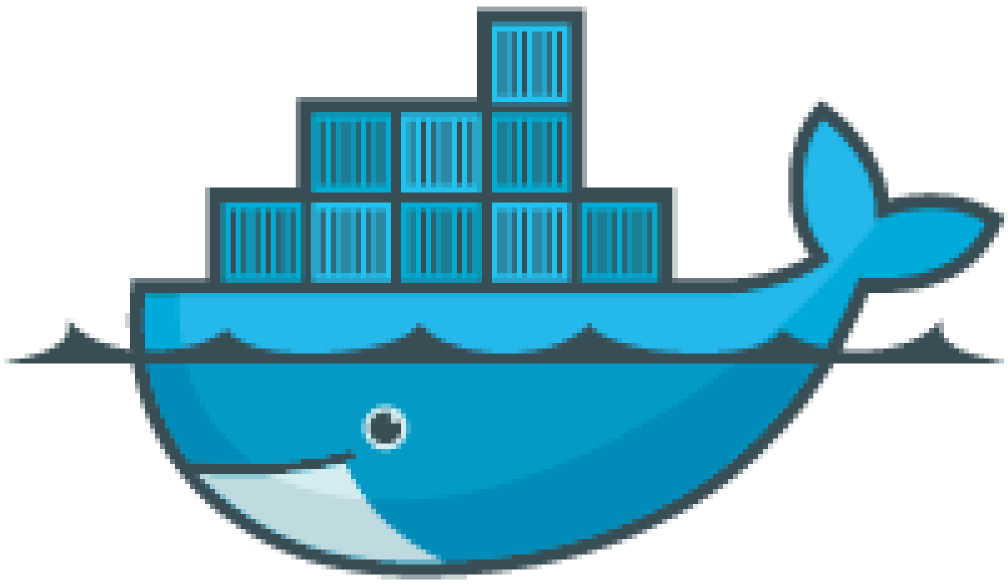
How-to build Docker from source



"Docker is an open-source project that automates the deployment of applications inside software containers".

Lately the Docker project has grown to include many different projects, but the core is still the docker server and client. The client and server is contained in the same binary and written in Go (golang). This is the binary we'll compile from source.

Docker is pretty easy to install from binaries. They release pre-compiled binaries, an install script and many linux distros have packages ready to install. So why would you want to build it from source? Well, I personally have two reasons: 1) I'm currently tinkering with Docker Plugins, a feature under development, and even the experimental build is lagging behind the latest developments. 2) Being able to modify the source and run the result can be very helpful when trying to understand how Docker works.

1 How

It turns out there are basically two ways of building docker from source: The hard way and the easy way. The hard way is to install all the dependencies and build everything from scratch. Although this is not that hard, docker has more dependencies than expected. The easy way is to use their build scripts to automate everything. This saves you all the trouble of finding and installing the right dependencies. The scripts builds and runs a docker image to compile the binary. So you need to have docker installed already to build docker in this way, which is the drawback.

Being lazy, I'm going to go with the easy way, but if you want to this the hard way (without using docker) this Dockerfile is a good starting point. From that you can extract all the dependencies and steps necessary.

The easy way is roughly:

- Install docker
- Get the docker source
- Run the included scripts to create the docker image and compile everything
- Test the binaries
- Copy the binaries to wherever you want to have them.

1.1 Installing Docker

If you don't have docker already, install it. You can find instructions on docs.docker.com

For ubuntu this is basically:

```
# as root (or prepend with sudo)
apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys
sh -c "echo deb https://get.docker.com/ubuntu docker main > /etc/apt/sources.list.d/docker. ↩
    list"
apt-get update
apt-get install lxc-docker

# Get some dependencies
apt-get install make git
```

1.2 Aside: On Digital Ocean? Create some swap space

I've had some issues with compiling docker on the smallest Digital Ocean plan. I think it's because it runs out of ram and does not have any swap space to fall back to. You can fix this, I think, by creating and enabling some swap space:

```
# as root (or prepend with sudo)
dd if=/dev/zero of=/swapfile bs=1M count=1000
chmod 600 /swapfile
mkswap /swapfile
swapon /swapfile
swapon -s

# Test that it worked:
free -h
```

Read more in this Digital Ocean tutorial and this AskUbuntu question. (If you're not using Digital Ocean but would like to be, you should use my affiliate link. Perks for you, perks for me :)

1.3 Get the docker source

This should be as easy as fetching the code from Github. Install git if you don't have it already and run:

```
git clone https://github.com/docker/docker.git
cd docker
```

1.4 Compile docker in a docker container

The Dockerfile and supporting scripts make it pretty easy to compile a new docker from source, they do all the heavy lifting for you. This works by first building a new docker image from the Dockerfile, running this image and compiling a new docker binary. Since it's building a new docker image, the docker server (docker -d) must be running. Hence the need for installing docker first. This docker image then has all the dependencies necessary to build docker from source. The docker binary is build inside this container and copied out afterwards. Ideally the included Makefile and scrips do all of this for us, and produce a binary ready to use:

```
# as root (or prepend with sudo)
make build
# this takes a while and prints the output from "Docker build ."
# It ends with: "Successfully built ..".

# as root (or prepend with sudo)
make binary
```

```
# This runs the image we created with "make build".
# Ends with "Created binary: bundles/1.8.0-dev/binary/docker-1.8.0-dev",
# where "1.8.0-dev" is the version I'm building,
# the next unreleased version at the time of writing.
```

1.5 Test the new binary

1.5.1 Stop the old docker daemon

Stop the docker daemon so w can run our new one. On Ubuntu / debian this is:

```
service docker stop
```

On other systems it might be.. something else. You can try *killall docker* or *pkill docker*

1.5.2 Start a daemon using our new binary

```
# in the docker/ directory where we ran make build and make binary
./bundles/1.8.0-dev/binary/docker -d  # the version, "1.8.0-dev", will vary according to ↩
    the version your building
```

1.5.3 Run the old binary as a client and connect to our new daemon

With the docker server (docker -d) still running, open a new console / terminal and run: *docker version*. This will use the original docker as a client and it should be able to contact our new docker binary. You should see the difference in server version and client version, like so:

```
docker version

Client version: 1.6.0
Client API version: 1.18
Go version (client): go1.4.2
Git commit (client): 4749651
OS/Arch (client): linux/amd64
Server version: 1.8.0-dev
Server API version: 1.20
Go version (server): go1.4.2
Git commit (server): c2346f6
OS/Arch (server): linux/amd64
```

As you can see "Client version" and "Server version" are different. This is expected since the server is the new binary we just started as a daemon (./bundles. . . docker -d). The client, however, is the docker installed on the system. Run *which docker* to see where this binary is located.

1.5.4 Run the new binary as a client and connect to our new daemon

You can test the new binary as a client as well:

```
# in the docker/ directory where we ran make build and make binary
./bundles/1.8.0-dev/binary/docker version

Client:
 Version:      1.8.0-dev
 API version:  1.20
 Go version:   go1.4.2
 Git commit:   c2346f6
 Built:        Thu Jul 23 15:03:21 UTC 2015
 OS/Arch:      linux/amd64

Server:
 Version:      1.8.0-dev
 API version:  1.20
 Go version:   go1.4.2
 Git commit:   c2346f6
 Built:        Thu Jul 23 15:03:21 UTC 2015
 OS/Arch:      linux/amd64
```

As you can see the Client and Server version is now the same. Both the server and client is running the same binary file; the one we just built. Also, it looks like the output of *docker version* changed between version 1.6 and 1.8.

1.5.5 Last test: Run a docker container

A basic test that everything works is to run the official hello-world image using our new binary. Make sure the server is still running and run:

```
./bundles/1.8.0-dev/binary/docker run -t -i --rm hello-world
```

1.6 Installing our new binary system-wide

As far as I know, installing the new binary is as simple as copying it into place, overwriting the old one. (I'm sure there's cases where this is not enough, or could break things, so, you know..)

```
# Where is our old binary?
which docker

# Mine is in /usr/bin/docker, so let's move it
# as root (or prepend with sudo)
mv /usr/bin/docker /usr/bin/docker-old

# and copy our new binary in
cp ./bundles/1.8.0-dev/binary/docker /usr/bin/

# run the server with "docker -d" or :
service docker start
```

Questions? Comments? Found something wrong or confusing? Hit me up on twitter or hello @ this domain