GOAL

To provide you with the tools you need to create and run your first containerized application using Docker, then be able to look for more by yourself when needed.

WHY DOCKER

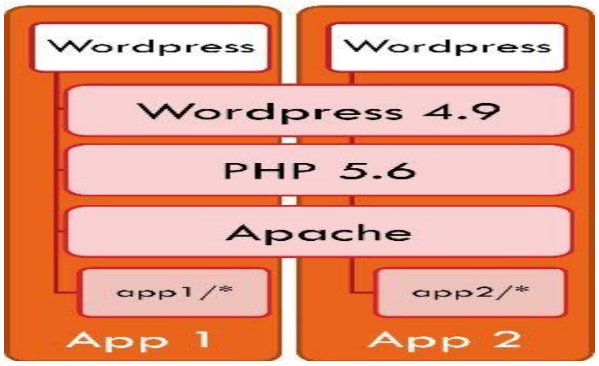
Docker is an engine that runs containers. As a tool, containers allow you to solve many challenges created in the growing DevOps trend.

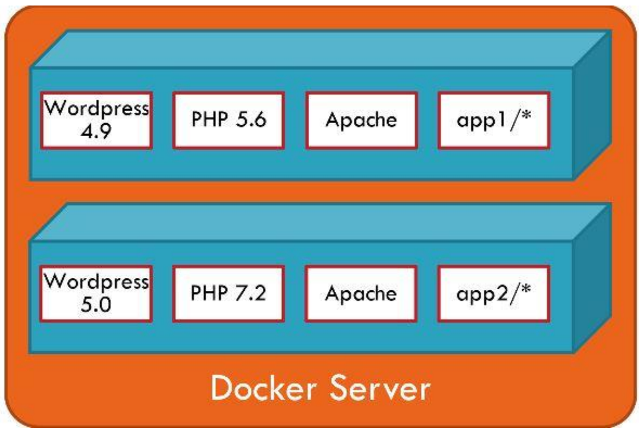
In DevOps, the Dev and Ops teams have conflicting goals:

Dev Team Seeks	**Ops Team Seeks**
Frequent deployments and updates	Stability of production apps
Easy creation of new resources	Manage infrastructure, not applications
	Monitoring and control

Containers make deployment easy. Deploying is as simple as running a new container, routing users to the new one, and trashing the old one. It can even be automated by orchestration tools. Since it's so easy, we can afford to have many containers serving a single application for increased stability during updates.

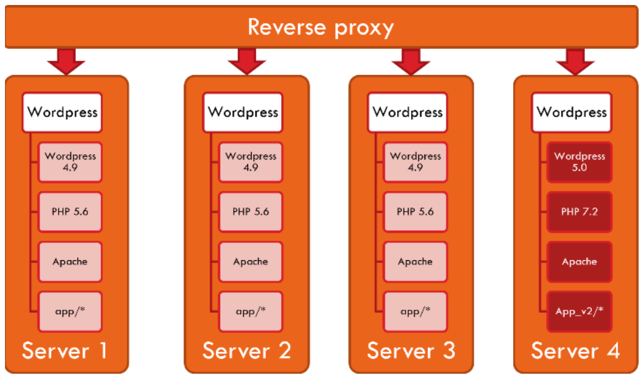
Typical Application
Now suppose you want to upgrade the PHP runtime from version 5.6 to 7.2. However, the version change induces breaking changes in the applications that therefore need to be updated. You need to update both *App 1* and *App 2* when proceeding with the upgrade. On a server that may host many apps of this type, this is going to be a daunting task, and you'll need to delay the upgrade until all apps are ready.
Another similar problem is when you want to host *App 3* on the same server, but *App 3* uses the Node.JS runtime together with a package that, when installed, changes a dependency used by the PHP runtime.





Allows Easy Scaling Up

When a server application needs to handle a higher usage than what a single server can handle, the solution is well-known, place a reverse proxy in front of it, and duplicate the server as many times as needed. That is only going to make things worse when upgrading: we'll need to upgrade each server's dependencies together with all of the conflicts that may induce. The update process depends on the application and its dependencies.



Again, containers have a solution for this. As we'll see in the *Basic Concepts* chapter, containers are based on images. You can run as many containers as you wish from a single image — all the containers will support the exact same dependencies. When using an orchestrator, you merely need to state how many containers you want and the image name and the orchestrator creates that many containers on all of your Docker servers. By using containers, it's a simple matter of telling the orchestrator that you want to run a new image version, and it gradually replaces every container with another one running the new version

.

When you create an image, you stuff your software into a container image. When a machine runs that image, a container is created. Container images and containers can be managed in a standardized way, which allows for standard solutions during a containerized software's lifecycle:

- common build chain
- common image storage
- common way to deploy and scale-up
- common hosting of running containers
- common control and monitoring of running containers
- common ways to update running containers to a new version

Various Products for Various Needs

Use	**Product**
Developer machine	*Docker Engine Community* or *Docker Desktop*
Small server, small expectations	*Docker Engine Community*
Serious stuff/Critical applications	*Docker Engine Enterprise* or *Kubernetes*

```
docker run hello-world
```

1. Your command asks Docker to create and run a container based on the *hello-world* image.
2. Since the *hello-world* image wasn't already present on your disk, Docker downloaded it from a default registry, the *Docker Hub*. More about that later.
3. Docker created a container based on the *hello-world* image.
4. The *hello-world* image states that, when started, it should output some text to the console, so this is the text you see as the container is running.
5. The container stopped.

Basic Concepts

Containers

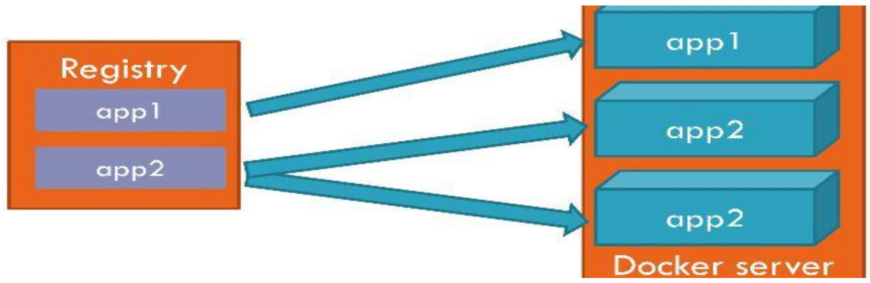
A *container* is what we eventually want to run and host in Docker. A *container* runs inside the Docker host isolated from the other containers and even the host OS. It cannot see the other containers, physical storage, or get incoming connections unless you explicitly state that it can. It contains everything it needs to run: OS, packages, runtimes, files, environment variables, standard input, and output.

Images

Any container that runs is created from an *image*. An image describes everything that is needed to create a container; it is a template for containers.

Registries

Images are stored in a *registry*. In the example above, the *app2* image is used to create two containers. Each container lives its own life, and they both share a common root: their image from the registry.



You may use the following commands for container management:

- *docker ps*: lists the containers that are still running. Add the **-a** switch in order to see containers that have stopped
- *docker logs*: retrieves the logs of a container, even when it has stopped
- *docker inspect*: gets detailed information about a running or stopped container
- *docker stop*: Stops a container that is running
- *docker rm*: deletes a container

```
• docker ps -a
• docker logs 48bab7f673b3
• docker inspect 48bab7f673b3
• docker rm 48bab7f673b3
```

More About Docker Run

You can think of the *docker run* command as the equivalent of buying a new computer. Each time a container is created from an image, you get a new isolated environment to play with inside that container.

```
docker container prune -f
```

-f : forcefully deletes all the stopped containers

Running a Server Container

We just saw how to run short-lived containers. They usually do some processing and display some output. However, there's a very common use for long-lived containers: server containers. Whether you want to host a web application, an API, or a database, you want a container that listens for incoming network connections and is potentially long-lived.

In short, a server container

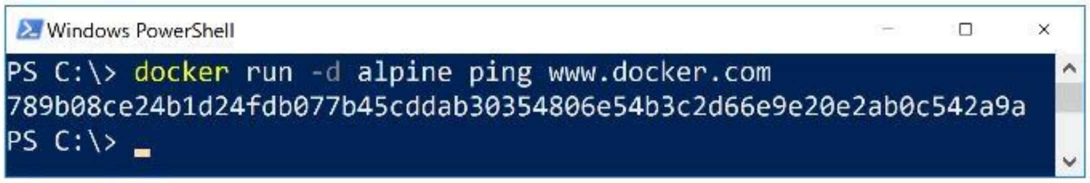
- is long-lived
- listens for incoming network connections

Running a long-lived container

1. Up until now, we remained connected to the container from our command line using the *docker run* command, making it impractical for running long-lived containers.
2. To disconnect while allowing the long-lived container to continue running in the background, we use the *-d* or *–detach* switch on the *docker run* command.
3. Running a container as detached means that you immediately get your command-line back and the standard output from the container is not redirected to your command-line anymore.
4.
```
docker run -d alpine ping www.docker.com
```

The ping command doesn't end since it keeps pinging the Docker server. That's a long-lived container.

Note the addition of a *-d* switch. When doing so, the container starts, but we don't see its output. Instead, the *docker run* command returns the ID of the container that was just created:



```
PS C:\> docker run -d alpine ping www.docker.com
789b08ce24b1d24fdb077b45cddab30354806e54b3c2d66e9e20e2ab0c542a9a
PS C:\> _
```

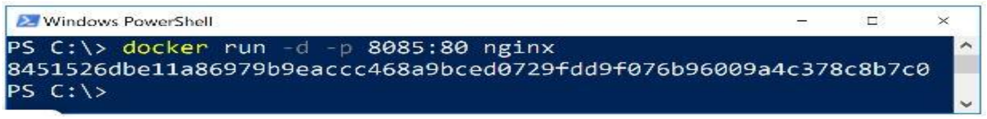
I can interact with the running container using the commands we saw above: *docker logs* to see its output, *docker inspect* to get detailed information and even *docker stop* in order to kill it.

Listening for Incoming Network Connections

By default, a container runs in isolation, and as such, it doesn't listen for incoming connections on the machine where it is running. You must explicitly open a port on the host machine and map it to a port on the container.
Suppose I want to run the NGINX web server. It listens for incoming HTTP requests on port 80 by default. If I simply run the server, my machine does not route incoming requests to it unless I use the *-p* switch on the *docker run* command.

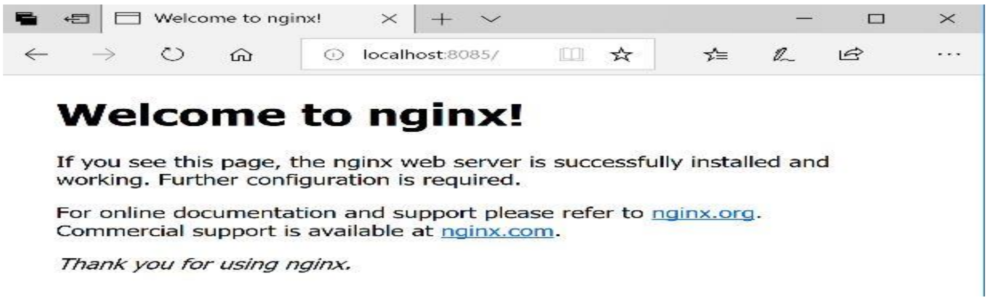
```
docker run -d -p 8085:80 nginx
```

The *-p* switch takes two parameters; the incoming port you want to open on the host machine, and the port to which it should be mapped inside the container. For instance, here is how I state that I want my machine to listen for incoming connections on port 8085 and route them to port 80 inside a container that runs NGINX:



```
Windows PowerShell
PS C:\> docker run -d -p 8085:80 nginx
8451526dbe11a86979b9eaccc468a9bced0729fdd9f076b96009a4c378c8b7c0
PS C:\>
```

Now I can run a browser and query that server using the *`http://localhost:8085`* URL:



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

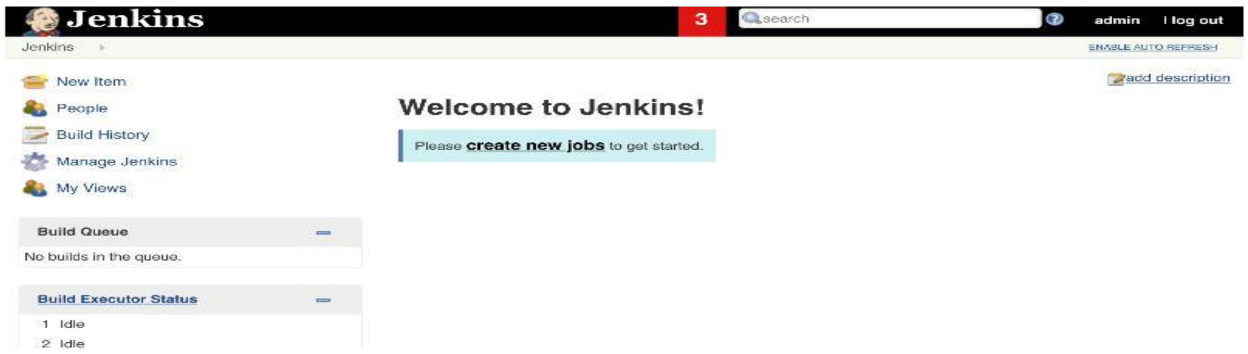
One thing I particularly like about containers is that they allow me to use any software without polluting my machine. Usually, you would hesitate before trying a new piece of software on your machine since it means installing several dependencies that may interfere with existing software and be leftover should you change your mind and uninstall the main software. Thanks to containers, I can even try big pieces of server software without polluting my machine.

Let me run a Jenkins server to illustrate that point. Jenkins is a full continuous integration server coded using Java. Thanks to Docker I don't need to install Java or any dependency on my machine in order to run a Jenkins server. Jenkins listens by default on port 8080, so I can go away and type:

```
docker run -p 8088:8080 jenkins
```

Note that I could add a *-d* switch since this is a long-running process. However, I am not using it here because I want to directly see the verbose output. For a real deployment, I would use the *-d* switch and inspect the output with the *docker logs* command when needed.

Then I can point my browser to `http://localhost:8088` and finish the setup:



Welcome to Jenkins!

Please **create new jobs** to get started.

No builds in the queue.

1 Idle
2 Idle

Should I decide not to continue with Jenkins and try another continuous integration server, I can simply run the *docker stop* and *docker rm* commands. I could also run two separate Jenkins servers by just executing the *docker run* command again using another port.

When using such images, you could wonder about where the data is stored. Docker uses **volumes** for this. Also, databases may be needed for storing data, and those may be run in containers as well.

Using Volumes

Volumes are used to Map a directory inside a specified container to a persistent storage. Lets see how?

When a container writes files, it writes them *inside* of the container. Which means that when the container dies (either due to the host machine restart, the container is moved from one node to another in a cluster, failure in container, etc.) all of that data is lost. It also means that if you run the same container several times in a load-balancing scenario, each container will have its own data, which may result in inconsistent user experience.

A rule of thumb for the sake of simplicity is to ensure that containers are stateless, for instance, storing their data in an external database (relational like an SQL Server or document-based like MongoDB) or distributed cache (like Redis). However, sometimes you want to store files in a place where they are persisted; this is done using volumes.

Using a volume, you map a directory inside the container to a persistent storage. Persistent storages are managed through drivers, and they depend on the actual Docker host. They may be an Azure File Storage on Azure or Amazon S3 on AWS. With Docker Desktop, you can map volumes to actual directories on the host system; this is done using the *-v* switch on the *docker run* command.

```
docker run -d mongo
```

Suppose you run a MongoDB database with no volume: Any data stored in that database will be lost when the container is stopped or restarted. In order to avoid data loss, you can use a volume mount:

```
docker run -v /your/dir:/data/db -d mongo
```

It will ensure that any data written to the */data/db* directory inside the container is actually written to the */your/dir* directory on the host system. This ensures that the data is not lost when the container is restarted.

Where Do Images Come From?

Each container is created from an image. You provide the image name to the *docker run* command. Docker first looks for the image locally and uses it when present. When the image is not present locally, it is downloaded from a *registry*.

docker images ls   You can list the local images using the following command

Format used for image name when an image is published to a registry:

```
<repository_name>/<name>:<tag>
```

- *tag* is optional; when missing, it is considered to be *latest* by default
- *repository_name* can be a registry DNS or the name of a registry in the Docker Hub

All of the images we've been using until now were downloaded from Docker Hub as they are not DNS names. When you have time, you should browse the Docker Hub and get familiar with the images it provides.

Although the *docker run* command downloads images automatically when missing, you may want to trigger the download manually.

docker pull : A pull command forces an image to download, whether it is already present or not.

- You want to ensure you have the latest version of an image tagged as "latest," which wouldn't be downloaded by the *docker run* command.
- You expect that the machine which runs the containers does not have access to the registries (e.g., no internet connection) at the time of running the containers.

Creating a Simple Image

The *Dockerfile* file contains instructions on how the image should be built. A *Dockerfile* file always begins with a *FROM* instruction because every image is based on another base image. This is a powerful feature since it allows you to extend images that may already be complex.

Just to make things crystal clear, here's what I did:

- Create an image:
    - Create a file named *Dockerfile*

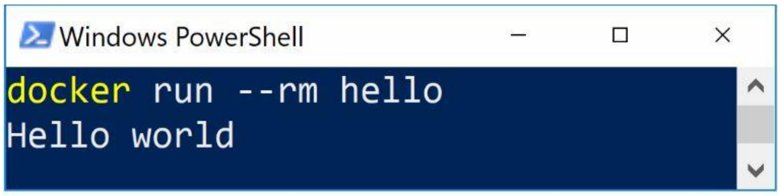
```
•  FROM debian:8
•
•  CMD ["echo", "Hello world"]
```

- Run a *docker build* command

```
docker build -t hello .
```

- Run a container from the image created

```
docker run --rm hello
```


Windows PowerShell

```
docker run --rm hello
Hello world
```

I'd like to expand on what this means; running a container is the virtual equivalent of starting a brand-new machine and then trashing it. In order to print that "Hello world" message, we essentially got a new computer, had it execute an *echo* command, and then trashed it. Docker makes fire-and-forget computing cheap.

The image we just created in the previous lesson didn't need anything other than what the base image already contained, which is why our *Dockerfile* file was so simple. In a real-world scenario, however, I'm very likely to want files to be part of an image I create.

Suppose I have a file named *index.html* on my disk with the following contents:

```
<html>

<body>

<h1>Hello !</h1>

    <div>I'm hosted by a container.</div>
  </body>
</html>
```

The Docker Hub contains an NGINX image where NGINX has already been installed with a configuration that serves files found in the */usr/share/nginx/html* directory.

I create the following *Dockerfile* file in the same folder as the HTML file:

```
FROM nginx:1.15

COPY index.html /usr/share/nginx/html
```

Apart from the *nginx* base image, you can see a *COPY* instruction. Its first parameter is the file to be copied from the build context and its second parameter is the destination directory inside the image.

You may have noticed that this time the *Dockerfile* file contains no *CMD* instruction. Remember that the *CMD* instruction states which executable should be run when a container is created from my image, so it's weird that I don't include a *CMD* instruction that runs an NGINX server. The reason why I didn't include a *CMD* instruction is because the base *nginx:1.15* image already contains a *CMD* instruction to run the NGINX server. This is part of my image and I don't need to include my own *CMD* instruction as long as I don't want to run *another* executable on container startup.

```
docker build -t webserver .
docker run --rm -it -p 8082:80 webserver
```

The above commands build a *webserver* from the *Dockerfile* file instructions, then start a container that listens to my machine's 8082 port and redirect the incoming connections to the container's 80 port. I can start a browser and point it to *http://localhost:8082*

- The *-it* switch allows me to stop the container using *Ctrl-C* from the command-line
- The *–rm* switch ensures that the container is deleted once it has stopped

When you issue a `docker build` command, the current working directory is called the *build context*. By default, the Dockerfile is assumed to be located here, but you can specify a different location with the file flag (`-f`). Regardless of where the `Dockerfile` actually lives, all recursive contents of files and directories in the current directory are sent to the Docker daemon as the build context.

Create a base image

A parent image is the image that your image is based on. It refers to the contents of the `FROM` directive in the Dockerfile. Each subsequent declaration in the Dockerfile modifies this parent image. Most Dockerfiles start from a parent image, rather than a base image. However, the terms are sometimes used interchangeably.

A base image has `FROM scratch` in its Dockerfile.

You can use Docker's reserved, minimal image, `scratch`, as a starting point for building containers. Using the `scratch` "image" signals to the build process that you want the next command in the `Dockerfile` to be the first filesystem layer in your image.

While `scratch` appears in Docker's repository on the hub, you can't pull it, run it, or tag any image with the name `scratch`. Instead, you can refer to it in your `Dockerfile`. For example, to create a minimal container using `scratch`:

```
FROM scratch
ADD hello /
CMD ["/hello"]

docker build --tag hello .
docker run --rm hello
```

Building your application in a Docker based project requires you to create your own custom Docker images.

Your Docker images contain your application’s source code, server binary, configuration and other dependencies.

The image that you use as the starting point to define a custom Docker image is called the **parent image**. This is the image that you specify with the `FROM` instruction in the Dockerfile.

Most of the time people call the parent image, the base image. I also do so and other people I work with do so. According to the official terminology, however the parent image is not the same as the base image. Still everybody calls all starting images base image, so you could just stop reading here. :)

To be precise however, the base image is an image that you build from scratch. This means that you are not using another Docker image as a starting point to add your image layers on top. way is to use `FROM scratch` in your Dockerfile.

Base images are usually OS base systems that are created as described above. I hope now the official documentation is easier to understand:

“A [base image](#) either has no `FROM` line in its Dockerfile, or has `FROM scratch`.”

Docker base image is the basic image on which you add layers (which are basically filesystem changes) and create a final image containing your App.
For example, in order to run a LAMP stack as a Docker containers, you might use either an Ubuntu 14.04 or Ubuntu 12.04 or CentOS 7, or any of your Linux OSes as a base image. Then, you would install Apache, MySQL and PHP on it and the result would be your final LAMP Docker image which can be run as a container.

As for your next question, with Linux Containers, we do not install the full OS. Rather, imagine it as a snapshot of an OS's filesystem. So a Ubuntu base image is like a snapshot of a Ubuntu filesystem. So, it does not have all the drivers installed on it like a full fledged Hyper-visor.

If the `WORKDIR` command is not written in the Dockerfile, it will automatically be created by the Docker compiler. Hence, it can be said that the command performs `mkdir` and `cd` implicitly.

Here’s a sample Dockerfile in which the working directory is set to `/project`:

```
FROM ubuntu:16.04
WORKDIR /project
RUN npm install
```

If the `project` directory does not exist, it will be created. The `RUN` command will be executed inside `project`.

Images Are Created Locally

When I run the *docker build* command to create an image from a *Dockerfile* file, the resultant image is stored locally on the computer where the *docker build* command is run.

I can see the images available locally on my computer by running the following command:

```
docker image ls
```

Having the images readily available locally makes it faster to run a container from them. However, there will be a time when some images are useless. I can remove them from my local machine using the *docker rmi* command and providing it the image name or image ID.

```
docker rmi c067edac5ec1
docker rmi webserver:latest
```

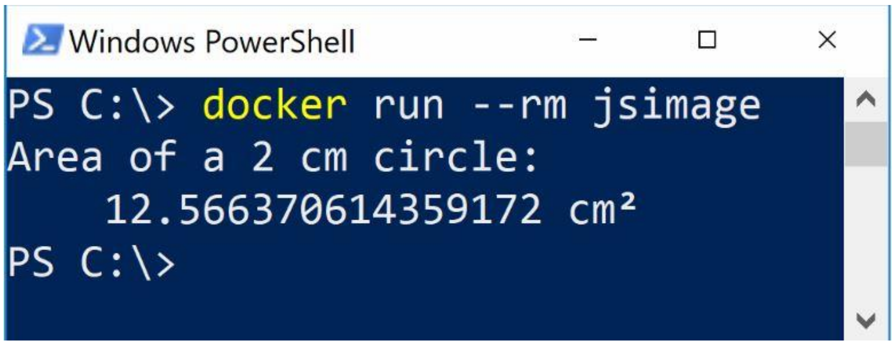
Exercise: Build an Image and Run It

1. Create a file named *compute.js* with the following code that computes and displays the area of a disk using JavaScript:
2. Create a Docker image that runs this code using the *node:11-alpine* image that contains Node.JS. The command that runs a JavaScript program using Node.JS is `node <filename>`.

Create a file named *Dockerfile* and add the following code to it:

```
3. FROM node:11-alpine
4. COPY compute.js .
5. CMD node compute.js
```

- Open a command-line. Change the current directory to the directory where you created the *Dockerfile* and the *compute.js* file
- Run the following command:
- `docker build -t jsimage .`
- Once the image has been built, run the following command:
- `docker run --rm jsimage`



```
PS C:\> docker run --rm jsimage
Area of a 2 cm circle:
    12.566370614359172 cm²
PS C:\>
```

Tags Matter

```
<repository_name>/<name>:<tag>
```

- *tag* is optional; when missing, it is considered to be *latest* by default
- *repository_name* can be a registry DNS or the name of a registry in the Docker Hub

  While your images aren't published to a registry (at least in this chapter, but that will change in the next one), you don't need to include a registry name. So, your image name is:

```
<name>:<tag>
```

```
<repository_name>/<name>:<tag>
```

Why Would You Tag Your Images? #

Other reasons come to mind once you become more serious with your CI/CD pipeline. For instance, you may want any or all of the following features:

- Be able to roll back to a previous version of an image if you detect a problem with the latest image.
- Run different versions in different environments. For instance, the latest version in a test environment and the previous version in a production environment.
- Run different versions at the same time, routing some users to the latest version and some to the previous versions. This is known as a canary release.
- Deploy different versions to different users, and be able to run whatever version on your development machine while you support them.

You're free to tag your images however you want. Common tags include:

- a version number, e.g. *hello:1.0*, *hello:1.1*, *hello:1.2*
- a Git commit tag, e.g. *hello:2cd7e376*, *hello:b43a14bb*

In order to apply a tag, just state it during your build command:

```
docker build -t hello:1.0 .
```

Tags for Base Images

It's quite tempting to base your images on the latest ones so that you're always running on up-to-date software, especially since it's so straightforward. All you need to do is omit the tag altogether or mention the *latest* one. You could be tempted to use the following instruction in your *Dockerfile* file:

```
FROM nginx:latest
```

Don't! First of all, it doesn't mean that any running container will be based on the latest available version of the *nginx* image. Docker is about having reproducible images, so the *latest* version is evaluated when you build your image, not when the container is run. This means that the version will not change unless you run the *docker build* command again.

Second, you're likely to run into trouble. What about the *nginx* image releasing a new version with breaking changes? If you build your image again, you're likely to get a broken image.

For these reasons, I recommend specifying the image tag. If you want to keep up to date with new releases of the base image, update the tag manually and make sure you test your image before releasing it.

Parameters as Environment Variables

Environment variables help programs know what directory to install files in, where to store temporary files, and where to find user profile settings. They help shape the environment in which programs run on your computer.

Environment variables are global system **variables** accessible by all the processes running under the Operating System (OS). **Environment variables** are useful to store system-wide values such as the directories to search for the executable programs (PATH) and the OS version.

Storage

```
VOLUME /path/to/directory
```

The */path/to/directory* is a path to a directory used inside the image. When a container is created using the *docker run* command, the *-v* switch can be used to map this directory to an actual volume on the host system.

This is only an indication. By default, if the user doesn't map this volume to an external store for the container, the data will be stored inside the container.

Networking

When your image hosts server software, it listens on one or several ports. For instance, an HTTP server generally listens on the TCP port 80.

```
EXPOSE 80
```

Using this instruction is purely for documentation purposes. It will NOT open a port to the outside world when a container is created from that image. Anyone who creates a container will need to explicitly bind that port to an actual port of the host machine using the *-p* switch of the *docker run* command.

Rationale for Publishing

As a developer, you create images for your software to run in a controlled environment. Since you want your images to run on other machines, you need to make sure they are distributed to those machines. Your option is to publish your images to a Registry. When the other machines need to create containers from your images, they will simply pull them from the Registry.

Registries

A Docker Registry is basically an image store that offers the following functions:

- Ability to store various images.
- Ability to store various tags for the same image.
- An HTTP API that allows pushing images from a machine that produces them, or pull images to a machine that runs containers from those images.
- TLS-secured connection to the API in order to avoid man-in-the-middle attacks.

There are many registries available. You can use the publicly available Docker Hub or use a private registry of your own.

Publishing an Image

Whichever Registry you choose, publishing an image is a three-step process:

1. Build your image (*docker build*) with the appropriate prefix name or tag (*docker tag*) an existing one appropriately.
2. Log into the Registry (*docker login*).
3. Push the image into the Registry (*docker push*).

Docker Hub

To publish images on Docker Hub, you need to create an account. I'm going to create one. For this, I head to https://hub.docker.com/ and click the *Sign Up* link. The *Docker ID* I select will be the prefix of images I publish to the Docker Hub.

When creating your account, make sure you select the right ID since it will be part of your images' names. Suppose your ID name is *short-name*, your images should be tagged:

```
<short_name>/<name>:<tag>
```

For this course, I created the *learnbook* account, and I want to publish the *webserver* image I created earlier.

The first is to run the *docker build* command again using the correct name. This is a good option when you're using it from the start, but if I do this now it will result in two separate images on my disk, the same contents but two different names and IDs which is not a good idea.

My second option is much better; use the *docker tag* command. A Docker image can have several names as needed, and they can be added to an already existing image thanks to the *docker tag* command. The *docker tag* doesn't duplicate the image contrary to running *docker build* again.

The *docker tag* command accepts two arguments; first, the name of an existing image, and second, the name you want to add to that image.

```
docker tag webserver learnbook/webserver
```

The *docker login* command asks for my Docker Hub credentials interactively, but I could have just as well provided them as arguments to the command.

```
docker login -u learnbook -p <your-password>
docker push learnbook/webserver
```

Run an Image on Another Machine

This is a straightforward process. In fact, you already know how to do this; use the *docker run* command. Alternatively, get the image with *docker pull* then use it with *docker run*. This would be useful in case you plan to disconnect from the internet before running a container, for instance.

```
docker run --rm -it -p 8085:80 learnbook/webserver
```

Then open your browser and head to the following URL in order to see my web page being served over HTTP by that container.

```
http://localhost:8085
```

Private Registries

Private registries ensure that you can keep your private images private. When we published our images to the Docker Hub using the *docker push* command, we had to use the *docker login* command first in order to authenticate. A private registry will require users to also use *docker login* before they can pull an image.

- **Docker Hub**, where you pay according to the number of private repositories used.

- **Azure Container Registry** allows you to have your own private registry in Azure.

- **GitLab** has an included optional Docker registry; enable it so that each project can store the images it creates.

Size Matters

In order to reduce the size of an image, you need to understand that it is influenced by several factors:

- The files included in your image
- The base image size
- Image layers

Files Included in Your Image

Include only necessary files in your image. That may sound obvious, but it's easy to include unwanted files.

First of all, avoid *COPY* instructions that are too broad. A typical example that should be avoided is:

Try to be as precise as possible. If necessary, split them into separate *COPY* instructions such as:

```
COPY ./Project/src/*.ts ./src
COPY ./Project/package.json .
```

There are times when you will need to use *COPY* instructions that copy whole folders, as below:

```
COPY ./js/built /app/js
```

However, you may want to exclude files from that copy. You can use a *.dockerignore* file

```
# Ignore .git folder
.git
# Ignore Typescript files in any folder or subfolder
**/*.ts
```

Base Image Size

Many images you can use as base images have smaller variants. For instance, the *debian* image I used in the *Create Docker Images* chapter is quite big. Here is the definition I used:

Dockerfile

```
FROM debian:8
CMD ["echo", "Hello world"]
```

That definition results in an image that weights **127 MB**. The size can be checked using the *docker image ls* command.

If I head to the description page of that image, I can see that there is a smaller variant using the *8-slim* tag. Here is an optimized definition:

Dockerfile

```
FROM debian:8-slim
CMD ["echo", "Hello world"]
```

That revised definition yields an image that weights **79.3 MB**. We just saved **38%** of the original image size.

The same can be done using a *node:alpine* image instead of the default *node* image. For instance, the image definition I used for my *webserver* image results in a **109 MB** image:

If I switch the base image from *nginx:1.15* to *nginx:1.15-alpine*, the resulting image weights only **16.1 MB**.

```
FROM nginx:1.15-alpine
COPY index.html /usr/share/nginx/html
```

Image Layers

When creating an image, Docker reads each instruction in order and the resulting partial image is kept separate; it is cached and labeled with a unique ID. Such caching is very effective because it is used at different moments of an image life:

- In a future build, Docker will use the cached part instead of recreating it as long as it is possible.
- When pushing a new version of the image to a Registry, the common part is not pushed.

- When pulling an image from a registry, the common part you already have is not pulled.

```
Reading state information...
The following extra packages will be installed:
  apache2 apache2-bin apache2-data apache2-utils file ifupdown isc-dhcp-
client
  isc-dhcp-common krb5-locales libalgorithm-c3-perl libapache2-mod-php5
[...]
Need to get 23.2 MB of archives.
After this operation, 88.3 MB of additional disk space will be used.
Get:1 http://deb.debian.org/debian/ jessie/main libgdbm3 amd64 1.8.3-
13.1 [30.0 kB]
Get:2 http://deb.debian.org/debian/ jessie/main libjson-c2 amd64 0.11-
4 [24.8 kB]
[...]
Preparing to unpack .../libkrb5support0_1.12.1+dfsg-19+deb8u5_amd64.deb ...
Unpacking libkrb5support0:amd64 (1.12.1+dfsg-19+deb8u5) ...
Selecting previously unselected package libk5crypto3:amd64.
[...]
Removing intermediate container d43a7a64606d
 ---> 3499256d8527
Successfully built 3499256d8527
```

Since this is the first build, each instruction is processed. However, each of the 4 steps has been cached.

If I run the same *docker build* command without changing anything, I can benefit from the cached layers. The whole build process takes less than a second and here's the output:

```
•  Step 1/4 : FROM debian:8
    ---> ec0727c65ed3
   Step 2/4 : COPY . .
    ---> Using cache
    ---> cb8153c1f09a
   Step 3/4 : RUN apt-get update && apt-get upgrade && apt-get dist-
   upgrade -y
    ---> Using cache
    ---> f4a652ba0849
   Step 4/4 : RUN apt-get install -y php5
    ---> Using cache
    ---> 3499256d8527
   Successfully built 3499256d8527
```

That's efficient but unrealistic. In real life I would rebuild my image because some of the files used in the *COPY* step changed. Let's change a file in the current directory and run the exact same *docker build* command. We can see that the cache isn't used, and the build process takes several seconds again because it needs to download, unpack, and install all the packages:

```
Step 1/4 : FROM debian:8
 ---> ec0727c65ed3
Step 2/4 : COPY . .
```

```
 ---> edec0c4e4746
Step 3/4 : RUN apt-get update && apt-get upgrade && apt-get dist-upgrade -y
 ---> Running in ddc4ad55ab7d
Ign http://deb.debian.org jessie InRelease
Get:1 http://deb.debian.org jessie-updates InRelease [145 kB]
Get:2 http://security.debian.org jessie/updates InRelease [44.9 kB]
[...]
Removing intermediate container ddc4ad55ab7d
 ---> 6d484f51c8f8
Step 4/4 : RUN apt-get install -y php5
 ---> Running in b6ba3afb3cc9
Reading package lists...
Need to get 23.2 MB of archives.
After this operation, 88.3 MB of additional disk space will be used.
[...]
Removing intermediate container b6ba3afb3cc9
 ---> 48c3c3bf2fad
Successfully built 48c3c3bf2fad
```

How can we improve this? Simply by changing the order of the instructions in the *Dockerfile* file. Indeed, the files copied by the *COPY* instruction are more likely to change than the packages needed to install PHP5. We can first install PHP and then copy the files. Here's the improved *Dockerfile* file:

```
FROM debian:8

RUN apt-get update && apt-get upgrade && apt-get dist-upgrade -y
RUN apt-get install -y php5

COPY . .
```

Note that the *COPY* instruction has been placed last. That's the only change.

If I run the *docker build* instruction, then change some files in the folder (as we did earlier), and run the *docker build* command again, here's what we get. On the first *docker build*, the whole build is done, taking up several seconds:

```
FROM debian:8

RUN apt-get update && apt-get upgrade && apt-get dist-upgrade -y
RUN apt-get install -y php5

COPY . .
Step 1/4 : FROM debian:8
 ---> ec0727c65ed3
Step 2/4 : RUN apt-get update && apt-get upgrade && apt-get dist-upgrade -y
[...]
 ---> 219277a22ec2
Step 3/4 : RUN apt-get install -y php5
[...]
 ---> e493f018171f
```

```
Step 4/4 : COPY . .
[...]
 ---> 5d1f58a397c7
Successfully built 5d1f58a397c7
```

Now, if I change a file and run the *docker build* command again, the whole process only takes a few seconds because cached layers are used for everything except the *COPY* instruction:

```
Step 1/4 : FROM debian:8
 ---> ec0727c65ed3
Step 2/4 : RUN apt-get update && apt-get upgrade && apt-get dist-upgrade -y
 ---> Using cache
 ---> 219277a22ec2
Step 3/4 : RUN apt-get install -y php5
 ---> Using cache
 ---> e493f018171f
Step 4/4 : COPY . .
 ---> 08c87dcf7267
Successfully built 08c87dcf7267
```

Note how the ID of the layers resulting from steps **1** to **3** are the same. Only step **4** produces a different layer.

More About Containers

Restart Mode

fadds When running server containers like we did earlier, we want them to always be up. It is very tempting to use the *always* restart mode. For instance:

```
docker run -d -p 80 --restart always nginx
```

Should the container start, or the Docker host itself restart, the container will restart so that it has a high uptime. That’s probably not what you want. If you want your container to always be running *except* when you explicitly stop it, use the *unless_stopped* restart mode:

```
docker run -d -p 80 --restart unless-stopped nginx
```

Monitoring

High availability Docker servers are monitored with tools that are up to the tasks such as collecting your logs and providing usage statistics. For simple needs or your development box however, you may use the following command:

```
docker stats
```

This will output a live list of running containers plus information about how many resources they consume on the host machine. Like a *docker ps* extended with live resource usage data.

Reclaim Your Disk

Here are some ways disk space is consumed unknowingly:

- Stopped containers that were not removed by using the `--rm` switch on the *docker run* command or using the *docker rm* command once they are stopped.
- Unused images: images that are not referenced by other images or containers.
- Dangling images: images that have no name. This happens when you *docker build* an image with the same tag as before, the new one replaces it and the old one becomes dangling.
- Unused volumes.

Most commands ask for an interactive confirmation, but if you want to run them unattended you can add the -f switch.

Here are the commands you can run to remove the items that you don't need:

```
docker container prune -f
docker volume prune -f
docker image prune -f
docker image prune --all
```

https://medium.com/@wkrzywiec/how-to-put-your-java-application-into-docker-container-5e0a02acdd6b

https://medium.com/@wkrzywiec/build-and-run-angular-application-in-a-docker-container-b65dbbc50be8

https://medium.com/@wkrzywiec/database-in-a-docker-container-how-to-start-and-whats-it-about-5e3ceea77e50

`The ENTRYPOINT specifies a command that will always be executed when the container starts. The CMD specifies arguments that will be fed to the ENTRYPOINT.`

1. Dockerfile should specify at least one of `CMD` or `ENTRYPOINT` commands.
2. `ENTRYPOINT` should be defined when using the container as an executable.
3. `CMD` should be used as a way of defining default arguments for an `ENTRYPOINT` command or for executing an ad-hoc command in a container.
4. `CMD` will be overridden when running the container with alternative arguments.

The `ENTRYPOINT` can be overridden by specifying an `--entrypoint` flag, followed by the new entry point you want to use.

```
        FROM ubuntu
        ENTRYPOINT ["sleep"]
        CMD ["10"]
```

The `ENTRYPOINT` is the program that will be run, and the value passed to the container will be appended to it.

What is the difference between the `COPY` and `ADD` commands in a Dockerfile, and when would I use one over the other?

```
COPY <src> <dest>
ADD <src> <dest>
```

Both the instructions will copy new files from `<src>` and add them to the container's filesystem at path `<dest>`

Multiple `<src>` resources may be specified but if they are files or directories, their paths are interpreted as relative to the source of the context of the build.

`ADD` allows `<src>` to be a URL

`COPY` is

Same as 'ADD', but without the tar and remote URL handling.

Note: The first encountered `ADD` instruction will invalidate the cache for all following instructions from the Dockerfile if the contents of `<src>` have changed. This includes invalidating the cache for `RUN` instructions. See the `Dockerfile` Best Practices guide for more information.

- The `<src>` path must be inside the *context* of the build; you cannot `ADD ../something /something`, because the first step of a `docker build` is to send the context directory (and subdirectories) to the docker daemon.

- If `<src>` is a URL and `<dest>` does not end with a trailing slash, then a file is downloaded from the URL and copied to `<dest>`.

- If `<src>` is a URL and `<dest>` does end with a trailing slash, then the filename is inferred from the URL and the file is downloaded to `<dest>/<filename>`. For instance, `ADD http://example.com/foobar /` would create the file `/foobar`. The URL must have a nontrivial path so that an appropriate filename can be discovered in this case (`http://example.com` will not work).

- If `<src>` is a directory, the entire contents of the directory are copied, including filesystem metadata.

Note: Whether a file is identified as a recognized compression format or not is done solely based on the contents of the file, not the name of the file. For example, if an empty file happens to end with `.tar.gz` this will not be recognized as a compressed file and **will not** generate any kind of decompression error message, rather the file will simply be copied to the destination.