Docker is a platform for developers and sysadmins to **develop, deploy, and run** applications with containers. The use of Linux containers to deploy applications is called *containerization*. Containers are not new, but their use for easily deploying applications is.

Containerization is increasingly popular because containers are:

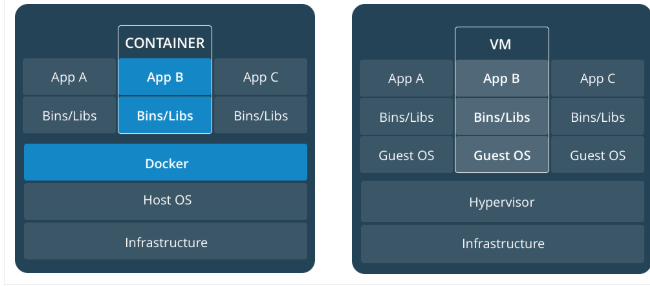
* Flexible: Even the most complex applications can be containerized.
* Lightweight: Containers leverage and share the host kernel.
* Interchangeable: You can deploy updates and upgrades on-the-fly.
* Portable: You can build locally, deploy to the cloud, and run anywhere.
* Scalable: You can increase and automatically distribute container replicas.
* Stackable: You can stack services vertically and on-the-fly.
* Images and containers
* A container is launched by running an image. An **image** is an executable package that includes everything needed to run an application--the code, a runtime, libraries, environment variables, and configuration files.
* A **container** is a runtime instance of an image--what the image becomes in memory when executed (that is, an image with state, or a user process). You can see a list of your running containers with the command, docker ps, just as you would in Linux

A **container** runs natively on Linux and shares the kernel of the host machine with other containers. It runs a discrete process, taking no more memory than any other executable, making it lightweight.

Containers and virtual machines

A **container** runs *natively* on Linux and shares the kernel of the host machine with other containers. It runs a discrete process, taking no more memory than any other executable, making it lightweight.

By contrast, a **virtual machine** (VM) runs a full-blown “guest” operating system with *virtual* access to host resources through a hypervisor. In general, VMs provide an environment with more resources than most applications need.



|  |  |
| --- | --- |
|  |  |

Docker Engine - Community is ideal for developers and small teams looking to get started with Docker and experimenting with container-based apps. Docker Engine - Community has three types of update channels, **stable**, **test**, and **nightly**:

* **Stable** gives you latest releases for general availability.
* **Test** gives pre-releases that are ready for testing before general availability.
* **Nightly** gives you latest builds of work in progress for the next major release.

# **Get Started, Part 2: Containers**

*Estimated reading time: 15 minutes*

* [1: Orientation](https://docs.docker.com/get-started/part1)
* [2: Containers](https://docs.docker.com/get-started/part2)
* [3: Services](https://docs.docker.com/get-started/part3)
* [4: Swarms](https://docs.docker.com/get-started/part4)
* [5: Stacks](https://docs.docker.com/get-started/part5)
* [6: Deploy your app](https://docs.docker.com/get-started/part6)

## **Prerequisites**

* [Install Docker version 1.13 or higher](https://docs.docker.com/engine/installation/).
* Read the orientation in [Part 1](https://docs.docker.com/get-started/).
* Give your environment a quick test run to make sure you’re all set up:
* docker run hello-world

## **Introduction**

It’s time to begin building an app the Docker way. We start at the bottom of the hierarchy of such app, a container, which this page covers. Above this level is a service, which defines how containers behave in production, covered in [Part 3](https://docs.docker.com/get-started/part3/). Finally, at the top level is the stack, defining the interactions of all the services, covered in [Part 5](https://docs.docker.com/get-started/part5/).

* Stack
* Services
* **Container** (you are here)

## **Your new development environment**

In the past, if you were to start writing a Python app, your first order of business was to install a Python runtime onto your machine. But, that creates a situation where the environment on your machine needs to be perfect for your app to run as expected, and also needs to match your production environment.

With Docker, you can just grab a portable Python runtime as an image, no installation necessary. Then, your build can include the base Python image right alongside your app code, ensuring that your app, its dependencies, and the runtime, all travel together.

These portable images are defined by something called a Dockerfile.

## **Define a container with Dockerfile**

Dockerfile defines what goes on in the environment inside your container. Access to resources like networking interfaces and disk drives is virtualized inside this environment, which is isolated from the rest of your system, so you need to map ports to the outside world, and be specific about what files you want to “copy in” to that environment. However, after doing that, you can expect that the build of your app defined in this Dockerfile behaves exactly the same wherever it runs.

### Dockerfile

Create an empty directory on your local machine. Change directories (cd) into the new directory, create a file called Dockerfile, copy-and-paste the following content into that file, and save it. Take note of the comments that explain each statement in your new Dockerfile.

# Use an official Python runtime as a parent image

FROM python:2.7-slim

# Set the working directory to /app

WORKDIR /app

# Copy the current directory contents into the container at /app

COPY . /app

# Install any needed packages specified in requirements.txt

RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container

EXPOSE 80

# Define environment variable

ENV NAME World

# Run app.py when the container launches

CMD ["python", "app.py"]

This Dockerfile refers to a couple of files we haven’t created yet, namely app.py and requirements.txt. Let’s create those next.

## **The app itself**

Create two more files, requirements.txt and app.py, and put them in the same folder with the Dockerfile. This completes our app, which as you can see is quite simple. When the above Dockerfile is built into an image, app.py and requirements.txt is present because of that Dockerfile’s COPY command, and the output from app.py is accessible over HTTP thanks to the EXPOSE command.

### requirements.txt

Flask

Redis

### app.py

from flask import Flask

from redis import Redis, RedisError

import os

import socket

# Connect to Redis

redis = Redis(host="redis", db=0, socket\_connect\_timeout=2, socket\_timeout=2)

app = Flask(\_\_name\_\_)

@app.route("/")

def hello():

try:

visits = redis.incr("counter")

except RedisError:

visits = "<i>cannot connect to Redis, counter disabled</i>"

html = "<h3>Hello {name}!</h3>" \

"<b>Hostname:</b> {hostname}<br/>" \

"<b>Visits:</b> {visits}"

return html.format(name=os.getenv("NAME", "world"), hostname=socket.gethostname(), visits=visits)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host='0.0.0.0', port=80)

Now we see that pip install -r requirements.txt installs the Flask and Redis libraries for Python, and the app prints the environment variable NAME, as well as the output of a call to socket.gethostname(). Finally, because Redis isn’t running (as we’ve only installed the Python library, and not Redis itself), we should expect that the attempt to use it here fails and produces the error message.

**Note**: Accessing the name of the host when inside a container retrieves the container ID, which is like the process ID for a running executable.

That’s it! You don’t need Python or anything in requirements.txt on your system, nor does building or running this image install them on your system. It doesn’t seem like you’ve really set up an environment with Python and Flask, but you have.

## **Build the app**

We are ready to build the app. Make sure you are still at the top level of your new directory. Here’s what ls should show:

$ ls

Dockerfile app.py requirements.txt

Now run the build command. This creates a Docker image, which we’re going to name using the --tag option. Use -t if you want to use the shorter option.

docker build --tag=friendlyhello .

Where is your built image? It’s in your machine’s local Docker image registry:

$ docker image ls

REPOSITORY TAG IMAGE ID

friendlyhello latest 326387cea398

Note how the tag defaulted to latest. The full syntax for the tag option would be something like --tag=friendlyhello:v0.0.1.

**Troubleshooting for Linux users**

*Proxy server settings*

Proxy servers can block connections to your web app once it’s up and running. If you are behind a proxy server, add the following lines before RUN pip in your Dockerfile, using the ENV command to specify the host and port for your proxy servers:

# Set proxy server, replace host:port with values for your servers

ENV http\_proxy host:port

ENV https\_proxy host:port

DNS settings

DNS misconfigurations can generate problems with pip. You need to set your own DNS server address to make pip work properly. You might want to change the DNS settings of the Docker daemon. You can edit (or create) the configuration file at /etc/docker/daemon.json with the dns key, as following:

{

"dns": ["your\_dns\_address", "8.8.8.8"]

}

In the example above, the first element of the list is the address of your DNS server. The second item is Google’s DNS which can be used when the first one is not available.

Before proceeding, save daemon.json and restart the docker service.

sudo service docker restart

Once fixed, retry to run the build command.

MTU settings

If the MTU (default is 1500) on the default bridge network is greater than the MTU of the host external network, then pip fails. Set the MTU of the docker bridge network to match that of the host by editing (or creating) the configuration file at /etc/docker/daemon.json with the mtu key, as follows:

{

"mtu": 1450

}

Before proceeding, save daemon.json and restart the docker service.

sudo systemctl restart docker

Re-run the build command.

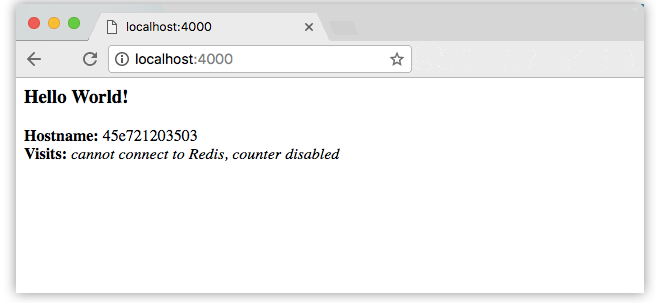
## **Run the app**

Run the app, mapping your machine’s port 4000 to the container’s published port 80 using -p:

docker run -p 4000:80 friendlyhello

You should see a message that Python is serving your app at http://0.0.0.0:80. But that message is coming from inside the container, which doesn’t know you mapped port 80 of that container to 4000, making the correct URL http://localhost:4000.

Go to that URL in a web browser to see the display content served up on a web page.



**Note**: If you are using Docker Toolbox on Windows 7, use the Docker Machine IP instead of localhost. For example, http://192.168.99.100:4000/. To find the IP address, use the command docker-machine ip.

You can also use the curl command in a shell to view the same content.

$ curl http://localhost:4000

<h3>Hello World!</h3><b>Hostname:</b> 8fc990912a14<br/><b>Visits:</b> <i>cannot connect to Redis, counter disabled</i>

This port remapping of 4000:80 demonstrates the difference between EXPOSE within the Dockerfile and what the publish value is set to when running docker run -p. In later steps, map port 4000 on the host to port 80 in the container and use http://localhost.

Hit CTRL+C in your terminal to quit.

**On Windows, explicitly stop the container**

On Windows systems, CTRL+C does not stop the container. So, first type CTRL+C to get the prompt back (or open another shell), then type docker container ls to list the running containers, followed by docker container stop <Container NAME or ID> to stop the container. Otherwise, you get an error response from the daemon when you try to re-run the container in the next step.

Now let’s run the app in the background, in detached mode:

docker run -d -p 4000:80 friendlyhello

Now let’s run the app in the background, in detached mode:

docker run -d -p 4000:80 friendlyhello

You get the long container ID for your app and then are kicked back to your terminal. Your container is running in the background. You can also see the abbreviated container ID with docker container ls (and both work interchangeably when running commands):

$ docker container ls

CONTAINER ID IMAGE COMMAND CREATED

1fa4ab2cf395 friendlyhello "python app.py" 28 seconds ago

Notice that CONTAINER ID matches what’s on http://localhost:4000.

Now use docker container stop to end the process, using the CONTAINER ID, like so:

docker container stop 1fa4ab2cf395

Now use docker container stop to end the process, using the CONTAINER ID, like so:

docker container stop 1fa4ab2cf395

## **Share your image**

To demonstrate the portability of what we just created, let’s upload our built image and run it somewhere else. After all, you need to know how to push to registries when you want to deploy containers to production.

A registry is a collection of repositories, and a repository is a collection of images—sort of like a GitHub repository, except the code is already built. An account on a registry can create many repositories. The docker CLI uses Docker’s public registry by default.

**Note**: We use Docker’s public registry here just because it’s free and pre-configured, but there are many public ones to choose from, and you can even set up your own private registry using [Docker Trusted Registry](https://docs.docker.com/datacenter/dtr/2.2/guides/).

### Log in with your Docker ID

If you don’t have a Docker account, sign up for one at [hub.docker.com](https://hub.docker.com/). Make note of your username.

Log in to the Docker public registry on your local machine.

$ docker login

### Tag the image

The notation for associating a local image with a repository on a registry is username/repository:tag. The tag is optional, but recommended, since it is the mechanism that registries use to give Docker images a version. Give the repository and tag meaningful names for the context, such as get-started:part2. This puts the image in the get-started repository and tags it as part2.

Now, put it all together to tag the image. Run docker tag image with your username, repository, and tag names so that the image uploads to your desired destination. The syntax of the command is:

docker tag image username/repository:tag

For example:

docker tag friendlyhello gordon/get-started:part2

Run [docker image ls](https://docs.docker.com/engine/reference/commandline/image_ls/) to see your newly tagged image.

$ docker image ls

REPOSITORY TAG IMAGE ID CREATED SIZE

friendlyhello latest d9e555c53008 3 minutes ago 195MB

gordon/get-started part2 d9e555c53008 3 minutes ago 195MB

python 2.7-slim 1c7128a655f6 5 days ago 183MB

...

### Publish the image

Upload your tagged image to the repository:

docker push username/repository:tag

Once complete, the results of this upload are publicly available. If you log in to [Docker Hub](https://hub.docker.com/), you see the new image there, with its pull command.

### Pull and run the image from the remote repository

From now on, you can use docker run and run your app on any machine with this command:

docker run -p 4000:80 username/repository:tag

If the image isn’t available locally on the machine, Docker pulls it from the repository.

$ docker run -p 4000:80 gordon/get-started:part2

Unable to find image 'gordon/get-started:part2' locally

part2: Pulling from gordon/get-started

10a267c67f42: Already exists

f68a39a6a5e4: Already exists

9beaffc0cf19: Already exists

3c1fe835fb6b: Already exists

4c9f1fa8fcb8: Already exists

ee7d8f576a14: Already exists

fbccdcced46e: Already exists

Digest: sha256:0601c866aab2adcc6498200efd0f754037e909e5fd42069adeff72d1e2439068

Status: Downloaded newer image for gordon/get-started:part2

\* Running on http://0.0.0.0:80/ (Press CTRL+C to quit)

No matter where docker run executes, it pulls your image, along with Python and all the dependencies from requirements.txt, and runs your code. It all travels together in a neat little package, and you don’t need to install anything on the host machine for Docker to run it.