2018.12

**How to use the DockerHub**

Revision History

|  |  |
| --- | --- |
| 문 서 번 호 |  |
| 문 서 명 |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 개정번호 | 개정일자 | 개정 내용 요약 | 작성자 | 확인자 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Table of Contents

[1. Docker 1](#_Toc532571158)

[1.1. Docker 개요 1](#_Toc532571159)

[1.2. Docker Install 2](#_Toc532571160)

[1.3. 이미지 이해하기 3](#_Toc532571161)

[1.4. Container 이해하기 5](#_Toc532571162)

[1.5. 버전 관리 시스템과 도커 7](#_Toc532571163)

[1.6. Dockerfile로 이미지 생성하고 어플리케이션 실행시키기 9](#_Toc532571164)

[1.7. 정리 15](#_Toc532571165)

[2. Dokku 15](#_Toc532571166)

[2.1. 소개 15](#_Toc532571167)

[2.2. 도쿠 설치하기 16](#_Toc532571168)

[2.3. 도메인 설정 17](#_Toc532571169)

[2.4. Git Server 설정 18](#_Toc532571170)

[2.5. 어플리케이션 배포 19](#_Toc532571171)

[2.6. 빌드스텝 (Buildstep) 20](#_Toc532571172)

[2.7. 도커 컨테이너와 도메인 연결 22](#_Toc532571173)

[2.8. 정리 25](#_Toc532571174)

[3. Heroku 26](#_Toc532571175)

[4. Vagrant 26](#_Toc532571176)

List of Tables

**그림 목차 항목을 찾을 수 없습니다.**

List of Figures

**그림 목차 항목을 찾을 수 없습니다.**

# Docker

Docker는 2013년에 등장한 새로운 컨테이너 기반 가상화 도구이다. Docker는 계층화된 파일 시스템(AUFS, BTRFS 등)을 사용해 가상화된 컨테이너의 변경사항을 모두 추적하고 관리한다. 으를 통해서 컨테이너의 특정 상태를 항상 보존 해두고 필요할 때 언제 어디서나(단, Docker가 설치만 되어 있다면) 이를 실행할 수 있도록 도와주는 도구이다.

## Docker 개요

클라우드와 같이 잘 짜여지고, 잘 나뉘어진 거대한 시스템에서야 그렇다 치더라도 가상 머신은 여러모로 손실이 많은 수단 중 하나이다. 가상 머신은 격리된 환경을 구축해준다는 데서 매력적이긴 하지만, 실제 배포용으로 쓰기에는 성능 면에서 매우 불리한 도구라고 할 수 있다. 당연한 이야기이지만 일단 운영체제 위에서 또 다른 운영체제를 통째로 돌린다는 것 자체가 리소스를 비효율적으로 활용할 수 밖에 없다.

이런 가운데 가상 머신의 단점은 극복하면서 장점만을 극대화하기 위한 최적화가 계속해서 이루어지고 있다. 도커 역시 이런 단점을 극복하기 위해서 나온 가상화 어플리케이션이다. Docker는 단순한 가상 머신을 넘어서 어느 플랫폼에서나 재현가능한 어플리케이션 컨테이너를 만들어주는 걸 목표로 한다. LXC(Linux 컨테이너)라는 독특한 개념에서 출발하는 Docker의 가상화는 기존에 운영체제를 통째로 가상화하는 것과는 접근 자체를 달리한다. 가상 머신이라고 하기에는 격리된 환경을 만들어주는 도구라고 하는게 맞을 지도 모른다.

예를 들어보자. 우분투에서 CentOS라는 Docker를 통해 가상 환경을 구축하는데 얼마만큼의 시간이 걸릴까? 먼저 여기서는 Ubuntu 상에서 작업을 한다고 가정한다.

|  |
| --- |
| $cat /etc/issue  Ubuntu 13.10 \n \1 |

실제로 도커는 LXC를 사용하기 때문에 특정 리눅스 배포판에서 사용할 수 있고, 윈도우나 맥에서는 사용이 불가능하다. Docker가 설치되어 있다는 가정 하에 Ubuntu에서 CentOS 가상 머신을 띄우는 데는 아래 두 명령어만 실행시키면 된다.

|  |
| --- |
| $ docker pull centos  $ docker run –rm –I –t centos:6.4 /bin/bash  bash-4.1# |

먼저 위의 pull 명령어를 통해서 centos 이미지를 다운로드 받습니다. 그리고 이 이미지에 쉘 명령어를 실행시킨다. 이걸로 끝이다. 먼저 첫번째 명령어를 실행시키는데 나의 컴퓨터에서 12초 정도가 걸렸다. 아래 명령어를 실행시키는데 0.3초가 걸리지 않았다. 쉘이 실행되면 bash 쉘로 바뀐 것을 알 수 있다. 실제로 CentOS인지 확인해 보도록 하자.

|  |
| --- |
| bash-4-1# cat /etc/issue  CentOS release 6.4 (Final)  Kernel \r on an \m |

어떻게 이런 일이 가능할까? 원리적인 부분은 이미 좋은 문서들이 있다. 한국어로 된 좋은 자료로는 Deview 2013에서 김영찬 님이 발표하신 이렇게 배포해야 할까? – Lightweight Linux Container Docker를 활용하여 어플리케이션 배포하기 세션과 xym 님이 쓴 docker the cloud를 추천한다. 하지만 분명한 건 VMWare를 사용한다고 해서 가상화 기술에 빠삭해야하는 것이 아니듯이, Docker 역시 기본적으로 툴이라는 사실을 이해 해야 한다. 제 생각에도 도코의 원리를 이해하는 것도 중요하지만, 막상 이 도구를 사용하는 동안에 컨테이너와 이미지의 차이를 이해하는 게 더 중요하고, **Dockerfile**을 만들어 자산만의 배포 프로세스를 저장하는 법을 익히는게 더 중요하다.

이 글에서는 바로 이러한 시점에서 개발 환경이자 배포 툴로써의 도커를 이해하기 위한 개념들을 소개하고 모니워키 어플리케이션을 도커로 설치하는 부분까지 다뤄보도록 하겠다.

## Docker Install

각 운영체제 별 Docker의 설치 방법은 공식 홈페이지에 잘 정리되어 있다. 제가 사용하는 Ubuntu의 경우에는 Docker에서 제공하는 스크립트 파일 하나를 통해서 원큐에 정말 쉽게 Docker를 설치할 수 있다. 가상 머신에서도 설치가 가능하니, 가능하다면 Ubuntu 환경을 사용하길 권장한다.

|  |
| --- |
| curl –s <https://get.docker.io/ubuntu/> | sudo sh  or  curl –fsSL <https://get.docker.io/ubuntu/> | sudo sh |

단, 당연히 curl이 설치 되어 있어야 한다.

설치확인은 아래와 같이 확인한다.

|  |
| --- |
| $ docker –v  Docker version 0.7.6, build bc3b2ec |

필요하다면 ufw(방화벽) 설정을 통해서 도커가 사용하는 4243/tcp 포트도 열어준다.

|  |
| --- |
| sudo ufw allow 4243/tcp |

기본적으로 docker의 대부분의 명령어를 실행 시 root 권한이 필요하다. 따라서 sudo를 사용해야하는 번거로움이 따라오는데, 이를 위해 현재 유저를 docker 그룹에 포함시켜야 한다.

|  |
| --- |
| $ sudo groupadd docker  $ sudo gpasswd –a ${USER} docker  $ sudo service docker restart |

명령어들을 실행하고 재로그인을 하게 되면 더 이상 sudo 명령어를 앞에 붙이지 않아도 docker 명령을 사용할 수 있다. 단, 이 방법을 사용할 경우엔, 공식 문서에서도 경고하고 있듯이 docker group은 root와 같은 권한 가지고 있다는 사실을 인지하고 있어야 한다.

[주의사항] https://subicura.com/2017/01/19/docker-guide-for-beginners-2.html

* 도커를 실행하기 위한 kernel 버전은 3.10x 이상입니다. Ubuntu 14.04 이상을 사용하면 큰 문제가 없고 kernel의 버전이 낮을 경우 제대로 동작을 안하거나 문제가 생길 수 있다. 가급적 최신 버전으로 업데이트 한다.
* Docker for Mac / Docker for Windows

도커를 맥이나 윈도우에 설치하려면 Docker for mac 또는 Docker for windows를 설치하면 된다. 파일을 다운받고 설치하고 재부팅하면 대부분 문제없이 완료된다. 소소한 옵션 들이 있는 특별히 건드릴 부분은 없으나 한번 살펴보고 적절하게 설정하면 되다. (windows는 공유 드라이브를 선택해준다)

참조 URL : <https://docs.docker.com/docker-for-windows/> (정리 필요)

마치 네이티브스럽게 설치된 것 같지만 도커는 리눅스 컨테이너이므로 실제로는 가상머신에 설치가 된다. 사용자는 가상머신을 사용한다는 느낌이 전혀 안드는데 그런 부분을 굉장히 신경써서 설계하였다. 예를 들면, 포트를 연결하기 위해 도커 컨테이너의 특정 포트를 가상머신에 연결하고 다시 max이나 windows의 포트와 연결해야 한다. 디렉토리를 연결한다면 디렉토리를 가상머신과 공유하고 그 디렉토리를 다시 컨테이너와 연결해야 한다. 이런 한단계 추가적으로 거쳐야하는 부분을 자연스럽게 처리해준다.

Docker for mac은 xhyve라는 maxOS에서 제공하는 가상환경을 이용하고 docker for windows는 Hyper-V 기능을 이용합니다. 따라서 OS가 최신저번이 아니면 동작하지 않을 수 있다.

* 윈도우 환경에서 Docker 설치하기

Docker는 Unbuntu 위에서만 작동한다.

참고 : <http://docs.docker.io/en/latest/installation/windows/>

그래서 윈도우 위에서 Ubuntu VM을 설치한 후, Ubuntu VM에서 Docker를 실행한다. 이를 위해서 VM을 수행하기 위한 환경을 설치한다.

O Hypervisor인 Virtual Box를 설치한다. <http://www.virtualbox.org>

O VM을 실행하기 위한 vagrant를 설치한다. <http://www.vagrantup.com>

O Docker코드를 다운받기 위해서 git 클라이언트를 설치한다. <http://git-scm.com/downloads>

여기까지 설치했으면, docker를 실행하기 위한 준비가 되었다. 다음 명령어를 수행해서, docker 코드를 git hub에서 다운로드 받은 후에, vagrant를 이용해서 Unbuntu host os를 구동한다.

|  |
| --- |
| git clone <https://github.com/dotcloud/docker.git>  cd docker  vagrant up |

Virtual Box를 확인해보면, Docker의 Host OS가 Ubuntu OS가 기동 되었음을 확인할 수 있다.



그러면, 기동된 Ubuntu OS로 SSH를 이용해서 log in을 해보자. Putty를 이용해서 127.0.0.1:2222 포트로, SSH를 통해서 로그인한다. (기본 id, password는 vagrant/vagrant이다). 이제 Docker를 이용해서, public repository에서 “busybox”라는 Ubuntu OS를 Container를 설치하고, 그 Container에서 “echo hello world” 명령어를 수행해보자.

|  |
| --- |
| sudo su  docker run busybox echo hello world |

Docker가 public repository에서 busybox image를 다운로드 받아서 설치하고, 아래와 같이 명령어를 수행했음을 확인할 수 있다.

## 이미지 이해하기

도커를 시작하면 이미지 개념을 이해할 필요가 있다. 처음에 보였던 예제를 보면 centos 이미지를 다운로드 받고, 이 이미지에 shell을 실행시킨다. 그런데 여기에 약간의 함정이 있다. Docker에서 실제로 실행되는 건 이미지가 아니다. 이미지는 추상적인 개념이다. 실행되는 건 이미지를 기반으로 생성된 컨테이너이다.

먼저 docker images 명령어로 시스템에 어떤 이미지가 있는지 확인해보자.

|  |
| --- |
| $ docker images  REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE |

docker images 명령어는 현재 시스템에서 사용가능한 이미지 일람을 보여준다. 이미지가 어디서 오는지 궁금하게 느껴질 수도 있다. 이미지는 크게 세 가지 방법을 통해서 추가할 수 있다.

하나는 처음 예제에서와 마찬가지로 docker pull <이미지 이름>을 통해서 가져오는 방법이다. 바로 이 명령어를 사용하면 docker.io의 공식 저장소에서 이미지를 다운로드 받아온다. 쉘을 활용하는 개발자라면 이런 개념이 낮 설지는 않을 것이다. 리눅스에서 apt-get이나 yum 혹은 gem이나 pip, cpan, npm 같은 명령어를 사용해 보셨다면 바로 이해할 수 있다. 이런 유틸리티를 사용해본 적이 없다고 하더라도 마찬가지 개념으로 docker 이미지 파일들을 관리하는 중앙 저장소가 있다고 이해해도 무방하다. 독특한 점은 install 이 아니 pull 명령어를 사용한다는 점이다. 이에 대해서는 개념적으로 VCS(버전 관리 시스템)을 알고 있다면 추가적인 설명이 필요없지만, 어쨌거나 단순히 다운로드라고 이해하셔도 현재 단계에서는 무방하다.

두 번째 방법은 커밋을 하는 방법이며 아래에 설명한다. 또 다른 방법은 Dockerfile을 통해서 기술된 과정을 거쳐 도커 이미지를 생성하는 방법이다. 이에 대해서는 맨 마지막에서 자세히 다룬다.

여기서는 편의상 Ubuntu 이미지를 다운로드 받아온다. 이 이미지에 대한 정보는 웹을 통해서 확인할 수 있다. 공식 저장소에 있는 이미지 정보들은 <https://index.docker.io>에서 확인할 수 있으며 우분투 이미지에 관해서는 이 페이지에서 찾을 수 있다.

|  |
| --- |
| $ docker pull Ubuntu  Pulling repository Ubuntu  04180f9bd8a6: Download complete  1418069bd8a6: Download complete  34150f9bd8a6: Download complete  b4130f9bd8a6: Download complete  … |

도커가 무언가를 열심히 다운로드 받는다. 사실 이것들 하나하나가 이미지라는 것을 이해할 필요가 있다. 우분투 하나를 설치했을 뿐인데, 무려 14개의 이미지를 다운로드 받는다. 다운로드가 전부 끝났으면 이제 다시 이미지들을 확인해 본다.

|  |
| --- |
| $ docker images  REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE  ubuntu saucy 43461fe97ba1 3 days ago 144.6 MB  <none> <none> 86f6383454b4 3 days ago 0 B  ubuntu raring 5e94ff221e91 3 days ago 133.6 MB  ubuntu quantal 3e47bae8d07a 3 days ago 127.6 MB  <none> <none> 1957a8106a4c 3 days ago 0 B  ubuntu lucid 04180f9bd8a6 3 days ago 139.6 MB  <none> <none> 7a4f87241845 3 days ago 0 B  ubuntu precise 1e548c932d40 3 days ago 125.9 MB  <none> <none> 46e4dee27895 5 weeks ago 0 B  <none> <none> b74728ce6435 5 weeks ago 0 B  <none> <none> 511136ea3c5a 7 months ago 0 B  ubuntu 12.04 8dbd9e392a96 9 months ago 128 MB  ubuntu latest 8dbd9e392a96 9 months ago 128 MB  ubuntu 12.10 b750fe79269d 10 months ago 175.3 MB  <none> <none> 27cf78414709 10 months ago 175.3 MB |

-a (-a=false: show all iamges) 플래그는 이미지를 빌드하는 과정에서 생성되는 모든 이미지를 보여준다. 자세히 보시면 저장소(Repository)가 <none>인 이미지들이 존재하는 것을 알 수 있다. –a 플러그의 설명에서 알 수 있듯이 이 이미지들은 최종적인 이미지를 생성하는 과정에서 생성되는 중간 이미지들이다. 나머지 다수의 우분투 이미지들은 TAG에서 보시듯이 다양한 버전의 이미지가 등록된 것을 알 수 있다.

## Container 이해하기

다시 한번 특정 이미지의 쉘에 접근해보겠다.

|  |
| --- |
| $ docker run –I –t ubuntu:12.04 /bin/bash  root@8bfd70fe7392:/# |

우분투 안에 우분투에 접속하는데 성공했다. 그런데 앞서 말했듯이 이미지에 접속했다는 말에는 함정이 있다. 이 말은 마치 가상머신 ssh 프로토콜을 사용해 접근한 것과 같은 착각을 일으킨다. 이제 새로운 명령어를 하나 배워보도록 하겠다. 앞서서 우리가 사용한 사용가능한 이미지들을 확인하는 명령어는 docker images 이다. 이번에 사용할 명령어는 현재 실행중인 컨테이너들을 출력하는 명령어 docker ps 이다. 별도의 쉘이다 터미널을 열고 docker ps 실행시켜 보자.

|  |
| --- |
| $ docker ps  CONTAINER ID IMAGES COMMAND CREATED STATUS PORTS NAMES  0fae5858c9c0 ubuntu:12.04 /bin/bash 8 seconds ago Up 7 seconds … |

8초전에 만들어진 컨테이너가 실행되고 있는 것을 알 수 있다. 여기서 우리가 실행한 컨테이너 정보를 알 수 있다. 우리는 ubuntu:12.04 이미지로부터 컨테이너를 생성했고, 이 컨테이너에 /bin/bash 라는 쉘을 실행시켰다. 그 외에도 맨 앞의 컨테이너 아이디는 앞으로 Docker에서 컨테이너를 조작할 때 사용하는 컬럼이기 때문에 필수적으로 알아둘 필요가 있다.

또한 위의 예제에서는 직접 명령어를 넘겨서 이미지를 컨테이너로 실행시켰지만, 보통 이미지들은 자신이 실행할 명령어들을 가지고 있다. 예를 들어 레디스, 마리아DB, 루이 온 레일즈 어플리케이션을 담고 있는 이미지라면, 각각의 어플리케이션을 실행하는 스크립트를 실행하면 된다. 컨테이너는 독립된 환경에서 실행 되지만, 컨테이너의 기본적인 역할은 이 미리 규정된 명령어를 실행하는 일이다. 이 명령어가 종료되면 컨테이너도 종료 상태(Exit)에 들어간다. 이러한 죽은 컨테이너의 목록까지 확인하려면 docker ps –a 명령어드를 사용하면 된다. 실제로 쉘을 종료하고 컨테이너 목록을 확인해 보자.

|  |
| --- |
| root@d02cd092f62d:/# exit  exit  $ docker ps –a  CONTAINE ID IMAGE COMMAND CREATED STATUS PORTS NAMES  e2af61348652 ubuntu:12.04 /bin/bash 2 minutes ago Exit 0 |

상태(Status) 컬럼에서 확인할 수 있듯이 컨테이너가 죽었다. 이번에 restart 명령어를 통해 이미지를 살려보자.

|  |
| --- |
| $ docker restart e2af613  e2af613  $ sudo docker ps  CONTAINER ID IMAGES COMMAND CREATED STATUS PORTS  e2af613f8562 ubuntu:12.04 /bin/bash 4 minutes ago Up 6 seconds |

컨테이너가 살아났다. 하지만 쉘이 실행되지는 않는다. 컨테이너 속으로 들어가기 위해서는 attach 명령어를 사용할 필요가 있다.

|  |
| --- |
| $ docker attach e2af613  root@e2af61348562:/# |

다시 docker 컨테이너 안으로 들어왔다. 여기까지 도커 컨테이너의 생명 주기를 보았다. 도커 컨테이너를 실행시키는 run 부터 실행이 종료되었을 때 다시 실행하는 restart를 배웠고 이 외에도 실행을 강제로 종료시키는 stop 명령어도 있으며, 종료된 컨테이너를 영면으로 이끌어주는 rm 명령어도 있다. 위의 예제에 보면 처음에 run 명령어와 함께 사용한 –-rm 플래그는 컨테이너가 종료 상태로 들어가면 자동으로 삭제를 해주는 옵션이다.

다시 말하지만, 이미지에는 접속한다는 개념이 없다. 실제로 실행되는 가상 머신은 항상 컨테이너이다. 분명 앞서 ‘명령어’를 실행시킨다고 했지만, 컨테이너란 격리된 환경에서 특정한 명령을 실행시켜주는 가상 머신과 같은 무언가이다. 그렇다면 쉡을 실행시키지 않았을 때 이 가상 머신을 조작할 수 있는 방법이 있을까? 원론적으로 불가능한 것은 아니다. 이 부분에 대해서는 다음에 다루도록 한다. 직접 파일 시스템을 조작할 수도 있고, 리눅스 컨테이너를 조작해 특정 컨테이너에 대해 쉘을 실행시킬 수도 있다. 하지만 좀 더 정상적인 방법은 조작이 필요한 컨테이너에 ssh 서비스를 올려서 ssh 프로토콜로 접근하는 방법이다. 실제 도커로 서비스를 운영하면 이런 부분이 절실할 날이 오겠지만, 일단 넘어가도록 하자.

## 버전 관리 시스템과 도커

다시 이미지로 돌아와서, 이미지와 컨테이너는 다르다. 그렇다면 컨테이너를 조작했을 때 이미지는 어떻게 될까?

당연히 이미지에는 아무런 변화도 생기지 않는다. 아주 세속적인 예를 들어보면 윈도우 CD로 윈도우로 설치해서 사용한다고 해서 설치한 윈도우 CD에 어떤 변화가 생기지 않는 것과 같은 이치이다. 이미지는 어디까지나 고정된 이미지이다. 도커에서는 이미지는 불변(Immutable)하는 저장 매체이다. 그런데 도커에서 이미지는 불변이지만 이 이미지 위에 무언가를 더해서 새로운 이미지를 만들어내는 일이 가능하다. 좀 더 정확히 말하면 컨테이너는 변경가능(Mutable)하다. 특정한 이미지로부터 생성된 컨테이너에 어떤 변경사항을 더하고, 이 변경된 상태를 이미지로 만들어내는 것이 가능하다.

앞서 실행한 컨테이너에 Git을 설치해 보겠다. 위에서 실행한 쉘에 다음과 같은 명령어를 입력한다.

|  |
| --- |
| root@e2af61348562:/# apt-get install –y git  ..  root@e2af61348562:/# git –version  git version 1.7.9.5 |

우분투의 패키지 관리자인 apt-get을 통해서 버전 관리 시스템인 Git을 설치했다. 여기서 도커는 마치 자신이 VCS인야, 어떤 컨테이너와 이 컨테이너의 부모 이미지 간의 파일의 변경사항을 확인할 수 있는 명령어를 제공한다. 마치 git diff 명령어로 프로젝트의 변경사항을 확인하듯이, docker diff 명령어로 부모 이미지와 여기서 파생된 컨테이너의 파일 시스템 간의 변경사항을 확인할 수 있다.

|  |
| --- |
| $ docker diff e2af61  A /.bash\_history  A /.wh..wh.plnk/310.4862541  A /.wh..wh.plnk/93.4862548  C /bin  A /bin/less  …  C /var/log/alternatives.log  C /var /log/apt  A /var/log/apt/history.log  A /var/log/apt/term.log  A /var/log/dpkg.log |

이제 기본 우분투 12.04 이미지에 Git가 설치된 새로운 이미지를 생성해보도록 하겠다. 이 작업도 VCS와 매우 비슷하다. 도커에서는 이 작업을 commit 이라고 한다.

|  |
| --- |
| $ docker commit e2af61 ubuntu:git  5ff1d6b1c5db272c3f1a88c96f78146ed48d18848f0a10e6afa066b462ff5ee  $docker images | grep git  REPOSITORY TAG IMAGES ID CREATED VIRTUAL SIZE  Ubuntu git 5ff1d6b1c5db About a minute ago 222.3 MB |

이미지는 만들기가 쉽다. 단지 커밋을 하고 뒤에 이름을 붙여주면 바로 새로운 도커 이미지가 생성된다. 이미지로부터 컨테이너를 실행시키고 이 컨테이너의 수정사항을 통해서 새로운 이미지를 만들었다. 그렇다면 정말로 이 이미지를 통해서 컨테이너를 실행시키면 (run) git가 실행될까? 직접 확인해 보자.

|  |
| --- |
| $ docker run –I –t ubuntu:git /bin/bash  root@27d4e3090750:/# git – version  git version 1.7.9.5 |

이미지가 별로 필요없다면 삭제해 버린다. 하나 알아두어야 할 것은 이미지에서 (종료상태를 포함한) 파생된 컨테이너가 하나라도 있다면 이미지는 삭제할 수 없다. 따라서 먼저 컨테이너를 종료하고, 삭제까지 해주어야 한다. docker rm은 컨테이너를 삭제하는 명령어이다. docker rmi는 이미지를 삭제하는 명령어이다. 이 두 명령어를 혼동하지 않아야 한다.

먼저 컨테이너를 지우고, 이미지를 삭제해보자.

|  |
| --- |
| $ docker ps –a  CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS  27d4e3090750 ubuntu:git /bin/bash 4 minutes ago Exit 0  $ docker rm 27d4e  27d4e  $ docker rmi ubuntu:git  Untagged: 5f~~~  Deleted : 5ff~~~ |

지금까지 도커 이미지의 생명주기를 보았따. 도커 이미지를 pull로 받아오고 commit으로 파생된 이미지를 만들고 rmi 명령어로 삭제까지 했다. 컨테이너와 이미지의 생명주기만 이해하고 나면 도커의 80%는 이해한 거나 다름 없다. 도커를 통해서 하는 일은 거의 다 이미지와 컨테이너 개념으로 커버가 가능하다. 이제 남은 일은 자신에게 필요한 이미지를 만들고 이 이미지를 통해서 컨테이너 (가상 머신)를 실행하는 일이다. 물론 이제 기본적인 개념들을 배웠으나 오픈된 중앙 저장소격에 해당하는 Docker Index에서 이미 만들어져 있는 다양한 이미지들을 활용할 수도 있다.

무엇을 하건 도커의 모토를 잊지 말아야 한다. 완성된 이미지는 도커만 설치되어 있다면 언제 어디에서는 가동 가능하다.

## Dockerfile로 이미지 생성하고 어플리케이션 실행시키기

앞서 도커 이미지를 추가하는 방법은 크게 세가지가 있다고 했다. 먼저 pull을 사용하는 방법은 이미 앞에서 다룬 바 있다. 그리고 컨테이너의 변경사항으로부터 이미지를 만드는 법에 대해서도 소개했다. 이러한 방법들은 매우 좋기는 하지만, 어딘가 2% 부족하다. 이를 보완해주는 Dockerfile 이라고 불리는 도커 이미지 생성용 배치 파일이 있다.

Dockerfile은 특정한 이미지를 출발점으로 새로운 이미지 구성에 필요한 일련의 명령어들을 저장해 놓은 파일이다. 미리 만들어둔 docker-moniwiki라는 프로젝트를 통해서 간단히 Dockerfile을 설명하고, 실제로 모나위키 어플리케이션을 가진 이미지를 생성하고 컨테이너를 통해서 실행시켜보도록 하자. 먼저 프로젝트를 clone 받는다.

|  |
| --- |
| git clone https://github.com/nacyot/docker-moniwiki.git |

이 프로젝트에는 3개의 파일이 있지만, 실질적으로 Dockerfile 하나밖에 없다고 보아도 무방하다. 전체 파일 내용은 다음과 같다.

|  |
| --- |
| FROM ubuntu:12.04  MAINTAINER Daekwon Kim <propellerheaven@gmail.com>  # Run upgrades  RUN echo "deb <http://archive.ubuntu.com/ubuntu> precise main universe" > /etc/apt/sources.list  RUN apt-get update  # Install basic packages  RUN apt-get -qq -y install git curl build-essential  # Install apache2  RUN apt-get -qq -y install apache2  ENV APACHE\_RUN\_USER www-data  ENV APACHE\_RUN\_GROUP www-data  ENV APACHE\_LOG\_DIR /var/log/apache2  RUN a2enmod rewrite  # Install php  RUN apt-get -qq -y install php5  RUN apt-get -qq -y install libapache2-mod-php5  # Install Moniwiki  RUN apt-get install rcs  RUN cd /tmp; curl -L -O http://dev.naver.com/frs/download.php/8193/moniwiki-1.2.1.tgz  RUN tar xf /tmp/moniwiki-1.2.1.tgz  RUN mv moniwiki /var/www/  RUN chown -R www-data:www-data /var/www/moniwiki  RUN chmod 777 /var/www/moniwiki/data/ /var/www/moniwiki/  RUN chmod +x /var/www/moniwiki/secure.sh  RUN ./var/www/moniwiki/secure.sh  EXPOSE 80  CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"] |

보는 바와 같이 Dockerfile은 모니위키를 설치하는 일련의 과정과 서버를 실행하는 명령어로 구성되어 있다. 각각의 부분을 간략히 살펴보자.

|  |
| --- |
| FROM Ubuntu:12.04  MAINTAINER Daekwon Kim <propellerheaven@gamil.com> |

먼저 맨 위에 정의된 FROM은 어떤 이미지로부터 새로운 이미지를 생성할 지를 지정한다. 다음으로 MAINTAINER는 이 Dockerfile을 생성-관리하는 사람을 입력한다.

|  |
| --- |
| RUN echo “deb <http://archive.ubuntu.com/ubuntu> precise main universe” > /etc/apt/sources.list  RUN apt-get update |

RUN은 직접 쉘 명령어를 실행하는 명령어이다. 이 때 바로 뒤에 명령어를 입력하게 되면 쉘을 통해서 명령어가 실행된다. 위의 두 줄은 패키지 관리자 apt-get에 저장소를 추가하고 저장소 정보를 갱신하는 명령어이다.

|  |
| --- |
| # Install basic packages  RUN apt-get –qq –y install git curl build-essential  # Install apache2  RUN apt-get –qq –y install apache2  ENV APACHE\_RUN\_USER www-data  ENV APACHE\_RUN\_GROUP www-data  ENV APACHE\_LOG\_DIR /var/log/apache2  RUN a2enmod rewrite  # Install php  RUN apt-get –qq –y install php5  RUN apt-get –qq –y install libapache2-mod-php5 |

다음 부분에서는 모니위키 설치에 필요한 패키지들과 apache2 서버, php 프로그램을 설치 및 설정해준다. 아파치 설치 과정에서 나오는 ENV를 통해 환경 변수를 지정할 수 있다.

|  |
| --- |
| # Install Moniwiki  RN apt-get install rcs  RUN cd /tmp; curl –L –O <http://dev.naver.com/frs/download.php/8193/moniwiki-1.2.1.tgz>  RUN tar xf /tmp/moniwiki-1.2.1.tgz  RUN mv moniwiki /var/www/  RUN chown –R www-data:www-data /var/www/moniwiki  RUN chmod 777 /var/www/moniwiki/data/ /var/www/moniwiki/  RUN chmod +x /var/www/moniwiki/secure.sh  RUN ./var/www/moniwiki/secure.sh |

이제 실제로 모니위키를 설치합니다. 여기서는 모니위키를 curl을 통해서 다운로드 받고, 압축을 풀고 모니위키 실행에 필요한 권한 관련 설정을 해준다.

|  |
| --- |
| EXPOSE 80  CMD [“/usr/sbin/apache2”, “-D”, “FOREGROUND”] |

이제 마지막이다. EXPOSE는 가상 머신에 오픈할 포트를 지정해준다. 마지막줄의 CMD는 컨테이너에서 실행될 명령어를 지정해준다. 이 글의 앞선 예에서 docker run을 통해서 /bin/bash를 실행했지만, 여기서는 아파치 서버를 FOREGROUND에 실행시킨다.

이제 직접 이 Dockerfile을 빌드할 차례이다.

|  |
| --- |
| **$ docker build –t nacyot/moniwiki .**  Uploading context 71.68 kB  Uploading context  Step 1 : FROM ubuntu:12.04  ---> 8dbd9e392a96  Step 2 : MAINTAINER Daekwon Kim <propellerheaven@gmail.com>  ---> Running in a2af31ca9d62  ---> c42835b9308b  Step 3 : RUN echo "deb <http://archive.ubuntu.com/ubuntu> precise main universe" > /etc/apt/sources.list  ---> Running in d305ce1fea04  ---> f4cb16c39b0e  Step 4 : RUN apt-get update  ...  ---> c63d093aacfb  Step 21 : EXPOSE 80  ---> Running in cee6a6048c83  ---> 7436a638e52c  Step 22 : CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]  ---> Running in 2f251c355290  ---> 0a148bb4de2f  Successfully built 0a148bb4de2f |

Docker build 명령어는 –t 플래그를 사용해 이름과 태그를 지정할 수 있다. 그리고 마지막에 . 은 빌드 대상 디렉토리를 가리킨다. 이 때 알아두면 좋은게 하나 있다. 위에서 정의한 RUN 명령 하나 하나는 명령 하나마다 이미지가 된다. 기본적으로 이 빌드를 통해서 생성되는 최종 이미지는 nacyot/moniwi가 된다. 다만, docker images –a를 통해서 살펴보면 이름없는 도커 이미지들이 다수 생성되는 것을 알 수 있다.

|  |
| --- |
| $ docker images -a  REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE  <none> <none> c63d093aacfb 4 minutes ago 670.4 MB  <none> <none> 7436a638e52c 4 minutes ago 670.4 MB  nacyot/moniwiki latest 0a148bb4de2f 4 minutes ago 670.4 MB  <none> <none> 49825025193f 4 minutes ago 670.4 MB  <none> <none> 5b374f859553 4 minutes ago 670.4 MB  <none> <none> c8afe13ab509 4 minutes ago 670.4 MB  <none> <none> bb65aa482123 4 minutes ago 663.1 MB  <none> <none> fa7f2059c9ba 4 minutes ago 655.8 MB  <none> <none> 1a18589e4d9a 4 minutes ago 648.5 MB  <none> <none> fdd759b53314 4 minutes ago 646.1 MB  <none> <none> 786c8ad1df43 4 minutes ago 610.5 MB  <none> <none> 14e8b032683a 4 minutes ago 610.5 MB  <none> <none> 6be08754c2ae 4 minutes ago 547.9 MB  <none> <none> 67e17f5a39f1 4 minutes ago 547.9 MB  <none> <none> 55ec9487188b 4 minutes ago 547.9 MB  <none> <none> 88a61604a1d0 4 minutes ago 547.9 MB  <none> <none> 7edabbb84352 4 minutes ago 547.9 MB  <none> <none> 78e9afc826cd 4 minutes ago 503 MB  <none> <none> 8066398c160f 5 minutes ago 272.5 MB  <none> <none> f4cb16c39b0e 6 minutes ago 128 MB  <none> <none> c42835b9308b 6 minutes ago 128 MB |

이미지 아이디가 빌드 과정에서는 출력되는 아이디와 같은 것을 알 수 있다. 필요한 경우 중간 이미지에 접근하거나 직접 중간 이미지로부터 다른 이미지를 생성하는 것도 가능하다. 정말 좋은 소식은 도커는 이러한 빌드 순서를 기억하고 각 이미지를 보존하기 때문에 같은 빌드 과정에 대해서는 캐시를 사용해 매우 빠르게 빌드가 가능하다느 점이다. 실제로 Docker 파일을 만드는 과정에서는 많은 시행 착오를 겪게 되는데, 중간에 빌드가 실패하더라도 성공했던 명령어까지는 거의 시간 소모 없이 빠르게 진행되도록 설게되어 있다.

빌드 자체는 꽤나 번거로운 일이다. 도커의 가상화가 굉장히 빠르다고 해도 어플리케이션 실행환경을 구축하는 일은 상당히 시간도 많이 걸린다. 더욱이 빌드 자체는 완벽히 ‘지현 가능’ 하지 않다. 하지만 이렇게 Dockfile을 통해서 배치화를 시켜두면 Dockfile이라는 정말 작은 파일 하나로 어플리케이션 배포 환경을 구축할 수 있다는 장점이 있으며, 또한 쉽게 유연하게 사용할 수 있다. 아주 흥미로운 이야기를 하다 하자면 Docker 생태계에 있는 오픈소스 어플리케이션들은 아예 Dockerfile을 프로젝트에 포함하고 있다. 대표적으로 도커 모니터링 툴인 Shipyard가 있다. 여기서 제공하는 Dockerfile을 빌드해서 이미지를 만들고, 이 이미지로 컨테이너를 가동하면 바로 shipyard 어플리케이션을 사용할 수 있다.

Dockerfile은 단순히 어플리케이션 설치를 스크립트로 만들어주는 게 아니라, 배포환경 구축까지 한꺼번에 해주는 역할을 한다. 오래전 제로보드나 테터툴즈 한번 설치해보겠다고 <?phpinfo()?> 찍어가면 php랑 apache랑 잘 붙었나 안 붙었나 확인해보고 안 되면 이유도 못 찾아 혼자 서러워했던 적인 있는 분이라면 이해하리라 믿는다. 다른 예를 들면 설치가 까다로운 걸로 악명높은 오픈소스 웹 어플리케이션 중에 Gitlab이라는 어플리케이션이 있다. Gitlab-docker에서 제공하는 Dockerfile 하나면 이제 Gitlab도 두렵지 않다. 그저 Build 하고 Run 하면 Gitlab이 뜬다.

이야기가 길어졌지만, 다시 모니위키로 돌아와서 빌드가 끝났으니 실행을 해보자.

|  |
| --- |
| $ docker run –d –p 9999:80 nacyot/moniwiki  746443ad118afdb3f254eedaeeada5abc2b125c7263bc5e67c2964b570166187 |

다시 docker run이다. 이번에는 –d와 –p 플래그를 사용했다. 앞서서 자세히 설명하지 않았지만, -d 플래그는 -i플래그의 반대 역할을 하는 플래그로, 컨테이너를 백그라운드에서 실행시켜준다. –p는 포트포워딩을 지정하는 플래그이다. :을 경계로 앞에는 외부 포트, 뒤로는 내부 포트이다. 참고로 컨테이너 안에서 아파치가 80 포트로 실행된다. 따라서 여기서는 localhost에 9999로 들어오는 연결을 여기서 실행한 컨테이너의 80포트로 보내도록 한다.

이제 접근해서 실행되는지 확인한다.

http://127.0.0.1:9999/moniwiki/

## 정리

도커의 활용가능성은 무궁무진하다. 당연히 실제 배포에도 사용할 수 있고, 유연하고 날렵한 격리된 환경을 활용해 실험적인 개발을 진행할 수도 있다. Vagrant 보다도 훨씬 빠른 가상 환경을 활용할 수 있으며 접근 방법은 다르지만 서버환경을 자동적으로 구성할 수도 있다. 미리 만든 이미지를 자신의 저장소(registry)에 등록해 여러대의 머신에 컨테이너들을 자동적으로 배포할 수도 있고, 오픈소스 프로젝트에서 Dockerfile을 제공해 설치를 색다른 방법으로 지원할 수도 있다. **Dokku**를 사용하면 정말 쉽게 mini-heroku와 같은 PasS 플랫폼을 구축해 불 수도 있다.

# Dokku

## 소개

Jeff Lindsay가 개발한 도쿠는 Docker를 활용해 100줄 남짓한 쉘 스크립트 본체와 도쿠를 둘러싼 몇 가지 모듈들을 활용해 미니 히로쿠(Heroku)와 같은 PaaS 환경을 구축할 수 있도록 해준다. 도쿠 스스로가 미니 히로쿠라고 표현하고 있지만, 사용해보면 정말 히루쿠와 비슷하다. 실제로 내부적으로 도쿠는 히루쿠에서 공개한 각 언어별 빌드팩을 그대로 사용하고 있다. 히로쿠를 통해서 서비스를 배포해보았거나 Rails Tutorial을 따라하면서 배포를 연습해 보았다면 반갑게 느껴질 것이다. 실제로 활용을 하건 안하건, 이런 훌륭한 시스템을 직접 구축해 볼 수 있다.

히로쿠나 PaaS에 대해서 잘 모르는 분들은 이런 개념이 낯설게 느껴질 수도 있지만 히로쿠나 PaaS 역시 클라우드 생태계의 한 축을 구성하고 있는 단어들이다. 클라우드하면 가상화이다. 클라우드의 본질적인 가치는 하드웨어 추상화에 있다. 하드웨어를 추상화하기 위해서는 다양한 기술과 관리를 위한 기술들이 필요하다. 이렇게 추상화된 하드웨어를 이용하는 입장에서는 ‘더 이상’ 하드웨어라는 물리적인 영역을 신경쓰지 않아도 된다. 개발과는 거리가 있지만, 이런 종류의 추상화를 가장 잘 보여주는게 구글 검색창이다. 그 안에서 어떠한 로직과 방법으로 데이터를 분석하는지는 전혀 알 필요 없이, 우리는 구글의 엄청나게 거대한 시스템을 사용할 수 있다. 그렇게 보자면 서비스는 가장 높은 단계의 추상화라고 할 수 있다.

IaaS나 Paas는 이보다 좀 더 낮은 단계의 추상화를 제공한다. 서비스를 이용하는 사람은 서버가 어떤 하드웨어로 구성되어 있고, 램은 몇 기가고 CPU는 뭘 쓰고 있고 신경 쓸 필요가 없을 뿐더러, 어떤 운영체제 위에서 돌아가는지, 어플리케이션이 어떤 언어를 사용하고, 어떤 구조로 만들어져 있고 데이터베이스가 어떻게 분산되어 있는지도 신경쓸 필요가 없다. 앞의 문장을 어디까지 신경써야 하는가를 가지고 단계를 나눌 수 있다. 앞서 이야기했듯이 서비스는 위에서 아무것도 신경쓰지 않는다. 여기서 한단계 내려가면 사용하는 언어와 어플리케이션만 신경써야 하는 단계가 있다. 이 아래에 있는 것들은 서비스로 지원한다. 여기가 PaaS가 위치한 곳이다. 여기서 더 아래로 내려가면 CPU의 성능과 램의 용량을 따지고 어떤 운영체제를 사용하고 각각의 시스템들을 어떻게 연동시켜야하는지 하는 부분이 나온다. 심지어 시스템의 크기까지 제어할 수 있다. 여기가 바로 IaaS가 위치한 곳이다. 더 아래로 내려가면 네, 실제 서버 구성부터 이것저것 모든 것을 다 해야 한다. 해보신 분들은 아시겠지만 이런 일은 한 마디로 말해서 고역이다. 운영체제도 하드웨어도 같지 않은 수십대의 시스템을 효과적으로 관리한다는 건 거의 불가능에 가깝다. 거기다가 고장까지 고려하면 더더욱 쉽지 않다.

클라우드의 본질적인 가치는 어플리케이션을 개발하는데 있어서 우리가 어디까지 신경서야 하는지 그 위치를 선택가능한 옵션으로 떨어뜨려줬다는데 있다. 그건 내가 해줄게 하는 것이다. AWS가 나오기 전에 서비스를 만드는 사람들에게 그런 종류의 선택권은 존재하지 않았다. 구체적인 서비스를 예를 들면 AmazonWS EC2가 IaaS라면 히로쿠나 구글 앱 엔진은 대표적인 PaaS 서비스이다. 더욱 재미있는 건 Heroku가 다름 아닌 AWS 위에서 돌아가고 있다는 사살이다.

특히 히로쿠는 루비 온 레일즈를 시작하는 현재는 자바, Node.js, 파이썬 어플리케이션까지 지원하고 있다. 히로쿠를 사용하면 회사나 개발자는 딱 어플리케이션만 만들어내면 된다. 거짓말 조금 보태어 어디서 어떻게 관리하고 배포할지 일절 고민할 필요가 없다. 어플리케이션을 만들고 Heroku에 있는 Git 서버에 푸쉬하기만 하면 어플리케이션을 자동으로 빌드하고 연결한 도메인으로 배포해준다. 이건 그저 감탄만 나올 정도로 너무 훌륭하다. 같은 크기에 있어서 IaaS 서비스보다 PaaS가 일반적으로 더 비싼 편이기 때문에 가격 정책이나 여러가지를 고려해서 맞는 서비스를 이용한게 현명한 일이다.

도쿠는 바로 Docker라는 가상화 기술을 활용해 이러한 PaaS를 직접 구축하게 도와주는 절대로 크지 않은 어플리케이션이다. 절대로 많은 것을 요구하지도 않고 어렵지도 않다.

참조 URL

https://blog.nacyot.com/articles/2014-01-30-deploying-application-with-dokku/

## 도쿠 설치하기

도큐를 사용하기 위해서는 먼저 설치를 해야 한다. 도큐는 현재 우분투 12나 13.04를 사용할 것을 권장하고 있다. 우분투 13.10에서도 사용한 바 정상적으로 사용이 가능하다. 몇가지 이슈가 있어서 13.10에서 사용하는 것은 권장하고 않고 있는 상황이다. 먼저 Docker는 설치가 되어 있다고 가정한다.

하나 주의해야 하는 점은 도큐 서버는 어디까지나 배포 서버이다. 불가능한 건 아니지만, 가능하면 개발 서버와 분리된 환경이나 서버가 따로 없다면 가상 머신 상에서 테스트 해보기를 권장한다. 또한 도메인 설정이 되어 있어야 정상적으로 사용가능하다. (없어도 테스트나 사용은 가능하지만 권장하지 않는다). 자세한 내용은 각 설정에 대해서 이야기하는 부분에서 자세히 다룬다.

바로 도큐를 설치해보자. 도큐 역시 원큐에 설치가능한 쉘 스크립트를 제공하고 있다.

|  |
| --- |
| $ wget –q0- <https://raw.github.com/progrium/dokku/v0.2.1/bootstrap.sh> | sudo DOKKU\_TAG=v0.2.1 bash |

현재 최신 안정 버전인 0.2.1을 설치한다. 도쿠 공식 문서에 따르면 우분투 12.04에서는 python-software-properties 패키지가 필요할 지도 모른다고 이야기하고 있다. 아래와 같은 메시지가 뜨면서 설치가 실패하는 경우가 있다.

|  |
| --- |
| /var/lib/dokku/plugins/nginx-vhosts/install: line 5: add-apt-repository: command not found  make: \*\*\* [plugins] Error 127 |

이때는 apt-get install **python-software-propertis**를 실행시켜 필요한 패키지를 설치해준다.

설치가 끝났으면 제대로 설치가 되었는지 확인한다.

|  |
| --- |
| …  Almost done! For next steps on configuration  $ dokku version  V0.2.1 |

위에 처럼 버전이 올바르게 출력되면 정상적으로 설치가 된 것이다.

## 도메인 설정

도커는 기본적으로 배포환경이기 때문에 도메인을 지정해줄 필요가 있다. 도커를 설치하면 자동적으로 dokku 유저가 시스템에 추가되고, /home/dokku 디렉토리에서 현재 배포중인 앱과 설정에 관련된 파일들을 확인할 수 있다. 먼저 여기에 VHOST라는 파일을 추가하고 자신이 사용할 도메인을 지정해준다. 예를 들어서 **<APP\_NAME>.nacyot.com**을 도메인으로 사용하고자 한다면 **/home/dokku/VHOST** 파일에 다음과 같이 추가해준다. (파일이 없으면 루터나 dokku 권한으로 직접 생성해준다.)

|  |
| --- |
| nacyot.com |

뒤에서 좀 더 이야기하겠지만, 여기서 도메인을 설정하게 되면 따로 아무런 설정을 하지 않더라도 앱 배포시 앱의 이름을 활용해 서브도메인으로 곧장 배포가 된다. 당연하지만 이를 위해서는 일반적으로 도메인을 사용하는 것과 마찬가지로 DNS 서버에 A Record를 추가해한다. 예를 들어 nacyot.com 아래의 모든 도메인을 도큐로 배포하는 어플리케이션의 서브 도메인을 사용하도록 하러면 \*.nacyot.com을 <Dokku\_IP\_ADDRESS> 도쿠 서버의 아이피 주소를 가리키도록 설정하면 된다.

이 때 도메인이 없는 경우에는 VHOST 파일을 삭제해주시기 바란다. 이렇게 하면 랜덤하게 생성된 (사실은 49000번대에서 순차적으로 생성되는) 포트에 앱이 자동으로 연결된다. 하지만 이 포트는 앱을 생성할 때마다 유동적으로 변하므로 미리 DNS 서버를 설정해놓고 도메인을 사용하는게 좋다.

## Git Server 설정

앞에서 히로쿠를 간단히 소개하면서 히로쿠의 Git 서버에 푸쉬만 하면 어플리케이션을 자동으로 빌드하고 배포해준다는 이야기를 했다. 도쿠도 마찬가지 방식을 사용한다. 따라서 Push를 보내는 쪽이 적절한 권한을 가지고 있는지 인증 절차를 거칠 필요가 있다. 도쿠 README 파일에서는 아래의 명령어를 실행시키라고 이야기한다.

|  |
| --- |
| $ cat ~/.ssh/id\_rsa.pub | ssh progriumapp.com “sudo sshcommand acl-add dokku progrium” |

이 명령어를 실행시키기 전에 약간의 이해야 필요하다. 도쿠는 어디까지나 배포서버라는 것을 전제로 하고 있다. 즉, 도쿠는 어디까지나 개발서버나 머나먼 어딘가에서 어플리케이션을 실제로 배포해주는 배포서버이다. 그리고 어플리케이션을 개발하고 배포하려는 사람(머신)은 다른 곳에 있다. 이 명령어는 도쿠 서버에서 사용하는 명령어가 아니라 개발 서버에서 ssh를 통해 Git 저장소를 접근할 수 있도록 권한 설정을 해주는 명령어이다. 즉 이 명령어는 개발 서버에 있는 ssh 공개키를 도쿠 서버에 넘겨서 접근 권한을 준다. 위 명령어를 좀 더 정확히 표현하자면 아래와 같다.

|  |
| --- |
| $ cat ~/.ssh/id\_rsa.pub | ssh <아이디>@<서버\_아이피\_주소\_혹은\_도메인> “ sudo sshcommand acl-add dokku progrium” |

꺽쇠 안의 부분을 도쿠 서버에 맞게 적당히 바꾸고 개발 서버에서 이 명령어를 실행해야 한다. 또한 이 때 root 계정이나 root에 준하는 권한을 가지 권한을 사용한다면 바로 실행이 되겠지만 sudo를 실행시 비밀번호를 묻는 경우 이 명령어가 정상적으로 처리되지 않는다. 정 실행이 어려울 경우엔 도쿠 서버에 직접 공개키를 echo 해서 실행 해도 무방하다. 실패하는 경우 failed라는 메시지가 뜨고 권한 등록에 성공하면 콜론으로 구분된 문자열이 출력된다.

제대로 등록되었는지 확인하기 위해 개발 서버에서 테스트로 접근해보자.

|  |
| --- |
| $ ssh dokku@<도큐\_서버\_아이피\_혹은\_도메인> |

이 때 연결이 끊어졌다는 메시지가 나오면 정상이고, dokku 계정의 패스워드를 물어본다면 제대로 등록이 안 된 상황이다. ssh키나 등록 부문에 문제가 없는지 확인해 볼 필요가 있다. 특히 얼마 전까지만 해도 계정을 git을 사용했었고, ssh 등록 명령어도 gitreceive upload-key dokku 를 사용했었기 때문에 예전 문서를 참조했는지도 확인이 필요하다.

## 어플리케이션 배포

도쿠의 README 문서를 따라 샘플 어플리케이션을 배포해보도록 한다. 샘플 어플리케이션으로는 히로쿠에서 제공하는 node-js-sample을 사용한다. 참고로 이 작업들은 개발 머신에서 진행한다.

|  |
| --- |
| $ git clone https://github.com/heroku/node-js-sample.git |

먼저 git 저장소에서 어플리케이션을 클론한다. 이제 node-js-sample 폴더로 이동해 원격 저장소로 도쿠 서버를 등록하고 푸쉬한다.

|  |
| --- |
| $ cd node-js-sample  $ git remote add dokku dokku@도쿠\_서버\_주소:앱\_이름  $ git push dokku master  Counting objects: 313, done.  Delta compression using up to 4 threads.  Compressing objects: 100% (270/270), done.  Writing objects: 100% (313/313), 200.76 KiB | 0 bytes/s, done.  Total 313 (delta 15), reused 313 (delta 15)  --🡪 Building node-sample … |

[도쿠 서버 주소]는 실제 도쿠 서버의 주소를 아이피나 도메인으로 지정하면 되고, 앱 이름은 원하는 이름을 사용하면 됩니다. 이 때 앱 이름이 곧 서브 도메인 이름이 된다. 예를 들어 node-sample 으로 이름을 지정하면 도메인은 VHOST에서 설정한 메인 도메인과 연결되어 node-sample.nacyot.com이 된다. 자동으로 빌드가 진행된다.

|  |
| --- |
| …  ----🡪 Building runtime environment  ----🡪 Discovering process types  Procfile declares types -> web  ----🡪 Releasing node-sample …  ----🡪 Deploying node-sample …  ----🡪 Cleaning up …  ===> Application deployed:  <http://node-sample.nacyot.com>  To [dokku@<.....>:node-sample](mailto:dokku@%3c.....%3e:node-sample)  \* [new branch] master -> master  $ |

1분여 남짓 기다리면, 빌드가 끝난다. 배포도 끝난다. DNS 설정이 이미 적용 되어 있다면 배포 즉시 node-sample.nacyot.com 도메인으로 접속해 어플리케이션을 확인해 볼 수 있다. 정상적으로 배포가 되었다면 Hello, world 라는 문자열이 출력될 것이다.

빌드도 알아서 해주고, 서버 설정도 알아서 해주고 훌륭한 PaaS 시스템이라고 할 수 있다. 여러분은 그저 어플리케이션만 만들고 Push 해주면 된다.

## 빌드스텝 (Buildstep)

히로쿠를 써보신 분들이라면 빌드 과정에서 나오는 메시지들이 히로쿠에서 어플리케이션을 푸시할 때 나오는 메시지들과 매우 닮아 있다. 실제로 도쿠 어플리케이션을 빌드하는 방법은 히로쿠에서 제공한느 빌드팩이라는 오픈 소스를 통해서 이루어진다. 사실 도쿠가 어떻게 도커를 사용하고 빌드팩을 사용하는지 이렇게 봐서는 이해가 어려울지 모른다.

도쿠는 분명 도커 기반으로 작동한다. 하지만 이번 글에서는 아직 도커에 관련된 부분이 아직 나오지 않았다. 이제 도커가 나설 차례이다. 도쿠를 설치하면 도커에서는 어떤일이 일어날까? 도쿠를 설치한 상태에서 도커의 이미지를 확인해보자. 도쿠 서버에서 아래 명령어를 실행시켜본다.

|  |
| --- |
| $ docker images  REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE  progrium/buildstep latest 107a86e32874 6 hours ago 1.42 GB  ubuntu quantal 42130da57f7 4 days ago 127.9 MB  centos 6.4 539c0211cd76 10 months ago 300.6 MB |

Progrium/buildstep 이라는 이미지가 하나 추가된 것을 알 수 있다. 이 예제의 이미지는 커스터마이징이 되어 있어서 아마 출력되는 용량은 더 작을 것이다. 이 빌드스텝이라는 이미지가 도쿠 시스템의 하이라이트이다. 빌드스텝은 어플리케이션이 빌드가 될 베이스가 되는 이미지이다. 즉 위의 node-js-sample 에서 푸시를 하게 되면 실제로 일어나는 일은 이 buildstep 이라는 이미지를 바탕으로 푸시된 어플리케이션 빌드해서 새로운 이미지를 생성하는 일이다.

따라서 실제로 어플리케이션을 푸쉬한 다음에 이미지들을 확인해보면 다음과 같다.

|  |
| --- |
| $ docker images  REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE  app/node-sample latest 4d88b88125cc 11 minutes ago 1.582 GB  progrium/buildstep latest 107a86e32874 6 hours ago 1.42 GB  ubuntu quantal 42130da57f7 4 days ago 127.9 MB  centos 6.4 539c0211cd76 10 months ago 300.6 MB |

보시면 아시겠지만 app/node-sample 이라는 새로운 이미지가 추가된 것을 확인할 수 있다. 추정컨테 마치 progrium/buildstep 이미지에서 뭔가가 추가된 것 같이 용량도 별반 다르지 않다. 그런데 이것만으로는 정말론 app/node-sample의 부모 이미지가 progrium/buildstep 인지 확인할 수 없다. 따라서 history 명령을 통해서 app/node-sample의 이력을 조회해보자.

|  |
| --- |
| $ docker history app/node-sample  IMAGE CREATED CREATED BY SIZE  4d88b88215cc 18 minutes ago /build/builder 162.4 MB  30848b99f896 19 minutes ago /bin/bash –c mkdir –p /app && tar –xC /app 1.976KB  107a86e32874 6 hours ago /bin/bash 517.1 MB  B1c42dee3ff 29 hours ago 875.5 MB |

아래에서부터 이력을 쫓아보면 아시겠지만 107a86e32874(progrium/buildstep)에서 4d88b88215cc(app/node-sample)이 파생된 것을 확인할 수 있다. 네, 이로써 buildstep이 도쿠를 통한 어플리케이션에 있어 모든 근원이라는 것을 확인했다. 이쯤되면 buildstep이 도대체 뭔가 하는 궁금증이 생기게 마련이다. 그렇다면 빌드스텝 안을 까보도록 한다.

|  |
| --- |
| $ docker run –rm –I –t progrium/buildstep:latest /bin/bash  root@3a6b0a823acb:/# ls  boot cache etc home lib64 mnt proc run selinux tmp var  bin build dev exec lib media opt root sbin srv sys usr  root@3a6b0a823acb:/# cd build  root@3a6b0a823acb:/build# ls  builder buildpacks buildpacks.txt packages.txt prepare  root@3a6b0a823acb:/build# |

빌드스탭 이미지에 쉘을 띄웁니다. 안에는 일반적인 리눅스 시스템에서는 보기 힘든 build라는 디렉토리가 있습니다. 들어가 보면 build와 관련된 파일들이 나온다. Buildpakcs 디렉토리에 들어가보면 다음과 같은 디렉토리들이 들어 있다.

|  |
| --- |
| root@3a6b0a823acb:/build/buildpacks# ls -al  total 64  drwxr-xr-x 16 root root 4096 Jan 29 07:16 .  drwxr-xr-x 3 root root 4096 Jan 29 06:18 ..  drwxr-xr-x 5 root root 4096 Dec 3 18:35 buildpack-nginx  drwxr-xr-x 5 root root 4096 Dec 3 18:35 heroku-buildpack-clojure  drwxr-xr-x 5 root root 4096 Dec 3 18:35 heroku-buildpack-dart  drwxr-xr-x 8 root root 4096 Dec 3 18:35 heroku-buildpack-go  drwxr-xr-x 6 root root 4096 Dec 3 18:35 heroku-buildpack-java  drwxr-xr-x 7 root root 4096 Dec 3 18:35 heroku-buildpack-meteorite  drwxr-xr-x 7 root root 4096 Dec 3 18:35 heroku-buildpack-nodejs  drwxr-xr-x 4 root root 4096 Dec 3 18:35 heroku-buildpack-perl  drwxr-xr-x 7 root root 4096 Dec 3 18:35 heroku-buildpack-php  drwxr-xr-x 6 root root 4096 Dec 3 18:35 heroku-buildpack-play  drwxr-xr-x 6 root root 4096 Dec 3 18:35 heroku-buildpack-python  drwxr-xr-x 9 root root 4096 Jan 29 07:16 heroku-buildpack-ruby  drwxr-xr-x 6 root root 4096 Dec 3 18:35 heroku-buildpack-scala  drwxr-xr-x 5 root root 4096 Dec 3 18:35 heroku-buildpack-**static**-apache |

Heroku-buildpack 관련 파일이 많이 있다. 실제 빌드스텝이 어떻게 구성되어 있는지는 빌드스텝 저장소를 참조해보면 도움이 될 것이라 판단된다. 여기는 어플리케이션 빌드가 가능하도록 각 언어별 heroku-buildpack 들을 clone 해 두었다고 보면 된다.

빌드스탭 이미지 역시 커스터마이징이 가능하다. 도쿠에서 빌드스텝이미지를 기반으로 빌드를 할 때는 단순히 progrium/buildstep:latest 라는 이름을 가진 이미지를 기반으로 빌드를 한다. 물론 기본적으로 내부에 위에서 간략히 보여드린 빌드팩들이 포함되어 있어야 한다. 어쨌거나 원한다면 이미지를 수정하거나 필요한 패키지들을 더 집어 넣을 수도 있다. 이미지 이름만 같으면 된다. 필자의 경우 [heroku-buildpack-ruby][buildpack-ruby]를 간단히 커스터마이징해서 쓰면 있고 그래서 이미지 안의 heroku-buildpack-ruby만 제 저장소에 있는 걸 클론해서 쓰고 있다. 위의 ls 출력 결과를 보면 ruby만 수정 날짜가 다른 걸 알 수 있다.

## 도커 컨테이너와 도메인 연결

즉 도쿠 서버에 푸시를 해서 실제로 일어나는 일이란, 이 빌드스텝이라는 이미지를 기반으로, /build 폴더에 있는 히로쿠 빌드팩을 통해 어플리케이션을 빌드하고, 도메인을 연결시켜준다. 이것은 어떻게 일어나는 것일까? 이제 buildstep 이미지를 빠져나와 dokku 홈으로 이동한다.

|  |
| --- |
| $ cd /home/dokku  $ ls -l  total 24  -rw-r--r-- 1 dokku root 13 Jan 29 11:31 dokkurc.log  -rw-r--r-- 1 dokku root 58 Jan 28 17:38 HOSTNAME  drwxr-xr-x 8 dokku dokku 4096 Jan 29 22:20 node-sample  -rw-r--r-- 1 root root 19 Jan 29 17:08 VHOST  $ cd node-sample  $ ls -l  total 52  drwxr-xr-x 2 dokku dokku 4096 Jan 29 22:19 branches  drwxr-xr-x 2 dokku dokku 4096 Jan 29 22:19 cache  -rw-r--r-- 1 dokku dokku 66 Jan 29 22:19 config  -rw-r--r-- 1 dokku dokku 65 Jan 29 22:20 CONTAINER  -rw-r--r-- 1 dokku dokku 73 Jan 29 22:19 description  -rw-r--r-- 1 dokku dokku 23 Jan 29 22:19 HEAD  drwxr-xr-x 2 dokku dokku 4096 Jan 29 22:19 hooks  drwxr-xr-x 2 dokku dokku 4096 Jan 29 22:19 info  -rw-r--r-- 1 dokku dokku 530 Jan 29 22:20 nginx.conf  drwxr-xr-x 4 dokku dokku 4096 Jan 29 22:19 objects  -rw-r--r-- 1 dokku dokku 6 Jan 29 22:20 PORT  drwxr-xr-x 4 dokku dokku 4096 Jan 29 22:19 refs  -rw-r--r-- 1 dokku dokku 38 Jan 29 22:20 URL |

앞에서도 어렴풋이 이야기 했지만, dokku 계정은 사실 git 원격 저장소를 관리하기 위한 계정이다. 그리고 그 아래에 있는 디렉토리들은 실제로 하나하나 git 저장소가 된다. node-samle 역시 git 저장소이자 도커 이미지를 생성하기 위한 설정들이 저장 되어 있는 디렉토리이다. 마술은 git의 hook 를 사용해서 일어난다. Git은 특정한 이벤트가 발생할 때 특정한 명령어를 실행시킬 수 있는 hook라는 특수한 기능을 지원한다. Hooks 디렉토리에 들어가보면 pre-receive 후크가 활성화되어 있는 것을 알 수 있다.

|  |
| --- |
| $ cd hooks  $ cat pre-receive  *#!/usr/bin/env bash*  set -e; set -o pipefail;  cat | DOKKU\_ROOT="/home/dokku" dokku git-hook node-sample |

내용은 매우 간단하다. Node-sample 어플리케이션에 대해 dokku 명령어에 git-hook 라는 플러그인 명령을 실행시키고 있다. 이렇게 보면 이 명령어의 정체가 잘 안보이니 git-hook도 실제도 살펴 보자. git-hook 플로그인의 내용은 아래 /var/lib/dokku/plugins/git/commands 에서 확인할 수 있다.

|  |
| --- |
| *#!/usr/bin/env bash*  set -eo pipefail; [[ $DOKKU\_TRACE ]] && set -x  **case** "$1" in  git-hook)  APP=$2  **while** read oldrev newrev refname  **do**  *# Only run this script for the master branch. You can remove this*  *# if block if you wish to run it for others as well.*  **if** [[ $refname = "refs/heads/master" ]] ; **then**  git archive $newrev | dokku receive $APP | sed -u "s/^/"$'\e'"/"  **fi**  **done**  ;;  ... |

여기서는 git-hook 명령어만 살펴보자. 여기서 git 브랜치가 master일 때 다시 dokku 명령어를 통해서 receive 명령을 실행시키는 걸 알 수 있다. Dokku receive 명령어는 dooku 메인 스크립트에 정의되어 있다.

|  |
| --- |
| **case** "$1" in  receive)  APP="$2"; IMAGE="app/$APP"  echo "-----> Cleaning up ..."  dokku cleanup  echo "-----> Building $APP ..."  cat | dokku build $APP  echo "-----> Releasing $APP ..."  dokku release $APP  echo "-----> Deploying $APP ..."  dokku deploy $APP  echo "=====> Application deployed:"  echo " $(dokku url $APP)"  echo  ;; |

여기서 다시 build, release, deply 명령을 실행한다. 자세한 사항은 전체 스크립트에서 확인할 수 있으며, 여기서는 deply 부분만 간략히 살펴보겠다.

|  |
| --- |
| deploy)  APP="$2"; IMAGE="app/$APP"  pluginhook pre-deploy $APP  # kill the app when running  if [[ -f "$DOKKU\_ROOT/$APP/CONTAINER" ]]; then  oldid=$(< "$DOKKU\_ROOT/$APP/CONTAINER")  docker kill $oldid > /dev/null 2>&1 || true  fi  # start the app  DOCKER\_ARGS=$(: | pluginhook docker-args $APP)  id=$(docker run -d -p 5000 -e PORT=5000 $DOCKER\_ARGS $IMAGE /bin/bash -c "/start web")  echo $id > "$DOKKU\_ROOT/$APP/CONTAINER"  port=$(docker port $id 5000 | sed *'s/0.0.0.0://')*  echo $port > "$DOKKU\_ROOT/$APP/PORT"  echo "http://$(< "$DOKKU\_ROOT/HOSTNAME"):$port" > "$DOKKU\_ROOT/$APP/URL"  pluginhook post-deploy $APP $port  ;; |

Deploy 부분에서는 빌드된 이미지로부터 docker container를 실행시킨다. 특히 마지막 부분에서는 post-deploy pluginhook 를 실행시킨다. 이는 /var/lib/dokku/plugins/nginx-vhosts/post-deply 에 정의되어 있는 부분으로 저장소 폴더에 앱 설정에 맞춰 nginx.conf를 생성하고 이 설정이 적용되도록 nginx 서비스를 재실행한다. 앞서 배포한 어플리케이션의 /home/dokku/node-sample/nginx.conf는 다음과 같다.

|  |
| --- |
| upstream node-sample { server 127.0.0.1:49175; }  server {  listen 80;  server\_name node-sample.nacyot.com;  location / {  proxy\_pass http://node-sample;  proxy\_http\_version 1.1;  proxy\_set\_header Upgrade $http\_upgrade;  proxy\_set\_header Connection "upgrade";  proxy\_set\_header Host $http\_host;  proxy\_set\_header X-Forwarded-Proto $scheme;  proxy\_set\_header X-Forwarded-For $remote\_addr;  proxy\_set\_header X-Forwarded-Port $server\_port;  proxy\_set\_header X-Request-Start $msec;  }  } |

자세히 보면 이는 node-sample.nacyot.com으로 들어온 요청을 받았을 때 127.0.0.1:49175 로 프록시를 보내주는 설정이다. 배포된 어플리케이션이 어느 포트로 실행되고 있는지는 실행중인 컨테이너 목록에서 확인할 수 있다.

|  |
| --- |
| $ docker ps  CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  bc259b925113 app/node-sample:latest /bin/bash -c /start 2 hours ago Up 2 hours 0.0.0.0:49175->5000/tcp ecstatic\_fermi |

포트 부분을 보게 되면 localhost:49175 요청에 대해서 node-sample 컨테이너의 5000번 포트로 포트포워딩이 이뤄지는 것을 알 수 있다. 즉 위에서 살펴본 nginx.conf 파일을 통해서 node-sample.nacyot.com 으로 요청을 받으면 node-sample 컨테이너의 5000번 포트에서 돌아가고 있는 어플리케이션으로 요청이 전달되는 것을 알 수 있다.

그런데 /home/dokku 아래의 nginix.conf 설정이 어떤 원리로 적용이 될까? 이 것은 도쿠 설치시 추가되는 /etc/nginx/conf.d/dokku.conf 파일을 보면 풀린다.

|  |
| --- |
| include /home/dokku/\*/nginx.conf; |

정말 단순하다. 이로서 git push를 하면 일어나는 부분을 살펴 보았다.

다시 한번 정리하자면 도쿠 서버에 푸시를 해서 실제로 일어나는 일이란, 이 빌드스텝(buildstep 이라는 이미지를 기반으로, /build 폴더에 있는 히로쿠 빌드팩을 통해 어플리케이션을 빌드하고, nginx 설정 파일을 동적으로 만들고 재실행해서 원하는 도메인으로 서비스를 연결시켜주는 일이다)

## 정리

지금까지 도쿠를 통해 샘플 어플리케이션을 배포해보고, 배포가 어떻게 이루어지는 지 간략하게 살펴보았다. 이제 도쿠가 어떻게 돌아가는지 감이 좀 잡혔을 것이다. 실제 서비스 운용을 위한 용도는 아니지만 도커와 도쿠를 개인서버는 물론 kt cloud와 SKT T developes VM, amazon ec2 마이크로 인스턴스에도 올려보았지만 신기할 정도로 잘 돌아간다.

# Heroku

# Vagrant

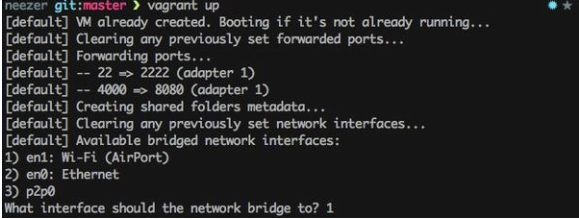
* 참조 사이트 : <http://taewan.kim/post/vagrant_intro/>

## Vagrant 소개

Vagrant는 가상화 인스턴스를 관리하는 소프트웨어이다. Vagrant의 실체는 가상 개발 환경을 생성하고 설정하기 위한 주요 가상화 소프트웨어의 고수준 레퍼이다. Vagrant가 지원하는 가상화 기술은 다음과 같다.

* VirtualBox
* VMware
* KVM
* Linux Container (LXC)
* Docker

Vagrant는 위에서 나열한 프로바이더 기술을 기반으로하여 가상화 인스턴스를 효과적으로 생성하고 관리 일관성을 확보하는 방안을 제공한다. Vagrant의 가장 매력적인 부분은 가상 인스턴스를 만들고 실행하는 과정이 매우 빠르고 편리하다는 것이다.



Vagrant의 사전적인 의미는 ‘부랑자’/’정처없는 사람’ 이라는 사전적인 의미를 갖는다. Vagrant는 가상화 이미지를 효과적으로 만들고 테스트하는 유연한 환경이라는 의미에서 명명된 것이라고 생각된다. 쉽게 만들고 쉽게 버릴 수 있고 다시 그 상태를 쉽게 복원할 수 있는 “Code as a Infrastructure” 개념을 반영한 오픈소스 프로젝트이다.

## Vagrant 프로젝트

Vagrant는 2010년 3월에 “Mitchell Hashimoto”가 시작한 오픈소스 프로젝트이다. 2016년 2월 현재 최신 버전은 1.8.1이다. 개발 언어는 Ruby이며 지원 플랫폼은 Linux, FreeBDS, OS X와 MS Window 이다. 2016년 현재 Vagrant의 주요 개발사는 HashiCorp이다. HashiCorp는 샌프란시스코에 위치한 오픈소스 기업으로 2012년 “Mitchell Hashimoto”가 설립하였다.

Vagrant는 초기에 VirtualBox만을 지원하는 형태로 개발되었다. 그 후 1.1버전부터 다른 가상화 기술을 지원하기 시작했다. VMWare와 KVM은 1.1 버전에서 지원이 추가되었다. 1.6 버전에서는 Docker를 새롭게 추가하였다.

## Vagrant 설치

앞에서 설명한 것처럼 Vagrant는 여러 가상화 프로바이더를 지원하는 툴이다. 따라서 Vagrant를 설치하기전에 사용할 가상화 프로바이더를 사전에 설치해야 한다.

Vagrant의 기본 가상화 프로바이더는 VirtualBox이다. VirtualBox는 오라클 프리웨어로 제공하는 가상화 기술이다. 지원하는 운영체제는 Window, OS X, Linux, Solaris이다. Virtual Box 설치 파일은 <https://www.virtualbox.org>에서 다운로드 받을 수 있다. VirtualBox 설치 파일의 크기는 운영체제에 따라 약간의 차이는 있지만 약 100Mb 내외 이다.

앞에서 설명한 것 처럼 Vagrant는 Ruby로 개발된 소프트웨어 이다. 따라서 RubyGem으로 설치가 가능하다. 그러나 Vagrant 프로젝트에서는 RubyGem을 시용한 설치하는 방법보다 Vagrant 홈페이지에서 바이너리로 다운로드 하는 것을 권장하고 있다.

RubyGem을 사용할 경우 이전 버전이 설치될 가능성이 높기 때문이다. Vagrant 다운로드 URL은 다음과 같다.

<https://www.vagrantup.com/downloads.html>

다운받은 Vagrant 설치 파일로부터 설치를 완료한 후, Vagrant의 설치는 다음과 같은 명령으로 확인할 수 있다.

|  |
| --- |
| root$ vagrant version  Installed Version: 1.8.1  Vagrant was unable to check for the latest version of Vagrant.  Please check manually at http://www.vagrantup.com  root$ |

## Vagrant 둘러보기

Vagrant Quick-Start를 하기 전에 앞서 간단히 Vagrant가 동작하는 방식에 대하여 알아보겠습니다. Vagrant는 일반적으로 다음과 같은 단계로 작업을 수행한다.

1. Box 이미지 다운로드
2. Box 이미지를 이용하여 프로젝트 생성
3. 프로젝트 최상위 디렉토리에 위치한 Vagrantfile 파일 수정
4. 프로젝트의 가상 인스턴스 시작
5. 가상 이미지 접속 및 작업
6. 가상 이미지 종료

Vagrant는 프로젝트를 생성하기 앞서 여러 프로젝트가 공통적으로 사용할 Box를 다운로드 한다. Box란 Vagrant에서 가상 이미지로 사용할 수 있도록 초기 설정된 바이너리이다. Vagrant의 메인 개발사인 hashicorp는 여러 운영체제로 만든 Box를 제공한다. Hasicorp의 “Atlas box catalog”에서 Box를 검색하고 다운로드 할 수 있다. Box를 검색하고 다운로드를 지원하는 “Atlas box catalog”의 URL은 다음과 같다.

* <https://atals.hashicorp.com/boxes/search>

“Atlas box catalog” 혹은 Box의 URL을 파라미터로 “vagrant box add” 명령을 이용하여 Box를 다운로드 할 수 있다. 다운받은 기본 Box는 ~/vagrant.d/boxes 디렉토리에 저장된다. 이 디렉토리에 위치한 각 Box는 프로바이더 별로 디렉토리를 구분하여 저장되며, 각 프로바이더 폴더에는 4개의 파일이 저장되어 있는 것을 확인할 수 있다.

|  |
| --- |
| taewan:demo taewan$ vagrant box add ubuntu/trusty64  ==> box: Loading metadata for box 'ubuntu/trusty64'  box: URL: https://atlas.hashicorp.com/ubuntu/trusty64  ==> box: Adding box 'ubuntu/trusty64' (v20160206.0.0) for provider: virtualbox  box: Downloading: https://atlas.hashicorp.com/ubuntu/boxes/trusty64/versions/20160206.0.0/providers/virtualbox.box  ==> box: Successfully added box 'ubuntu/trusty64' (v20160206.0.0) for 'virtualbox'!  taewan:demo taewan$ vagrant box list  ubuntu/trusty64 (virtualbox, 20160206.0.0)  taewan:demo taewan$  taewan:demo taewan$ ls -al ~/.vagrant.d/boxes/  total 0  drwxr-xr-x 3 taewan staff 102 2 10 00:00 .  drwxr-xr-x 9 taewan staff 306 2 10 00:00 ..  drwxr-xr-x 4 taewan staff 136 2 10 00:00 ubuntu-VAGRANTSLASH-trusty64  taewan:demo taewan$ tree ~/.vagrant.d/boxes/ubuntu-VAGRANTSLASH-trusty64/  /Users/taewan/.vagrant.d/boxes/ubuntu-VAGRANTSLASH-trusty64/  ├── 20160206.0.0  │   └── virtualbox  │   ├── Vagrantfile  │   ├── box-disk1.vmdk  │   ├── box.ovf  │   └── metadata.json  └── metadata\_url  2 directories, 5 files  taewan:demo taewan$ |

Vagrant는 가상 이미지 관리하기 위해서 프로젝트를 만든다. 각 프로젝트의 최상위 디렉토리에는 Vagrantfile 파일이 위치한다. 이 파일은 Vagrant로 관리되는 가상 이미지 설정을 포함한다.

프로젝트 최상위 디렉토리에서 “vagrant up” 명령을 최초로 실행하면 vagrant는 “~/vagrant.d/boxes”에 위치한 box 이미지를 복사하여 지정된 프로바이더에 맞는 가상 이미지를 생성하고 Guset OS를 실행한다.

## Vagrant 사용해보기

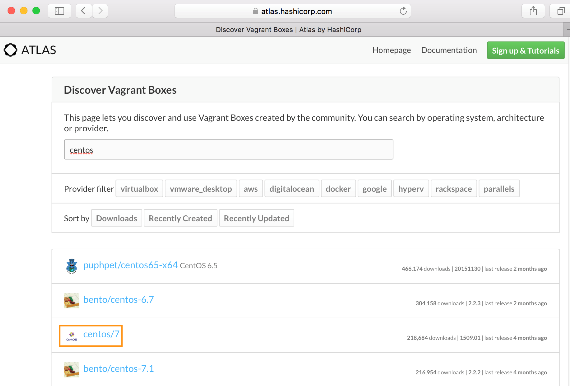
Vagrant로 CentOS 가상 인스턴스를 만들고 로그인하여 사용하는 간단한 데모를 진행해보자.

### Box 이미지 카탈로그 조회

현재 호스트에 머신에 설치된 Box 이미지의 목록을 조회한다.

|  |
| --- |
| root$ vagrant box list  ubuntu/trusty64 (virtualbox, 20160206.0.0)  root$ |

현재 호스트 머신에는 CentOS 이미지가 없는 것을 확인할 수 있다. “Atals box catalog”에서 CentOS를 검색한다.



위 검색 결과에서 “centos/7”을 사용할 것을 결정한다.

### Box 이미지 다운로드

“vagrant box add” 명령으로 앞으로 사용할 CentOS 기본 이미지를 다운로드 한다.

|  |
| --- |
| demo taewan$ vagrant box add centos/7  ==> box: Loading metadata for box 'centos/7'  box: URL: https://atlas.hashicorp.com/centos/7  This box can work with multiple providers! The providers that it  can work with are listed below. Please review the list and choose  the provider you will be working with.  1) libvirt  2) virtualbox  Enter your choice: 2  ==> box: Adding box 'centos/7' (v1509.01) for provider: virtualbox  box: Downloading: https://atlas.hashicorp.com/centos/boxes/7/versions/1509.01/providers/virtualbox.box  ==> box: Successfully added box 'centos/7' (v1509.01) for 'virtualbox'!  demo taewan$ |

설치가 완료되면 “vagrant box list” 명령으로 설치 결과를 확인할 수 있다.

|  |
| --- |
| demo taewan$ vagrant box list  centos/7 (virtualbox, 1509.01)  ubuntu/trusty64 (virtualbox, 20160206.0.0)  demo taewan$ |

### Vagrant 프로젝트 생성 및 설정 파일 확인

Vagrant 프로젝트를 생성할 디렉터리에 생성한 후 Vagrant 프로젝트 초기화를 수행한다. 프로젝트 초기화 명령(vagrant init)을 실행할 때 사용할 box를 지정할 수 있다.

|  |
| --- |
| workspace taewan$ pwd  /Users/taewan/workspace  workspace taewan$ vagrant init centos/7  A `Vagrantfile` has been placed in this directory. You are now  ready to `vagrant up` your first virtual environment! Please read  the comments in the Vagrantfile as well as documentation on  `vagrantup.com` for more information on using Vagrant.  workspace taewan$ ls  Vagrantfile  workspace taewan$ tree  .  └── Vagrantfile  0 directories, 1 file  workspace taewan$ cat Vagrantfile  # 주석 제거  Vagrant.configure(2) do |config|  config.vm.box = "centos/7"  end  workspace taewan$ |

### Vagrant 가상 이미지 실행 및 ssh 접근

“vagrant up” 명령으로 가상 이미지를 실행하고 “vagrant ssh” 명령으로 가상 이미지에 ssh 접근이 가능하다.

|  |
| --- |
| workspace taewan$ vagrant up  Bringing machine 'default' up with 'virtualbox' provider...  ==> default: Importing base box 'centos/7'...  ==> default: Matching MAC address for NAT networking...  ==> default: Checking if box 'centos/7' is up to date...  ==> default: Setting the name of the VM: workspace\_default\_1455162125926\_16977  ==> default: Clearing any previously set network interfaces...  ==> default: Preparing network interfaces based on configuration...  default: Adapter 1: nat  ==> default: Forwarding ports...  default: 22 (guest) => 2222 (host) (adapter 1)  ==> default: Booting VM...  ==> default: Waiting for machine to boot. This may take a few minutes...  default: SSH address: 127.0.0.1:2222  default: SSH username: vagrant  default: SSH auth method: private key  default:  default: Vagrant insecure key detected. Vagrant will automatically replace  default: this with a newly generated keypair for better security.  default:  default: Inserting generated public key within guest...  default: Removing insecure key from the guest if it's present...  default: Key inserted! Disconnecting and reconnecting using new SSH key...  ==> default: Machine booted and ready!  ==> default: Checking for guest additions in VM...  default: No guest additions were detected on the base box for this VM! Guest  default: additions are required for forwarded ports, shared folders, host only  default: networking, and more. If SSH fails on this machine, please install  default: the guest additions and repackage the box to continue.  default:  default: This is not an error message; everything may continue to work properly,  default: in which case you may ignore this message.  ==> default: Installing rsync to the VM...  ==> default: Rsyncing folder: /Users/taewan/workspace/ => /home/vagrant/sync  workspace taewan$ vagrant ssh  [vagrant@localhost ~]$ uname -a  Linux localhost.localdomain 3.10.0-229.14.1.el7.x86\_64 #1 SMP Tue Sep 15 15:05:51 UTC 2015 x86\_64 x86\_64 x86\_64 GNU/Linux  [vagrant@localhost ~]$ exit  logout  Connection to 127.0.0.1 closed.  workspace taewan$ |

### ssh config 파일 수정

간편한 접근을 위하여 vagrant는 ssh config 설정을 제공하는 명령을 제공한다. “vagrant ssh-config-host [hostname]” 명령으로 ssh config를 위한 내용을 확인할 수 있다. 이 결과는 리다이렉션을 이용하여 ~/.ssh/config에 적용할 수 있다. Ssh config가 적용된 이후에는 ssh [hostname]으로 접근 가능하다.

|  |
| --- |
| workspace taewan$ # ssh config 설정 확인  workspace taewan$ vagrant ssh-config --host demo  Host demo  HostName 127.0.0.1  User vagrant  Port 2222  UserKnownHostsFile /dev/null  StrictHostKeyChecking no  PasswordAuthentication no  IdentityFile "/Users/taewan/workspace/.vagrant/machines/default/virtualbox/private\_key"  IdentitiesOnly yes  LogLevel FATAL  workspace taewan$ # ssh config 적용  workspace taewan$ vagrant ssh-config --host demo >> ~/.ssh/config  workspace taewan$ # ssh 접근  workspace taewan$ ssh demo  Last login: Wed Feb 10 22:43:45 2016 from 10.0.2.2  [vagrant@localhost ~]$ uname -a  Linux localhost.localdomain 3.10.0-229.14.1.el7.x86\_64 #1 SMP Tue Sep 15 15:05:51 UTC 2015 x86\_64 x86\_64 x86\_64 GNU/Linux  [vagrant@localhost ~]$ exit  logout  Connection to 127.0.0.1 closed.  workspace taewan$ |

### Vagrant 상태 확인 및 종료

다음 명령으로 Vagrant 인스턴스의 상태를 확인하고 종료할 수 있다.

|  |
| --- |
| workspace taewan$ # 인스턴스 상태 확인  workspace taewan$ vagrant status  Current machine states:  default running (virtualbox)  The VM is running. To stop this VM, you can run `vagrant halt` to  shut it down forcefully, or you can run `vagrant suspend` to simply  suspend the virtual machine. In either case, to restart it again,  simply run `vagrant up`.  workspace taewan$ # 인스턴스 종료  workspace taewan$ vagrant destroy  default: Are you sure you want to destroy the 'default' VM? [y/N] y  ==> default: Forcing shutdown of VM...  ==> default: Destroying VM and associated drives...  workspace taewan$ # 인스턴스 상태 확인  workspace taewan$ vagrant status  Current machine states:  default not created (virtualbox)  The environment has not yet been created. Run `vagrant up` to  create the environment. If a machine is not created, only the  default provider will be shown. So if a provider is not listed,  then the machine is not created for that environment.  taewanui-MacBook-Pro:workspace taewan$ |

## Vagrant 명령 요약

* vagrant –v : 버전 확인
* vagrant box add [title] [download-url]

box를 vagrant에 등록한다.

Ex) vagrant box add bento/centos-7.2

Vagrant box add centos65 https://github.com/2creatives/vagrant-centos/releases/download/v6.5.3/centos65-x86\_64-20140116.box

* vagrant box list : box가 vagrant에 잘 등록되었는지 확인
* vagrant init [title]

vagrant init centos64 [title : box 등록시 사용한 title명]

* vagrant status : 현재 프로젝트의 가상 이미지 상태 요약
* vagrant global-status : 호스트 머신 전체의 Vagrant 가상 이미지들의 상태 확인
* vagrant up : Vagrant 가상 이미지 시작
* vagrant halt : 가상 인스턴스 강제 종료
* vagrant destroy : 가상 이미지 종료 및 기존 이미지 삭제
* vagrant suspend : 가상 인스턴스 하이버네이트, 상태 보존
* vagrant resume : 중지된 인스턴스 시작
* vagrant reload : 변경된 VagrantFile 적용
* vagrant ssh : 현재 프로젝트의 가상 이미지에 ssh 접근 (디폴트 vagrant/vagrant )
* vagrant share : 외부에서 vm에서 구동되고 있는 웹 서버에 접속이 가능하게 함

## Reference

* <https://gist.github.com/wpscholar/a49594e2e2b918f4d0c4>
* https://www.vagrantup.com/docs/

# Docker Container Orchestration Tool, Kubernetes

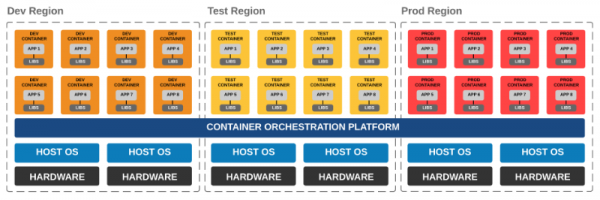
참조 URL : https://www.popit.kr/kubernetes-introduction/

Kubernetes가 무엇인지 설명하고, 아키텍쳐와 구성요소에 대해서 살펴본다. 그리고 minikube를 이용해 로컬에서 Kubernetes를 사용해본다.

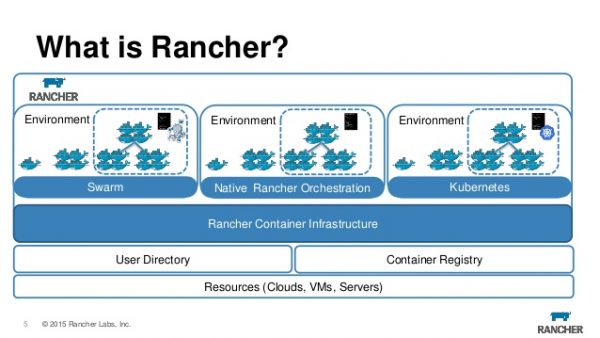
* What is Kubernetes?
* Kubernetes Architecture (abbreviated)
* Getting Started with minikube
* Kubernetes Object: Pod
* Kubernetes Object: Service
* Kubernetes Object: Deployment
* Kubernetes Object: Pet Set
* Kubernetes Object: Others
* Summary
* Furrthemore
* References

## What is Kubernetes?

Kubernetes이하( k8s)는 docker-swarm, marathon과 같은 container orchestration 툴이다. Docker-swarm#production demo 영상에서는 docker-swarm을 이용해 1000개 노드에 50000개의 컨테이너를 띄우는 것을 보여준다.



Host 머신 한대라면 docker run이나 docker-compose 등 무엇으로 container를 실행해도 상관 없다. 그러나 사용자가 많아지면 하나의 host에서 모든 container 를 실행할 수 없다. (host에서 fail이라도 발생하면..) 그러나 여러대의 host에서 container를 실행하려면 inter-host container 네트워킹과, host machine의 리소스를 고려한 container 분배 등을 고려해야 한다. 이런 문제들을 해결해주는 것이 바로 container orchestration 툴이다. 요즘은 여러 container orchestration 툴을 골라 쓸 수 있게 해주는 **rancher(https://rancher.com/)**와 같은 플랫폼도 나오고 있다.



Kubernetes가 하는 일은.

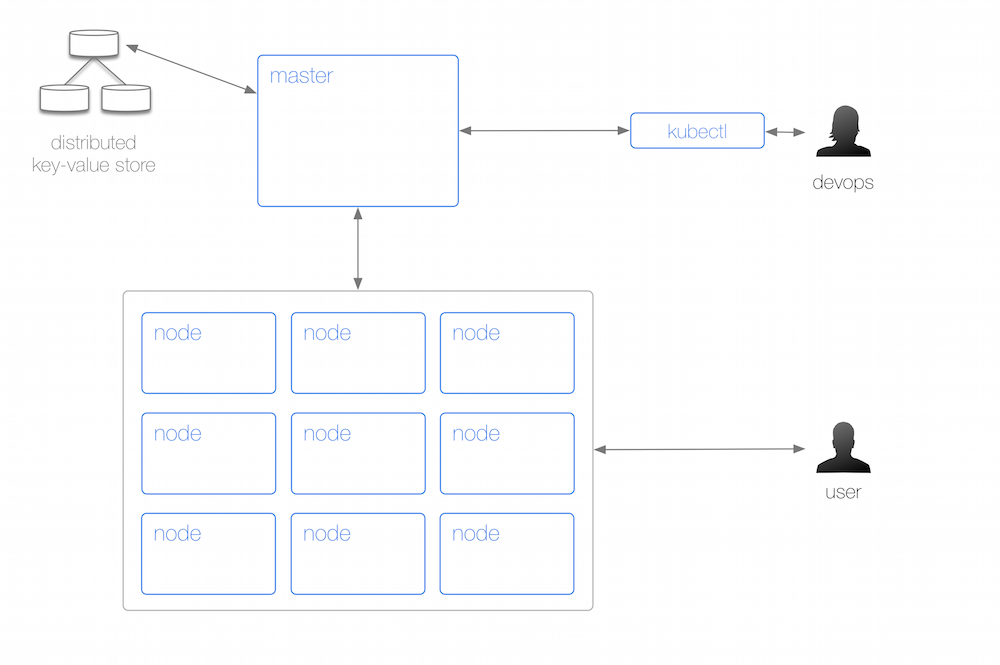
* 여러 host(=node in k8s)를 묶어 클러스터를 구성하고
* Container를 적절한 위치에 배포하고 (auto-placement)
* Container가 죽으면 자동으로 복구하며 (auto-restart)
* 필요에 따라 container를 매끄럽게 추가(scaling), 복제(replication), 업데이트(rolling update), 롤백(rollback) 할 수 있다.
* 이 외에도 수 많은 기능이 있으며, **What is k8s?**( https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/) 에서 확인 할 수 있다.

Kubernetes를 사용하기 위해서는 아래의 내용들을 이해해야 한다. 이 글에서 object와 internal에 대해서만 간략히 설명한다. 필요한 내용은 kubernetes docs를 통해 얻을 수 있는데, 다행히도 문서가 장황하지 않다. 필요한 내용을 필요한 만큼만 설명하고 있기 때문에 날 잡아서 읽기에도 괜찮다.

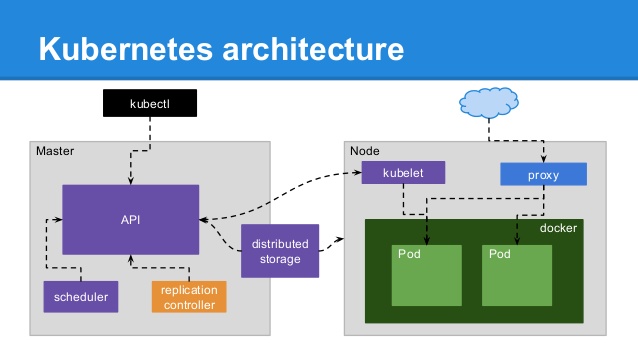
* Kubernetes object (e.g pod, pet set, service, selector 등)
* Kubernetes internal
* Multi-host 위에서 container 실행시 고려해야 할 것들
* Service discovery
* Networking
* Volume management
* Log aggregation 등

## Kubernetes Architecture (abbreviated)

아래 간략한 구성 요소이다.

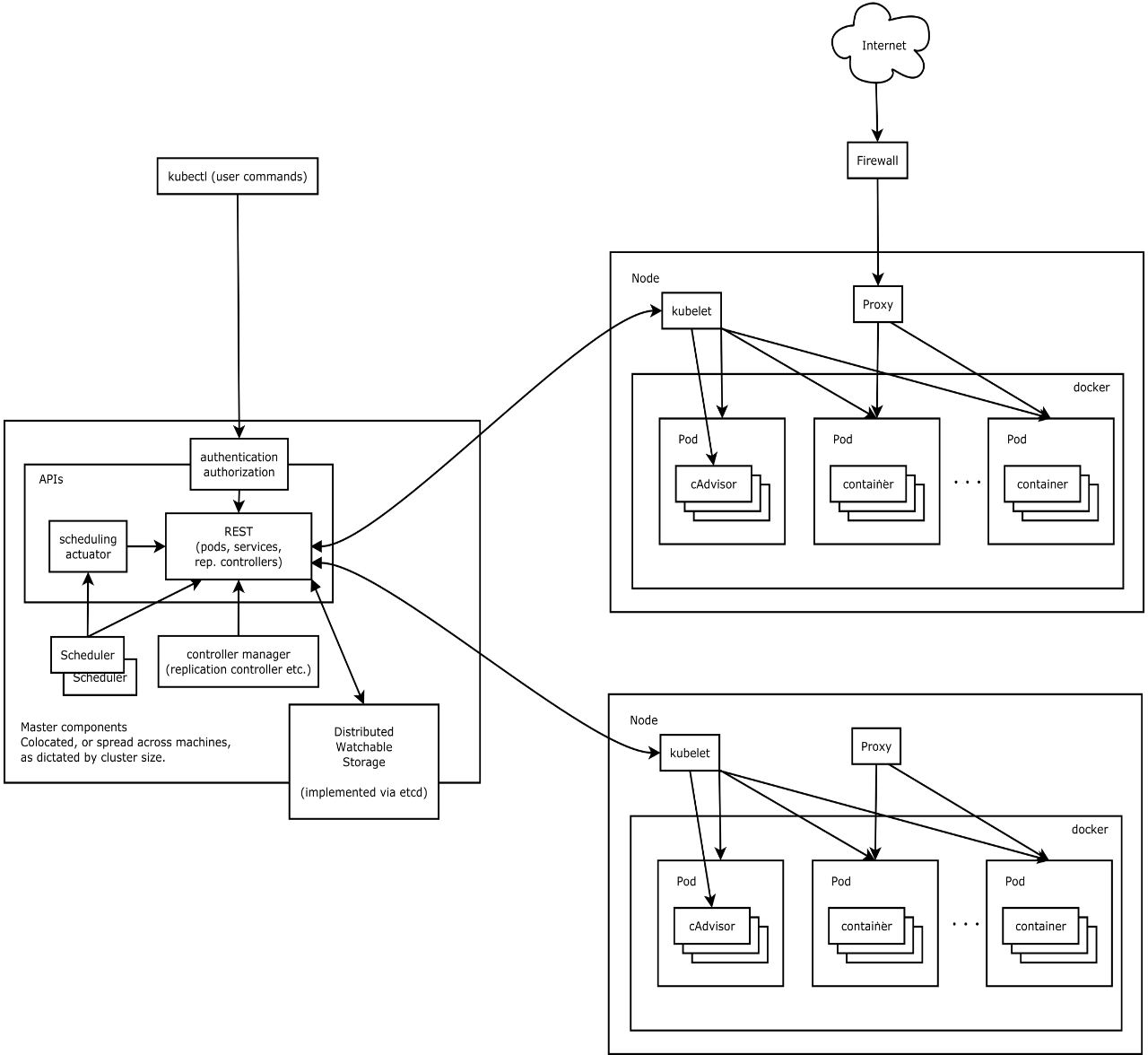


하나의 k8s 클러스터는 하나의 master와 여러 개의 node로 구성되어 있다. 개발자는 kubectl을 이용해서 master에 명령을 내리고, node를 관리하는 반면 사용자가 (endpoing user)는 node에 접속해서 서비스를 이용한다. 위 그림을 조금 더 자세히 보면.



* master에는 (현재는 master가 단일 노드이지만 추후 multi-node master 가 지원될 예정)
* 작업을 위한 api server
* State를 관리하기 위한 분산 저장소 (default는 etcd)
* Scheduler
* Controller manager등이 있다.
* node(=minion)에는
* master와 통신하는 kubelet (agent, 현재는 containerized되어있지 않음)이 있고
* 외부의 요청을 처리하는 kube-proxy
* Container 리소스 모니터링을 위한 cAdvier 등이 있다.

이제 다시 큰 그림을 자세히 보면 다음과 같다.



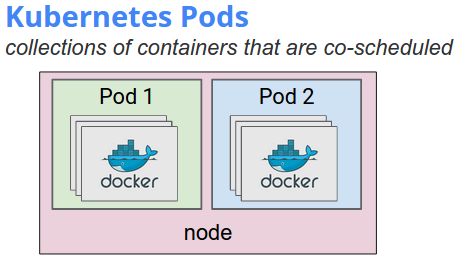
## Getting Started with minikube

Minikube를 설치하면 로컬에서 kubernetes를 실행해볼 수 있다. 진행하기 전에 GCE에 가입하신 후 Kubernetes: Hello World Walkthrough를 진행하고 오시면 내용을 이해하시는데 더욱 도움이 될 것이다.

추가적으로 kubectl 1.3.0+의 경우에는 bash와 zsh completion이 들어 있따. 이걸 이용하면 편한다. 만약 gcloud이용해서 kubectl를 설치하셨다면 1.2.5 버전이 깔릴 수 있다. 이때는 kubectl install 참고하시어 설치하면 된다.

|  |
| --- |
| $ kubectl version  Client Version: version.Info{Major:"1", Minor:"3", GitVersion:"v1.3.0", GitCommit:"283137936a498aed572ee22af6774b6fb6e9fd94", GitTreeState:"clean", BuildDate:"2016-07-01T19:26:38Z", GoVersion:"go1.6.2", Compiler:"gc", Platform:"darwin/amd64"}  Server Version: version.Info{Major:"1", Minor:"3", GitVersion:"v1.3.3", GitCommit:"c6411395e09da356c608896d3d9725acab821418", GitTreeState:"dirty", BuildDate:"1970-01-01T00:00:00Z", GoVersion:"go1.6.2", Compiler:"gc", Platform:"linux/amd64"}  $ source <(kubectl completion zsh |

## Kubernetes Object: Pod



하나 또는 여러 개의 container 묶음을 pod 이라 부른다. docker에서 container 끼리 통신하려면 같은 network위에 있도록 구성해야 하는 반면 (compose 도 동일), 하나의 pod 내에 있는 container 끼리는 그럴 필요가 없다. (pod의 종료, 삭제 관련해서는 Termination of Pods를 참고)

* 같은 IP 와 post space를 가지고 때문에 localhost로 통신이 가능하며
* Volume을 공유한다.
* 만약 어느 container가 죽고 재시작되어도 pod이 살아있는 한 shared volume은 유지된다.

위에서 k8s가 auto-restart 등을 해준다고 했었는데 테스트 해보겠다. 그 전에 먼저 클러스터가 정상적으로 세팅이 되었는지 확인해 본다. Minikube를 이용해서 로컬에서 실행했으므로 아래와 같은 결과가 나온다. Kubectl을 여러 클러스터 중 하나에 붙어서 커맨드를 날릴 수 있도록 도와주는 docker-machine 정도로 이해하면 된다.

|  |
| --- |
| $ kubectl config view  apiVersion: v1  clusters:  - cluster:  certificate-authority: /Users/1ambda/.minikube/ca.crt  server: <https://192.168.64.2:8443>  name: minikube  contexts:  - context:  cluster: minikube  user: minikube  name: minikube  current-context: minikube  kind: Config  preferences: {}  users:  - name: minikube  cser:  client-certificate: /Users/1ambda/.minikube/apiserver.crt  client-key: /Users/1ambda/.minikube/apiserver.key |

이제 아래와 같이 nginx.yaml을 만들겠습니다.

|  |
| --- |
| apiVersion: v1  kind: ReplicationController  metadata:  name: nginx  spec:  replicas: 3  selector:  app: nginx  template:  metadata:  name: nginx  labels:  app: nginx  spec:  containers:  - name: nginx  Image: nginx  ports:  - containerPort: 80 |

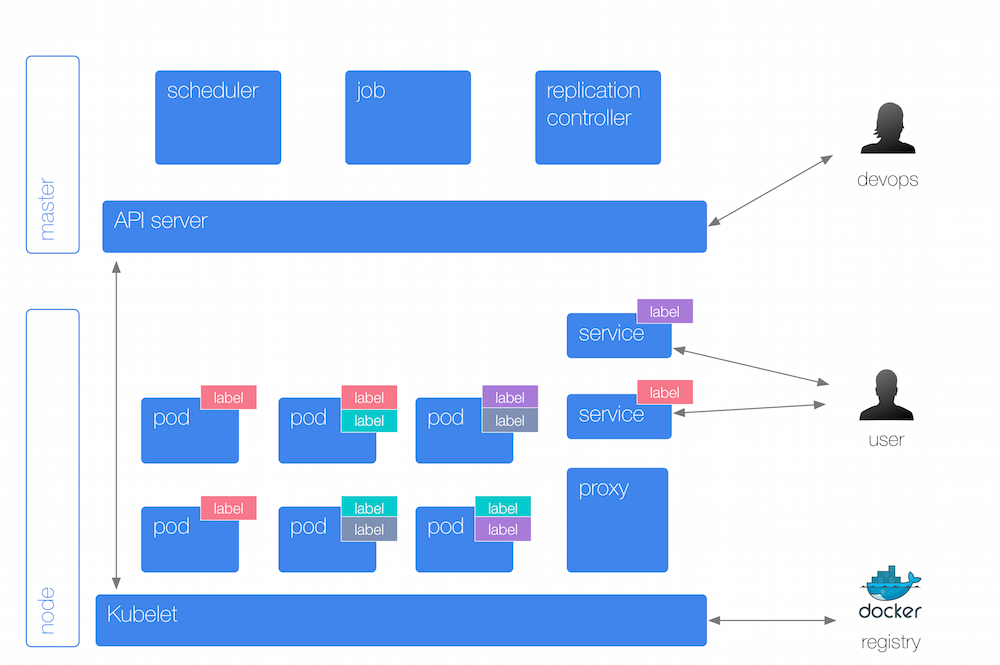
만들어진 replication controller 를 k8s에 올려보면,

|  |
| --- |
| $ kubectl create –f nginx.yaml  Replicationcontroller “nginx” created  $ kubectl get replicationcontroller nginx  NAME DESIRED CURRENT AGE  Nginx 3 3 41s  $ kubectl get pods  NAME READY STATUS RESTARTS AGE  nginx-5aa7m 1/1 Running 0 45s  nginx-5frtc 1/1 Running 0 45s  nginx-sg1s5 1/1 Running 0 45s |

Pod 하나를 죽여 보겠습니다. 재생성되는 것을 확인할 수 있습니다.

|  |
| --- |
| $ kubectl delete pod nginx-5aa7m  Pod “nginx-5aa7m” deleted  $ kubectl get pods  NAME READY STATUS RESTARTS AGE  nginx-5aa7m 1/1 Running 0 1s  nginx-5frtc 0/1 ContainerCreating 0 1s  nginx-sg1s5 1/1 Running 0 1s  $ kubectl get pods  NAME READY STATUS RESTARTS AGE  nginx-6tub7 1/1 Running 0 5s  nginx-5frtc 1/1 Running 0 1s  nginx-sg1s5 1/1 Running 0 1s |

## Kubernetes Object: Service



Pod은 생성/삭제 될 수 있다. Replication-controller를 이용하면 심지어 동적으로도 scale up/down 이 가능한데 이럴 경우 IP가 변경/추가/제거 되므로 k8s 에서는 external pods이나 pods pods간의 안정적인 통신을 위해 service라는 object를 도입했다.

위 그림을 잘 보면, pod에 있는 label과 동일한 컬러의 것이 service에도 있고 해당 service가 같은 label 컬러를 가진 pod을 위한 것임을 쉽게 알 수 있다.

위에서 생성한 nginx pods를 위한 service를 nginx-svc.yaml이란 이름으로 만들어 보겠다. Selector를 잘 보자.

|  |
| --- |
| apiVersion: v1  kind: Service  metadata:  name: nginx-svc  spec:  ports:  - name: nginx-svc  port: 8090  targetPort: 80  type: NodePort  selector:  app: nginx |

이제 생성하면, nginx-svc.service의 EXTERNAL-IP가 <nodes>로 보인다. 전체 node에 대해서 외부 포트를 열어서 그렇다. NodePort대신 LoadBalancer(cloud provider가 지원할 경우만 사용가능) 타입을 이용하거나 ingress를 이용할 수 있다.

|  |
| --- |
| $ kubectl create –f nginx-svc.yaml  service “nginx-svc” created  $ kubtctl get service  NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE  kubernetes 10.0.0.1 443/TCP 3d  nginx-svc 10.0.0.196 8090/TCP 1m  $ kubectl describe service nginx-svc  …  Endpoints: 172.17.0.3:80, 172.17.0.4:80, 172.17.0.5:80  $ kubectl get node  NAME STATUS AGE  Boot2docker Ready 3d  $ kubectl describe node boot2docker | grep Address  Addresses: 192.168.64.2, 192.168.64.2  $ curl 192.168.64.2:31968  Welcome to nginx!  … |

192.168.64.2:31968을 접근해보면, nginx가 떠있음을 확인할 수 있다. Pod이 생성될 때 active service에 대해서 kubelet이 service의 IP, port와 관련된 환경변수를 pod에 주입한다. (service가 먼저 생성되어 있어야 함) 예를 들어 service name이 redis-master라면 다음과 같은 값들이 주입된다.

|  |
| --- |
| REDIS\_MASTER\_SERVICE\_HOST=10.0.0.11  REDIS\_MASTER\_SERVICE\_PORT=6379  REDIS\_MASTER\_PORT=tcp://10.0.0.11:6379  REDIS\_MASTER\_PORT\_6379\_TCP=tcp://10.0.0.11:6379  REDIS\_MASTER\_PORT\_6379\_TCP\_PROTO=tcp  REDIS\_MASTER\_PORT\_6379\_PORT=6379  REDIS\_MASTER\_PORT\_6379\_TCP\_ADDR=10.0.0.11 |

하여 pod간 통신에는 env variable을 이용할 수 있으나, DNS를 이용하는 것이 더 권장된다. 그리고 label값은 kubernetes/example/guestbook 처럼 붙이는 것이 권장된다.

|  |
| --- |
| apiVersion: extensions/v1beta1  kind: Deployment  metadata:  name: redis-master  spec:  replicas: 1  template:  metadata:  labels:  app: redis  role: master  tier: backend |

## Kubernetes Object: Deployment

참조 URL

https://blog.nacyot.com/articles/2014-01-27-easy-deploy-with-docker/