**[Docker Hello World Application](http://www.bogotobogo.com/DevOps/Docker/Docker_Hello_World_Application.php)**

**Note**

Docker allows us to run applications inside containers. Running an application inside a container takes a single command: **docker run**.

Ref : [Dockerizing applications: A "Hello world"](https://docs.docker.com/userguide/dockerizing/" \t "_blnak)

$ docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]

The **docker run** command runs a process in a new container. It starts a process with its own file system, its own networking, and its own isolated process tree.

The **IMAGE** which starts the process may define defaults related to the process that will be run in the container, the networking to expose, and more, but **docker run** gives final control to the operator or administrator who starts the container from the image. For that reason, **docker run** has more options than any other Docker command.

If the **IMAGE** is not already loaded, then **docker run** will pull the **IMAGE**, and all image dependencies, from the repository in the same way running docker pull **IMAGE**, before it starts the container from that image.

**Hello world**

We need to have a disk image to make the virtualization work. The disk image represents the system we're running on and they are the basis of containers. Docker registry is a registry of already existing images that we can use to run and create containerized applications.

$ docker run ubuntu:14.04 /bin/echo 'Hello world'

Hello world

Note that we just launched our first container!

**What happened?**

Let's check what the docker run command did.

1. First, we specified the docker binary and the command we wanted to execute, run. The docker run combination runs containers.
2. Next we specified an image: ubuntu:14.04. This is the source of the container we ran. Docker calls this an image. In this case, we used an Ubuntu 14.04 operating system image. When we specify an image, Docker looks first for the image on our Docker host. If it can't find it, then it downloads the image from the public image registry: Docker Hub.
3. Next, we told Docker what command to run inside our new container:

/bin/echo 'Hello world'

When our container was launched, Docker created a new Ubuntu 14.04 environment, and then executed the /bin/echo command inside it. We saw the result on the command line:

Hello world

Docker containers only run as long as the command we specify is active. So, as soon as the "Hello world" was echoed, the container stopped. It's not running anymore.

**An interactive container**

Let's try the docker run command again. But this time, let's specify a new command to run in our container.

$ docker run -t -i ubuntu:14.04 /bin/bash

root@ec2a31ad35da:/#

Here, as we did before, we issued the docker run command, and launched an ubuntu:14.04 image.

We used two flags: **-t** and **-i**.

1. The **-t** flag assigns a pseudo-tty or terminal inside our new container
2. The **-i** flag allows us to make an interactive connection by grabbing the standard input (STDIN) of the container.

We have a new command for our container to run: **/bin/bash** which launches a Bash shell inside our container.

So, when our container is launched, we can see what we've got: a **command prompt** inside it!

root@ec2a31ad35da:/#

We can play around inside this container. When we're done, we can use the exit command or enter Ctrl-D to finish.

root@ec2a31ad35da:/# exit

exit

The same as happened with our previous container, once the Bash shell process has finished, the container is stopped.

**daemon - Hello world app**

Let's create a container that runs as a daemon, and we can do this with the docker run command:

$ docker run -d ubuntu:14.04 /bin/sh -c "while true; do echo hello world; sleep 1; done"

7a7fa9cc294d9ef7b32d5fe6c8a9da5b82d4500ab0710013d96bea26dbadbd6d

Where's our "Hello world" output? Let's look at what we've run here. We ran **docker run** but this time we used **-d**. The **-d** flag tells Docker to run the container and put it in the background, to **daemonize** it.

Note that we also used the same image: **ubuntu:14.04** as before.

Then, we run the following command:

/bin/sh -c "while true; do echo hello world; sleep 1; done"

This is our daemon: a shell script that echoes "hello world" forever.

However, we cannot see any "hello world", instead, Docker has returned a really long string:

7a7fa9cc294d9ef7b32d5fe6c8a9da5b82d4500ab0710013d96bea26dbadbd6d

This is a container ID, and it uniquely identifies a container. So, we can play with it. Actually, we can use this container ID to see what's happening with our "hello world" daemon.

First, let's make sure our container is running. We can do that with the docker ps command. The docker ps command queries the Docker daemon for information about all the containers it knows about.

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

7a7fa9cc294d ubuntu:14.04 "/bin/sh -c 'while t 16 minutes ago Up 16 minutes jovial\_franklin

Here we can see our daemonized container. The docker ps has returned some useful information about it, starting with a shorter variant of its **container ID: 7a7fa9cc294d**. We can also see the image we used to build it, **ubuntu:14.04**, the **command** it is running, its **status** and an automatically assigned **name, jovial\_franklin**.

$ docker logs jovial\_franklin

hello world

hello world

hello world

...

The docker logs command looks inside the container and returns its standard output: in this case, the output of our command is "hello world".

Now we can see our daemon is working, and we've just created our first Docker application!

How can we stop our daemonized container?

**docker stop**!

The docker stop command tells Docker to stop the running container. If it succeeds, it will return the name of the container it has just stopped:

$ docker stop jovial\_franklin

jovial\_franklin

Let's check if it really stopped. We can do it with the docker ps command:

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

No docker container is running now!