200 XP

How Docker images work

10 minutes

Recall we said the container image becomes the unit we use to distribute applications. We also mentioned the container is in a standardized format used by both our developer and operation teams.

Here we'll look at the differences between software, packages, and images as used in Docker. Knowing the differences between these concepts will help us better understand how Docker images work.

We'll also briefly discuss the roles of the OS running on the host and the OS running in the container.

Software packaged into a container

The software packaged into a container isn't limited to the applications our developers build. When we talk about software, we refer to application code, system packages, binaries, libraries, configuration files, and the operating system running in the container.

For example, assume we're developing an order tracking portal that our company's various outlets will use. We need to look at the complete stack of software that will run our web application. The application we're building is a .NET Core MVC app, and we plan to deploy the application using Nginx as a reverse proxy server on Ubuntu Linux. All of these software components form part of the container image.

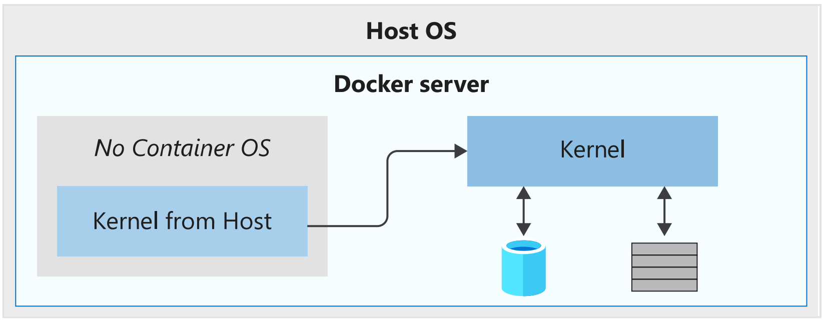
What is a container image?

A container image is a portable package that contains software. It's this image that, when run, becomes our container. The container is the in-memory instance of an image.

A container image is immutable. Once you've built an image, the image can't be changed. The only way to change an image is to create a new image. This feature is our guarantee that the image we use in production is the same image used in development and QA.

What is the host OS?

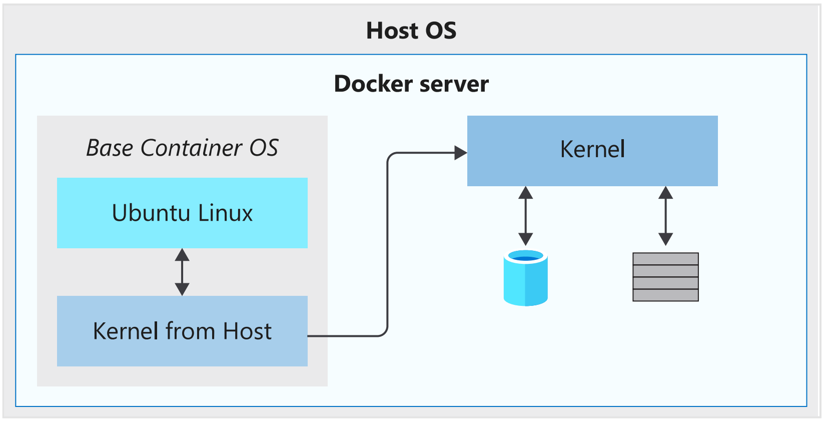
The host OS is the OS on which the Docker engine runs. Docker containers running on Linux share the host OS kernel and don't require a container OS as long as the binary can access the OS kernel directly.



However, Windows containers need a container OS. The container depends on the OS kernel to manage services such as the file system, network management, process scheduling, and memory management.

What is the container OS?

The container OS is the OS that is part of the packaged image. We have the flexibility to include different versions of Linux or Windows OSs in a container. This flexibility allows us to access specific OS features or install additional software our applications may use.

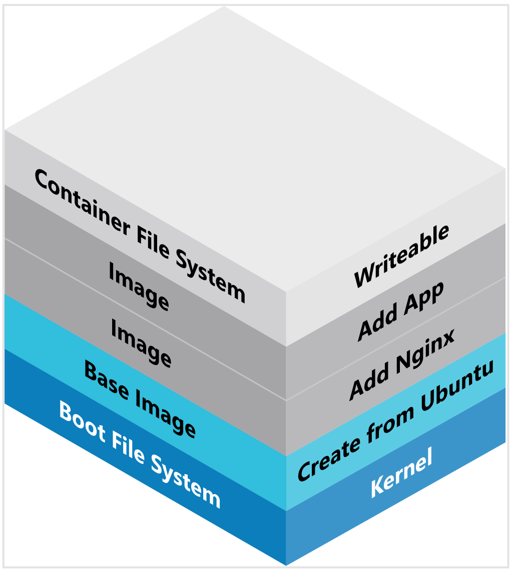


The container OS is isolated from the host OS and is the environment in which we deploy and run our application. Combined with the image's immutability, this isolation means the environment for our application running in development is the same as in production.

In our example, we're using Ubuntu Linux as the container OS and this OS doesn't change from development or production. The image we use is always the same.

What is the Stackable Unification File System (`Unionfs`)?

`Unionfs` is used to create Docker images. `Unionfs` is a filesystem that allows you to stack several directories, called branches, in such a way that it appears as if the content is merged. However, the content is physically kept separate. `Unionfs` allows you to add and remove branches as you build out your file system.



For example, assume we're building an image for our web application from earlier. We'll layer the Ubuntu distribution as a base image on top of the boot file system. Next we'll install Nginx and our web app. We're effectively layering Nginx and the web app on top of the original Ubuntu image.

A final writeable layer is created once the container is run from the image. This layer however, does not persist when the container is destroyed.

What is a base image?

A base image is an image that uses the Docker `scratch` image. The `scratch` image is an empty container image that doesn't create a filesystem layer. This image assumes that the application you're going to run can directly use the host OS kernel.

What is a parent image?

A parent image is a container image from which you create your images.

For example, instead of creating an image from `scratch` and then installing Ubuntu, we'll rather use an image already based on Ubuntu. We can even use an image that already has Nginx installed. A parent image usually includes a container OS.

What is the main difference between base and parent images?

Both image types allow us to create a reusable image. However, base images allow us more control over the contents of the final image. Recall from earlier that an image is immutable, you can only add to an image and not subtract.

What is a Dockerfile?

A Dockerfile is a text file that contains the instructions we use to build and run a Docker image. The following aspects of the image are defined:

- The base or parent image we use to create the new image
- Commands to update the base OS and install additional software
- Build artifacts to include, such as a developed application
- Services to expose, such a storage and network configuration
- Command to run when the container is launched

Let's map these aspects to an example Dockerfile. Suppose we're creating a Docker image for our ASP.NET Core website. The Dockerfile may look like the following example.

Bash Copy

```bash
# Step 1: Specify the parent image for the new image
FROM ubuntu:18.04

# Step 2: Update OS packages and install additional software
RUN apt -y update &&  apt install -y wget nginx software-properties-common apt-transport-https \
    && wget -q https://packages.microsoft.com/config/ubuntu/18.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb \
    && dpkg -i packages-microsoft-prod.deb \
    && add-apt-repository universe \
    && apt -y update \
    && apt install -y dotnet-sdk-3.0

# Step 3: Configure Nginx environment
CMD service nginx start

# Step 4: Configure Nginx environment
COPY ./default /etc/nginx/sites-available/default

# STEP 5: Configure work directory
WORKDIR /app

# STEP 6: Copy website code to container
COPY ./website/. .
```

```
# STEP 7: Configure network requirements
EXPOSE 80:8080

# STEP 8: Define the entry point of the process that runs in the container
ENTRYPOINT ["dotnet", "website.dll"]
```

We're not going to cover the Dockerfile file specification here or the detail of each command in our above example. However, notice that there are several commands in this file that allow us to manipulate the structure of the image.

Recall, we mentioned earlier that Docker images make use of *unionfs*. Each of these steps creates a cached container image as we build the final container image. These temporary images are layered on top of the previous and presented as single image once all steps complete.

Finally, notice the last step, step 8. The `ENTRYPOINT` in the file indicates which process will execute once we run a container from an image.

How to manage Docker images

Docker images are large files that initially get stored on your PC and we need tools to manage these files.

The Docker CLI allows us to manage images by building, listing, removing, and running them. We manage Docker images by using the `docker` client. The client doesn't execute the commands directly and sends all queries to the `dockerd` daemon.

We aren't going to cover all the client commands and command flags here, but we'll look at some of the most used commands. The *learn more* section of this module includes links to Docker documentation, which covers all commands and command flags in detail.

How to build an image

We use the `docker build` command to build Docker images. Let's assume we use the Dockerfile definition from earlier to build an image. Here is an example that shows the build command.

Bash Copy

```bash
docker build -t temp-ubuntu .
```

Here is the output generated from the build command:

Output Copy

```
Sending build context to Docker daemon  4.69MB
Step 1/8 : FROM ubuntu:18.04
 ---> a2a15febcdf3
Step 2/8 : RUN apt -y update && apt install -y wget nginx software-properties-
common apt-transport-https && wget -q
https://packages.microsoft.com/config/ubuntu/18.04/packages-microsoft-prod.deb -O
packages-microsoft-prod.deb && dpkg -i packages-microsoft-prod.deb && add-apt-
repository universe && apt -y update && apt install -y dotnet-sdk-3.0
 ---> Using cache
 ---> feb452bac55a
Step 3/8 : CMD service nginx start
 ---> Using cache
 ---> ce3fd40bd13c
Step 4/8 : COPY ./default /etc/nginx/sites-available/default
 ---> 97ff0c042b03
Step 5/8 : WORKDIR /app
 ---> Running in 883f8dc5dcce
Removing intermediate container 883f8dc5dcce
 ---> 6e36758d40b1
Step 6/8 : COPY ./website/. .
 ---> bfe84cc406a4
Step 7/8 : EXPOSE 80:8080
 ---> Running in b611a87425f2
Removing intermediate container b611a87425f2
 ---> 209b54a9567f
Step 8/8 : ENTRYPOINT ["dotnet", "website.dll"]
 ---> Running in ea2efbc6c375
Removing intermediate container ea2efbc6c375
 ---> f982892ea056
Successfully built f982892ea056
Successfully tagged temp-ubuntu:latest
```

Don't worry if you don't understand the above output. However, notice the steps listed in the output. When each step executes, a new layer gets added to the image we're building.

Also, notice that we execute a number of commands to install software and manage configuration. For example, in step 2, we run the `apt -y update` and `apt install -y` commands to update the OS. These commands execute in a running container that is created for that step. Once the command has run, the intermediate container is removed. The underlying cached image is kept on the build host and not automatically deleted. This optimization ensures that later builds reuse these images to speed up build times.

What is an image tag?

An image tag is a text string that is used to version an image.

In the example build from earlier, notice the last build message that reads "Successfully tagged *temp-ubuntu: latest*". When building an image, we name and optionally tag the image using the `-t` command flag. In our example, we named the image using `-t temp-ubuntu`, while the resulting image name was tagged *temp-ubuntu: latest*. An image is labeled with the `latest` tag if you don't specify a tag.

A single image can have multiple tags assigned to it. By convention, the most recent version of an image is assigned the *latest* tag and a tag that describes the image version number. When you release a new version of an image, you can reassign the latest tag to reference the new image.

Here is another example. Suppose you want to use the .NET Core samples Docker images. Here we have four platforms versions that we can choose from:

- `mcr.microsoft.com/dotnet/core/samples:dotnetapp`
- `mcr.microsoft.com/dotnet/core/samples:aspnetapp`
- `mcr.microsoft.com/dotnet/core/samples:wcfservice`
- `mcr.microsoft.com/dotnet/core/samples:wcfclient`

How to list images

The Docker software automatically configures a local image registry on your machine. You can view the images in this registry with the `docker images` command.

code Copy

```
docker images
```

The output looks like the example below.

Output Copy

```
REPOSITORY          TAG                     IMAGE ID            CREATED
SIZE
tmp-ubuntu          latest             f89469694960        14 minutes ago
1.69GB
tmp-ubuntu          version-1.0        f89469694960        14 minutes ago
1.69GB
ubuntu              18.04                   a2a15febcdf3        5 weeks ago
64.2MB
```

Notice how the image is listed with its *Name*, *Tag*, and an *Image ID*. Recall that we can apply multiple labels to an image. Here is such an example. Even though the image names are different, we can see the IDs are the same.

The image ID is a useful way to identify and manage images where the name or tag of an image might be ambiguous.

How to remove an image

You can remove an image from the local docker registry with the `docker rmi` command. Specify the name or ID of the image to remove. This example removes the image for the sample web app using the image name:

code Copy

```
docker rmi temp-ubuntu:version-1.0
```

You can't remove an image if the image is still in use by a container. The `docker rmi` command returns an error message, which lists the container relying on the image.

We've explored the basics of Docker images, how to manage these images and how to run a container from an image. Next, we'll look at how to manage containers.

Check your knowledge

1. On which of the following operating systems does Docker for desktop run?

- ○ Windows only
- ○ Linux and Windows only
- [✓ Linux, macOS, and Windows] ✓

The desktop version of Docker runs on Linux, macOS, and Windows.

2. Which is correct Docker command to rebuild a container image?

- ○ docker rebuild
- ○ docker compile
- [✓ docker build] ✓

You use the docker build command to rebuild a container image. Once you've built an image, the image can't be changed. The only way to change an image is to create a new image.

3. Which of the following sentences describe a container image the best?

- [✓ A container image is a read-only portable package that contains software and may include an operating system.] ✓

  A container image is an immutable package that contains all the application code, system packages, binaries, libraries, configuration files, and the operating system running in the container. Docker containers running on Linux share the host OS kernel and don't require a container OS as long as the binary can access the OS kernel directly.

- A container image is a set of commands that builds a container.
- A container image is a read-only portable package that contains software.

Next unit: How Docker containers work

Continue >