Introducing Docker

****

Before we go any further with this book, we need to look at a little thing called Docker, before we begin don't forget to clone the example code repository [https://github.com/bu](https://github.com/building-microservices-with-go/chapter3.git) [ilding-microservices-with-go/chapter3.git](https://github.com/building-microservices-with-go/chapter3.git).

# Introducing Containers with Docker

Docker is a platform that has risen to prominence in the last three years; it was born out of the desire to simplify the process of building, shipping, and running applications. Docker is not the inventor of the container, Jacques Gélinas created the VServer project back in 2001, and since then the other main projects have been LXC from IBM and rkt from CoreOS.

If you would like to read more about the history, then I recommend this excellent blog post by Redhat: <http://rhelblog.redhat.com/2015/08/28/the-history-of-containers>, this section is going to concentrate on Docker which is by far the most popular current technology.

The concept of a container is process isolation and application packaging. To quote Docker:

*A container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings.*

*...*

*Containers isolate software from its surroundings, for example, differences between development and staging environments and help reduce conflicts between teams running different software on the same infrastructure.*

Where they benefit application development is that we can take advantage of this when deploying these applications as it allows us to pack them closer together, saving on hardware resources.

From a development and test lifecycle, containers give us the capability to run production code on our development machines with no complicated setup; it also allows us to create that Clean Room environment without having different instances of the same database installed to trial new software.

Containers have become the primary choice for packaging microservices, and as we progress through the examples in this book, you will learn how invaluable it is to your workflow.

Containers work by isolating processes and filesystems from each other. Unless explicitly specified, containers cannot access each other's file systems. They also cannot interact with one another via TCP or UDP sockets unless again specified.

Docker is made up of many parts; however, at its core is the Docker Engine, a lightweight application runtime with features for orchestration, scheduling networking, and security. Docker Engine can be installed anywhere on a physical or virtual host, and it supports both Windows and Linux. Containers allow developers to package large or small amounts of code and their dependencies together into an isolated package.

We can also draw from a huge array of pre-created images, just about all software vendors from MySQL to IBM's WebSphere have an official image that is available for us to use.

Docker also uses Go, in fact nearly all of the code that goes into the Docker Engine and other applications are written in Go.

Rather than write an essay on how Docker works, let's examine each of the features by example. By the end of this chapter, we will take one of the simple examples that we created in Chapter 1, *Introduction to Microservices*, and create a Docker image for it.

# Installing Docker

Head over to <https://docs.docker.com/engine/installation/>and install the correct version of Docker on your machine. You will find versions for Mac, Windows, and Linux.

# Running our first container

To validate Docker has been installed correctly, let's run our first container, **hello-world** is actually an image, an image is an immutable snapshot of a container. Once we start these with the following command they become containers, think of it like types and instances, a type defines fields and methods making up behavior. An instance is a living instantiation of this type, you can assign other types to the fields and call the methods to perform actions.

**$ docker run --rm hello-world**

The first thing you should see is:

**Unable to find image 'hello-world:latest' locally latest: Pulling from library/hello-world c04b14da8d14: Pull complete**

**Digest: sha256:0256e8a36e2070f7bf2d0b0763dbabdd67798512411de4cdcf9431a1feb60fd9 Status: Downloaded newer image for hello-world:latest**

When you execute a docker run the first thing the engine does is check to see if you have the image installed locally. If it doesn't then it connects to the default registry, in this case, [h](https://hub.docker.com/) [ttps://hub.docker.com/](https://hub.docker.com/) to retrieve it.

Once the image has been downloaded, the daemon can create a container from the downloaded image, all the output is streamed to the output on your terminal:

**Hello from Docker!**

**This message shows that your installation appears to be working correctly.**

The --rm flag tells the Docker engine to remove the container and delete any resources such as volumes it was using on exit. Unless we would like to re-start a container at some point it is good practice to use the --rm flag to keep our filesystem clean, otherwise, all of the temporary volumes which are created will sit around and consume space.Let's try something a little more complicated, this time, we will start a container and create a shell inside of it to show how you can navigate to the internal file system. Execute the following command in your terminal:

**$ docker run -it --rm alpine:latest sh**

Alpine is a lightweight version of Linux and is perfect for running Go applications. The -it flags stand for **interactive terminal** it maps the standard in from your terminal to the input of the running container. The sh statement after the name of the image we want to run is the name of the command we would like to execute in the container when it starts.

If all went well, you should now be inside a shell of the container. If you check the current directory by executing the ls command, you will see the following, which hopefully is not the directory you were in before running the command:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **bin** | **etc** | **lib media** | **proc** | **run** | **srv** | **tmp** | **var** |
| **dev** | **home** | **linuxrc mnt** | **root** | **sbin** | **sys** | **usr** |  |

This is the root folder of the newly started container, containers are immutable, so any changes you make to the file system in a running container is disposed of when the container is stopped. While this may seem to be a problem, there are solutions for persisting data, which we will look at in a little bit, however, for now, the important concept to remember is that:

*"Containers are immutable instances of images, and the data volumes are by default non- persistent"*

You need to remember this when designing your services, to illustrate how this works take a look at this simple example.

Open another terminal and execute the following command:

**$ docker ps**

You should see the following output:

|  |  |  |  |
| --- | --- | --- | --- |
| **CONTAINER ID**  **STATUS** | **IMAGE**  **PORTS** | **COMMAND**  **NAMES** | **CREATED** |
| **43a1bea0009e** | **alpine:latest** | **"sh"** | **6 minutes ago** |
| **Up 6 minutes** |  | **tiny\_galileo** |  |

The docker ps command queries the engine and returns a list of the containers, by default this only shows the running containers, however, if we add the -a flag we can also see stopped containers.

The Alpine Linux container that we started earlier is currently running, so jump back to your previous terminal window and create a file in the root file system:

**$ touch mytestfile.txt**

If we list the directory structure again, we can see that a file has been created in the root of the file system:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **bin** | **lib** | **mytestfile.txt** | **sbin** | **usr** |
| **dev** | **linuxrc** | **proc** | **srv** | **var** |
| **etc** | **media** | **root** | **sys** |  |
| **home** | **mnt** | **run** | **tmp** |  |

Now exit the container using the exit command and run docker ps again, you should see the following output:

|  |  |  |  |
| --- | --- | --- | --- |
| **CONTAINER ID** | **IMAGE** | **COMMAND** | **CREATED** |
| **STATUS** | **PORTS** | **NAMES** |  |

If we add the -a flag command to see stopped containers too, we should see the container we started earlier:

**CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES**

**518c8ae7fc94 alpine:latest "sh" 5 seconds ago Exited (0) 2 seconds ago pensive\_perlman**

Now, start another container again using the docker run command and list the directory contents in the root folder.

No mytestfile.txt right? The reason this does not exist is because of the principle we were discussing earlier, which I think is important to mention again as if this is the first time you have used Docker it will catch you out:

*"Containers are immutable instances of images, and the data volumes are by default non- persistent."*

There is something worth noting, however, unless you explicitly remove a container it will persist in a stopped state on the Docker host.

Removing containers is important to remember for two reasons; the first is that if you do not remember this, you will fill up the disk on your host quickly as every time you create a container Docker will allocate space on the host for the container volumes. The second is that the container can be restarted.

Restarted that sounds cool, in fact, it is a handy feature, not something you should use in your production environment, for that you need to remember the golden rule and design your application accordingly:

*"Containers are immutable instances of images, and the data volumes are by default non- persistent."*

However, the use of Docker extends far beyond simply running applications for your microservices. It is an awesome way to manage your development dependencies without cluttering up your development machine. We will look at that a little later on, but for now, we are interested in how we can restart a stopped container.

If we execute the docker ps -a command, we will see that we now have two stopped containers. The oldest one is the first container we started to which we added our mytestfile.txt. This is the one we want to restart, so grab the ID of the container and execute the following command:

**$ docker start -it [container\_id] sh**

Again, you should be in a shell at the root of the container if you check the directory contents what do you think you will find?

That's right, mytestfile.txt; this is because when you restarted the container, the engine remounted the volumes that were attached the first time you ran the command. These are the same volumes you mutated to add the file as mentioned earlier.

So we can restart our container; however, I just want to repeat the golden rule one last time:

*"Containers are immutable instances of images, and the data volumes are by default non- persistent."*

When running in a production environment, you cannot ensure that you can restart a container. There are a million reasons for this, one of the main ones that we will look at more in depth when we look at orchestration is that containers are generally run on a cluster of hosts. Since there is no guarantee which host the container will be restarted on or even that the host the container was previously running on actually exists. There are many projects that attempt to solve this, but the best approach is to avoid the complexity altogether. If you need to persist files, then store them in something that is designed for the job such as Amazon S3 or Google Cloud Storage. Design your applications around this principle and you will spend far less time panicking when the inevitable happens, and your super sensitive data container disappears.

OK, before we look at Docker volumes in more depth let's clean up after ourselves.

Exit your container and get back to the shell on the Docker host. If we run docker ps -a

,we will see that there are two stopped containers. To remove these, we can use the docker rm containerid command.

Run this now using the first containerid in your list, if this is successful, the container ID you asked to be removed would be echoed back to you, and the container will is deleted.

If you want to remove all the stopped containers you can use the following command:

**$ docker rm -v $(docker ps -a -q)**

The docker ps -a -q the -a flag will list all the containers including the stopped ones, -q will return a list of the container IDs rather than the full details. We are passing this as a parameter list to docker rm, which will remove all the containers in the list.

To avoid having to remove a container we can use the --rm flag when starting a new container. This flag tells Docker to remove the container when it stops.

# Docker volumes

We have seen how Docker containers are immutable; however, there are some instances when you may wish to write some files to a disk or when you want to read data from a disk such as in a development setup. Docker has the concept of volumes, which can be mounted either from the host running the Docker machine or from another Docker container.

## Union filesystem

To keep our images efficient and compact Docker uses the concept of a Union File System. The Union filesystem allows us to represent a logical file system by grouping different directories and or files together. It uses a **Copy on Write** technique, which copies the layer when we modify the file system, this way we only use about 1MB of space when creating a new image. When data is written to the file system Docker copies the layer and puts it on the top of the stack. When building images and extending existing images we are leveraging this technique, also when starting an image and creating a container the only difference is this writable layer, which means we do not need to copy all the layers every time and fill up our disk.

## Mounting volumes

The -v, or --volume parameter allows you to specify a pair of values corresponding to the file system you wish to mount on the host and the path where you would like to mount the volume inside the container.

Let's try our example from earlier, but this time mounting a volume on the local file system:

**$ docker run -it -v $(pwd):/host alpine:latest /bin/sh**

If you change into the host folder, you will see that there is access to the same folder from where you ran the docker run command. The syntax for the values for -v is hostfolder:destinationfolder, one thing I think is important to point out is that these paths need to be absolute, and you cannot use a relative path like ./ or ../foldername.

The volume you have just mounted has read/write access, any changes you make will be synchronized to the folder on the host so be careful to not go running rm -rf \*. Creating Volumes on a production environment should be used very sparingly, I would advise that where possible you avoid doing it all together as in a production environment there is no guarantee if a container dies and is re-created that it will be replaced on the same host where it was previously. This means that any changes you have made to the volume will be lost.

# Docker ports

When running web applications inside a container, it is quite common that we will need to expose some ports to the outside world. By default, a Docker container is completely isolated, and if you start a server running on port 8080 inside your container unless you explicitly specify that port is accessible from the outside, it will not be accessible.

Mapping ports is a good thing from a security perspective as we are operating on a principle of no trust. It is also effortless to expose these ports. Using one of the examples we created in Chapter 1, *Introduction to Microservices*, let's see just how easy this is.

Move to the folder where you checked out the sample code, and run the following Docker command:

**$ docker run -it --rm -v $(pwd):/src -p 8080:8080 -w /src golang:alpine**

**/bin/sh**

The -w flag we are passing is to set the working directory that means that any command we run in the container will be run inside this folder. When we start the shell, you will see that rather than having to change into the folder we specify in the second part of the volume mounting we are already in that folder and can run our application. We are also using a slightly different image this time. We are not using alpine:latest, which is a lightweight version of Linux, we are using golang:alpine, which is a version of Alpine with the most recent Go tools installed.

If we start our application using the go run main.go command; we should see the following output:

**2016/09/02 05:53:13 Server starting on port 8080**

Now change to another shell and try to curl the API endpoint:

**$ curl -XPOST localhost:8080/helloworld -d '{"name":"Nic"}'**

You should see something like the following message returned:

**{"message":"Hello Nic"}**

If we run the docker ps command to inspect the running containers, we will see that there are no ports exposed. Go back to your previous terminal window and kill the command and then exit the container.

This time, when we start it, we will add the -p argument to specify the port. Like volumes, this takes a pair of values separated by a colon (:). The first is the destination port on the host that we would like to bind to the second is the source port on the Docker container to which our application is bound.

Because this binds to the port on the host machine, in the same way that you would not be able to start the program locally twice because of the port binding, you cannot do this with the host port mappings in Docker either. Of course, you can start multiple instances of your code in separate containers and bind to different ports, and we will see how you can do that in just a bit.

But first let's take a look at that port command, rather than starting a container and creating a shell to run our application we can do this in one command by replacing the /bin/sh command with our go run command. Give that a try and see if you can get your application running.

Got it?

You should have typed something like the following:

**$ docker run -it --rm -v $(pwd):/src -w /src -p 8080:8080 golang:alpine go run reading\_writing\_json\_8.go**

Now try your curl to send some data to the API again, you should see the following output:

**{"message":"Hello Nic"}**

Like volumes, you can specify multiple instances of the -p argument, which enables you to set up the binding for multiple ports.

# Removing a container starting with an explicit name

Containers that start with a name parameter are not automatically removed even if you specify the --rm argument. To remove a container started in this way, we must manually use the docker rm command. If we append the -v option to the command, we can also remove the volumes that are associated with it. We should really do this now, or when we try to recreate the container later in the chapter, you might be left a little puzzled:

**$ docker rm -v server**

# Docker networking

I never intended this chapter to be a full reproduction of the official Docker documentation; I am just trying to explain some of the key concepts that will help you as you progress through the rest of this book.

Docker networking is an interesting topic, and by default, Docker supports the following network modes:

bridge host none overlay

## Bridge networking

The bridge network is the default network that your containers will connect to when you launch them; this is how we were able to join our containers together in the last example. To facilitate this, Docker uses some of the core Linux capabilities such as networking namespaces and virtual Ethernet interfaces (or veth interfaces).

When the Docker engine starts, it creates the docker0 virtual interface on the host machine. The docker0 interface is a virtual Ethernet bridge that automatically forwards packets between any other network interfaces that are attached to it. When a container starts it creates a veth pair, it gives one to the container, which becomes its eth0, and the other connects to the docker0 bridge.

## Host networking

The host network is essentially the same network that the Docker engine is running on. When you connect a container to the host network all of the ports that are exposed by the container are automatically mapped to the hosts, it also shares the IP address of the host. While this may seem like a nice convenience, Docker was always designed to be capable of running multiple instances of the same container on the engine, and since you can only bind a socket to one port in Linux using the host network limits this feature.

The host network can also pose a security risk to your container as it is no longer protected by the principle of no trust and you no longer have the ability to explicitly control if a port is exposed or not. That being said, due to the efficiencies of host networking it may in some instances be appropriate to connect a container to the host network if you anticipate that it is going to heavily use the network. An API gateway might be one such example, this container would still be possible to route requests to other API containers that are sitting on the bridge network.

## No network

Removing your container from any network might in some instances be something you wish to do. Consider the situation where you have an application that only processes data stored in a file. Utilizing the principle of no trust, we may determine that the securest thing to do is to not connect it to any container and to only allow it to write to a volume that is mounted on the host. Attaching your container to the none network provides exactly this capability, and while the use case might be somewhat limited it is there, and it's nice to know about it.

## Overlay network

The Docker overlay network is a unique Docker network that is used to connect containers running on separate hosts to one another. With the bridge network as we have already learned, network communication is localized to the Docker host and this is generally fine when you are developing software. When you run your code in production however, all this changes, as you will typically be running multiple hosts, each running multiple containers as part of your high availability setup. The containers still need to talk to one another, and while we could route all traffic through an **ESB** (**enterprise service bus**), this is a little bit of an anti-pattern in the microservice world. The recommended approach as we will see in a later chapter, is for the service to be responsible for its own discovery and load balancing client calls. The Docker overlay network solves this problem, it is in effect a network tunnel between machines which passes the traffic unmodified over the physical network. The problem with the overlay is that you can no longer rely on Docker to update the etc/hosts file for you, and you must depend on a dynamic service registry.

## Custom network drivers

Docker also supports plugins for networking, based around its open source libnetwork project, you can write custom networking plugins that can replace the networking subsystem of the Docker engine. They also give the capability for you to connect non- Docker applications to your container network such as a physical database server.

### Weaveworks

Weaveworks is one of the most popular plugins, it gives you the capability to securely link your Docker hosts and also provides a whole host of additional tools such as service discovery with weavedns and visualization with weavescope, so you can see how your network is connected together.

[https://www.weave.works](https://www.weave.works/)

### Project Calico

Project Calico attempts to solve the speed and efficiency problems that using virtual LANs, bridging, and tunneling can cause. It achieves this by connecting your containers to a vRouter, which then routes traffic directly over the L3 network. This can give huge advantages when you are sending data between multiple data centers as there is no reliance on NAT and the smaller packet sizes reduce CPU utilization.

[https://www.projectcalico.org](https://www.projectcalico.org/)

## Creating custom bridge networks

Implementing a custom overlay network is beyond the scope of this book, however, understanding how you can create custom bridge networks is something that we should look at as Docker-Compose, which we are going to introduce later in this chapter, utilizes these concepts.

Like many of the Docker tools, creating a bridge network is quite straightforward. To see the currently running networks on your Docker engine, we can execute the following command:

**$ docker network ls**

The output should be something like the following:

|  |  |  |  |
| --- | --- | --- | --- |
| **NETWORK ID** | **NAME** | **DRIVER** | **SCOPE** |
| **8e8c0cc84f66** | **bridge** | **bridge** | **local** |
| **0c2ecf158a3e** | **host** | **host** | **local** |
| **951b3fde8001** | **none** | **null** | **local** |

You will find that there are three networks created by default, which is three of the ones we discussed earlier. Because these are default networks, we are unable to remove these, Docker requires these networks to function correctly and allowing you to remove them would be a bad thing indeed.

### Creating a bridge network

To create a bridge network, we can use the following command:

**$ docker network create testnetwork**

Run this now in your terminal and list the networks again to see the results.

You will see that there is now a fourth network in your list that uses the bridge driver and that has the name you specified as one of the arguments. By default, when you create a network, it uses the bridge as a default driver, of course, it is possible to create a network to a custom driver, and this can be easily facilitated by specifying the additional argument,

-d drivername.

**Connecting containers to a custom network**

To connect a container to a custom network, let's again use the example application that we created in Chapter 1, *Introduction to Microservices*:

**$ docker run -it --rm -v $(pwd):/src -w /src --name server -- network=testnetwork golang:alpine go run main.go**

Did you get the error message that the name is already in use because you forgot to remove the container in the earlier section? If so, it might be time to head back a few pages.

Assuming all went well, you should see the server starting message, now let's try to curl the container using the same command we executed earlier:

**$ docker run --rm appropriate/curl:latest curl -i -XPOST server:8080/helloworld -d '{"name":"Nic"}'**

You should have received the following error message:

**curl: (6) Couldn't resolve host 'server'**

This was expected, have a go to see if you can update the docker run command to make it work with our API container.

Got it?

If not, here is the modified command with the added network argument:

**$ docker run --rm --network=testnetwork appropriate/curl:latest curl -i - XPOST server:8080/helloworld -d '{"name":"Nic"}'**

This command should have worked just fine the second time, and you should see the expected output. Now remove the server container, and we will take a look at how you can write your own Docker files.

# Writing Dockerfiles

Dockerfiles are the recipes for our images; the define the base image, software to be installed and give us the capability to set the various structure that our application needs.

In this section, we are going to look at how we can create a Docker file for our example API. Again, this is not going to be a comprehensive overview of how Dockerfiles work as there are many books and online resources that exist for that explicit purpose. What we will do is to look at the salient points that will give us the basics.

The first thing we are going to do is build our application code as when we package this into a Docker file we will be executing a binary, not using the go run command. The image we are going to create will have only the software installed that we need to run our application. Limiting the software installed is a Docker best practice when creating images as it reduces the attack surface by only including what is necessary.

## Building application code for Docker

We are going to execute a slightly different command for creating our files from the usual

go build:

**$ CGO\_ENABLED=0 GOOS=linux GOARCH=386 go build -a -installsuffix cgo - ldflags '-s' -o server**

In the preceding command, we are passing the argument -ldflags '-s', this argument passes the -s argument to the linker when we build the application and tells it to statically link all dependencies. This is very useful when we use the popular Scratch container as a base; Scratch is the lightest base you can get it has no application frameworks or applications this is opposed to Ubuntu, which takes about 150MB. The difference between Scratch and Ubuntu is that Scratch does not have access to the standard C library GLibC.

If we do not build a static binary, then it will not execute if we try to run it in a Scratch container. The reason for this is that while you may think that your Go application is a static binary it still has a dependency on GLibC, both the net and the os/user packages link to GLibC so if we are to run our application with a Scratch base image we need to statically link this. The benefit, however, is an incredibly small image, we end up with an image which is roughly 4MB in size, exactly the size of our compile Go application.

Because the Docker engine is running on Linux, we also need to build our Go binary for the Linux architecture. Even if you are using Docker for Mac or Docker for Windows, what is happening under the hood is that the Docker engine is running a lightweight virtual machine on either HyperV or the Mac's xhyve virtual machine.

If you are not using Linux to run your go build command and since Go has excellent capability for cross-platform compilation, you don't need to do much. All you do need to do is prefix the architecture variables GOOS=linux GOARCH=386 to your go build command as we did in the earlier example.

Now that we have created a binary for our application, let's take a look at the Docker file:

1. **FROM scratch**
2. **MAINTAINER** [**jackson.nic@gmail.com**](mailto:jackson.nic@gmail.com) **3**

**4 EXPOSE 8080 5**

**6 COPY ./server ./ 7**

**8 ENTRYPOINT ./server**

### FROM

The FROM instruction set the base image for subsequent instructions. You can use any image that is either stored in a remote registry or locally on your Docker Engine. When you execute docker build, if you do not already have this image, then Docker will pull it from the registry as the first step of the build process. The format for the FROM command is the same as you would use when issuing a docker run command it is either:

FROM image // assuming latest

FROM image:tag // where you can specify a tag to use

In **line 1**, we are using the image name scratch, this is a particular kind of image, which is basically a blank canvas. We could use Ubuntu or Debian or Alpine or pretty much anything really, but since all we need to run our Go application is the application itself then we can use scratch to produce the smallest possible image.

### MAINTAINER

The MAINTAINER instruction allows you to set the author of the generated image. This is an optional instruction; however, it can be good practice to include this even if you are not planning on publishing your image to the public registry.

### EXPOSE

The EXPOSE instruction informs Docker that the container listens on the specified networks ports at runtime. Expose does not make the ports accessible to the host; this function still needs to be performed with the -p mapping.

### COPY

The COPY instruction copies files from the source in the first part of this instruction to the destination specified in the second part:

COPY <src> <dest>

COPY ["<src">, "<dest>"] // useful when paths contain whitespace

The <src> in the COPY instruction may contain wildcards with the matching done using Go's filepath.Match rules.

Note:

<src> must be part of the context for the build, you cannot specify relative folders such as ../;

A root / specified in the <src> will be the root of the context

A root / specified in the <dest> will map to the containers root file system Specifying a COPY instruction without a destination will copy the file or folder into the WORKDIR with the same name as the original

### ENTRYPOINT

An ENTRYPOINT allows you to configure the executable that you would like to run when your container starts. Using ENTRYPOINT makes it possible to specify arguments as part of the docker run command which is appended to the ENTRYPOINT.

ENTRYPOINT has two forms:

ENTRYPOINT ["executable", "param1", "param2"] // preferred form

ENTRYPOINT command param1 param2 //shell form

For example, in our Docker file, we are specifying the ENTRYPOINT ./server. This is our Go binary that we would like to run. When we start our container with the following docker run helloworld command, we do not need to explicitly tell the container to execute the binary and launch the server. We can, however, pass additional arguments to the application via the docker run command arguments; these would then be appended to the ENTRYPOINT before the application is run. For example:

**$ docker run --rm helloworld --config=/configfile.json**

The preceding command would append the arguments to the executed statement defined in the entry point, which would be the equivalent of executing the following shell command:

**$ ./server --config=configfile.json**

### CMD

The CMD instruction has three forms:

CMD ["executable", "param1", "param2"] // exec form

CMD ["param1", "param2"] // append default parameters to ENTRYPOINT CMD command param1 param2 // shell form

When CMD is used to provide default arguments for the ENTRYPOINT instruction then both the CMD and ENTRYPOINT instructions should be specified using the JSON array format.

If we specify a default value for CMD, we can still override it by passing the command arguments to the docker run command.

Only one CMD instruction is permitted in a Docker file.

### Good practice for creating Dockerfiles

Taking all of this into account, we need to remember how the union file system works in Docker and how we can leverage it to create small and compact images. Every time we issue a command in the Dockerfile, Docker will create a new layer. When we mutate this command, the layer must be completely recreated and potentially all the following layers too, which can dramatically slow down your build. It is therefore recommended a good practice that you should attempt to group your commands as tightly as possible to reduce the possibility of this occurring.

Quite often, you will see Dockerfiles which instead of having a separate RUN command for every command we would like to execute, we chain these using standard bash formatting.

For example, consider the following, which would install software from a package manager.

#### Bad Practice:

**RUN apt-get update**

**RUN apt-get install -y wget RUN apt-get install -y curl RUN apt-get install -y nginx**

#### Good Practice:

**RUN apt-get update && \**

**apt-get install -y wget curl nginx**

The second example would only create one layer, which in turn would create a much smaller and more compact image, it is also good practice to organize your COPY statements placing the statement which changes the least further up in the Dockerfile, this way you avoid invalidation of subsequent layers even if there are no changes to these layers.

## Building images from Dockerfiles

To build an image from our Dockerfile, we can execute a straightforward command:

**$ docker build -t testserver .**

Breaking this down the -t argument is the tag we wish to give the container, this takes the form name:tag, If we omit the tag portion of the argument as we have in our example command, then the tag latest will be automatically assigned.

If you run docker images, you will see that our testserver image has been given this tag.

The final argument is the context we would like to send to the Docker Engine. When you run a Docker build, the context is automatically forwarded to the server. This may seem strange, but you have to remember that it is not uncommon that the Docker Engine will not be running on your local machine, and therefore it will not have access to your local filesystem. For this reason, we should be careful about where we are setting our context as it can mean that potentially a large amount of data is being sent to the engine, which will slow things down. Context then becomes the root for your COPY commands.

Now that we have our running container, let's test it out. Why not start a container from our newly built image and check the API by curling the endpoint:

**$ docker run --rm -p 8080:8080 testserver**

**$ curl -XPOST localhost:8080/helloworld -d '{"name":"Nic"}'**

### Docker build context

When we run our Docker build command, we set the context path as the final argument. What actually happens when the command executes is that the context is transferred to the server. This can cause problems if you have a large source folder, so it is good practice to only send the files you need to be packaged inside the container or the files you need when building the container. There are two ways we can mitigate this problem. The first is to ensure that our context only has the files on it we require. Since this is not always possible we have a secondary option of using a .dockerignore file.

### Docker Ignore files

The .dockerignore file is similar to a git ignore file before the CLI sends the context to the Engine, it excludes files and directories that match patterns in the .dockerignore file. It uses the patterns which are defined in Go's filepath.Match rules you can find more information about them in the following Go documentation: <https://godoc.org/path/filepath#Match>

|  |  |
| --- | --- |
| **Rule** | **Behavior** |
| # comment | Ignored. |
| \*/temp\* | Exclude files and directories whose names start with temp in any immediate subdirectory of the root. For example, the plain file  /somedir/temporary.txt is excluded, as is the directory /somedir/temp. |
| \*/\*/temp\* | Exclude files and directories starting with temp from any subdirectory that is two levels below the root. For example, /somedir/subdir/temporary.txt is excluded. |
| temp? | Exclude files and directories in the root directory whose names are a one- character extension of temp. For example, /tempa and /tempb are excluded. |

[https://docs.docker.com/engine/reference/builder/#/dockerignore-file](https://docs.docker.com/engine/reference/builder/%23/dockerignore-file)

## Running Daemons in containers

One of the things you might be used to when deploying an application to a VM or physical server is to use a Daemon runner such as initd or systemd to ensure that the application is started in the background and continues to run even if it crashes. This is an anti-pattern when you are using Docker containers, for Docker to successfully stop the application it will attempt to kill the process running with PID 1. Daemons will generally start with PID 1 and start your application with another process ID, which will mean they are not killed when you stop the Docker container. This can cause containers to hang when the docker stop command is executed.

In the instance that you need to ensure that your application keeps running even after a crash then you delegate this responsibility to the orchestrator who is starting your Docker container. We will learn more about this when we look at orchestration in a later chapter.

# Docker Compose

That was all super easy-ish, let's now take a look at a compelling feature of Docker that allows you to start multiple containers at once with your stack definition stored in a handy YAML file.

## Installing Docker Compose on Linux

If you have either Docker for Mac or Docker for Windows installed then it already comes bundled with docker-compose, if however, you are using Linux, then you may need to install this yourself as it does not come as part of the default Docker package.

To install Docker Compose on Linux, execute the following command in your terminal:

**$ curl -L**

**https://github.com/docker/compose/releases/download/1.8.0/docker-compose-`u name -s`-`uname -m` > /usr/local/bin/docker-compose && chmod +x**

**/usr/local/bin/docker-compose**

Before we look at how we can run our application with docker-compose, let's take a look at the file we are going to run and some of the important facets of it:

1. **version: '2'**
2. **services:**
3. **testserver:**
4. **image: testserver**
5. **curl:**
6. **image: appropriate/curl**
7. **entrypoint: sh -c "sleep 3 && curl -XPOST testserver:8080/helloworld**

**-d '{\"name\":\"Nic\"}'"**

Docker Compose files are written in YAML, inside this file you can define services that will make up your application. In our simple example, we are only describing two services. The first is our example code that we have just built and the second is a simple service that curls this API. As a production example, this is not particularly useful I admit, but it is only intended to show how to set up these files. As we progress through later chapters, we will heavily rely on compose files to create our databases and other data stores that make up our application.

**Line 1** defines the version of the Docker compose file we are using, version 2 is the latest version and is a breaking change from version 1 which along with the --link directive is now deprecated and will be removed in a future release.

In **line 2** we define the services. Services are the containers that you would like to start with your stack. Each service has to have a unique name to the compose file, but not necessarily to all the containers running on your Docker Engine. To avoid conflicts when starting a stack, we can pass -p projectname to the docker-compose up command; this will prefix the name of any of our containers with the specified project name.

The minimum information you need to specify for a service is the image, which is the image you wish to start a container from. In the same way that docker run works, this can either be a local image on the Docker Engine or it can be a reference to an image in a remote registry. When you start a stack, compose will check to see if the image is available locally and if not it will automatically pull it from the registry.

**Line 6** defines our second service; this is simply going to execute a command to curl a request to the API exposed by the first service.

In this service definition block, we are both specifying the image and an entry point.

## Service startup

The previous command looks a little weird, but there is a gotcha with Docker compose, which quite a few people fall foul too, there is no real way for compose to know when an application is running. Even if we use the depends-on configuration, we are only informing compose that there are dependencies and that it should control the start order of the services.

sh -c "sleep 3 && curl -XPOST testserver:8080/helloworld -d '{\"name\":\"Nic\"}'"

All compose will do is check that the container has been started. The general problem occurs with a misunderstanding that a container being started equals it is ready to receive requests. More often than not this is not the case, it can take time for your application to start and be ready to accept requests. If you have a dependency like we have specified in our entry point to curl the endpoint in another service, then we cannot assume that the dependent service is ready for requests before we execute our command. We will cover a pattern for dealing with this in Chapter 6, *Microservice Frameworks*, but for now we can be aware that:

*"Container started, and service ready is not the same thing."*

In our simple example, we know that it roughly takes a second or so for the service to start, so we will just sleep for three seconds to give it plenty of time to get ready before executing our command. This method is not good practice, and it is only to illustrate how we can use compose to link services. In reality, you would probably never start a single command like we are here in your compose file.

When you use a Docker network, Docker automatically adds a mapping to the containers resolve.conf pointing to the built in Docker DNS server, we can then contact other containers connected to the same network by referencing them by name. Looking at our curl command, this DNS capability is exactly what allows us to use the hostname testserver.

OK, time to test it out, run the following command from your terminal:

**$ docker-compose up**

All being well you should see the following message returned in the output:

**{"message":"Hello Nic"}**

*Ctrl* + *C* will exit compose, however, since we did run this with the docker run command and passed the arguments --rm to remove the container, we need to ensure that we clean up after ourselves. To remove any stopped container that you have started with docker- compose, we can use the particular compose command rm and pass the -v argument to remove any associated volumes:

**$ docker-compose rm -v**

## Specifying the location of a compose file

Whenever you run docker-compose, it looks for a file named docker-compose.yml in the current folder as a default file. To specify an alternate file, we can pass the -f argument to compose with a path to the compose file we would like to load:

**$ docker-compose -f ./docker-compose.yml up**

## Specifying a project name

As we discussed earlier when we start docker-compose, it will create services with the given names in your Compose file appending the project name default to them. If we need to run multiple instances of this compose file, then docker-compose will not start another instance as it will check to see if any services are running with the given names first. To override this, we can specify the project name replacing the default name of default. To do this we just need to specify the -p projectname argument to our command as follows:

**$ docker-compose -p testproject up**

This will then create two containers:

testproject\_testserver testproject\_curl

# Summary

In summary, we have learned how to work with Docker in this chapter, and while this is only a brief overview, I suggest you head over to the documentation and read more in depth on the concepts of Dockerfiles, Composefiles, the Docker Engine, and Docker Compose. Docker is an invaluable tool for development, testing, and production and as we progress through the following chapters, we will use these concepts extensively. In the next chapter, we are going to look at testing, which builds on all of the things you have learned so far.