How to install Docker in Ubuntu:

<https://docs.docker.com/engine/installation/linux/docker-ce/ubuntu/#install-using-the-repository>

Docker has two types of edition includes **Community Edition(CE)** and **Enterprise Edition(EE).**

Community is suitable for POC, whereas Enterprise Edition is suitable for companies setting docker with all enterprise licenses.

**To start with CE:**

# Get Docker CE for Ubuntu

*Estimated reading time: 10 minutes*

To get started with Docker CE on Ubuntu, make sure you [meet the prerequisites](https://docs.docker.com/engine/installation/linux/docker-ce/#prerequisites), then [install Docker](https://docs.docker.com/engine/installation/linux/docker-ce/#install-docker).

## Prerequisites

### Docker EE customers

To install Docker Enterprise Edition (Docker EE), go to [Get Docker EE for Ubuntu](https://docs.docker.com/engine/installation/linux/docker-ee/ubuntu/) **instead of this topic**.

To learn more about Docker EE, see [Docker Enterprise Edition](https://www.docker.com/enterprise-edition/).

### OS requirements

To install Docker CE, you need the 64-bit version of one of these Ubuntu versions:

$ lsb\_release -a

* Zesty 17.04
* Xenial 16.04 (LTS)
* Trusty 14.04 (LTS)

Docker CE is supported on Ubuntu on x86\_64, armhf, and s390x (IBM z Systems) architectures.

**s390x limitations**: System Z is only supported on Ubuntu Xenial and Zesty.

### Uninstall old versions

Older versions of Docker were called docker or docker-engine. If these are installed, uninstall them:

$ sudo apt-get remove docker docker-engine docker.io

It’s OK if apt-get reports that none of these packages are installed.

The contents of /var/lib/docker/, including images, containers, volumes, and networks, are preserved. The Docker CE package is now called docker-ce.

### Recommended extra packages for Trusty 14.04

Unless you have a strong reason not to, install the linux-image-extra-\* packages, which allow Docker to use the aufs storage drivers.

$ sudo apt-get update

$ sudo apt-get install \

linux-image-extra-$(uname -r) \

linux-image-extra-virtual

For Ubuntu 16.04 and higher, the Linux kernel includes support for OverlayFS, and Docker CE will use the overlay2 storage driver by default.

## Install Docker CE

You can install Docker CE in different ways, depending on your needs:

* Most users [set up Docker’s repositories](https://docs.docker.com/engine/installation/linux/docker-ce/#install-using-the-repository) and install from them, for ease of installation and upgrade tasks. This is the recommended approach.
* Some users download the DEB package and [install it manually](https://docs.docker.com/engine/installation/linux/docker-ce/#install-from-a-package) and manage upgrades completely manually. This is useful in situations such as installing Docker on air-gapped systems with no access to the internet.
* In testing and development environments, some users choose to use automated [convenience scripts](https://docs.docker.com/engine/installation/linux/docker-ce/#install-using-the-convenience-script) to install Docker.

### Install using the repository

Before you install Docker CE for the first time on a new host machine, you need to set up the Docker repository. Afterward, you can install and update Docker from the repository.

#### SET UP THE REPOSITORY

1. Update the apt package index:
2. $ sudo apt-get update
3. Install packages to allow apt to use a repository over HTTPS:

sudo apt-get install \

apt-transport-https \

ca-certificates \

curl \

software-properties-common

1. Add Docker’s official GPG key:
2. $ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

Verify that the key fingerprint is 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88.

$ sudo apt-key fingerprint 0EBFCD88

pub 4096R/0EBFCD88 2017-02-22

Key fingerprint = 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88

uid Docker Release (CE deb) <docker@docker.com>

sub 4096R/F273FCD8 2017-02-22

1. Use the following command to set up the **stable** repository. You always need the **stable** repository, even if you want to install builds from the **edge** or **testing** repositories as well. To add the **edge** or **testing** repository, add the word edge or testing (or both) after the word stable in the commands below.

**Note**: The lsb\_release -cs sub-command below returns the name of your Ubuntu distribution, such as xenial. Sometimes, in a distribution like Linux Mint, you might have to change $(lsb\_release -cs) to your parent Ubuntu distribution. For example, if you are using Linux Mint Rafaela, you could use trusty.

**amd64**:

$ sudo add-apt-repository \

"deb [arch=amd64] https://download.docker.com/linux/ubuntu \

$(lsb\_release -cs) \

stable"

**armhf**:

$ sudo add-apt-repository \

"deb [arch=armhf] https://download.docker.com/linux/ubuntu \

$(lsb\_release -cs) \

stable"

**s390x**:

$ sudo add-apt-repository \

"deb [arch=s390x] https://download.docker.com/linux/ubuntu \

$(lsb\_release -cs) \

stable"

**Note**: Starting with Docker 17.06, stable releases are also pushed to the **edge** and **testing** repositories.

[Learn about **stable** and **edge** channels](https://docs.docker.com/engine/installation/).

#### INSTALL DOCKER CE

1. Update the apt package index.
2. $ sudo apt-get update
3. Install the latest version of Docker CE, or go to the next step to install a specific version. Any existing installation of Docker is replaced.
4. $ sudo apt-get install docker-ce

**Got multiple Docker repositories?**

If you have multiple Docker repositories enabled, installing or updating without specifying a version in the apt-get install or apt-get update command will always install the highest possible version, which may not be appropriate for your stability needs.

1. On production systems, you should install a specific version of Docker CE instead of always using the latest. This output is truncated. List the available versions.
2. $ apt-cache madison docker-ce
3. docker-ce | 17.06.0~ce-0~ubuntu | https://download.docker.com/linux/ubuntu xenial/stable amd64 Packages

The contents of the list depend upon which repositories are enabled. Choose a specific version to install. The second column is the version string. The third column is the repository name, which indicates which repository the package is from and by extension its stability level. To install a specific version, append the version string to the package name and separate them by an equals sign (=):

$ sudo apt-get install docker-ce=<VERSION>

The Docker daemon starts automatically.

1. Verify that Docker CE is installed correctly by running the hello-world image.
2. $ sudo docker run hello-world

This command downloads a test image and runs it in a container. When the container runs, it prints an informational message and exits.

Docker CE is installed and running. You need to use sudo to run Docker commands. Continue to [Linux postinstall](https://docs.docker.com/engine/installation/linux/linux-postinstall/) to allow non-privileged users to run Docker commands and for other optional configuration steps.

<https://docs.docker.com/engine/reference/commandline/create/>

## Uninstall Docker CE

1. Uninstall the Docker CE package:
2. $ sudo apt-get purge docker-ce
3. Images, containers, volumes, or customized configuration files on your host are not automatically removed. To delete all images, containers, and volumes:
4. $ sudo rm -rf /var/lib/docker

You must delete any edited configuration files manually.

## Setup

Before we get started, make sure your system has the latest version of Docker installed using above steps.

You should be able to run docker run hello-world and see a response like this:

$ docker run hello-world

Hello from Docker!

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

To generate this message, Docker took the following steps:

...(snipped)...

Now would also be a good time to make sure you are using version 1.13 or higher. Run docker --version to check it out.

$ docker --version

Docker version 17.05.0-ce-rc1, build 2878a85

If you see messages like the ones above, you are ready to begin your journey.

**Alright we are ready to setup a app in the development environment(local box).**

## 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 has to be just so in order for your app to run as expected; ditto for the server that runs your app.

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 a Dockerfile

Dockerfile will define 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 have 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 will behave exactly the same wherever it runs.

### Dockerfile

Create an empty directory. 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:latest

# Set the working directory to /app

WORKDIR /app

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

ADD . /app

# Install any needed packages specified in requirements.txt

RUN pip install -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 will be present because of that Dockerfile’s ADD command, and the output from app.py will be 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} Chario team welcome to Docker Container..!</h3>" \

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

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

return html.format(name=os.getenv("NAME", "Itrain"), 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 will fail and produce 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 will 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 tag using -t so it has a friendly name.

docker build --tag evolvePro .

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

$ docker images

REPOSITORY TAG IMAGE ID

friendlyhello latest 326387cea398

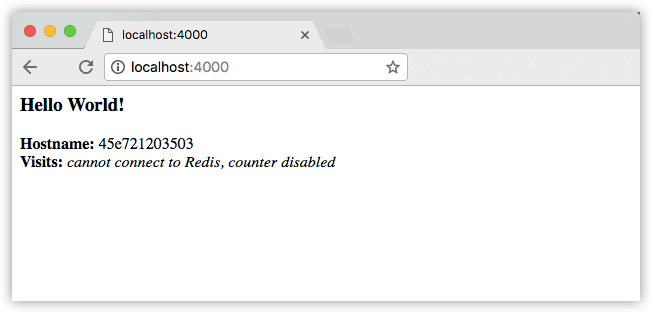
## Run the app

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

docker run -p 4000:80 friendlyhello

You should see a notice 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, including “Hello World” text, the container ID, and the Redis error message.



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>

**Note**: This port remapping of 4000:80 is to demonstrate the difference between what you EXPOSE within the Dockerfile, and what you publish using docker run -p. In later steps, we’ll just map port 80 on the host to port 80 in the container and use http://localhost.

Hit CTRL+C in your terminal to quit.

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 ps (and both work interchangeably when running commands):

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED

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

You’ll see that CONTAINER ID matches what’s on http://localhost:4000.

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

docker stop 1fa4ab2cf395

**Push your image built above to docker registry(otherwise respository in docker hub):**

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’ll be using 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 [cloud.docker.com](https://cloud.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 isusername/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:part1. This will put the image in the get-started repository and tag it as part1.

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

docker tag image username/repository:tag

For example:

docker tag friendlyhello john/get-started:part1

Run [docker images](https://docs.docker.com/engine/reference/commandline/images/) to see your newly tagged image. (You can also use docker image ls.)

$ docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

friendlyhello latest d9e555c53008 3 minutes ago 195MB

john/get-started part1 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 will see the new image there, with its pull command.

**How to configure docker container as build slaves on Jenkins:**

<https://www.tecmint.com/install-run-and-delete-applications-inside-docker-containers/>

## Docker Containers As Build Slaves

<https://blog.hypriot.com/post/setting-up-100-nodes-jenkins-cluster-with-docker-swarm-in-less-than-10-minutes/>

In this guide, I will walk you through the steps for configuring docker container as build slaves.

I assume that you have a Jenkins server up and running. If you do not have one, follow this tutorial. [How to setup jenkins 2](http://devopscube.com/install-configure-jenkins-2-0/)

### Create A Jenkins Slave Docker Image

Next step is to create a slave image. The image should contain the following minimum configurations to act as a slave.

1. sshd service running on port 22.

2. Jenkins user with password.

3. All the required application dependencies for the build. For example, for a java maven project, you need to have git, java, and maven installed on the image.

I have created a [Jenkins image for maven](https://hub.docker.com/r/bibinwilson/jenkins-slave/). You can use this image or use its [Dockerfile](https://github.com/bibinwilson/jenkins-docker-slave" \t "_blank) a reference for creating your own.

Make sure sshd service is running and can be logged into the containers using a username and password. Otherwise, Jenkins will not be able to start the build process.

### Configure Jenkins Server

1. Head over to Jenkins Dashboard –> Manage jenkins –> Manage Plugins.

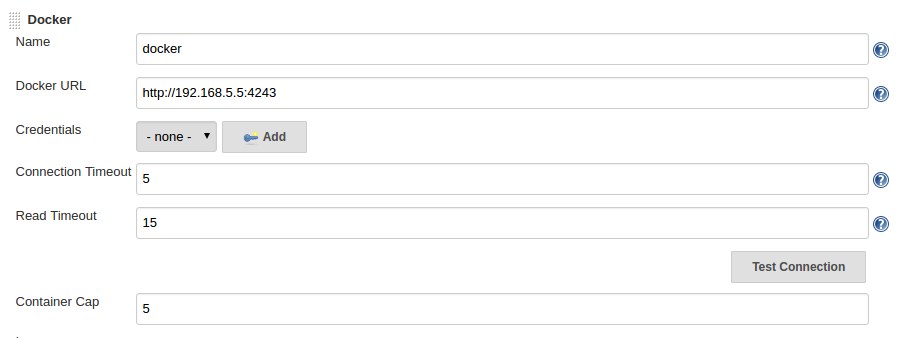
2. Under available tab, search for “Docker Plugin” and install it.

3. Once installed, head over to jenkins Dashboard –> Manage jenkins –>Configure system.

4. Under “Configure System”, if you scroll down, there will be a section named “cloud” at the last. There you can fill out the docker host parameters for spinning up the slaves.

5. Under docker, you need to fill out the details as shown in the image below.

Note: Replace “Docker URL” with your docker host IP. You can use the “Test connection” to test if jenkins is able to connect to the docker host.

[](https://devopscube.com/wp-content/uploads/2016/04/2-2.png)

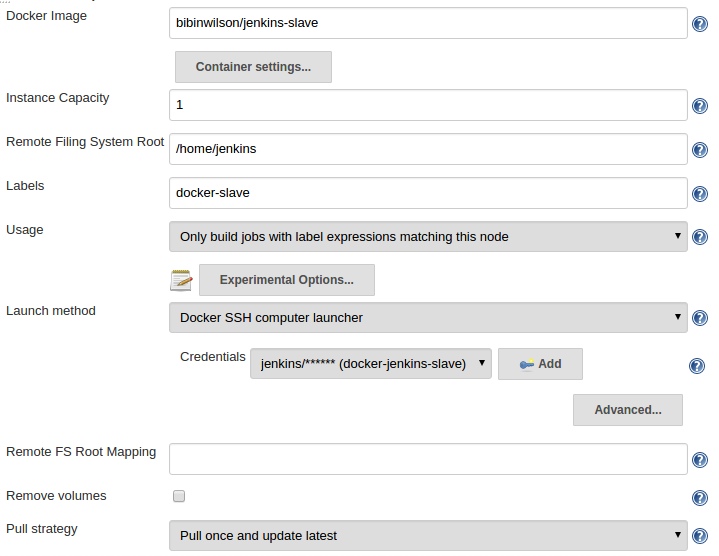
5. Now, from “**Add Docker Template**” dropdown, click “**docker template**” and fill in the details based on the explanation and the image given below.

**Docker Image** – Image that you created for the slave.

**Remote Filing System Root** – Home folder for the user you have created. In our case it’s jenkins.

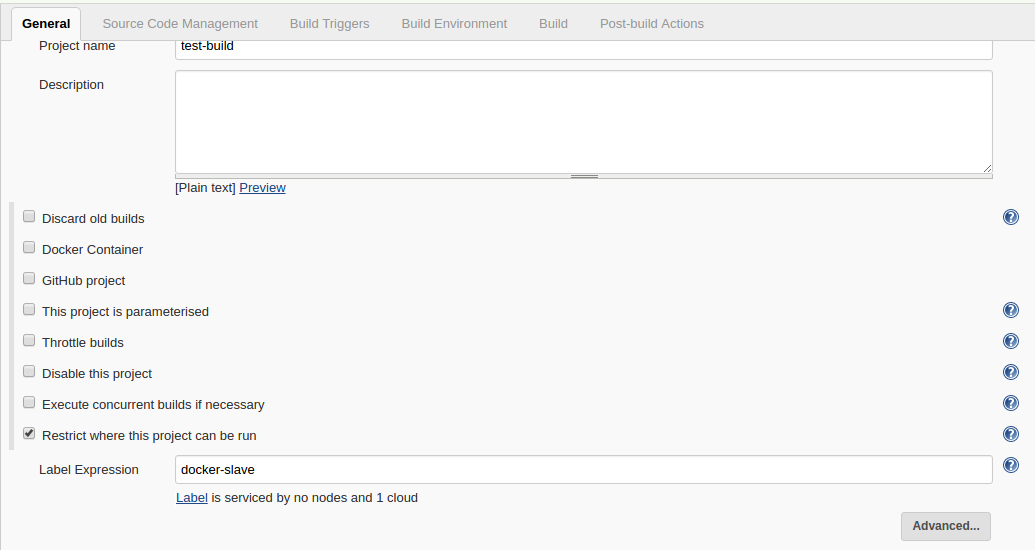
**Labels** – Identification for the docker host. It will be used in the Job configuration.

**Credentials**– click add and enter the username and password that you have created for the docker image. Leave the rest of the configuration as shown in the image below and click save.

[](https://devopscube.com/wp-content/uploads/2016/04/3-2.png)

### Building Jobs On Docker Slaves

Now that you have the slave configurations ready, you can create a job, select “Restrict where this project can be run” option and select the docker host as slave using the label as shown below.

[](https://devopscube.com/wp-content/uploads/2016/04/4-1.png)

If you have done all the configurations right, Jenkins will spin up a container, builds the project and destroys the container once the build is done. You can check the build logs in your jobs console output.

<https://blog.nimbleci.com/2016/08/31/how-to-build-docker-images-automatically-with-jenkins-pipeline/>

<http://fishi.devtail.io/weblog/2016/11/20/docker-build-pipeline-as-code-jenkins/>

<http://www.stuartellis.name/articles/jenkins-pipeline/>

<https://blog.couchbase.com/deployment-pipeline-docker-jenkins-java-couchbase/>

<https://github.com/arun-gupta/docker-jenkins-pipeline>

<https://devopscube.com/docker-containers-as-build-slaves-jenkins/>

<https://devopscube.com/setup-and-configure-sonarqube-on-linux/>

<https://www.digitalocean.com/community/tutorials/how-to-remove-docker-images-containers-and-volumes>

<https://getintodevops.com/blog/building-your-first-docker-image-with-jenkins-2-guide-for-developers>

**Github Webhook configuration:**

<https://support.cloudbees.com/hc/en-us/articles/224543927-GitHub-webhook-configuration>