemon was started with the LXC driver enabled. Use the `--exec-driver=lxc` option when you start the Docker daemon. The daemon is usually configured to start as one of your system's services. Check the installation instructions on www.docker.com to find details for your distribution.

Once Docker is configured for LXC, you can use the `--lxc-conf` flag on `docker run` or `docker create` to set the LXC configuration for a container:

```
docker run -d \
  --lxc-conf="lxc.cgroup.cpuset.cpus=0,1" \
  --name ch6_stresser dockerinaction/ch6_stresser
```

← Limited to two CPU cores by LXC

```
docker run -it --rm dockerinaction/ch6_htop
```

```
docker rm -vf ch6_stresser
```

As when you ran a similar example earlier in this chapter, when you've finished with `htop`, press `Q` to quit.

If you decide to use the LXC provider and specify LXC-specific options for your containers, Docker won't be aware of that configuration. Certain configurations can be provided that conflict with the standard container changes made for every Docker container. For this reason, you should always carefully validate the configuration against the actual impact to a container.

6.7 **Build use-case-appropriate containers**

Containers are a cross-cutting concern. There are more reasons and ways that people could use them than I could ever enumerate. So it's important, when you use Docker to build containers to serve your own purposes, that you take the time to do so in a way that's appropriate for the software you're running.

The most secure tactic for doing so would be to start with the most isolated container you can build and justify reasons for weakening those restrictions. In reality, people tend to be a bit more reactive than proactive. For that reason I think Docker hits a sweet spot with the default container construction. It provides reasonable defaults without hindering the productivity of users.

Docker containers are not the most isolated by default. Docker does not require that you enhance those defaults. It will let you do silly things in production if you want to. This makes Docker seem much more like a tool than a burden and something people generally want to use rather than feel like they have to use. For those who would rather not do silly things in production, Docker provides a simple interface to enhance container isolation.

6.7.1 **Applications**

Applications are the whole reason we use computers. Most applications are programs that other people wrote and work with potentially malicious data. Consider your web browser.

A web browser is a type of application that's installed on almost every computer. It interacts with web pages, images, scripts, embedded video, Flash documents, Java applications, and anything else out there. You certainly didn't create all that content, and most people were not contributors on web browser projects. How can you trust your web browser to handle all that content correctly?

Some more cavalier readers might just ignore the problem. After all, what's the worst thing that could happen? Well, if an attacker gains control of your web browser (or other application), they will gain all the capabilities of that application and the permissions of the user it's running as. They could trash your computer, delete your files, install other malware, or even launch attacks against other computers from yours. So, this isn't a good thing to ignore. The question remains: how do you protect yourself when this is a risk you need to take?

The best approach is to isolate the risk. First, make sure the application is running as a user with limited permissions. That way, if there's a problem, it won't be able to change the files on your computer. Second, limit the system capabilities of the browser. In doing so, you make sure your system configuration is safer. Third, set limits on how much of the CPU and memory the application can use. Limits help reserve resources to keep the system responsive. Finally, it's a good idea to specifically whitelist devices that it can access. That will keep snoopers off your webcam, USB, and the like.

6.7.2 High-level system services

High-level system services are a bit different from applications. They're not part of the operating system, but your computer makes sure they're started and kept running. These tools typically sit alongside applications outside the operating system, but they often require privileged access to the operating system to operate correctly. They provide important functionality to users and other software on a system. Examples include `cron`, `syslogd`, `dbus`, `sshd`, and `docker`.

If you're unfamiliar with these tools (hopefully not all of them), it's all right. They do things like keep system logs, run scheduled commands, and provide a way to get a secure shell on the system from the network, and `docker` manages containers.

Although running services as root is common, few of them actually need full privileged access. Use capabilities to tune their access for the specific features they need.

6.7.3 Low-level system services

Low-level services control things like devices or the system's network stack. They require privileged access to the components of the system they provide (for example, firewall software needs administrative access to the network stack).

It's rare to see these run inside containers. Tasks such as file-system management, device management, and network management are core host concerns. Most software run in containers is expected to be portable. So machine-specific tasks like these are a poor fit for general container use cases.

The best exceptions are short-running configuration containers. For example, in an environment where all deployments happen with Docker images and containers, you'd want to push network stack changes in the same way you push software. In this case, you might push an image with the configuration to the host and make the changes with a privileged container. The risk in this case is reduced because you authored the configuration to be pushed, the container is not long running, and changes like these are simple to audit.

6.8 Summary

This chapter introduced the isolation features provided by Linux and talked about how Docker uses those to build configurable containers. With this knowledge, you will be able to customize that container isolation and use Docker for any use case. The following points were covered in this chapter:

- Docker uses cgroups, which let a user set memory limits, CPU weight, and CPU core restrictions and restrict access to specific devices.
- Docker containers each have their own IPC namespace that can be shared with other containers or the host in order to facilitate communication over shared memory.

- Docker does not yet support the UTS namespace, so user and group IDs inside a container are equivalent to the same IDs on the host machine.
- You can and should use the `-u` option on `docker run` and `docker create` to run containers as non-root users.
- Avoid running containers in privileged mode whenever possible.
- Linux capabilities provide operating system feature authorization. Docker drops certain capabilities in order to provide reasonably isolating defaults.
- The capabilities granted to any container can be set with the `--cap-add` and `--cap-drop` flags.
- Docker provides tooling for integrating with enhanced isolation technologies like SELinux and AppArmor. These are powerful tools that any serious Docker adopter should investigate.

Part 2

Packaging Software for Distribution

It may not be a frequent occasion, but inevitably a Docker user will need to create an image. There are times when the software you need is not packaged in an image. Other times you will need a feature that has not been enabled in an available image. The four chapters in this part will help you understand how to originate, customize, and specialize the images you intend to deploy or share using Docker.



Packaging software in images

This chapter covers

- Manual image construction and practices
- Images from a packaging perspective
- Working with flat images
- Image versioning best practices

The goal of this chapter is to help you understand the concerns of image design, learn the tools for building images, and discover advanced image patterns. You will accomplish these things by working through a thorough real-world example. Before getting started, you should have a firm grasp on the concepts in part 1 of this book.

You can create a Docker image by either modifying an existing image inside a container or defining and executing a build script called a Dockerfile. This chapter focuses on the process of manually changing an image, the fundamental mechanics of image manipulation, and the artifacts that are produced. Dockerfiles and build automation are covered in chapter 8.

7.1 *Building Docker images from a container*

It's easy to get started building images if you're already familiar with using containers. Remember, a union file system (UFS) mount provides a container's file system.

Any changes that you make to the file system inside a container will be written as new layers that are owned by the container that created them.

Before you work with real software, the next section details the typical workflow with a Hello World example.

7.1.1 *Packaging Hello World*

The basic workflow for building an image from a container includes three steps. First, you need to create a container from an existing image. You will choose the image based on what you want to be included with the new finished image and the tools you will need to make the changes.

The second step is to modify the file system of the container. These changes will be written to a new layer on the union file system for the container. We'll revisit the relationship between images, layers, and repositories later in this chapter.

Once the changes have been made, the last step is to commit those changes. Once the changes are committed, you'll be able to create new containers from the resulting image. Figure 7.1 illustrates this workflow.

With these steps in mind, work through the following commands to create a new image named `hw_image`.

	<code>docker run --name hw_container \</code>	
	<code>ubuntu:latest \</code>	
	<code>touch /HelloWorld</code>	← Modify file in container
Commit change to new image	→ <code>docker commit hw_container hw_image</code>	
	<code>docker rm -vf hw_container</code>	← Remove changed container
	<code>docker run --rm \</code>	
	<code>hw_image \</code>	
	<code>ls -l /HelloWorld</code>	← Examine file in new container

If that seems stunningly simple, you should know that it does become a bit more nuanced as the images you produce become more sophisticated, but the basic steps will always be the same.

Now that you have an idea of the workflow, you should try to build a new image with real software. In this case, you'll be packaging a program called Git.

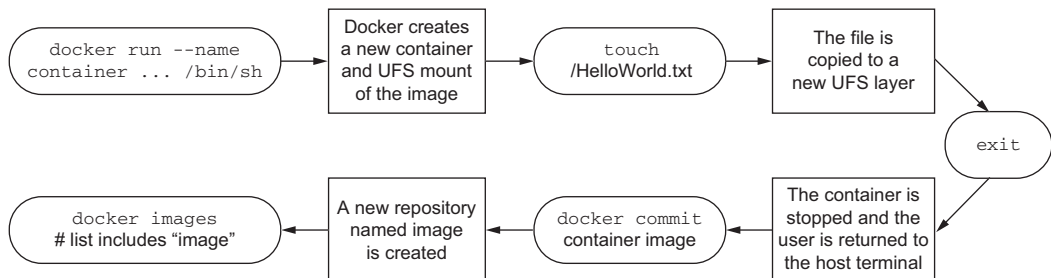


Figure 7.1 Building an image from a container

7.1.2 Preparing packaging for Git

Git is a popular, distributed version-control tool. Whole books have been written about the topic. If you're unfamiliar with it, I recommend that you spend some time learning how to use Git. At the moment, though, you only need to know that it's a program you're going to install onto an Ubuntu image.

To get started building your own image, the first thing you'll need is a container created from an appropriate base image:

```
docker run -it --name image-dev ubuntu:latest /bin/bash
```

This will start a new container running the bash shell. From this prompt, you can issue commands to customize your container. Ubuntu ships with a Linux tool for software installation called `apt-get`. This will come in handy for acquiring the software that you want to package in a Docker image. You should now have an interactive shell running with your container. Next, you need to install Git in the container. Do that by running the following command:

```
apt-get -y install git
```

This will tell APT to download and install Git and all its dependencies on the container's file system. When it's finished, you can test the installation by running the `git` program:

```
git version
# Output something like:
# git version 1.9.1
```

Package tools like `apt-get` make installing and uninstalling software easier than if you had to do everything by hand. But they provide no isolation to that software and dependency conflicts often occur. You can be sure that other software you install outside this container won't impact the version of Git you have installed.

Now that Git has been installed on your Ubuntu container, you can simply exit the container:

```
exit
```

The container should be stopped but still present on your computer. Git has been installed in a new layer on top of the `ubuntu:latest` image. If you were to walk away from this example right now and return a few days later, how would you know exactly what changes were made? When you're packaging software, it's often useful to review the list of files that have been modified in a container, and Docker has a command for that.

7.1.3 Reviewing file system changes

Docker has a command that shows you all the file-system changes that have been made inside a container. These changes include added, changed, or deleted files and directories. To review the changes that you made when you used APT to install Git, run the `diff` subcommand:

```
docker diff image-dev
```

← # Outputs a LONG list of file changes...

Lines that start with an **A** are files that were added. Those starting with a **C** were changed. Finally those with a **D** were deleted. Installing Git with APT in this way made several changes. For that reason, it might be better to see this at work with a few specific examples:

```
docker run --name tweak-a busybox:latest touch /HelloWorld
docker diff tweak-a
# Output:
#   A /HelloWorld
```

← **Add new file to busybox**

```
docker run --name tweak-d busybox:latest rm /bin/vi
docker diff tweak-d
# Output:
#   C /bin
#   D /bin/vi
```

← **Remove existing file from busybox**

```
docker run --name tweak-c busybox:latest touch /bin/vi
docker diff tweak-c
# Output:
#   C /bin
#   C /bin/busybox
```

← **Change existing file in busybox**

Always remember to clean up your workspace, like this:

```
docker rm -vf tweak-a
docker rm -vf tweak-d
docker rm -vf tweak-c
```

Now that you've seen the changes you've made to the file system, you're ready to commit the changes to a new image. As with most other things, this involves a single command that does several things.

7.1.4 **Committing a new image**

You use the `docker commit` command to create an image from a modified container. It's a best practice to use the `-a` flag that signs the image with an author string. You should also always use the `-m` flag, which sets a commit message. Create and sign a new image that you'll name `ubuntu-git` from the `image-dev` container where you installed Git:

```
docker commit -a "@dockerinaction" -m "Added git" image-dev ubuntu-git
# Outputs a new unique image identifier like:
# bbf1d5d430cdf541a72ad74dfa54f6faec41d2c1e4200778e9d4302035e5d143
```

Once you've committed the image, it should show up in the list of images installed on your computer. Running `docker images` should include a line like this:

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
ubuntu-git	latest	bbf1d5d430cd	5 seconds ago	226 MB

Make sure it works by testing Git in a container created from that image:

```
docker run --rm ubuntu-git git version
```

Now you've created a new image based on an Ubuntu image and installed Git. That's a great start, but what do you think will happen if you omit the command override? Try it to find out:

```
docker run --rm ubuntu-git
```

Nothing appears to happen when you run that command. That's because the command you started the original container with was committed with the new image. The command you used to start the container that the image was created by was `/bin/bash`. When you create a container from this image using the default command, it will start a shell and immediately exit. That's not a terribly useful default command.

I doubt that any users of an image named `ubuntu-git` would expect that they'd need to manually invoke Git each time. It would be better to set an entrypoint on the image to `git`. An entrypoint is the program that will be executed when the container starts. If the entrypoint isn't set, the default command will be executed directly. If the entrypoint is set, the default command and its arguments will be passed to the entrypoint as arguments.

To set the entrypoint, you'll need to create a new container with the `--entrypoint` flag set and create a new image from that container:

```

docker run --name cmd-git --entrypoint git ubuntu-git
docker commit -m "Set CMD git" \
    -a "@dockerinaction" cmd-git ubuntu-git
Cleanup └─ docker rm -vf cmd-git
docker run --name cmd-git ubuntu-git version

```

Now that the entrypoint has been set to `git`, users no longer need to type the command at the end. This might seem like a marginal savings with this example, but many tools that people use are not as succinct. Setting the entrypoint is just one thing you can do to make images easier for people to use and integrate into their projects.

7.1.5 Configurable image attributes

When you use `docker commit`, you commit a new layer to an image. The file-system snapshot isn't the only thing included with this commit. Each layer also includes meta-data describing the execution context. Of the parameters that can be set when a container is created, all the following will carry forward with an image created from the container:

- All environment variables
- The working directory
- The set of exposed ports
- All volume definitions
- The container entrypoint
- Command and arguments

