The technology partner for entrepeneurs
www.talpor.com

Jan
28

16 Comments

Docker - Beginner's tutorial

Docker is a relatively new and rapidly growing project that allows to create very light “virtual machines”. The quotation marks here are important, what Docker allows you to create are not really virtual machines, they’re more akin to chroots on steroids, a lot of steroids. In this tutorial we'll explore what Docker can do for you and how does it do what it does.

Before we continue, let me clear something up. As of right now (4th of January of 2015) Docker works only on Linux, it cannot function natively on Windows or OSX. I’ll be talking about the architecture of Docker later on and the reason will become obvious. So if you want Docker on a platform that is not Linux, you’ll need to run Linux on a VM.

This tutorial has three objectives: explaining **what problem it solves**, explaining **how it solves it** at a high level, and explaining **what technologies does it use** to solve it. This is not a step-by-step tutorial, there are already many good step-by-step tutorials on Docker, including an online interactive one from the authors of Docker. That said, there is a little step-by-step at the end, it's just there to connect all of the theory I present during the post with a clearcut realworld example, but is by no means exhaustive.

What can Docker can do for you?

Docker solves many of the same problem that a VM solves, plus some other that VMs could solve if they didn't were so resource intensive. Here are some of the things that Docker can deal with:

- Isolating an application dependencies
- Creating an application image and replicating it
- Creating ready to start applications that are easily distributable
- Allowing easy and fast scalation of instances
- Testing out applications and disposing them afterwards

The idea behind Docker is to create **portable lightweight containers for software applications** that can be run on any machine with Docker installed, regardless of the underlying OS, akin to the cargo containers used on ships. Pretty ambitious, and they're succeeding.

What does Docker do exactly?

In this section I will not be explaining what technologies Docker uses to do what it does, or what specific commands are available, that's on the last section, here I'll explain the resources and abstractions that Docker offers.

The two most important entities in Docker are **images** and **containers**. Aside from those, **links** and **volumes** are also important. Let's start with images.

Images

Images on Docker are like the snapshot of a virtual machine, but way more lightweight, way way way more lightweight (more on the next section).

There are several ways to create an image on Docker, most of them rely on creating an new image based on an already existing image, and since there are public images to pretty much everything

you need, including for all the major linux distributions, it's not likely that you will not find one that suit your needs. If you however feel the need to build and image from scratch, there are ways.

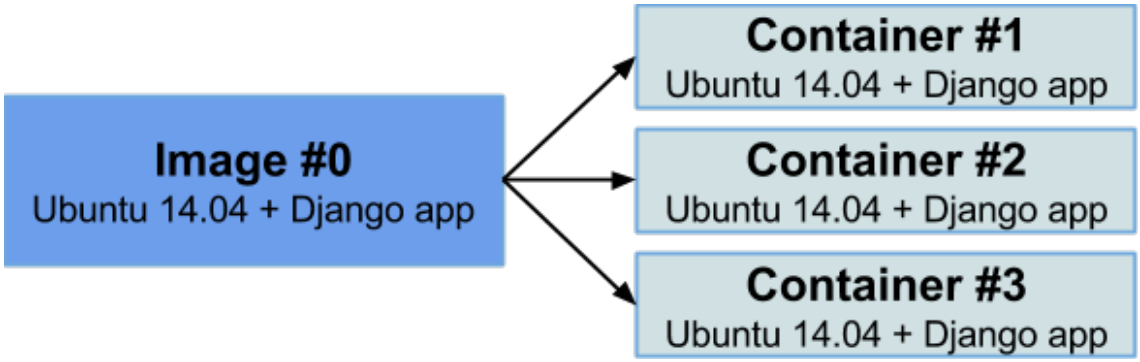
To create an image you take one image and modify it to create a child image. This can be done either through a file that specifies a base image and the modifications that are to be done, or live by "running" an image, modifying it and committing it. There are advantages to each method, but generally you'll want to use a file to specify the changes.

Images have an unique ID, and an unique human-readable name and tag pair. Images can be called, for example, ubuntu:latest, ubuntu:precise, django:1.6, django:1.7, etc.

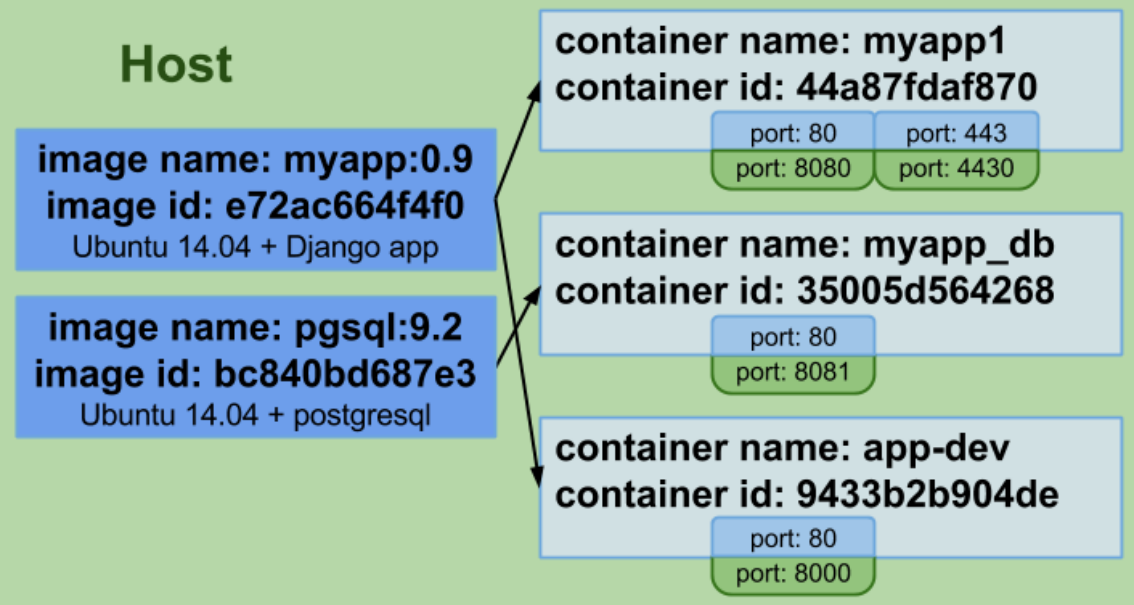
Containers

Now onto containers. From images you can create containers, this is the equivalent of creating a VM from a snapshot, but way more lightweight. Containers are the ones that run stuff.

Let use an example, you could download an image of ubuntu (there is a public repository of images called the docker registry), modify it by installing Gunicorn and your Django app with all its dependencies, and then create a container from that image that runs your app when it starts.



Containers, like VMs, are isolated (with one little caveat that I'll discuss later). They also have an unique ID and a unique human-readable name. It's necessary for containers to expose services, so Docker allows you to expose specific ports of a container.



Containers have one big difference that separate them from VMs, they are designed to **run a single process**, they don't simulate well a complete environment (if that's what you need check out LXC). You may be tempted to run a runit or supervisord instance and get several processes up, but it's really not necessary (in my humble opinion).

The whole single process vs multiple processes is somewhat of an outstanding debate. You should know that the Docker designers heavily promote the "one process per container approach", and that the only case where you really have no other option but to run more than one process is to run something like ssh, to access the container while it is running for debugging purposes, however the command docker exec solves that problem.

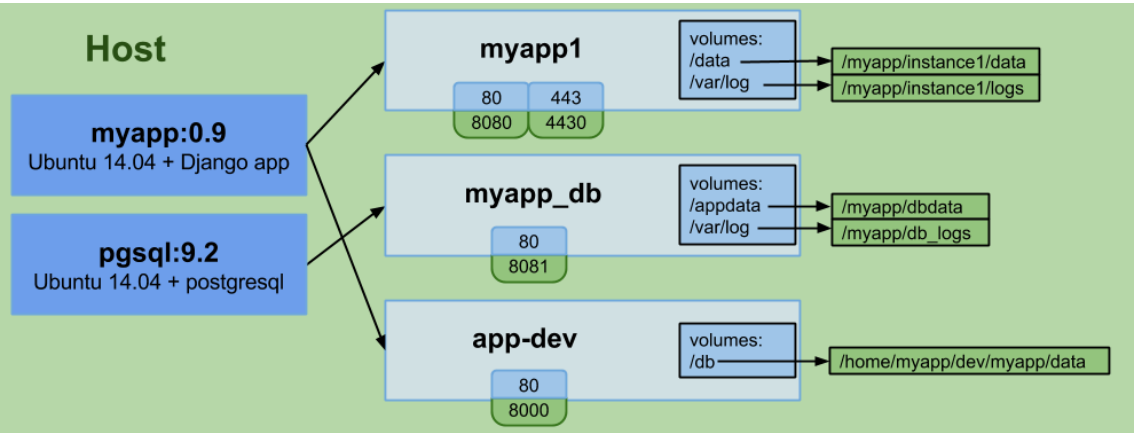
~~The second big difference between containers and VMs is that when you stop a VM, no files are erased besides maybe some temporary files, when you stop a Docker container all **changes** done to the initial **state** (the state of the image from which the container was created) are **lost**~~ (This is not true, thanks to Sean for pointing it out).

Containers are desing thus, to run an **application**, not a machine. You may use containers as VMs, but as we will see, you would lose a lot of flexibility, since Docker gives tools to **separate your application from your data**, allowing you to update your running code/systems rapidly and easily without putting at risk your data.

Volumes

Volumes are how you persist data beyond the lifespan of your container. They are spaces inside the container that store data outside of it, allowing you to destroy/rebuild/change/exile-to-oblivion your container, keeping your data intact. Docker ~~forces~~ allows you to define what parts are your **application** and what parts are your **data**, and ~~demands that you~~ gives you the tools to keep them **separated**. One of the biggest changes in mindset that one must make when working with Docker is that **containers should be ephemeral and disposable**.

Volumes are specific to each container, you can create several containers from a single image and define different volumes for each. Volumes are stored in the filesystem of the host running Docker. Whatever is not a volume is stored in other type of filesystem, but more on that later.



Volumes can also be used to share data between containers, I recommend reading the volumes documentation for details.

Links

Links are another very important part of Docker.

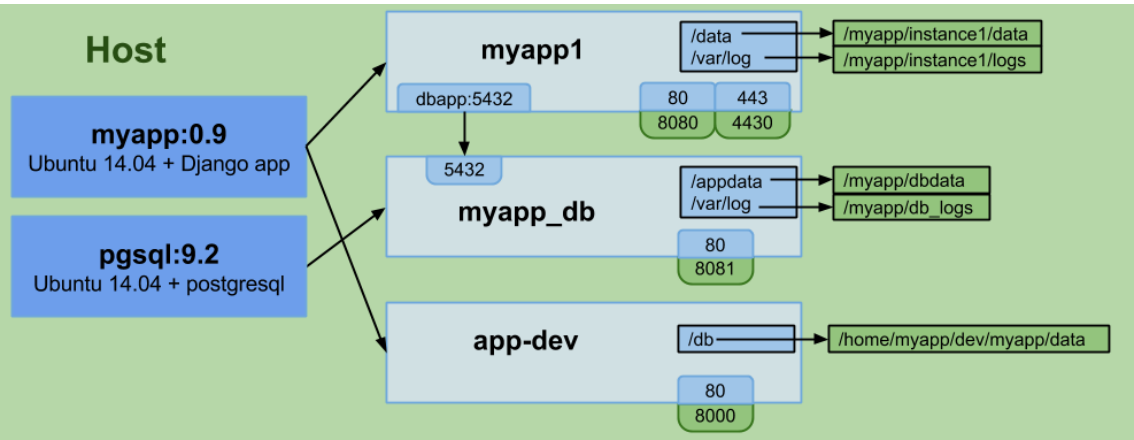
Whenever a container is started, a random private IP is assigned to it, other containers can use this IP address to communicate with it. This is important for 2 reasons: first it provides a way for containers to talk to each other, second containers share a local network; I had a problem once when I started two elasticsearch containers for two clients on the same machine, but left the cluster name to the default setting, the two elasticsearch servers promptly made an unsolicited cluster. *Edit: Restricting intercontainer communication is possible, read the advanced networking documentation of Docker for details.*

To allow intercontainer communication Docker allows you to reference other existing containers

when spinning up a new one, those referenced containers receive an alias (that you specify) inside the container you just created. We say that the two containers are **linked**.

So if my DB container is already running, I can create my webserver container and reference the DB container upon creation, giving it an alias, *dbapp* for example. When inside my newly created webserver container I can use the hostname *dbapp* to communicate with my DB container at any time.

Docker takes it one step further, requiring you to state which ports a container will ~~make available~~ advertise to other containers when it is linked. ~~otherwise no ports will be available~~ When new containers link to a container that advertises ports, Docker will generate environment variables inside the new containers with information about the exposed ports.



Portability of Docker images

There is one caveat when creating images. Docker allows you specify volumes and ports in an image. Containers created from that image inherit those settings. However, Docker doesn't allow you to specify anything on an image that is not portable.

You can define exposed ports, but only those ports that are exposed to other containers when links are created, you can't specify ports exposed to the host, since you don't know wich ports will be available on the hosts that might use that image.

You can't define links on an image either. Making a link requires you to reference another container by name, and you can't know beforehand how will the containers be named on every host that might use the image.

There are some other things that you can't do that are out of the scope of this post, but the point is that **images must be completely portable, Docker doesn't allow otherwise**.

So those are the primary moving parts, you create images, use those to create containers that expose ports and have volumes if needed, and connect several containers together with links. How can this all work with little to no overhead?

How does Docker do what it needs to be done?

Two words: cgroups and union filesystems. Docker uses cgroups to provide container isolation, and union filesystem to store the images and make containers ephemeral.

Cgroups

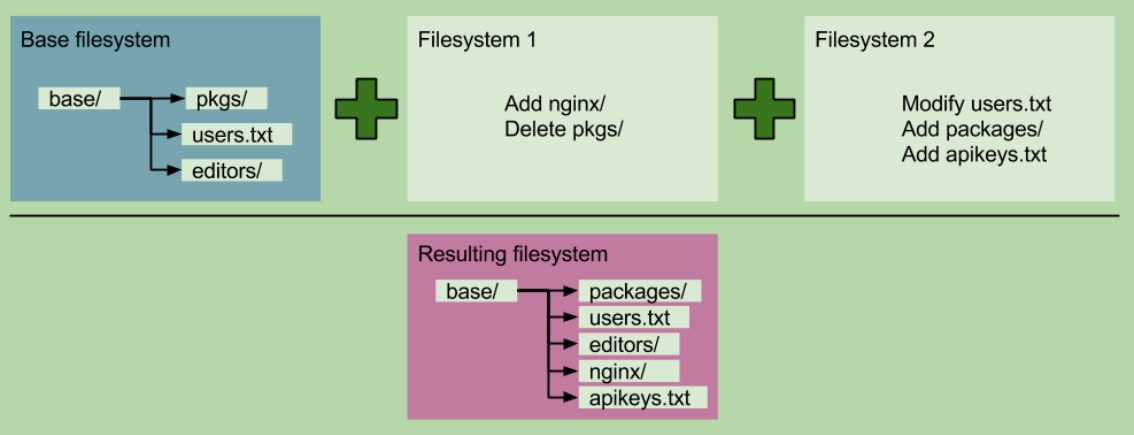
This is a Linux kernel feature that makes two things possible:

1. Limit resource utilization (RAM, CPU) for Linux process groups
2. Make PID, UTS, IPC, Network, User and mount namespaces for process groups

The keyword here is namespace. A PID namespace, for example, permits processes in it to use PIDs isolated and independent of the main PID namespace, so you could have your own init process with a PID of 1 within a PID namespace. Analogous for all the other namespaces. You can then use cgroups to create an environment where processes can be executed isolated from the rest of your OS, but the key here is that the processes on this environment **use your already loaded and running kernel**, so the overhead is pretty much the same as running another process. Chroot is to cgroups what I am to The Hulk, Bane and Venom combined.

Union filesystems

An union filesystem allows a layered accumulation of changes through an union mount. In an union filesystem several filesystems can be mounted on top of each other, the result is a layered collection of changes. Each filesystem mounted represents a collection of changes to the previous filesystem, like a diff.



When you download an image, modify it, and store your new version, you've just made a new union filesystem to be mounted on top of the initial layers that conformed your base image. This makes Docker images very light, for example: your DB, Nginx and Syslog images can all share the same Ubuntu base, each one storing only the changes from this base that they need to function.

As of January 4th 2015, Docker allows to use either aufs, btrfs or device mapper for union filesystems.

Images

Let me show you an image of postgresql:

```
[{
    "AppArmorProfile": "",
    "Args": [
        "postgres"
    ],
    "Config": {
        "AttachStderr": true,
        "AttachStdin": false,
        "AttachStdout": true,
        "Cmd": [
            "postgres"
```

```
        ],
        "CpuShares": 0,
        "Cpuset": "",
        "Domainname": "",
        "Entrypoint": [
            "/docker-entrypoint.sh"
        ],
        "Env": [
            "PATH=/usr/lib/postgresql/9.3/bin:/usr/local/sbin:/usr/local/
            "LANG=en_US.utf8",
            "PG_MAJOR=9.3",
            "PG_VERSION=9.3.5-1.pgdg70+1",
            "PGDATA=/var/lib/postgresql/data"
        ],
        "ExposedPorts": {
            "5432/tcp": {}
        },
        "Hostname": "6334a2022f21",
        "Image": "postgres",
        "MacAddress": "",
        "Memory": 0,
        "MemorySwap": 0,
        "NetworkDisabled": false,
        "OnBuild": null,
        "OpenStdin": false,
        "PortSpecs": null,
        "StdinOnce": false,
        "Tty": false,
        "User": "",
        "Volumes": {
            "/var/lib/postgresql/data": {}
        },
        "WorkingDir": ""
    },
    "Created": "2015-01-03T23:56:12.354896658Z",
    "Driver": "devicemapper",
    "ExecDriver": "native-0.2",
    "HostConfig": {
        "Binds": null,
        "CapAdd": null,
        "CapDrop": null,
        "ContainerIDFile": "",
        "Devices": null,
        "Dns": null,
        "DnsSearch": null,
        "ExtraHosts": null,
        "IpcMode": "",
        "Links": null,
        "LxcConf": null,
        "NetworkMode": "",
        "PortBindings": null,
        "Privileged": false,
```

```
            "PublishAllPorts": false,
            "RestartPolicy": {
                "MaximumRetryCount": 0,
                "Name": ""
            },
            "SecurityOpt": null,
            "VolumesFrom": [
                "bestwebappever.dev.db-data"
            ]
        },
        "HostnamePath": "/mnt/docker/containers/6334a2022f213f9534b45df33c644
        "HostsPath": "/mnt/docker/containers/6334a2022f213f9534b45df33c644370
        "Id": "6334a2022f213f9534b45df33c64437081a38d50c7f462692b019185b8cbc6
        "Image": "aaab661c1e3e8da2d9fc6872986cbd7b9ec835dcd3886d37722f1133baa
        "MountLabel": "",
        "Name": "/bestwebappever.dev.db",
        "NetworkSettings": {
            "Bridge": "docker0",
            "Gateway": "172.17.42.1",
            "IPAddress": "172.17.0.176",
            "IPPrefixLen": 16,
            "MacAddress": "02:42:ac:11:00:b0",
            "PortMapping": null,
            "Ports": {
                "5432/tcp": null
            }
        },
        "Path": "/docker-entrypoint.sh",
        "ProcessLabel": "",
        "ResolvConfPath": "/mnt/docker/containers/6334a2022f213f9534b45df33c6
        "State": {
            "Error": "",
            "ExitCode": 0,
            "FinishedAt": "0001-01-01T00:00:00Z",
            "OOMKilled": false,
            "Paused": false,
            "Pid": 21654,
            "Restarting": false,
            "Running": true,
            "StartedAt": "2015-01-03T23:56:42.003405983Z"
        },
        "Volumes": {
            "/var/lib/postgresql/data": "/mnt/docker/vfs/dir/5ac73c52ca86600a
            "postgresql_data": "/mnt/docker/vfs/dir/abace588b890e9f4adb604f63
        },
        "VolumesRW": {
            "/var/lib/postgresql/data": true,
            "postgresql_data": true
        }
    }
    ]
```

Thats it, images are just a json that specifies the characteristic of the containers that will be run from that image, where the union mount is stored, what ports are exposed, etc. Each image is associated with one union filesystem, each union filesystem on Docker has a parent, so images have a hierarchy. Several Docker images can be created from a same base, but each image may only have **one parent**, just like a computer science tree (unlike some other trees that have a bigger family group). Don't worry if it looks daunting or some things don't quite add up, you'll not be handling these files directly, this is for educational purposes only.

Containers

The reason containers are ephemeral is that, when you create a container from an image, Docker creates a blank union filesystem to be mounted on top of the union filesystem associated to that image.

Since the union filesystem is blank it means no changes are applied to the image's filesystem, when you create some change it gets reflected, but when the container is stopped the union filesystem of that container is discarded, leaving you with the original image's filesystem you started with. Unless you create a new image, or make a volume, your changes will always disappear on container stop.

What volumes do is to specify a directory within the container that will be stored it outside the union filesystem.

Here is a container for the ***bestwebappever:***

```
[{
    "AppArmorProfile": "",
    "Args": [],
    "Config": {
        "AttachStderr": true,
        "AttachStdin": false,
        "AttachStdout": true,
        "Cmd": [
            "/sbin/my_init"
        ],
        "CpuShares": 0,
        "Cpuset": "",
        "Domainname": "",
        "Entrypoint": null,
        "Env": [
            "DJANGO_CONFIGURATION=Local",
            "HOME=/root",
```

```
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin
            "TALPOR_ENVIRONMENT=local",
            "TALPOR_DIR=/opt/bestwebappever"
        ],
        "ExposedPorts": {
            "80/tcp": {}
        },
        "Hostname": "44a87fdaf870",
        "Image": "talpor/bestwebappever:dev",
        "MacAddress": "",
        "Memory": 0,
        "MemorySwap": 0,
        "NetworkDisabled": false,
        "OnBuild": null,
        "OpenStdin": false,
        "PortSpecs": null,
        "StdinOnce": false,
        "Tty": false,
        "User": "",
        "Volumes": {
            "/opt/bestwebappever": {}
        },
        "WorkingDir": "/opt/bestwebappever"
    },
    "Created": "2015-01-03T23:56:15.378511619Z",
    "Driver": "devicemapper",
    "ExecDriver": "native-0.2",
    "HostConfig": {
        "Binds": [
            "/home/german/bestwebappever/:/opt/bestwebappever:rw"
        ],
        "CapAdd": null,
        "CapDrop": null,
        "ContainerIDFile": "",
        "Devices": null,
        "Dns": null,
        "DnsSearch": null,
        "ExtraHosts": null,
        "IpcMode": "",
        "Links": [
            "/bestwebappever.dev.db:/bestwebappever.dev.app/db",
            "/bestwebappever.dev.redis:/bestwebappever.dev.app/redis"
        ],
        "LxcConf": null,
        "NetworkMode": "",
        "PortBindings": {
            "80/tcp": [
                {
                    "HostIp": "",
                    "HostPort": "8887"
                }
            ]
```

```
        },
        "Privileged": false,
        "PublishAllPorts": false,
        "RestartPolicy": {
            "MaximumRetryCount": 0,
            "Name": ""
        },
        "SecurityOpt": null,
        "VolumesFrom": [
            "bestwebappever.dev.requirements-data"
        ]
    },
    "HostnamePath": "/mnt/docker/containers/44a87fdaf870281e86160e9e844b8
    "HostsPath": "/mnt/docker/containers/44a87fdaf870281e86160e9e844b8987
    "Id": "44a87fdaf870281e86160e9e844b8987cfefd771448887675fed99460de491
    "Image": "b84804fac17b61fe8f344359285186f1a63cd8c0017930897a078cd09d6
    "MountLabel": "",
    "Name": "/bestwebappever.dev.app",
    "NetworkSettings": {
        "Bridge": "docker0",
        "Gateway": "172.17.42.1",
        "IPAddress": "172.17.0.179",
        "IPPrefixLen": 16,
        "MacAddress": "02:42:ac:11:00:b3",
        "PortMapping": null,
        "Ports": {
            "80/tcp": [
                {
                    "HostIp": "0.0.0.0",
                    "HostPort": "8887"
                }
            ]
        }
    },
    "Path": "/sbin/my_init",
    "ProcessLabel": "",
    "ResolvConfPath": "/mnt/docker/containers/44a87fdaf870281e86160e9e844
    "State": {
        "Error": "",
        "ExitCode": 0,
        "FinishedAt": "0001-01-01T00:00:00Z",
        "OOMKilled": false,
        "Paused": false,
        "Pid": 21796,
        "Restarting": false,
        "Running": true,
        "StartedAt": "2015-01-03T23:56:47.537259546Z"
    },
    "Volumes": {
        "/opt/bestwebappever": "/home/german/bestwebappever",
        "requirements_data": "/mnt/docker/vfs/dir/bc14bec26ca311d5ed9f2a8
    },
```

```
        "VolumesRW": {
            "/opt/bestwebappever": true,
            "requirements_data": true
        }
    }
    ]
```

Basically the same as an image, but now some exported ports to the host are also specified, where volumes are located on the host is also stated, the container state is present towards the end, etc. As before, don't worry if it looks daunting, you will not be handling these json directly.

Tiny and small and puny step-by-step

So, step 1. Install Docker.

The Docker cmd utilities need root permissions to work. You may include your user in the docker group to avoid having to sudo everything.

Step two, lets download an image from the public registry using the following command:

```
$> docker pull ubuntu:latest
ubuntu:latest: The image you are pulling has been verified
3b363fd9d7da: Pull complete
.....<bunch of downloading-stuff output>.....
8eaa4ff06b53: Pull complete
Status: Downloaded newer image for ubuntu:latest
$>
```

There are images for pretty much everything you may need on this public registry: Ubuntu, Fedora, Postgresql, MySQL, Jenkins, Elasticsearch, Redis, etc. The Docker developers maintain several images in the public registry, but the bulk of what you can pull from it come from users that publish their own creations.

There may be come a time when you need/want a private registry (for containers for developing apps and such), you should read this first. Now, there are ways to setup your own private registry. You could also just pay for one.

Step three, list your images:

```
$> docker images
REPOSITORY  TAG     IMAGE ID      CREATED     VIRTUAL SIZE
ubuntu      latest  8eaa4ff06b53  4 days ago  192.7 MB
```

Step four, create a container from that image.

```
$> docker run --rm -ti ubuntu /bin/bash
root@4638a40c2fbb:/# ls
bin boot dev etc home lib lib64 media mnt opt proc root......
root@4638a40c2fbb:/# exit
```

Quick rundown of what you did on that last command:

- --rm: tells Docker to remove the container as soon as the process is running exits. Good for making tests and avoiding clutter
- -ti: tell Docker to allocate a pseudo tty and put me on interactive mode. This is for entering the container and is good for rapid prototyping and playing around, but for production containers you will not be turning these flags on
- ubuntu: this is the image we're basing the container on
- /bin/bash: the command to run, and since we started on interactive mode, it gives us a prompt to the container

On the run command you specify your links, volumes, ports, name of the container (Docker assings default name if you do not provide one) etc.

Now let's run a container on the background:

```
$> docker run -d ubuntu ping 8.8.8.8
31c68e9c09a0d632caae40debe13da3d6e612364198e2ef21f842762df4f987f
$>
```

The output is the assigned ID, yours will vary as it is random. Let's check out what our container is up to:

```
$> docker ps
CONTAINER ID IMAGE          COMMAND          CREATED         STATUS         P
31c68e9c09a0 ubuntu:latest "ping 8.8.8.8"  2 minutes ago   Up 2 minutes
```

There he is, his automated assigned human-readable name is loving_mcclintock. Now lets check inside the container to see what's happening:

```
$> docker exec -ti loving_mcclintock /bin/bash
root@31c68e9c09a0:/# ps -aux|grep ping
root 1 0.0 0.0 6504 636 ? Ss 20:46 0:00 ping 8.8.8.8
root@31c68e9c09a0:/# exit
```

What we just did is to execute a program inside the container, in this case the program was /bin/bash. The flags -ti serves the same purpose as in docker run, so it just placed us inside a shell in the container.

Wrap up

This about wraps it up. There is so much more to cover, but that's beyond the scope of this blogpost.

I'll however leave you with some links and further reading material that I believe is important/interesting

Docker basic structure:

- https://docs.docker.com/introduction/understanding-docker/
- http://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/

Further reading:

- Dockerfiles: These allow you to define an image using a text file, they are really important
- Did I mentioned that dockerfiles are very important?
- You really should check out dockerfiles
- docker build: you need this to build your dockerfiles
- docker push/docker pull
- docker create/docker run
- docker rm/docker rmi
- docker start/docker stop
- docker exec
- docker inspect
- docker tag
- Links
- Volumes

Interesting links:

- ANNOUNCING DOCKER MACHINE, SWARM, AND COMPOSE FOR ORCHESTRATING DISTRIBUTED APPS
- Docker at Shopify: How we built containers that power over 100,000 online shops
- Why CoreOS is a game-changer in the data center and cloud

- Docker Misconceptions
- Microservices - Not a Free Lunch!
- Feature Preview: Docker-Based Development Environments
- Docker can now run within Docker. (We actually have used this to run a Jenkins instance in Docker that builds and runs other Docker containers)
- How to compile Docker on Windows (thanks to computermedic on reddit for the link)
- A chinese version of this post (Thanks to DockerOne for taking the time to do the translation)

Useful projects and links

- Phusion Docker baseimage
- Shipyard
- DockerUI
- CoreOS
- Decking
- Docker-py
- Docker-map
- Docker-fabric

Posted on Jan. 28, 2015, 5 p.m.

Topics : virtual machine Docker DevOps virtualenv virtual Provisioning VM

16 Comments talPor Solutions - Blog 1 Login

Recommend 5 Share Sort by Best

Join the discussion…

Nan Xiao · a year ago
"Containers are desing thus, to run an application, not a machine. ", "desing" should be "designed"?
· Reply · Share ›

Sean · a year ago
"when you stop a Docker container all changes done to the initial state are lost", we know that "docker stop" doesn't remove the changed data, but "docker rm" does. Would you please clarify this? Thanks.
· Reply · Share ›

German Jaber Sean · a year ago
Thank your for pointing out that mistake, I've updated the affected sections. I'll be more careful next time. If you have anymore questions post them here.
· Reply · Share ›

郭蕾 · a year ago
We have translated your great article to Chinese:http://dockerone.com/article/1... you can add the Chinese version in the article.Thanks.
· Reply · Share ›

German Jaber 郭蕾 · a year ago
Thank you!!! Who should I credit the translation to?
· Reply · Share ›

郭蕾 German Jaber · a year ago
DockerOne
· Reply · Share ›

German Jaber 郭蕾 · a year ago
I've corrected a very big mistake in the "containers", "volumes" and "portability of images" sections (I had to change the images too).

29th May, 2015, pingback from http://kevinmccauley.com/2015/05/29/t...
25th April, 2015, pingback from http://jesuslc.com/2015/04/25/empezan...
25th April, 2015, pingback from http://jesuslc.com/2015/04/25/823/
5th March, 2015, pingback from http://cookandy.com/working-with-dock...
22nd Jan., 2015, pingback from http://hugnew.com/wordpress/docker%e7...
20th Jan., 2015, pingback from https://juliomunoz.wordpress.com/2015...
19th Jan., 2015, pingback from http://www.javaba.net/2015/01/2100.html
9th Jan., 2015, pingback from http://rpestano.wordpress.com/2015/01...
6th Jan., 2015, pingback from http://www.scoop.it/t/docker-by-docke...