Article  Sources

https://medium.com/@andrenit/buildind-an-ethereum-playground-with-docker-part-1-introduction-80be173aaa7a

André Fernandes Follow
@vertigobr Founder & CTIO, disrupting things for fun.
Sep 11, 2016 · 4 min read

# Building an Ethereum Playground with Docker (part 1 — Introduction)

*I have updated this article on December/2017. I no longer create a custom Docker image, because Ethereum official image already brings an "all-tools" version that includes bootnode and other treats.*

This is the first article in an ongoing series "Building an Ethereum Playground with Docker". The articles already published are:

- Building an E. P. Part 1 — Introduction (this one)
- Building an E. P. Part 2 — Docker Image
- Building an E. P. Part 3 — Ethereum Wallet
- Building an E. P. Part 4 — Provisioning

We will cover using the official "ethereum/go-ethereum" Docker image, playing with Ethereum Wallet, provisioning the ethereum nodes on public clouds and deployment of a sample app.

## Introduction

Both blockchain and Docker are technologies that have been under a lot of hype for a few years now.

Docker itself is already past-hype and has become the *de facto* standard for containers — and containers are now the standard for pretty much everything on software development and delivery (unless, of course, you have been hiding in a cave).

Blockchain, on the other hand, is yet to deliver the wonders it is praised for. Beyond Bitcoin and its variants, blockchain technology can be used in several other scenarios (like digital property, voting, identity management, to name a few). In fact, from an entrepreneur point of view, a blockchain network can be seen as a platform for crafting a whole new set of products.

BTW, these articles assume some understanding about blockchain technology and will not try to explain it in any way.

## Enter Ethereum

The Ethereum network was built to deliver the platform concept, not focusing on the digital currency aspect like the Bitcoin network does. Quoting Ethereum documentation:

> *Ethereum is an open blockchain platform that lets anyone build and use decentralized applications that run on blockchain technology.*

So one can actually see an Ethereum network as a platform for running decentralized applications — and indeed it is such a thing. The blockchain technology allows this platform "…extreme levels of fault tolerance, ensures zero downtime, and makes data stored on the blockchain forever unchangeable and censorship-resistant".

Indeed a rich ecosystem of apps (or "dapps", that stands for "distributes applications") is flourishing on the Ethereum network.

## Wait: is it "the network" or "a network"?

This question can be rephrased: is there a single blockchain Ethereum network ("the" network) or a plethora of networks? Well, both.

There is The Ethereum blockchain network, public and controlled by no one in particular. Well, actually there is also The Ethereum "classic" network, but that is another story.

Unlike a regular cloud computing platform where you "own" (control and trust) the nodes that execute your applications, a public blockchain network is quite the opposite: it is entirely composed of *untrusted* participants that build *consensus* in a trusted way (thanks to the blockchain). So the meaning of *public* is that anyone can read it, transact with and participate in the consensus process (just install the Ethereum Wallet and *voilà*) — the decentralization actually adds to its value.

This is somewhat hard to understand: nobody "owns" all of the network's nodes, but since Ethereum developers need to improve their software and protocol everybody agrees on updating the software from time to time. So there are soft forks and hard forks, but the whole point is to preserve the network's value while evolving its software.

Since Ethereum is open-source software nothing stops anyone from starting a new public network — it just doesn't make sense to do so. From a technical point of view blockchain networks differ only on their genesis block, algorithm and protocol but a public network's value relies intrinsically on widespread use.

Private networks, on the other hand, can be spun at will for any purpose that comes to mind (there are actually many valid use cases). Once again, just create a new genesis block, start a set of nodes and put them to work.

It is very easy to stumble upon heated discussions on whether private blockchain networks actually make sense or not (why use a decentralized and inefficient solution if you actually trust the participants), but from a developer point of view it is very desirable to be able to create and blow away a platform — several times a day. *This is the point of this series of articles.*

## Why Docker?

Seriously? Call me a fanboy, but nowadays one would actually have a hard time to explain why not to use containers.

The next article in this series will discuss provisioning an entire private Ethereum network — locally or on a public cloud — relying on containers. Despite the fact that there are several offers around for ethereum-vms provisioning (like Azure's Blockchain as a Service offerings), going the container way is always a wiser choice. During the time it take to re-provision an Azure set of VMs (even with automated scripts) you can restart or recreate a Docker swarm

André Fernandes Follow
@vertigobr Founder & CTIO, disrupting things for fun.
Sep 12, 2016 · 7 min read

# Building an Ethereum playground with Docker (part 2 — Docker Image)

*I have updated this article on December/2017. I no longer create a custom Docker image, because Ethereum official image already brings an “all-tools” version that includes bootnode and other treats.*

This is the second article in an ongoing series “Building an Ethereum Playground with Docker”. The articles already published are:

- Building an E. P. Part 1 — Introduction
- Building an E. P. Part 2 — Docker Image (this one)
- Building an E. P. Part 3 — Ethereum Wallet
- Building an E. P. Part 4 — Provisioning

We will cover using the official “ethereum/go-ethereum” Docker image, playing with Ethereum Wallet, provisioning the ethereum nodes on public clouds and deployment of a sample app.

This article uses "ethereum/client-go" Docker image to run several Ethereum nodes locally (and safely). It also assumes you have a Docker engine available to you (and know a bit about it), probably after installing Docker for Mac or Docker for Windows on your notebook.

All sources are located in https://github.com/vertigobr/ethereum.git.

## The Docker Image

There is a public official image "docker pull ethereum/client-go" that will serve as the base of this work. We will create scripts with some functionality and configuration options to make it generally useful for our evil machinations.

This original public base image is a nice piece of work: you can use it to participate on the main public Ethereum network with a simple command:

```
docker run -d --name ethereum-node \
    -v $HOME/.ethereum:/root \
    -p 8545:8545 -p 30303:30303 \
    ethereum/client-go:v1.7.3 --fast --cache=512
```

Or, similarly, to participate on the public test network:

```
docker run -d --name ethereum-node \
    -v $HOME/.ethereum:/root \
    -p 8545:8545 -p 30303:30303 \
    ethereum/client-go:v1.7.3 --testnet --fast --cache=512
```

To check the node logs:

```
docker logs -f ethereum-node
```

And to attach the the running node (main net) console:

```
docker exec -ti ethereum-node geth attach
```

Or to the running node on testnet:

```
docker exec -ti ethereum-node \
    geth attach ipc:/root/.ethereum/testnet/geth.ipc
```

But we won't be doing any of this — we want a private network of our own.

## Getting Started

There is a Github project containing several helper scripts that will be used along this document:

```
git clone https://github.com/vertigobr/ethereum.git
```

Assuming you have Docker up and running:

- Run a bootnode
- Run a common node
- Run a miner node
- Check a node's peers

That would be running the commands:

```
./bootnode.sh
./runnode.sh
./runminer.sh
./showpeers.sh
```

The starting point is the creation of a "genesis.json" file that defines the *genesis block* of the blockchain. The "genesis.sh" script does that for you, and it can be edited to provide custom values for some variables

Some variables on the top of this script can be modified to define your very own genesis block, the main point being that all containers will mount the same genesis.json file when launched with the helper scripts.

Nodes that share the same genesis block and are capable of finding each other (in our case using a bootnode to locate peers) will compose your private Ethereum network. Also the helper scripts mount data folders for each container, so if you stop and destroy the containers and run them again with the same arguments they will resume work as if nothing had happened.

## The scripts

The project cloned from Github holds a few utility scripts we will use a lot to save time from typing long docker commands:

> **genesis.sh**: creates a new "genesis.json" file based on the template at "src/genesis.json.template" and on a few variables you can mess around with;
>
> **bootnode.sh**: runs an Ethereum bootnode in a container named "ethereum-bootnode";
>
> **runnode.sh**: runs an Ethereum non-mining node in a container named from your argument (ex: `runnode.sh node1` runs an `ethereum-node1` container);
>
> **runminer.sh**: identical to `runnnode.sh`, but starts a mining node;
>
> **showpeers.sh**: shows all peers that are connected to this node. Example: `showpeers.sh ethereum-node1`;
>
> **killall.sh**: "docker stop" and "docker rm" on all containers you ran with `runnode.sh` and `bootnode.sh`, but preserves the volume folders;
>
> **wipeall.sh**: “docker stop” and “docker rm” on all containers you ran with `runnode.sh` and `bootnode.sh` and also removes the volume folders.

All these scripts rely on the genesis.json file generated by "genesis.sh". *When running them locally there is zero chance your network will chat with another Ethereum network, because everyone is restricted to the peers that found your bootnode *and* because your nodes are the only ones that can use your blockchain*.

A few scripts deserve more explanation:

## bootnode.sh

This script does the job below (reduced for simplicity):

```
docker stop ethereum-bootnode
docker rm ethereum-bootnode
docker run -d --name ethereum-bootnode \
 -v $(pwd)/.bootnode:/opt/bootnode \
 ethereum/client-go:alltools-v1.7.3 bootnode --nodekey (...)
```

This script starts a dumb bootnode instead of a "full" geth node. Also notice the local volume mounted on ".bootnode".

### runnode.sh

This script does the job below (reduced for simplicity):

```
NODE_NAME=$1
CONTAINER_NAME="ethereum-$NODE_NAME"
docker stop $CONTAINER_NAME
docker rm $CONTAINER_NAME
BOOTNODE_URL=$(./getbootnodeurl.sh)
docker run -d --name $CONTAINER_NAME \
    -v $(pwd)/genesis.json:/opt/genesis.json \
    ethereum/client-go:v1.7.3 --cache=512 --bootnodes=(...)
```

Notice that the bootnode URL is detected with the utility `getbootnode.sh` and both container name and local volume are created based on the node name provided as an argument.

### showpeers.sh

This script does the job below (reduced for simplicity):

```
docker exec -ti "$1" geth --exec 'admin.peers' attach
```

Notice the use of "docker exec" to submit a command to a current running node with "geth attach".

## Let us Play

These are the steps we will follow:

Create the docker network

Run the bootnode;

Run a non-mining node;

Run a second non-mining node;

Check if both found their peer;

Run a mining node;

Check if everyone found their peers.

It is assumed, of course, that you have cloned the repository and have these scripts in your current folder. Once again, you don't need to build the image, Docker will pull it for you.

Here we go.

## Step 1: Create network and run bootnode

This is very simple:

```
./bootnode.sh
```

You can check the container log:

```
docker logs ethereum-bootnode
```

It will result on something like this:

```
INFO [12-06|17:31:44] UDP listener up
self=enode://d92e0ce77861919c516d8e6a65f58e441df0ec73b640551f
3dd5e83c2e6bf41aa189e7709f261e633db0f906fe9b1e37c92eeb9d80b42
918cb240726078439e3@[::]:30301
```

Notice that finding the bootnode from another container is possible in several ways, but I have opted to a low-level trick to detect the container internal IP address — this is an useless IP outside your engine (or swarm), but is good enough for us right now. Check, just to be sure, if the `getbootnode.sh` script returns a valid IP properly:

```
./getbootnode.sh
```

It will result on something like this (the actual IP may differ, obviously):

```
enode://xxxxx...yyyy@172.18.0.2:30301
```

## Step 2: Run a non-mining node

This is also very simple:

```
./runnode.sh node1
```

You can check the container log:

```
docker logs ethereum-node1
```

It will result on something like this:

```
INFO [12-06|17:38:34] Starting peer-to-peer node
instance=Geth/v1.7.3-stable/linux-amd64/go1.9.2
(...)
INFO [12-06|17:38:34] Starting P2P networking
INFO [12-06|17:38:36] UDP listener up
self=enode://454bc00e461761e8b3c7aa22f85a713c5f5a9b8032f253c5
eca96c24acd1cbcc7af3058047aee53128d673f060c0a236b77c5da194715
```

Congratulations, you already have a private Ethereum network in your hands!

## Step 5: Run a mining node

A non-mining blockchain kinda makes no sense, so we will spin a mining node to make it work (and just for the fun of it).

```
./runminer.sh miner1
```

It will take quite some time until the mining node begins mining. You will find this slow progression on the logs:

```
docker logs -f ethereum-miner1
...
INFO [12-06|18:08:53] Generating DAG in progress
epoch=0 percentage=0 elapsed=3.893s
INFO [12-06|18:08:57] Generating DAG in progress
epoch=0 percentage=1 elapsed=7.695s
...
```

After that you will begin to see successful mining logs:

```
INFO [12-06|18:16:06] 🔗 block reached canonical chain
number=10 hash=90370f…e58ba1
INFO [12-06|18:16:06] 🔨 mined potential block
number=15 hash=44b3de…25e7d9
INFO [12-06|18:16:06] Commit new mining work
number=16 txs=0 uncles=0 elapsed=230.484µs
DEBUG[12-06|18:16:06] Reinjecting stale transactions
count=0
DEBUG[12-06|18:16:06] Recalculated downloader QoS values
rtt=20s confidence=1.000 ttl=1m0s
INFO [12-06|18:16:10] Successfully sealed new block
number=16 hash=b171fe…003c8f
```

The non-mining nodes will get copies of mined blocks, as they are expected to. Just check back the first node log:

```
docker logs ethereum-node1
```

And you will see lines like these:

```
DEBUG[12-06|18:16:33] Queued propagated block
peer=06055abb9386a745 number=22 hash=f8e9b2…5cfb00 queued=1
DEBUG[12-06|18:16:33] Importing propagated block
peer=06055abb9386a745 number=22 hash=f8e9b2…5cfb00
DEBUG[12-06|18:16:33] Trie cache stats after commit
misses=7 unloads=0
DEBUG[12-06|18:16:33] Inserted new block
number=22 hash=f8e9b2…5cfb00 uncles=0 txs=0 gas=0
elapsed=13.009ms
```

## Step 6: Check if everyone found their peers

It is quite obvious that all nodes found the other peers (or the mining block would not propagate). You can easily check if it out anyway:

```
./showpeers.sh miner1
```

A response like the one below is expected:

```
[{
    caps: ["eth/63"],
    id:
"06055abb9386a74596cc0486430abfc3b967d9bba643bed799e8666e5caf
f3661c9eaa639a98f5c7c235ec003070dd0792d5838505252b4b5ce83d90a
451e77b",
    name: "Geth/v1.7.3-stable/linux-amd64/go1.9.2",
    network: {
      localAddress: "172.18.0.5:41094",
      remoteAddress: "172.18.0.4:30303"
    },
    protocols: {
      eth: {
        difficulty: 3820928,
        head:
"0x0ba0118d469b82875abfc6706de5d2463b6519fd8a71a63d939834cc14
fd4ebd",
        version: 63
      }
    }
}, {
    caps: ["eth/63"],
    id:
"204a74e1251671597e16207675a78853b24fddad96a795c1a2a14717a12d
eabcb440d2bcce9e5c6066af6f86a9c2e327ded161d9f9ebf01b0bc64fd2e
d3100b0",
    name: "Geth/v1.7.3-stable/linux-amd64/go1.9.2",
    network: {
      localAddress: "172.18.0.5:30303",
      remoteAddress: "172.18.0.3:56028"
    },
    protocols: {
      eth: {
        difficulty: 131072,
        head:
"0x2084e6ce93b07f22db9e3e5960b5911fd493214d886161c49e1a007530
52b987",
        version: 63
      }
    }
}]
```

## I Wanna Do It Again

Why not? After all these are containers, not VMs. You can stop an destroy them all:

```
./killall.sh
```

Running the scripts again with the same arguments will "resurrect" your network (the mounted volumes are still there):

```
./bootnode.sh
./runnode.sh node1
./runnode.sh node2
./runminer.sh miner1
```

The mined blocks are still there, of course — all pertinent state lives on the mounted volumes — so this second run is a lot faster.

Wanna really restart from scratch?

```
./wipeall.sh
```

This script also empties the mounted volumes.

# Building an Ethereum playground with Docker (part 3 — Ethereum Wallet)

This is the third article in an ongoing series “Building an Ethereum Playground with Docker”. The articles already published are:

We will cover using the official “ethereum/go-ethereum” Docker image, playing with Ethereum Wallet, provisioning the ethereum nodes on public clouds and deployment of a sample app.

All sources and scripts are located in https://github.com/vertigobr/ethereum.git.

In this article we will introduce and install the Ethereum Wallet, connect it to our private Ethereum network, create an account and mine into it.

## The Wallet

Not just an UI to manage your Ethereum accounts and ether, the Wallet is "…a gateway to decentralized applications on the Ethereum blockchain".

For now we will simply consider the Wallet as a ready-to-use front-end for our playground.

When ran with its default arguments the Wallet also becomes a full Ethereum node connected to the public network (main or testnet, you can choose) — something we don't need in this case. With some (documented) tinkering we will be able to use it to connect to one of our local mining container nodes.

### Installing the Wallet

Pick the proper version and download it from Github or from the main Ethereum site. There are installers for many platforms (OSX and Windows, for example) and also a ZIP alternative for manual installation.

Since *we will not* run the Wallet double-clicking an icon (we wil need to supply command line arguments) please take note of where the

Wallet executable file is. This is kinda catchy on OSX but straightforward on the other platforms. The reason is that we will have to run it from the command-line with specific arguments. Let us check this procedure by running the Wallet from a Terminal/Prompt with a "--help" option.

On Windows (use the folder you picked):

```
D:\Ethereum-Wallet\Ethereum-Wallet.exe --help
```

On Linux (assuming "/opt" folder):

```
/opt/Ethereum-Wallet/Ethereum-Wallet --help
```

On OSX (assuming you placed the app in "Applications"):

```
/Applications/Ethereum\ Wallet.app/Contents/MacOS/Ethereum\
Wallet --help
```

## The Wallet node

Now we will run a specific node configured for the wallet.

### Enabling the JSON-RPC Interface

The default endpoint the Wallet looks for is an local IPC socket with a known path (or a named pipe in Windows). This is hard coded on the Wallet code in a way that becomes somewhat inconvenient to use in our scenario — the "normal" use for the Wallet assumes that it is an ethereum node itself, so an embedded node runs connected to the main net or to the test net (you choose). We need to connect the Wallet to one of our containers instead.

*Remember, most of us are running the nodes with Docker for*

*Mac/Windows, so the nodes are inside an "invisible" VM (xhyve or Hyper-V at the time of this writing) and the Wallet app is running in your local OS. Technically your OS is not the Docker host (the "invisible" VM is), so the `geth.ipc` IPC socket file you find in the volume folders is useless for the Wallet.*

Instead we will enable another interface in one of our nodes: the JSON-RPC interface, in port 8545. That will work on any of the Wallet versions (OSX, Linux or Windows) — be warned that exposing the JSON-RPC port to the world isn't exactly a good idea, but for now we are working locally in a ephemeral private network that will be discarded anyway. Fear not, little Padawan.

Luckily our helper scripts already provide a way to enable (and expose) the JSON-RPC port easily, by providing an environment variable with the desired port for binding:

```
./bootnode.sh
# wait for DAG before running wallet
RPC_PORT=8545 ./runminer.sh wallet
```

> *Remember that the long "Generating DAG…" process must finish before mining. Just check with `docker logs -f ethereum-miner1`. If you run a second miner before it ends both will do the same work. Be patient. Life is beautiful.*

The lines above will start a bootnode (as from the last article), a non-exposed mining node and an exposed mining node. The magic behind the script is quite simple: it configures a port binding for the container and runs the node with additional arguments.

The generated port binding is like this, in case you are curious (this is a "docker" argument):

```
-p $RPC_PORT:8545
```

And the generated "geth" arguments are these:

```
--rpc --rpcaddr=0.0.0.0 --rpcapi="db,eth,net,web3,personal"
--rpccorsdomain "*"
```

The JSON-RPC port can be tested from your Terminal easily (you get an JSON response):

```
> curl -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"web3_clientVersion","params":
[],"id":67}' localhost:8545
{"jsonrpc":"2.0","id":67,"result":"Geth/v1.7.3-stable/linux-
amd64/go1.9.2"}
```

Oh, no curl in your Windows prompt? You can always use some tool like Postman to send POST requests directly from the browser.

> *Please note that Docker for Mac/Windows "magic" already takes care of NAT'ing the container exposed ports to "localhost" in your local OS. This is not the default behavior, for example, of "docker-machine" or custom Vagrant VMs, where you most likely work with a host-only IP (ex: "docker-machine ip default"). Just change the hostname for the RPC URL from “localhost” on the next topic to whatever works for you.*

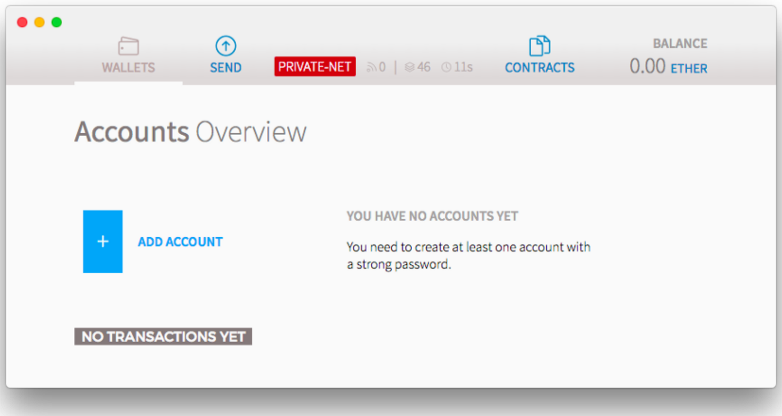
## Running the Wallet

Now that the JSON-RPC port is exposed in localhost:8545 you can run the Wallet with the proper arguments from the command line.

In OSX:

```
/Applications/Ethereum\ Wallet.app/Contents/MacOS/Ethereum\
Wallet --rpc http://localhost:8545
```

This will bring the Wallet UI (ignore the security alert) as seen below:



Accounts Overview

YOU HAVE NO ACCOUNTS YET

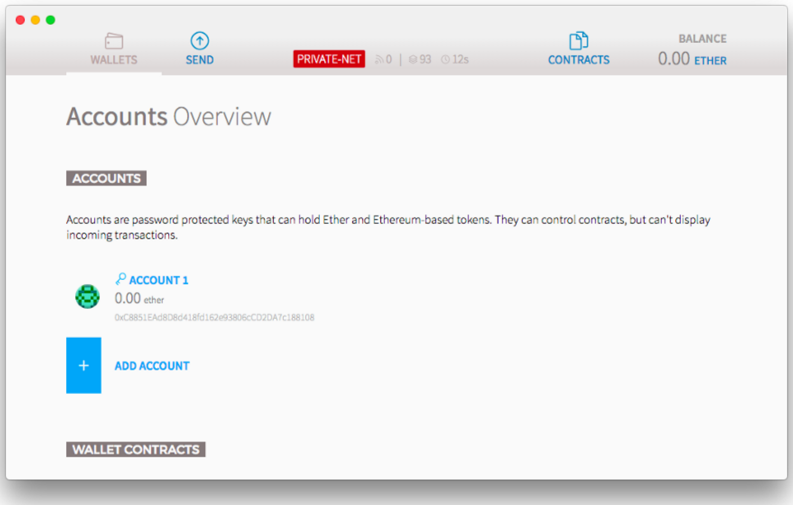
You need to create at least one account with a strong password.

Wallet on your private network

Notice the "PRIVATE-NET" alert (if it's not there you are doing something wrong). Notice as well that mined blocks count keeps going up — your network is alive and kicking.

## Creating an account

This is easy. Click "ADD ACCOUNT" and provide a password you are capable to remember in a few minutes (it gets harder with age, believe me). The UI now looks like this:



Accounts Overview

ACCOUNTS

Accounts are password protected keys that can hold Ether and Ethereum-based tokens. They can control contracts, but can't display incoming transactions.

ACCOUNT 1
0.00 ether
0xC8851EAd8D8d418fd162e93806cCD2DA7c188108

OMG, an account id!

Now copy this account id somewhere (the UI has a helper tool anyway) — we will start another mining node in order to mine into this account. Ignore all the warning messages that assume you don't know what you are doing while copying the id on the Wallet UI and finally close the Wallet.

### Mining into an account

In order to restart the mining node and to choose the account it mines into just supply an "ETHERBASE" variable to the script with the account id:

```
ETHERBASE=0x5600...fc050 RPC_PORT=8545 ./runminer.sh wallet
```

This variable is used in the script to configure the mining container properly.

Reopen the Wallet (run it in the prompt the same way before) and see that it now mines ether into the recently created account.

Actually you don't even have to mine in the same node the account was created, but anywhere in the network (remember: wait for the DAG to end):

```
ETHERBASE=0x5600...fc050 ./runminer.sh miner1
```

The node you are connected to with the Wallet will eventually receive the mined blocks from the "miner1" node.

## Homework

This can go on and on. Things you can do now just for the fun of it:

### Transfer ether between accounts

Start another mining node exposing the JSON-RPC on another port (just use another port number in the RPC_PORT variable), and **wait for the DAG process to finish**;

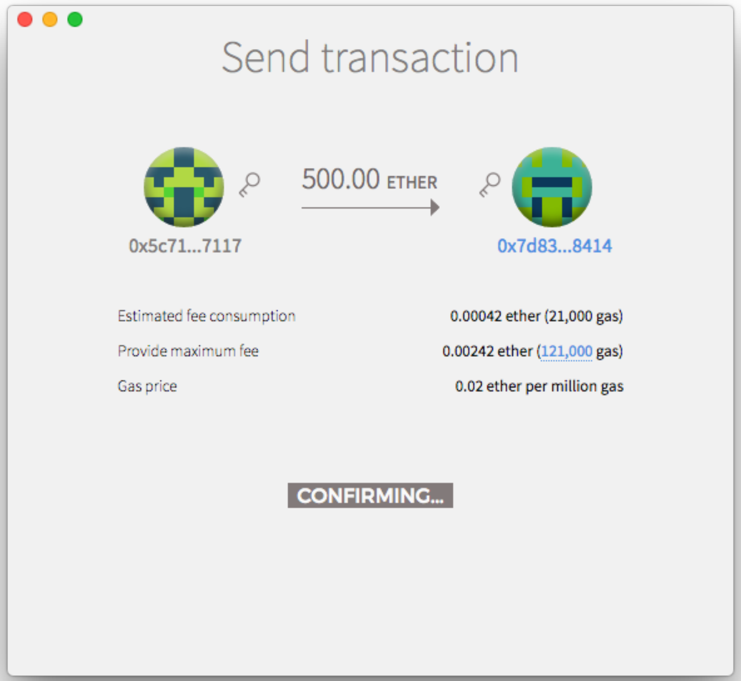
Run another Wallet UI (use this new port on the command-line URL argument);

Create another account in this new node;

Copy the new account id to the clipboard;

Go back to the first Wallet and transfer ether from that account into the new account (you just need the destination account id);

Wait a few seconds and the funds will go to the new account!



Estimated fee consumption 0.00042 ether (21,000 gas)
Provide maximum fee 0.00242 ether (121,000 gas)
Gas price 0.02 ether per million gas

Ether "wire transfer"

This is really awesome!

•

# Building an Ethereum playground with Docker (part 4— Provisioning)

This is the fourth article in a series on “Building an Ethereum Playground…”. The articles already published are:

The series will cover creating a proper Docker image, playing with Ethereum Wallet, provisioning the ethereum nodes on public clouds and deployment of a sample app.

This article is about the art of provisioning servers on public cloud providers in a way that you forget about them as soon as possible. We will always pursue a "CaaS" (Container-as-a-Service) workflow — in real life this results on portable projects, immutable infrastructure and better sleep time.

All sources are located in https://github.com/vertigobr /ethereum.git.

## The Point of CaaS Provisioning

We will discuss in this article how to provision a a cluster (swarm) of containers to run our private Ethereum, wether locally (using Docker for Mac or Docker for Windows) or on public cloud providers like Azure and Carina.

### There is not a wrong way to use Docker

Well, of course there are many ways to screw things with Docker or not, but the point is that containers are incredibly useful in many different situations. Whether you are an old grey-bearded sysadmin or a pimpled millennial, you will count on your fingers how many times in your lifetime there was a new technology so disruptive and yet so simple to understand, experiment and use. A small startup that in a few years impacted deeply the business of all bullies that ruled the party for decades? You have to watch in awe with a sincere smile.

### Localhost or Cloud, whatever

I am one of those who consider the term "serverless" a very confusing and silly one (or at least quite a poor choice). No software runs on ether (not even Ethereum) — there is always a server somewhere that costs money and suffers from bad code (deal with it).

The philosophy of serverless, on the other hand, is a smart one: *you don't need to care about provisioning and may outsource (or ignore) it completely*. Martin Fowler wrote a good piece about it: serverless on the real world materializes on offers on BaaS and FaaS space. This is a very rich subject — and also a sketchy one, for some FaaS choices put us back in the old vendor lock-in trap.

Back to containers, though, the serverless philosophy comes forward on CaaS (Container as a Service) offerings, like Carina. With CaaS we are talking more of a vm-less than a serverless case, but the point is that you get to deal directly with a Docker engine/swarm ignoring the existence of an underlying VM — and that is a good thing. How many actual VMs support you engines is a completely irrelevant (and invisible) issue in a "pure" CaaS product.

With CaaS you live on a optimal workflow when dealing with containers — even if you are not using a pure CaaS cloud provider, this workflow should be pursued by all means available.

Provisioning a container infrastructure in your development machine in the old days was done with Boot2Docker, later on with Docker Toolbox and (more recently) with Docker for Mac/Windows. Besides a more lightweight local environment (VirtualBox replaced by xhyve/hyper-v) what else has changed? In Docker for Mac/Windows the underlying VM is invisible — a good deal of effort was put in place in order to make it even unreachable, so you are naturally forced to work CaaS-style.

When provisioning Docker engines on a physical datacenter of your own you might have to decide going bare metal (like in HPE offer) or with a layer above your current virtualization solution (loads of choices). When provisioning engines on a public cloud you don't get that choice, but to some point it is also an irrelevant one.

Alas in all cases you should be pursuing a "CaaS lifestyle" — reaching a workflow where your "docker" commands talk to a local or remote infrastructure the same way. This is why the new swarm mode on Docker 1.12 sent so many people hopping mad on twitter about forking Docker: *the orchestration layer isn't a lock-in ground anymore* (and tooling was kept simple).

In other words, you want to ignore the VMs existence whenever you can.

So this article will tread on a few provisioning choices for a generic container-based clustered deployment where, at the end, the reader will be able to start, stop, wipe out and fool around with its nodes the same way, regardless of how/where/by whom they were provisioned.

## Docker beyond localhost — Clusters

Let us get some background on a few public cloud provider tools. You can get some freebie time ou beta access on pretty much all of them, so it is assumed you've already done that and have your logins, keys and/or subscription ids at hand (each provider has its own authentication waltz).

We are going to provision clusters in:

- Rackspace Carina
- Azure (with Azure CLI)
- Azure (with docker-machine)
- Amazon ECS (with ECS-CLI)
- Amazon EC2 (with docker-machine)

### Your local tools

This document assumes you are currently in a shell where:

- Docker client is available (like in an OS prompt after Docker for Mac/Windows is installed);
- You have cloned this sample repository and "cd" into it;
- Your Docker client is compatible with the Docker Engine you are remotely talking to (you should install Carina DVM to switch it freely);
- You are capable of generating an SSH key (you can do it with Putty in Windows). C'mon, you did it when you created your Github account;
- You have a few unix tools in the command-line (like "ssh") — Windows users could try a few options like Cygwin, or work

around with tools like Putty.

> *In all scenarios here where a VM is created we will favor using passwordless SSH login (i.e. private/public SSHkeys).*

## The "setenv.sh" script

This document assumes you have a local `setenv.sh` script that sets several environment variables to whatever values are needed to connect to your cloud provider.

A template to such a file is present on the repository, named `setenv.sh.template`. Copy it into `setenv.sh` and modify it to what your provider requires:

Carina CLI: requires environment variables CARINA_USERNAME and CARINA_APIKEY;

Azure CLI: needs a few environment variables that will guide resource creation on Azure (check the template);

Azure on docker-machine: needs a valid subscription id from Azure;

Amazon ECS-CLI: needs an AWS access key and its secret;

IMPORTANT: You should change names that would conflict with any other resources on Azure cloud (ex: the "AZURE_DNS_PREFIX" variable that names the VMs hosts created on Azure).

## Carina

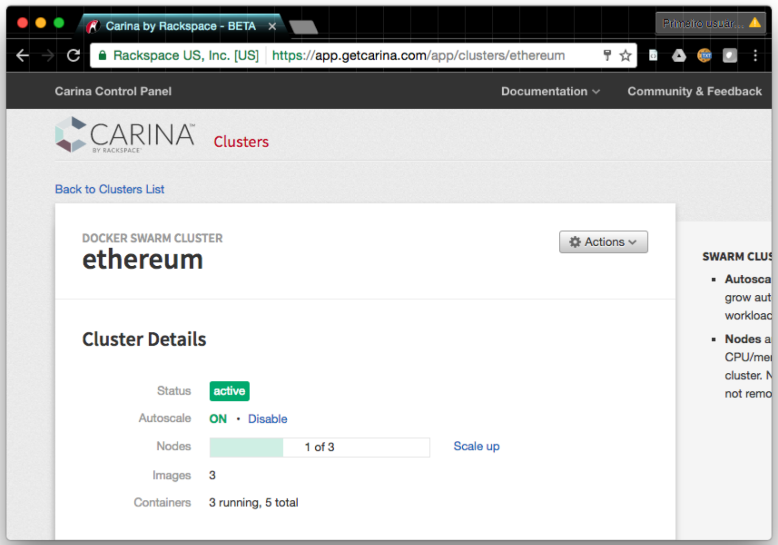
From all CaaS providers Carina from Rackspace has the simplest workflow I know of. Carina itself has been in beta for ages, and I am one of the blessed ones who hold a beta account.

The proper tooling for Carina play are the "Carina CLI" and the optional "DVM". DVM is also useful even if you are not using Carina at all — it is a very easy way to switch your Docker client at will. Download and install instructions for Carina CLI are here.

You must create a cluster (i.e. a swarm) in Carina before running containers in it. You can do this on the web console or on the command-line:

```
carina create --autoscale ethereum
```

This gives us a new swarm (a cluster of Docker engines) name "ethereum" and ready to roll in a very short time. You can always check it out on getcarina.com web console:



DOCKER SWARM CLUSTER
ethereum

Cluster Details

Status active
Autoscale ON · Disable
Nodes 1 of 3
Images 3
Containers 3 running, 5 total

SWARM CLUS
- Autosca grow aut workload
- Nodes a CPU/me cluster. N not remo

Carina admin console

At any time you can configure your current shell to use this swarm as the target for `docker` commands:

```
eval $(carina env ethereum)
```

This command sets several environment variables recognized by the Docker client. Check these with the command below:

```
➜  env | grep DOCKER
DOCKER_HOST=tcp://146.20.69.138:2376
DOCKER_TLS_VERIFY=1
DOCKER_CERT_PATH=(...)/.carina/clusters/xxx@yyy/ethereum
DOCKER_VERSION=1.11.2
```

The exposed engine API port (2376) is useless unless you have its keys, so this is a safe setup. This is actually what Docker recommends.

The resulting behavior is that *all* docker commands are now sent to Carina (and not to your local engine). On my Mac, for example, `docker version` shows these results (notice the difference between the client and the engine):

```
➜  docker version
Client:
 Version:      1.12.1
 API version:  1.24
 Go version:   go1.6.3
 Git commit:   23cf638
 Built:        Thu Aug 18 17:32:24 2016
 OS/Arch:      darwin/amd64
 Experimental: true

Server:
 Version:      swarm/1.2.3
 API version:  1.22
 Go version:   go1.5.4
 Git commit:   eaa53c7
 Built:        Fri May 27 17:25:03 UTC 2016
 OS/Arch:      linux/amd64
```

Congratulations, you are now living on CaaS style. Ideally you should be able to run containers the same way you did before — the fact that the engine is somewhere else should be irrelevant. We'll get back to it later.

> *Notice: at the time of this writing the Carina swarm is not an implementation of the new swarm mode of Docker 1.12 (i.e. no "docker service" commands). Instead Carina swarm works in a way that your "docker run" commands are attended by any node in the swarm. Rackspace states that the new swarm mode will be supported in the future.*

## Azure Container Service (Azure CLI)

Azure is Microsoft's cloud provider that competes (or tries to) with AWS. It is a generic cloud provider (not focused solely on containers like Carina is), but it has some VM templates aimed at Docker users.

One way to create a swarm cluster is using Azure CLI and a recently available template for that: the Azure Container Service. You can follow this link to create the ACS swarm manually or try the semi-scripted path below.

Please note that, at the end of the day, with ACS you get a somewhat cumbersome experience and still have to deal with VMs — this is nothing at all a CaaS workflow, but we will get as close as possible.

Download and install instructions for Azure CLI are here.

First you have to login into Azure with `azure login`:

```
➜  azure login
info:    Executing command login
/info:    To sign in, use a web browser to open the page
https://aka.ms/devicelogin. Enter the code XXXXXXXXX to
authenticate.
```

Azure CLI authentication asks you to validate this login from any device with a browser. Do as you are told and the command above will continue:

```
/info:    Added subscription Visual Studio Enterprise
info:    Added subscription Azure Pass
info:    Setting subscription "Visual Studio Enterprise" as
default
+
info:    login command OK
```

Test the current session with the command below:

```
azure account show
```

Now try the command:

```
➜ . ./setenv.sh
➜ azure config mode arm
info:    Executing command config mode
info:    New mode is arm
info:    config mode command OK
➜ azure group create $AZURE_RESOURCE_GROUP $AZURE_LOCATION
(...)
info:    group create command OK
➜ azure group deployment create $AZURE_RESOURCE_GROUP
$AZURE_DEPLOYMENT_NAME --template-uri $AZURE_TEMPLATE_URL -e
azure/azuredeploy.parameters.json
```

The ACS will then be created:

```
info:    Executing command group deployment create
+ Initializing template configurations and parameters
+ Creating a deployment
...
data:    Outputs            :
data:    Name        Type    Value
data:    ----------  ------
-------------------------------------------------------------
-------------
data:    masterFQDN  String  ethereum-
playgroundmgmt.eastus.cloudapp.azure.com
data:    sshMaster0  String  ssh etheruser@ethereum-
playgroundmgmt.eastus.cloudapp.azure.com -A -p 2200
data:    agentFQDN   String  ethereum-
playgroundagents.eastus.cloudapp.azure.com
info:    group deployment create command OK
```

Our template creates a Swarm with one master node and three agent nodes (this is the "old" Docker Swarm, not the Swarm mode from Docker 1.12.1).

Azure takes quite some time to finish, but eventually you will be able to see on its web admin console all the resources created under "EthereumPlayground" resource group.

> *IMPORTANT: our script created all these VMs as Ubuntu 14.04 with a local (sudoer) user named "etheruser". Also this user accepts key-based SSH logins with your default "id_rsa" key (your public key "id_rsa.pub" was used to configure the VMs).*

You can — but you usually WON'T — access your VMs with SSH (port 2200). Your default private key would work (passwordless login):

```
➜ ssh etheruser@ethereum-
playgroundmgmt.eastus.cloudapp.azure.com -p 2200
```

Now the closest we can get to a CaaS lifestyle on ACS (where you issue “docker” commands in your workstation talking to the remote swarm master) is achieved creating a SSH tunnel to the master Engine API port. Yes, it feels weird, but that's what Microsoft recommends. You create the tunnel and set DOCKER_HOST variable to use it:

```
➜ ssh -L 3375:localhost:2375 -f -N etheruser@ethereum-
playgroundmgmt.eastus.cloudapp.azure.com -p 2200
➜ export DOCKER_HOST=localhost:3375
➜ docker ps
Client:
 Version:      1.12.1
 API version:  1.24
 Go version:   go1.6.3
 Git commit:   23cf638
 Built:        Thu Aug 18 17:32:24 2016
 OS/Arch:      darwin/amd64
 Experimental: true

Server:
 Version:      swarm/1.1.0
 API version:  1.22
 Go version:   go1.5.3
 Git commit:   a0fd82b
 Built:        Thu Feb  4 08:55:18 UTC 2016
 OS/Arch:      linux/amd64
```

There you go, your swarm cluster is online.

> *Notice: at the time of this writing, the Docker swarm created this way is the old version of Swarm (usually refered as "standalone Swarm" on Docker docs), not the new "Swarm mode" introduced in Docker 1.12.*

## Azure on Docker-Machine

Another way to provision a Docker Swarm on Azure is using the `docker-machine` utility, a tool provided by Docker to simplify the provisioning of Docker engines/clusters on any cloud provider its drivers know about.

The `docker-machine` workflow delegates to its drivers all the hassle of dealing with cloud vendor specific mumbo-jumbo.

> *Notice: at the time of this writing, docker-machine can indeed be used to provision a swarm with its "--swarm" arguments but the Docker swarm created this way is the old version of Swarm (usually refered as "standalone Swarm" on Docker docs), not the new "Swarm mode" introduced in Docker 1.12. We are going to provision a cluster with the new Swarm mode — so we will use docker-machine just to provision simple Docker nodes and later on we will turn them into a swarm.*

To provision a simple Docker node (i.e. an Azure VM with Docker pre-installed) on Azure we run :

```
➜ source setenv.sh
➜ docker-machine create --driver azure \
  --azure-subscription-id $AZUREDM_SUBSCRIPTION_ID \
  --azure-docker-port 2376 \
  --azure-location $AZUREDM_LOCATION \
  --azure-resource-group $AZUREDM_RESOURCE_GROUP \
  --azure-ssh-user etheruser \
  docker-node1
Running pre-create checks...
(docker-node1) Completed machine pre-create checks.
Creating machine...
...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine
running on this virtual machine, run: docker-machine env
docker-node1
```

> *Notice: Azure authentication will hang your command-line and ask you to open a specific URL in a browser on any device available and complete a web login. This will surely happen on the first time you run docker-machine, but authentication is cached and won't bother you again for some time.*

Repeat the command above to create nodes `docker-node2` and `docker-node3`. At the time of this writing the VMs created on Azure this way are running Ubuntu 16.04 LTS and Docker 1.12.

After creating all three VMs you can list them with the command below:

```
➜  docker-machine ls
NAME           ACTIVE   DRIVER   STATE     URL   SWARM
DOCKER...
docker-node1            azure    Timeout
docker-node2            azure    Timeout
docker-node3            azure    Timeout
```

To set your local shell to work with the remote "docker-node1" engine you just have to run:

```
➜ eval $(docker-machine env docker-node1)
```

This sets a bunch of environment variables that affect your Docker client. To test the connectivity with the remote Engine run `docker version`:

```
➜  docker version
Client:
 Version:      1.12.1
 API version:  1.24
 Go version:   go1.7.1
 Git commit:   6f9534c
 Built:        Thu Sep  8 10:31:18 2016
 OS/Arch:      darwin/amd64

Server:
 Version:      1.12.1
 API version:  1.24
 Go version:   go1.6.3
 Git commit:   23cf638
 Built:        Thu Aug 18 05:33:38 2016
 OS/Arch:      linux/amd64
```

Each of these Azure VMs has a public IP (a "real" IP visible to the world) and a private IP (visible to other VMs in the same Azure virtual network). To find `docker-node1` public IP just type:

```
➜ docker-machine ip docker-node1
```

And to find its private IP just type (look for the eth0 device):

```
➜ docker-machine ssh docker-node1 ip addr
...
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq
state UP group default qlen 1000
    link/ether XX:XX:XX:XX:XX:XX brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.4/16 brd 192.168.255.255 scope global eth0
...
```

Ok, now it is time to "convert" these three independent Docker engines into a swarm (the new Swarm mode from Docker 1.12). We will start with `docker-node1` and move to each node afterwards (please replace "**192.168.0.4**" with your own private IP discovered as above explained).

```
➜ eval $(docker-machine env docker-node1)
➜ docker swarm init --advertise-addr 192.168.0.4
Swarm initialized: current node (23coeyfn8tsexzlvcehxfpjac)
is now a manager.

To add a worker to this swarm, run the following command:

docker swarm join \
    --token SWMTKN-1-20m9v...2y68rl \
    192.168.0.4:2377

To add a manager to this swarm, run 'docker swarm join-token
manager' and follow the instructions..
```

Easy! Write down the join snippet from the output (specially the token string), we will run it for each node. To check the swarm members:

```
➜  docker node ls
ID               HOSTNAME      STATUS  AVAILABILITY  MANAGER
STATUS
23co...fpjac *  docker-node1  Ready   Active        Leader
```

To add `docker-node2` to the swarm just run the command below:

```
➜ eval $(docker-machine env docker-node2)
➜ docker swarm join \
    --token SWMTKN-1-20m9vvf...2y68rl \
    192.168.0.4:2377
This node joined a swarm as a worker.
```

Repeat for `docker-node3` and you will have the swarm ready:

```
➜ eval $(docker-machine env docker-node1)
➜  docker node ls
ID             HOSTNAME      STATUS  AVAILABILITY  MANAGER
STATUS
0wz4...c2i2    docker-node3  Ready   Active
23co...pjac *  docker-node1  Ready   Active        Leader
9s5k...scke    docker-node2  Ready   Active
```

## Amazon on ECS-CLI

A cluster of Docker engines can be provisioned on AWS with a fast-food tool called `ecs-cli` that can be downloaded here. Configuring ECS-CLI is simple:

```
➜ # sets env vars
➜ source setenv.sh
➜ ecs-cli configure --region=$AWS_REGION --cluster
$AWS_CLUSTER
INFO[0000] Saved ECS CLI configuration for cluster (ethereum-
playground)
```

Assuming you already created a keypair in AWS console, a 3-node cluster is created in a simple instruction:

```
➜ ecs-cli up --keypair id_rsa --capability-iam --size 3
--instance-type t2.nano
INFO[0002] Created cluster
cluster=ethereum-playground
INFO[0005] Waiting for your cluster resources to be created
...
```

## Amazon on docker-machine

As usual, docker-machine keeps it simple. To create a Docker-enabled VM on AWS you can type:

```
➜ # sets env vars
➜ source setenv.sh
➜ docker-machine create --driver amazonec2 \
  --amazonec2-region $AWS_REGION \
  --amazonec2-zone b \
  --amazonec2-instance-type t2.micro \
  --amazonec2-ami ami-c60b90d1 \
  etherplay-1
Running pre-create checks...
Creating machine...
(etherplay-1) Launching instance...
...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine
running on this virtual machine, run: docker-machine env
etherplay-1
```

> *Notice: the "--amazonec2-zone" is "b" because the default "a" is a non-existant (or non-available) zone, and the "--amazonec2-instance-type" is "t2.micro" because this is the instance that falls in the monthly free tier. The "--amazonec2-zone" is forcing a Ubuntu 16.04 LTS, because in my tests the default VM image was 15.01. If a future docker-machine release fixes this you may remove these arguments. BTW, AMI images per region can be found here.*

Do the same for other VMs named "etherplay-2" and "etherplay-3". Just like in the Azure example we will configure these nodes into Swarm mode. We choose "etherplay-1" to become the swarm master, so we need to know its private IP (look for the "eth0" device):

```
➜ docker-machine ssh etherplay-1 ip addr
...
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 ...
    link/ether xx:xx:...:xx brd ff:ff:ff:ff:ff:ff
    inet 172.31.63.152/20 brd 172.31.63.255 scope global eth0
       valid_lft forever preferred_lft forever
...
```

We can now enable Swarm mode on "etherplay-1" (please use the IP you found):

```
➜ eval $(docker-machine env etherplay-1)
➜ docker swarm init --advertise-addr 172.31.63.152
Swarm initialized: current node (1qvglfulku1lxiz0o57vqnc2t)
is now a manager.

To add a worker to this swarm, run the following command:

docker swarm join \
    --token SWMTKN-1-5aqjx...bp0wzs \
    172.31.63.152:2377

To add a manager to this swarm, run 'docker swarm join-token
manager' and follow the instructions.
```

Write down the token string to use it later. To add "etherplay-2" and "etherplay-3" as worker nodes (use the token string obtained on the last step):

```
➜ eval $(docker-machine env etherplay-2)
➜ docker swarm join \
    --token SWMTKN-1-5aqjxa...bp0wzs \
    172.31.63.152:2377
```

Finally check the swarm member list — the swarm is ready!

```
➜ eval $(docker-machine env etherplay-1)
➜ docker node ls
ID                      HOSTNAME     STATUS  AVAILABILITY
MANAGER..
16fidk7...jvddy34x0     etherplay-3  Ready   Active
3wdbip1...63mjmqen6 *   etherplay-1  Ready   Active
Leader
4tntm5f...2zce00rlo     etherplay-2  Ready   Active
```

> *Notice: for some reason during my tests port 2377 in was blocked on the private IP. In that case joining fails and `docker node ls` returns only the swarm master. To fix this you must edit the security group "docker-*

> *machine" in AWS console and enable an inbound rule (incoming port=2377, source=the security group itself).*

### Other Cloud Providers

Other cloud providers will behave the same way: there is a command-line tool that can be used to provision VMs based on well-known templates, but there is also a docker-machine driver that will provision a specific VM with Docker pre-installed.

Digital Ocean has `doctl` and Google GCE has `gcloud`, for example. Both have also a docker-machine driver in place as well. *The docker-machine workflow takes care of a lot of things and results in a more portable and consistent process.*

Considering the developer convenience and the will to avoid vendor lock-in, it is my belief that the best tool for provisioning small-scale operations is docker-machine. Any reasonably complex project can be provisioned and replicated automatically in a way that the cloud provider becomes a commodity with little impact for development teams.

Cloud providers like Digital Ocean deliver a very simple model of provisioning; others, like AWS or Azure, have a more complex feature set. Still, docker-machine handles all providers gracefully. Remember: less is more.

## Sidenotes and finishing up

Using docker-machine, once you get it to work with your cloud provider, is irresistible. You get to control remote Docker engines securely and with little effort; turning these engine into a swarm is also quite easy (and cloud-agnostic).

A few sidenotes…

### …on docker-machine

Well, docker-machine is awesomely simple to use — it creates secure nodes and generates the keys for SSL communication between client and engine without a sweat.

Carina CLI does the same, with one practical advantage: it recreates all these client settings anywhere you run it with proper credentials. Unlike Carina CLI, docker-machine settings are kept on the machine that ran the original "docker-machine create" commands.

There are a few "export/import" tools for docker-machine, but those are still a little crude. Just keep that in mind that all you need is in the "$HOME/.docker" folder if you want to rescue it later (or manage the machines from somewhere else).

> *Notice: I have moved the ".docker" folder from a Mac into a Linux machine and I just had to fix the paths in each "config.json" under "~/.docker/machine/machines".*

### …on swarm managers

There can be (and should be) several managers in a swarm with high availability. Given the fluid nature of clouds, it is a wise choice to use reserved (fixed) IPs when creating a swarm manager. The "docker-machine" utility has an option for doing this on Azure ("--azure-static-public-ip").

Once again, don't be confused with the "--swarm" options on docker-machine, because they still refer to the old Swarm.

### …on proprietary tools and Kubernetes

I have been insisting on calling "CaaS-style" those providers that, at the end of the day, allow you to continue to use the "docker" client naturally. ECS-CLI is not such a case, even though it is somewhat compatible with "docker-compose" syntax.

Until recently Kubernetes was praised as the winning standard for container orchestration (specially if you believe in Gartner reports) — since Docker 1.12 nobody can be that sure. The new "Swarm Mode" is still very crude but surely will evolve very fast, resulting in HA clusters that are managed by the same tools of a single node local development workstation. *It is more likely that Swarm Mode will evolve into a complete large-scale solution than Kubernetes will ever become simpler. But, hey, that's just my opinion.*

Still Kubernetes is open-sourced and freely used on several cloud vendors (look for "Turn-key Cloud Solutions") — including Google GCE and Amazon AWS, so it is not vendor-locking *per se* nor proprietary at all. It just forces the developer into a whole new set of tools and overlapping syntax, and I really really really hate that.

### ...on old swarm and new swarm mode

To keep it bluntly (even if somewhat innacurate):

> Old swarm: "docker run", "docker-compose" etc. target the swarm transparently;
>
> Swarm mode (new in Docker 1.12): you use "docker service" instead.

Just be careful if provisioning the cluster with the cloud provider tools — most of them refer to the old swarm (even "docker-machine" does it until version 0.8.1 if you use the "--swarm" option). If you want the new swarm mode be sure to provision simple Docker nodes (with docker-machine or not) and then turn them into a swarm.

### ...on volumes

Dealing with volumes on swarms is an entire different beast — it makes no sense to rely on local folder mounts for that — containers are expected to "float" around the swarm and local folders kinda kill the whole purpose.

We will cover this subject on the next article.