https://subicura.com/2017/01/19/docker-guide-for-beginners-1.html

도커는 2013년 3월 산타클라라에서 열린 Pycon Conference에서 dotCloud의 창업자인 Solomon Hykes가 [The future of Linux Containers](https://youtu.be/wW9CAH9nSLs) 라는 세션을 발표하면서 처음 세상에 알려졌습니다.

이 발표 이후 도커가 인기를 얻으면서 2013년 10월 아예 회사이름을 도커(Docker Inc.)로 바꾸고 2014년 6월 도커 1.0을 발표합니다. 2014년 8월 도커에 집중하기 위해 dotCloud 플랫폼을 매각하고 2015년 4월 $95M(약 1,100억원) 투자를 유치한 후 계속해서 빠르게 성장하고 있습니다. (현재까지 총 투자액은 $180M이며 2016년 6월 MS에서 $4B/₩4조에 인수하려 했다는 [기사](https://www.sdxcentral.com/articles/news/sources-microsoft-tried-to-buy-docker-for-4b/2016/06/)가 있습니다.)

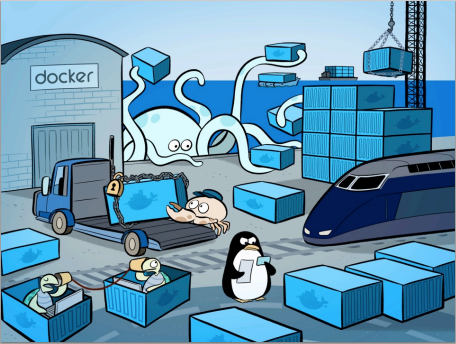
## 도커란?

도커는 **컨테이너 기반의 오픈소스 가상화 플랫폼**입니다.

컨테이너라 하면 배에 실는 네모난 화물 수송용 박스를 생각할 수 있는데 각각의 컨테이너 안에는 옷, 신발, 전자제품, 술, 과일등 다양한 화물을 넣을 수 있고 규격화되어 컨테이너선이나 트레일러등 다양한 운송수단으로 쉽게 옮길 수 있습니다.

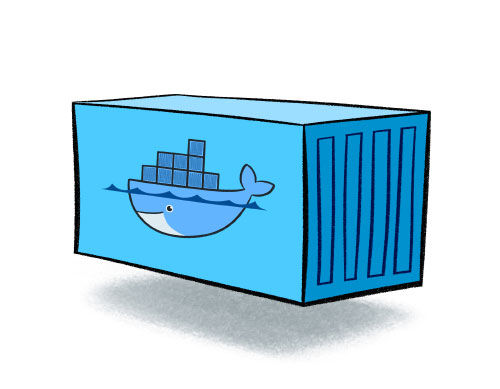
서버에서 이야기하는 컨테이너도 이와 비슷한데 다양한 프로그램, 실행환경을 컨테이너로 추상화하고 동일한 인터페이스를 제공하여 프로그램의 배포 및 관리를 단순하게 해줍니다. 백엔드 프로그램, 데이터베이스 서버, 메시지 큐등 어떤 프로그램도 컨테이너로 추상화할 수 있고 조립PC, AWS, Azure, Google cloud등 어디에서든 실행할 수 있습니다.

컨테이너를 가장 잘 사용하고 있는 기업은 구글인데 [2014년 발표](https://speakerdeck.com/jbeda/containers-at-scale) 에 따르면 구글은 모든 서비스들이 컨테이너로 동작하고 매주 20억 개의 컨테이너를 구동 한다고 합니다.

****

*도커는 컨테이너를 관리하는 플랫폼*

**컨테이너(Container)**

****

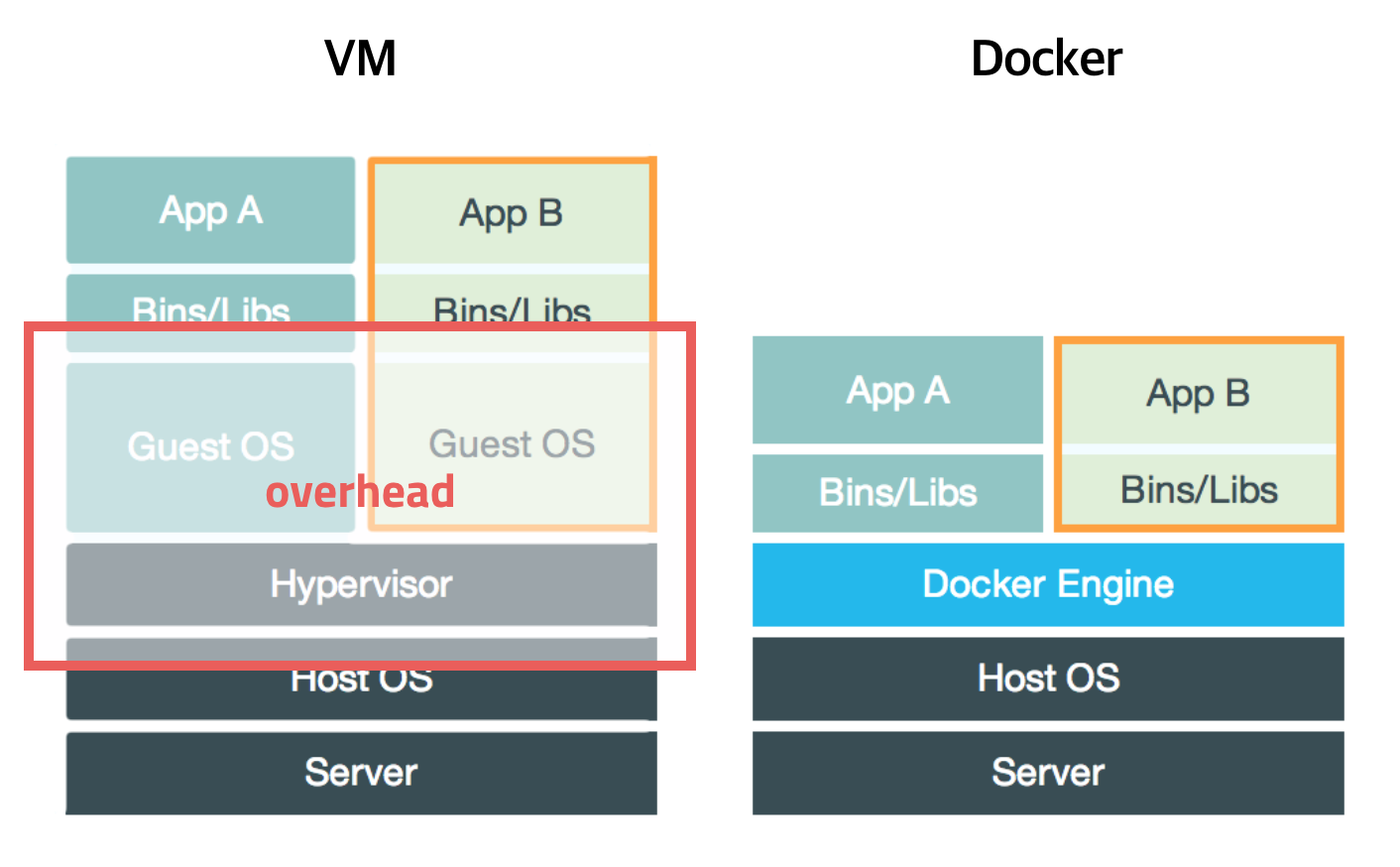
*docker container*

컨테이너는 격리된 공간에서 프로세스가 동작하는 기술입니다. 가상화 기술의 하나지만 기존방식과는 차이가 있습니다.

기존의 가상화 방식은 주로 **OS를 가상화**하였습니다.

우리에게 익숙한 [VMware](http://www.vmware.com/)나 [VirtualBox](https://www.virtualbox.org/)같은 가상머신은 호스트 OS위에 게스트 OS 전체를 가상화하여 사용하는 방식입니다. 이 방식은 여러가지 OS를 가상화(리눅스에서 윈도우를 돌린다던가) 할 수 있고 비교적 사용법이 간단하지만 무겁고 느려서 운영환경에선 사용할 수 없었습니다.

이러한 상황을 개선하기 위해 CPU의 가상화 기술([HVM](https://en.wikipedia.org/wiki/Hardware-assisted_virtualization))을 이용한 [KVM](http://www.linux-kvm.org/)Kernel-based Virtual Machine과 [반가상화](https://en.wikipedia.org/wiki/Paravirtualization) Paravirtualization방식의 [Xen](https://www.xenproject.org/)이 등장합니다. 이러한 방식은 게스트 OS가 필요하긴 하지만 전체OS를 가상화하는 방식이 아니였기 때문에 호스트형 가상화 방식에 비해 성능이 향상되었습니다. 이러한 기술들은 [OpenStack](https://www.openstack.org/)이나 AWS, [Rackspace](https://www.rackspace.com/)같은 클라우드 서비스에서 가상 컴퓨팅 기술의 기반이 되었습니다.

****

*가상머신과 도커*

전가상화든 반가상화든 추가적인 OS를 설치하여 가상화하는 방법은 어쨋든 성능문제가 있었고 이를 개선하기 위해 **프로세스를 격리** 하는 방식이 등장합니다.

리눅스에서는 이 방식을 리눅스 컨테이너라고 하고 단순히 프로세스를 격리시키기 때문에 가볍고 빠르게 동작합니다. CPU나 메모리는 딱 프로세스가 필요한 만큼만 추가로 사용하고 성능적으로도 거어어어어의 손실이 없습니다.

도커의 기본 네트워크 모드는 Bridge모드로 약간의 성능 손실이 있습니다. 네트워크 성능이 중요한 프로그램의 경우 --net=host 옵션을 고려해야 합니다.

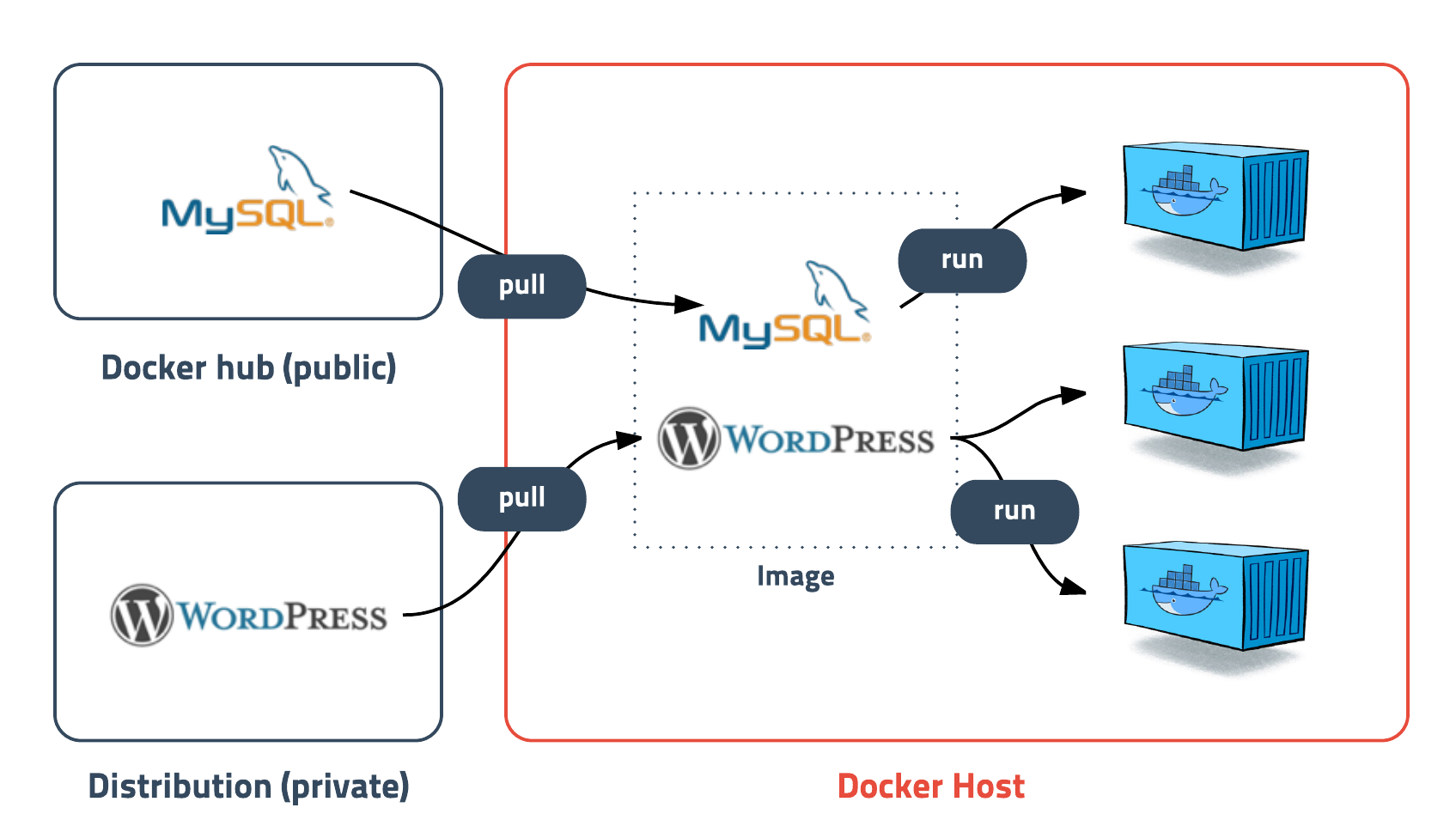
하나의 서버에 여러개의 컨테이너를 실행하면 서로 영향을 미치지 않고 독립적으로 실행되어 마치 가벼운 VMVirtual Machine을 사용하는 느낌을 줍니다. 실행중인 컨테이너에 접속하여 명령어를 입력할 수 있고 apt-get이나 yum으로 패키지를 설치할 수 있으며 사용자도 추가하고 여러개의 프로세스를 백그라운드로 실행할 수도 있습니다. CPU나 메모리 사용량을 제한할 수 있고 호스트의 특정 포트와 연결하거나 호스트의 특정 디렉토리를 내부 디렉토리인 것처럼 사용할 수도 있습니다.

새로운 컨터이너를 만드는데 걸리는 시간은 겨우 1-2초로 가상머신과 비교도 할 수 없이 빠릅니다.

이러한 컨테이너라는 개념은 도커가 처음 만든 것이 아닙니다. 도커가 등장하기 이전에, 프로세스를 격리하는 방법으로 리눅스에서는 cgroupscontrol groups와 namespace를 이용한 [LXC](https://linuxcontainers.org/lxc/)Linux container가 있었고 FreeBSD에선 [Jail](https://www.freebsd.org/doc/handbook/jails.html), Solaris에서는 [Solaris Zones](https://docs.oracle.com/cd/E18440_01/doc.111/e18415/chapter_zones.htm#OPCUG426)이라는 기술이 있었습니다. 구글에서는 고오오급 기술자들이 직접 컨테이너 기술을 만들어 사용하였고 [lmctfy(Let Me Contain That For You)](https://github.com/google/lmctfy)라는  오픈소스 컨테이너 기술을 공개했지만 성공하진 못했습니다.

도커는 LXC를 기반으로 시작해서 0.9버전에서는 자체적인 [libcontainer](https://github.com/docker/libcontainer) 기술을 사용하였고 추후 [runC](http://runc.io/)기술에 합쳐졌습니다.

**이미지(Image)**

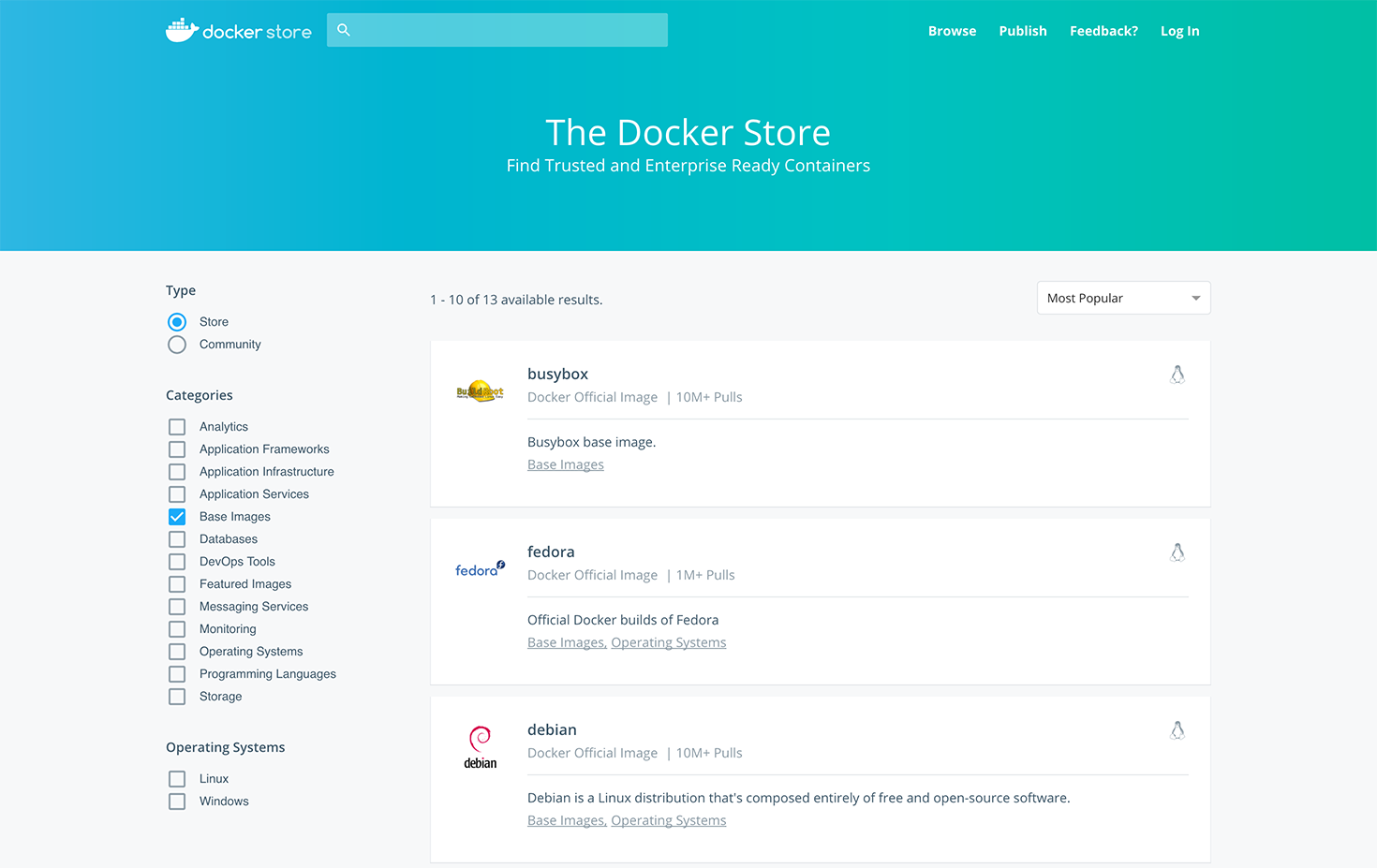
Docker image

도커에서 가장 중요한 개념은 컨테이너와 함께 이미지라는 개념입니다.

이미지는 **컨테이너 실행에 필요한 파일과 설정값등을 포함하고 있는 것**으로 상태값을 가지지 않고 변하지 않습니다(Immutable). 컨테이너는 이미지를 실행한 상태라고 볼 수 있고 추가되거나 변하는 값은 컨테이너에 저장됩니다. 같은 이미지에서 여러개의 컨테이너를 생성할 수 있고 컨테이너의 상태가 바뀌거나 컨테이너가 삭제되더라도 이미지는 변하지 않고 그대로 남아있습니다.

ubuntu이미지는 ubuntu를 실행하기 위한 모든 파일을 가지고 있고 MySQL이미지는 debian을 기반으로 MySQL을 실행하는데 필요한 파일과 실행 명령어, 포트 정보등을 가지고 있습니다. 좀 더 복잡한 예로 Gitlab 이미지는 centos를 기반으로 ruby, go, database, redis, gitlab source, nginx등을 가지고 있습니다.

말그대로 이미지는 컨테이너를 실행하기 위한 모오오오오든 정보를 가지고 있기 때문에 더 이상 의존성 파일을 컴파일하고 이것저것 설치할 필요가 없습니다. 이제 새로운 서버가 추가되면 미리 만들어 놓은 이미지를 다운받고 컨테이너를 생성만 하면 됩니다. 한 서버에 여러개의 컨테이너를 실행할 수 있고, 수십, 수백, 수천대의 서버도 문제없습니다.

Docker Store

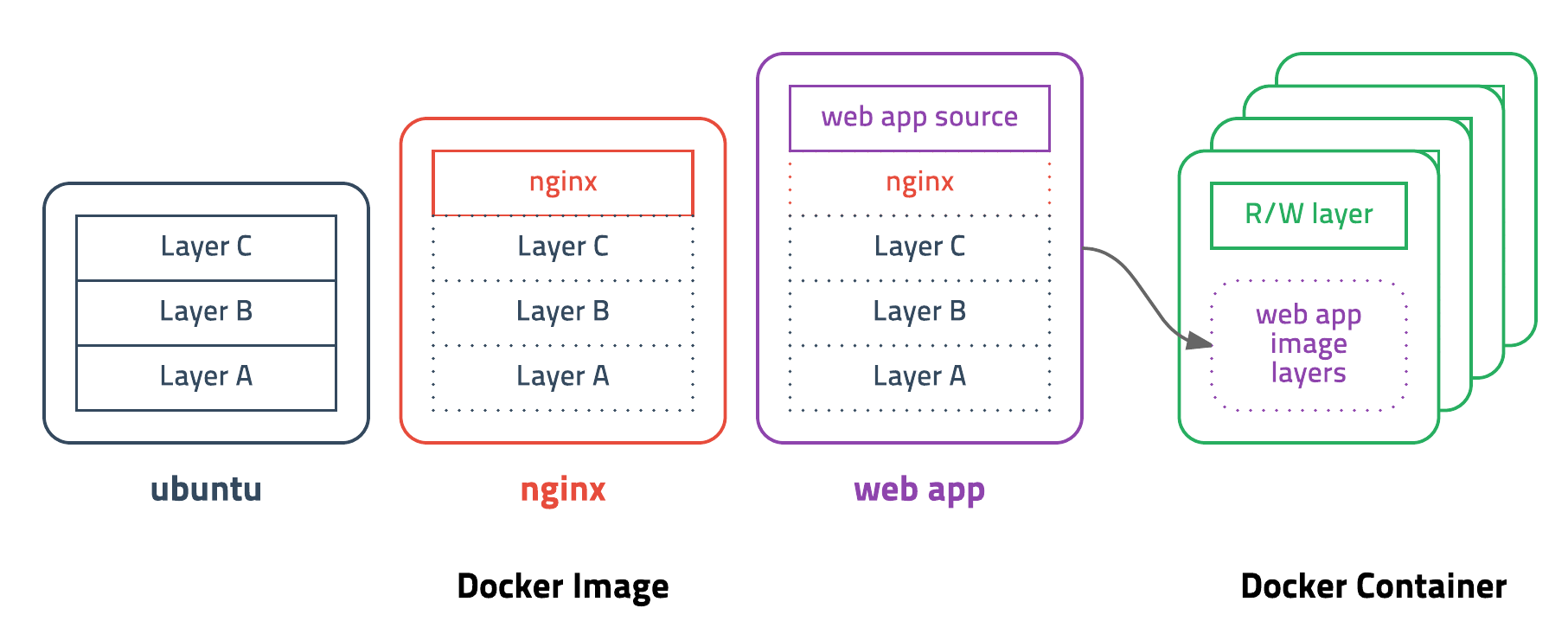
도커 이미지는 [Docker hub](https://hub.docker.com/)에 등록하거나 [Docker Registry](https://docs.docker.com/registry/) 저장소를 직접 만들어 관리할 수 있습니다. 현재 공개된 도커 이미지는 50만개가 넘고 Docker hub의 이미지 다운로드 수는 80억회에 이릅니다. 누구나 쉽게 이미지를 만들고 배포할 수 있습니다.

## 왜 이렇게 핫한가?

도커는 완전히 새로운 기술이 아니며 이미 존재하는 기술을 잘 포장했다고 볼 수 있습니다.

컨테이너, [오버레이 네트워크](https://en.wikipedia.org/wiki/Overlay_network)overlay network, [유니온 파일 시스템](https://en.wikipedia.org/wiki/UnionFS)union file systems등 이미 존재하는 기술을 도커처럼 잘 조합하고 사용하기 쉽게 만든 것은 없었고 사용자들이 원하는 기능을 간단하지만 획기적인 아이디어로 구현하였습니다.

### **레이어 저장방식**

Docker Layer

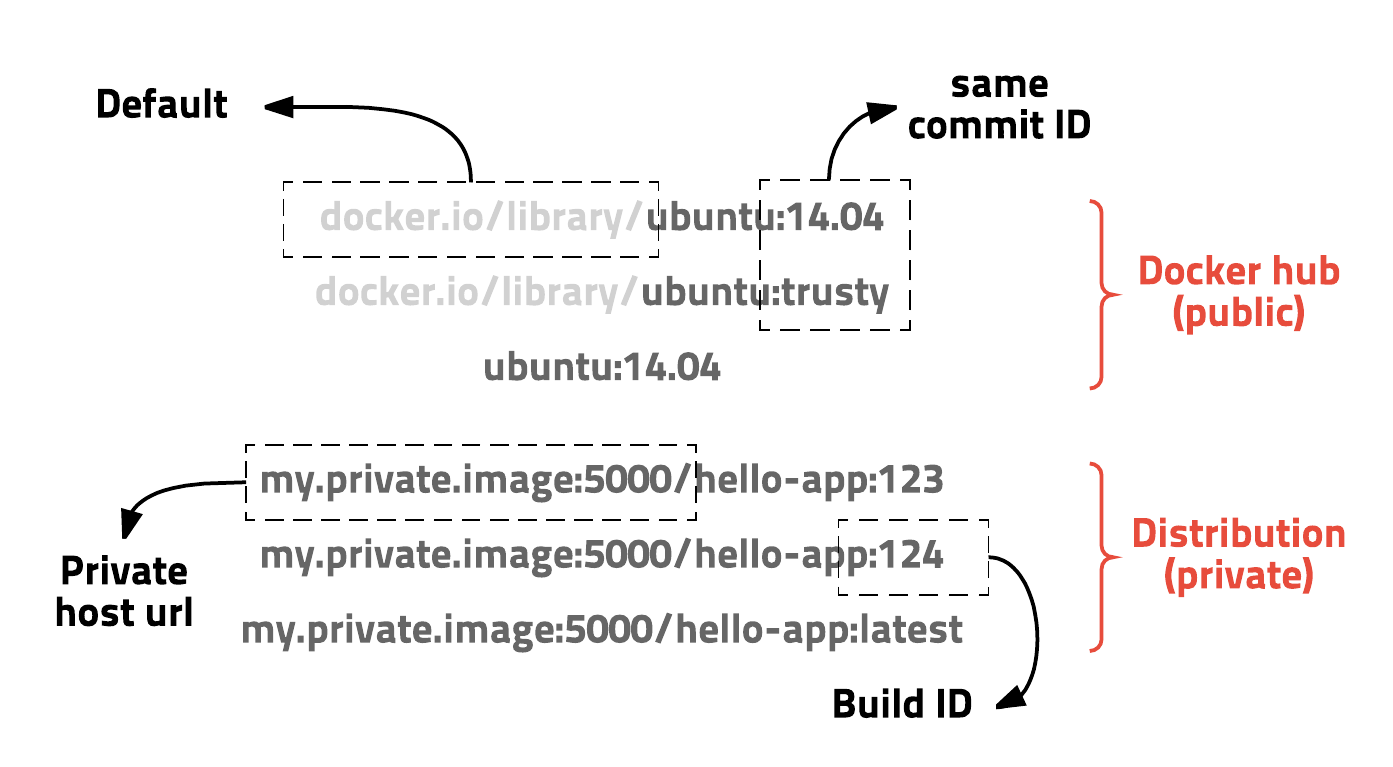
도커 이미지는 컨테이너를 실행하기 위한 모든 정보를 가지고 있기 때문에 보통 용량이 수백메가MB에 이릅니다. 처음 이미지를 다운받을 땐 크게 부담이 안되지만 기존 이미지에 파일 하나 추가했다고 수백메가를 다시 다운받는다면 매우 비효율적일 수 밖에 없습니다.

도커는 이런 문제를 해결하기 위해 **레이어layer**라는 개념을 사용하고 유니온 파일 시스템을 이용하여 여러개의 레이어를 하나의 파일시스템으로 사용할 수 있게 해줍니다. 이미지는 여러개의 읽기 전용read only 레이어로 구성되고 파일이 추가되거나 수정되면 새로운 레이어가 생성됩니다. ubuntu 이미지가 A + B + C의 집합이라면, ubuntu 이미지를 베이스로 만든 nginx 이미지는 A + B + C + nginx가 됩니다. webapp 이미지를 nginx 이미지 기반으로 만들었다면 예상대로 A + B + C + nginx + source 레이어로 구성됩니다. webapp 소스를 수정하면 A, B, C, nginx 레이어를 제외한 새로운 source(v2) 레이어만 다운받으면 되기 때문에 굉장히 효율적으로 이미지를 관리할 수 있습니다.

컨테이너를 생성할 때도 레이어 방식을 사용하는데 기존의 이미지 레이어 위에 읽기/쓰기read-write 레이어를 추가합니다. 이미지 레이어를 그대로 사용하면서 컨테이너가 실행중에 생성하는 파일이나 변경된 내용은 읽기/쓰기 레이어에 저장되므로 여러개의 컨테이너를 생성해도 최소한의 용량만 사용합니다.

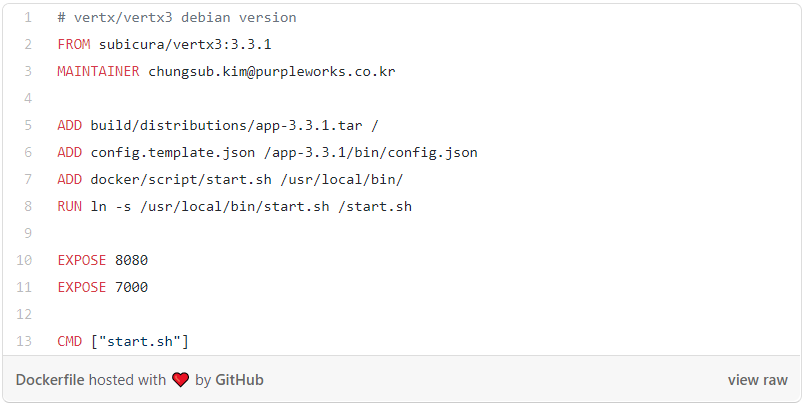
가상화의 특성상 이미지 용량이 크고 여러대의 서버에 배포하는걸 감안하면 단순하지만 엄청나게 영리한 설계입니다.

### **이미지 경로**

Docker image url

이미지는 url 방식으로 관리하며 태그를 붙일 수 있습니다. ubuntu 14.04 이미지는 docker.io/library/ubuntu:14.04 또는 docker.io/library/ubuntu:trusty 이고 docker.io/library는 생략가능하여 ubuntu:14.04 로 사용할 수 있습니다. 이러한 방식은 이해하기 쉽고 편리하게 사용할 수 있으며 태그 기능을 잘 이용하면 테스트나 롤백도 쉽게 할 수 있습니다.

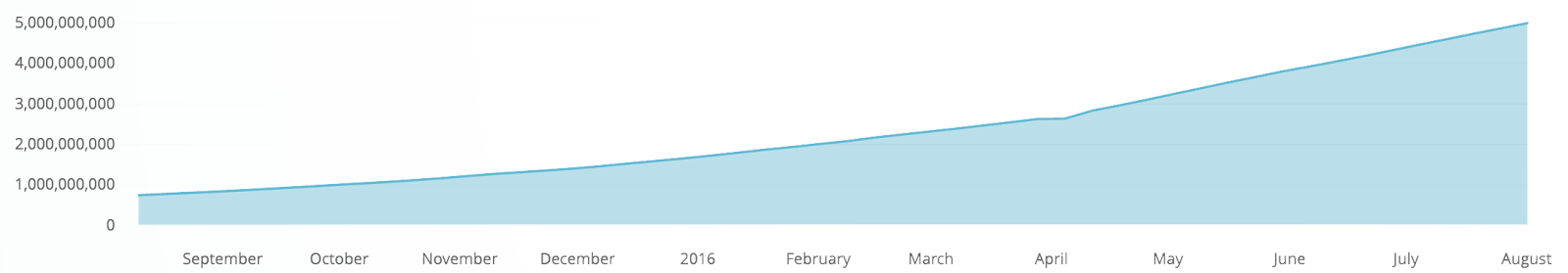
### **Dockerfile**



도커는 이미지를 만들기 위해 Dockerfile이라는 파일에 자체 DSLDomain-specific language언어를 이용하여 이미지 생성 과정을 적습니다. 추후에 문법에 대해 자세히 다루겠지만 위 샘플을 보면 그렇게 복잡하지 않다는 걸 알 수 있습니다.

이것은 굉장히 간단하지만 유용한 아이디어인데, 서버에 어떤 프로그램을 설치하려고 이것저것 의존성 패키지를 설치하고 설정파일을 만들었던 경험이 있다면 더 이상 그 과정을 블로깅 하거나 메모장에 적지 말고 Dockerfile로 관리하면 됩니다. 이 파일은 소스와 함께 버전 관리 되고 원한다면 누구나 이미지 생성과정을 보고 수정할 수 있습니다.

### **Docker Hub**

Docker Hub Hits 5 Billion Pulls(2016/08)

도커 이미지의 용량은 보통 수백메가로 수기가가 넘는 경우도 흔합니다. 이렇게 큰 용량의 이미지를 서버에 저장하고 관리하는 것은 쉽지 않은데 도커는 Docker hub를 통해 공개 이미지를 무료로 관리해 줍니다. 하루에도 엄청난 용량의 이미지가 전세계에서 다운로드 되고 트래픽 비용만 해도 어마어마 할 것 같은데 그것이 다 무료!입니다.

### **Command와 API**

도커 클라이언트의 커맨드 명령어는 정말 자아아알 만들어져 있습니다. 대부분의 명령어는 직관적이고 사용하기 쉬우며 컨테이너의 복잡한 시스템 구성을 이해하지 못하더라도 편하게 사용할 수 있습니다. 또한 http기반의 Rest API도 지원하여 확장성이 굉장히 좋고 훌륭한 3rd party 툴이 나오기 좋은 환경입니다.

### **유용한 새로운 기능들**

도커는 발전속도가 아주 빠른 오픈소스입니다. 사용하면서 부족하다고 느꼈던 부분은 빠르게 개선되고 새로운 버전이 나오면 유용한 기능이 대폭 추가됩니다. 어떻게 보면 프로그램을 작은 조각으로 나누고 여러개의 프로그램을 조합하여 동작시키는 유닉스의 철학에는 맞지 않지만.. 너무 잘합니다. ㅠㅠ

이번 [1.13버전](https://blog.docker.com/2017/01/whats-new-in-docker-1-13/)에서는 [Docker stacks](https://github.com/docker/docker/blob/master/experimental/docker-stacks-and-bundles.md)이라는 여러개의 컨테이너를 한번에 관리하는 기능이 정식으로 릴리즈 되었고 [system](https://docs.docker.com/engine/reference/commandline/system/) 커맨드가 추가되어 이미지, 컨테이너 관리가 더 편해졌습니다. [Secrets Management](https://github.com/docker/docker/pull/27794)라는 비밀정보를 관리하는 기능도 추가됩니다.

새로운 기능이 계속 추가되고 있고 다음 릴리즈가 기대됩니다.

### **훌륭한 생태계**

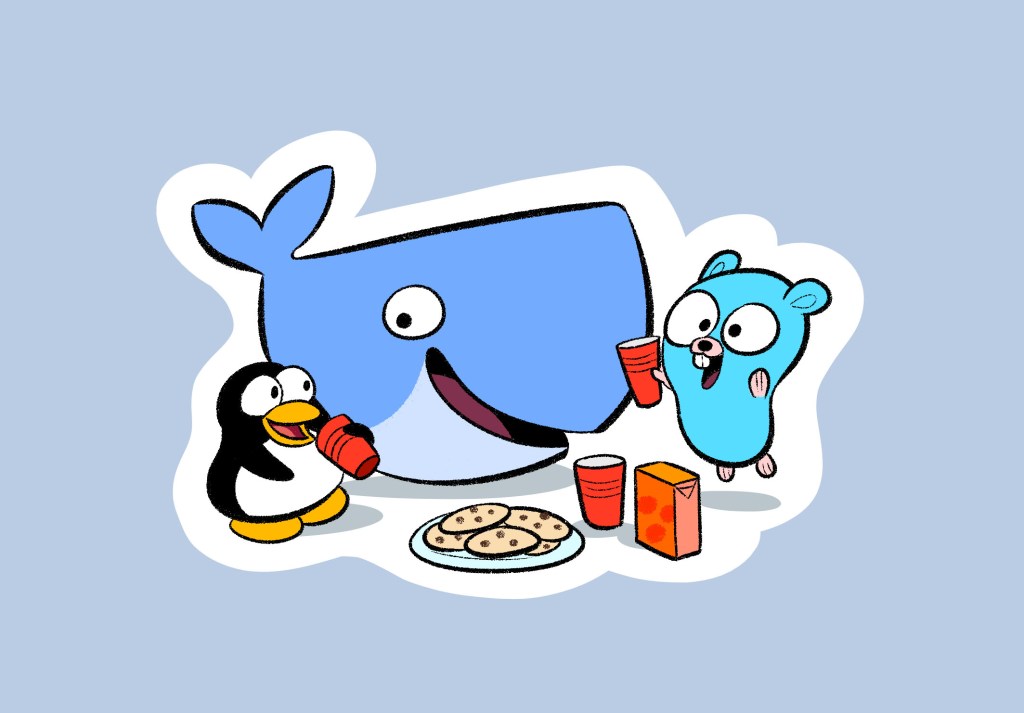
도커는 굉장히 큰 생태계를 가지고 있고 커다란 기업과 협력하여 사실상 클라우드 컨테이너 세계의 [de facto](https://ko.wikipedia.org/wiki/%EC%82%AC%EC%8B%A4%EC%83%81_%ED%91%9C%EC%A4%80)가 되었습니다. 로깅, 모니터링, 스토리지, 네트워크, 컨테이너 관리, 배포등 다양한 분야에서 다양한 툴들이 존재하며 아예 [도커를 위한 OS(coreos-> container linux)](https://coreos.com/why/)도 존재합니다.

현재 도커를 기반으로한 오픈소스 프로젝트는 10만개가 넘고 굉장히 활발하게 진행되고 있습니다.

### **커뮤니티 지원**

도커는 기술기업답지 않게 홍보와 커뮤니티 관리에 굉장히 신경쓰고 있습니다. 커뮤니티를 위한 스티커나 티셔츠를 무료로 제공하고 필요하면 연사요청도 할 수 있습니다. 홈페이지에서는 전세계에서 열리는 밋업 상황을 볼 수 있고 일주일마다 발송되는 [뉴스레터](https://blog.docker.com/docker-weekly-archives/)에는 다양한 개발자들의 글이 실려있습니다.

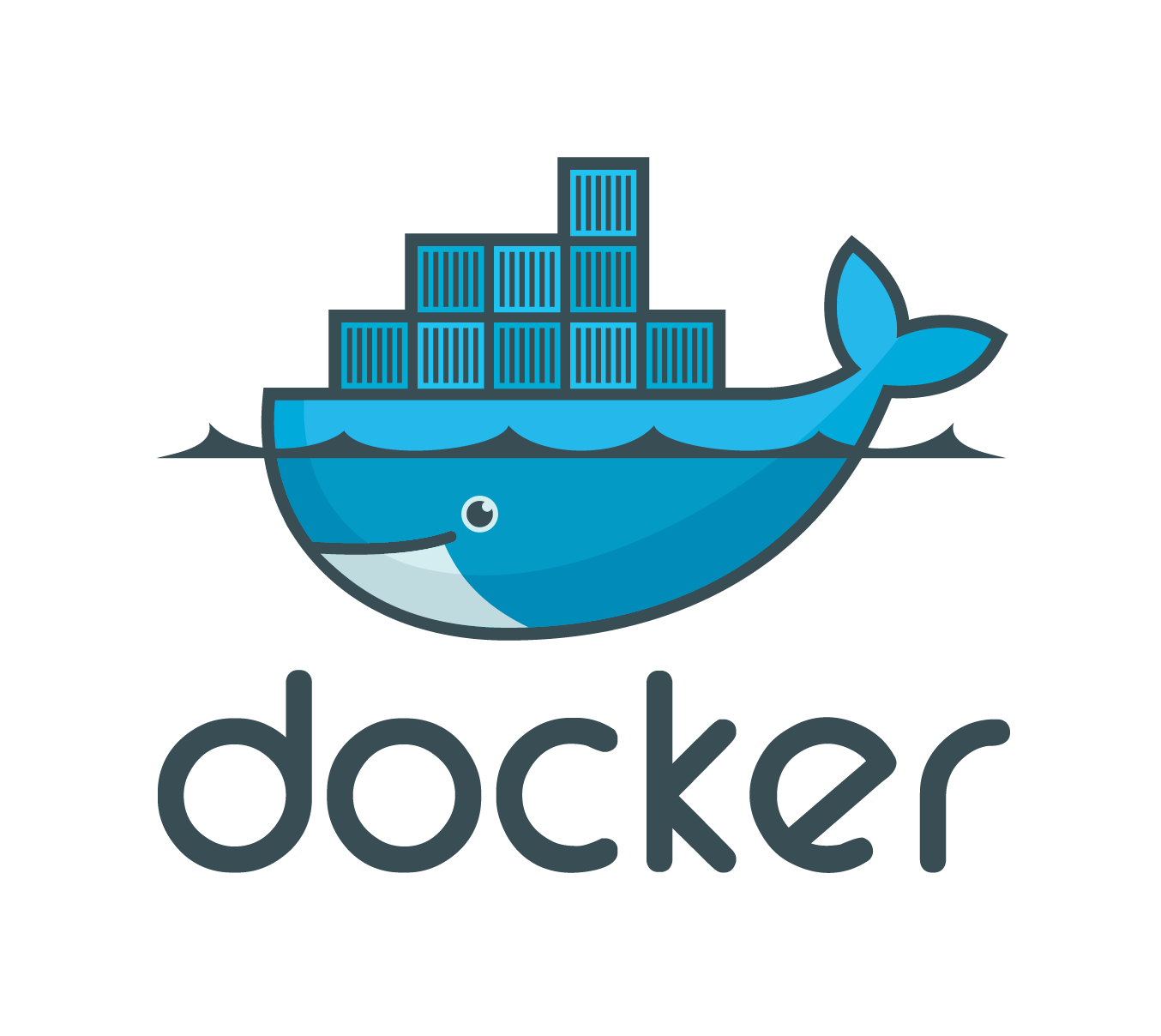
### **moby dock**



Tux(linux) - Moby Dock(docker) - Gopher (golang)

도커는 넘나 귀여운 고래를 로고로 하고 있습니다. 로고 스티커는 항상 인기가 넘치고 로고가 그려진 티셔츠는 입고 돌아다녀도 개발자처럼 보이지 않습니다. 도커가 성공한 가장 큰 이유는 귀여운 고래 덕분이라고 생각합니다.(엄근진)

<https://subicura.com/2017/01/19/docker-guide-for-beginners-2.html>



Docker logo

**도커의 모든 명령어 : https://docs.docker.com/engine/reference/commandline/docker/**

**도커 설치하기**

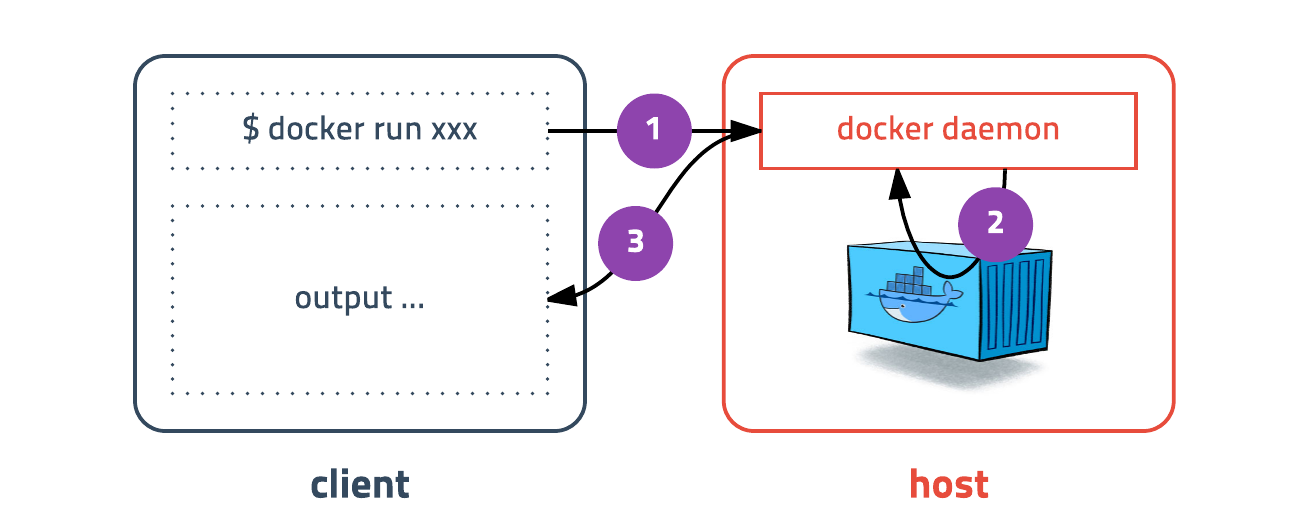
도커는 리눅스 컨테이너 기술이므로 macOS나 windows에 설치할 경우 가상머신에 설치가 됩니다.

**주의사항**

* 도커를 실행하기 위한 kernel 버전은 3.10.x 이상입니다. ubuntu 14.04 이상을 사용하면 큰 문제가 없고 kernel의 버전이 낮을 경우 제대로 동작을 안하거나 문제가 생길 수 있습니다. 가급적 최신버전으로 업데이트 해주세요.
* ubuntu나 centos가 아닌 경우는 다른 방법이 필요합니다. 다른 리눅스를 쓰시는 분은 대부분 고오오급 개발자 분이시니 따로 설명하지 않아도 될 것 같아 링크(https://docs.docker.com/engine/install/)로 대신하겠습니다.

Server 정보가 안나오고 Error response from daemon: Bad response from Docker engine이라는 메시지가 출력되는 경우는 보통 docker daemon이 정상적으로 실행되지 않았거나 sudo를 입력하지 않은 경우입니다.

버전정보가 클라이언트와 서버로 나뉘어져 있습니다. 도커는 하나의 실행파일이지만 실제로 클라이언트와 서버역할을 각각 할 수 있습니다. 도커 커맨드를 입력하면 도커 클라이언트가 도커 서버로 명령을 전송하고 결과를 받아 터미널에 출력해 줍니다.



Docker client-host

기본값이 도커 서버의 소켓을 바라보고 있기 때문에 사용자는 의식하지 않고 마치 바로 명령을 내리는 것 같은 느낌을 받습니다. 이러한 설계가 mac이나 windows의 터미널에서 명령어를 입력했을때 가상 서버에 설치된 도커가 동작하는 이유입니다.

**컨테이너 실행하기**

도커를 실행하는 명령어는 다음과 같습니다.

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

다음은 자주 사용하는 옵션들입니다.

| **옵션** | **설명** |
| --- | --- |
| -d | detached mode 흔히 말하는 백그라운드 모드 |
| -p | 호스트와 컨테이너의 포트를 연결 (포워딩) |
| -v | 호스트와 컨테이너의 디렉토리를 연결 (마운트) |
| -e | 컨테이너 내에서 사용할 환경변수 설정 |
| –name | 컨테이너 이름 설정 |
| –rm | 프로세스 종료시 컨테이너 자동 제거 |
| -it | -i와 -t를 동시에 사용한 것으로 터미널 입력을 위한 옵션 |
| –link | 컨테이너 연결 [컨테이너명:별칭] |

엄청나게 직관적인 옵션으로 몇번 실행해보면 자연스럽게 익숙해집니다.

### **ubuntu 16.04 container**

### **redis container**

### **MySQL 5.7 container (**[**https://hub.docker.com/\_/mysql/**](https://hub.docker.com/_/mysql/)**)**

### **WordPress container**

### **Tensorflow (**[**https://www.tensorflow.org/**](https://www.tensorflow.org/)**)**

: 손쉽게 머신러닝을 할 수 있는 툴입니다. tensorflow는 python으로 만들어져 python과 관련 패키지를 설치해야 합니다.

## 도커 기본 명령어

앞에서 도커의 run 명령어를 이용하여 여러개의 컨테이너를 실행했습니다. 이제 컨테이너의 상태를 살펴보고 어떤 이미지가 설치되어 있는지 확인하는 명령어를 알아봅니다.

### **컨테이너 목록 확인하기 (ps)**

컨테이너 목록을 확인하는 명령어는 다음과 같습니다.

docker ps [OPTIONS]

일단 기본옵션과 -a, --all 옵션만 살펴봅니다.

docker ps

ps 명령어는 실행중인 컨테이너 목록을 보여줍니다. detached mode로 실행중인 컨테이너들이 보입니다. 어떤 이미지를 기반으로 만들었는지 어떤 포트와 연결이 되어있는지 등 간단한 내용을 보여줍니다.

이번에는 -a 옵션을 추가로 실행해보겠습니다.

docker ps -a

맨 처음 실행했다가 종료된 컨테이너(Exited (0))가 추가로 보입니다. 컨테이너는 종료되어도 삭제되지 않고 남아있습니다. 종료된 건 다시 시작할 수 있고 컨테이너의 읽기/쓰기 레이어는 그대로 존재합니다. 명시적으로 삭제를 하면 깔끔하게 컨테이너가 제거됩니다.

### **컨테이너 중지하기 (stop)**

실행중인 컨테이너를 중지하는 명령어는 다음과 같습니다.

docker stop [OPTIONS] CONTAINER [CONTAINER...]

옵션은 특별한게 없고 실행중인 컨테이너를 하나 또는 여러개 (띄어쓰기로 구분) 중지할 수 있습니다.

앞에서 실행한 tensorflow 컨테이너는 더이상 필요가 없으니 중지해 보겠습니다. 중지하려면 컨테이너의 ID 또는 이름을 입력하면 됩니다. tensorflow 컨테이너의 ID를 ps명령을 통해 확인하고 중지해 봅니다.

도커 ID의 전체 길이는 64자리 입니다. 하지만 명령어의 인자로 전달할 때는 전부 입력하지 않아도 됩니다. 예를 들어 ID가 abcdefgh...라면 abcd만 입력해도 됩니다. 앞부분이 겹치지 않는다면 1-2자만 입력해도 됩니다.

잠시 기다리면 tensorflow 컨테이너가 종료됩니다. ps -a명령어를 입력하여 종료되었는지 확인합니다.

### **컨테이너 제거하기 (rm)**

종료된 컨테이너를 완전히 제거하는 명령어는 다음과 같습니다.

docker rm [OPTIONS] CONTAINER [CONTAINER...]

종료 명령어도 옵션은 특별한게 없습니다. 종료된 컨테이너를 하나 또는 여러개 삭제할 수 있습니다. 종료된 ubuntu 컨테이너와 tensorflow 컨테이너를 삭제해보겠습니다.

|  |  |
| --- | --- |
| 1  2  3 | docker ps -a *# get container ID*  docker rm **${**UBUNTU\_CONTAINER\_ID**}** **${**TENSORFLOW\_CONTAINER\_ID**}**  docker ps -a *# check exist* |

컨테이너가 말끔히 삭제되었습니다. 호스트 OS는 아무런 흔적도 남아있지 않고 컨테이너만 격리된 상태로 실행되었다가 삭제되었습니다. 시스템이 꼬일 걱정이 없습니다.

중지된 컨테이너를 일일이 삭제 하는 건 귀찮은 일입니다. docker rm -v $(docker ps -a -q -f status=exited) 명령어를 입력하면 중지된 컨테이너 ID를 가져와서 한번에 삭제합니다.

### **이미지 목록 확인하기 (images)**

도커가 다운로드한 이미지 목록을 보는 명령어는 다음과 같습니다.

docker images [OPTIONS] [REPOSITORY[:TAG]]

간단하게 도커 이미지 목록을 확인해보겠습니다.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | | | docker images |
|  |  |

이미지 주소와 태그, ID, 생성시점, 용량이 보입니다. 이미지가 너무 많이 쌓이면 용량을 차지하기 때문에 사용하지 않는 이미지는 지우는 것이 좋습니다.

### **이미지 다운로드하기 (pull)**

이미지를 다운로드하는 명령어는 다음과 같습니다.

docker pull [OPTIONS] NAME[:TAG|@DIGEST]

ubuntu:14.04를 다운받아보겠습니다.

|  |  |
| --- | --- |
| 1 | docker pull ubuntu:14.04 |

run명령어를 입력하면 이미지가 없을 때 자동으로 다운받으니 pull명령어를 언제 쓰는지 궁금할 수 있는데 pull은 최신버전으로 다시 다운 받습니다. 같은 태그지만 이미지가 업데이트 된 경우는 pull명령어를 통해 새로 다운받을 수 있습니다.

### **이미지 삭제하기 (rmi)**

이미지를 삭제하는 방법은 다음과 같습니다.

docker rmi [OPTIONS] IMAGE [IMAGE...]

images명령어를 통해 얻은 이미지 목록에서 이미지 ID를 입력하면 삭제가 됩니다. 단, 컨테이너가 실행중인 이미지는 삭제되지 않습니다. 컨테이너는 이미지들의 레이어를 기반으로 실행중이므로 당연히 삭제할 수 없습니다.

tensorflow는 더 이상 사용하지 않으니 이미지를 제거해보겠습니다.

|  |  |
| --- | --- |
| 1  2 | docker images *# get image ID*  docker rmi **${**TENSORFLOW\_IMAGE\_ID**}** |

이미지는 여러개의 레이어로 구성되어 있기 때문에 모든 레이어가 삭제된 것을 알 수 있습니다.

## 컨테이너 둘러보기

도커에 대한 아주아주아주 기본적인 명령어를 살펴보았습니다. 사실 저 명령어들과 이번에 살펴볼 log, exec 명령어를 익히면 도커에서 사용하는 명령어는 거의 다 익혔다고 할 수 있습니다. 다른 명령어는 필요에 따라 하나하나 살펴보면 됩니다.

### **컨테이너 로그 보기 (logs)**

컨테이너가 정상적으로 동작하는지 확인하는 좋은 방법은 로그를 확인하는 것 입니다. 로그를 확인하는 방법은 다음과 같습니다.

docker logs [OPTIONS] CONTAINER

기본 옵션과, -f, --tail 옵션을 살펴봅니다.

기존에 생성해 놓은 워드프레스 컨테이너 로그를 확인해 보겠습니다.

|  |  |
| --- | --- |
| 1  2 | docker ps  docker logs **${**WORDPRESS\_CONTAINER\_ID**}** |

옵션을 주지 않았을 때는 전체 로그를  전부 다 출력합니다. 너무 많으니 --tail옵션으로 마지막 10줄만 출력해 보겠습니다.

|  |  |
| --- | --- |
| 1 | docker logs --tail 10 **${**WORDPRESS\_CONTAINER\_ID**}** |

마지막 10줄만 보니 좀 나아 보입니다. 이제 실시간으로 로그가 생성되는 걸 확인해보겠습니다. -f 옵션으로 실행합니다.

|  |  |
| --- | --- |
| 1 | docker logs -f **${**WORDPRESS\_CONTAINER\_ID**}** |

로그를 켜 놓은 상태에서 워드프레스 페이지를 새로고침하면 브라우저 접속 로그가 실시간으로 보입니다. 가장 흔하게 사용하는 옵션이고 로그 보기를 중지하려면 ctrl + c를 입력하면 됩니다.

**로그에 대해 좀 더 자세히**

프로그램마다 로그 파일은 제각각 생길텐데 어떻게 저 로그가 나올까 라는 의문이 생깁니다. 도커는 로그파일을 자동으로 알아채는게 아니라 [표준 스트림](https://en.wikipedia.org/wiki/Standard_streams)Standard streams 중 stdout, stderr를 수집합니다. 따라서 컨테이너에서 실행되는 프로그램의 로그 설정을 파일이 아닌 표준출력으로 바꾸어야 합니다. 단지 출력 방식만 바꾸는 것으로 모든 컨테이너는 로그에 대해 같은 방식으로 관리할 수 있게 됩니다.

또하나 중요한 점은 컨테이너의 로그파일은 json 방식으로 어딘가에 저장이 됩니다. 로그가 많으면 은근히 파일이 차지하는 용량이 커지므로 주의해야합니다. 도커는 다양한 플러그인을 지원하여 json이 아닌 특정 로그 서비스에 스트림을 전달할 수 있습니다. 어느 정도 앱의 규모가 커지면 기본적인 방식 대신 로그 서비스를 이용하는 걸 고려해야 합니다.

### **컨테이너 명령어 실행하기 (exec)**

컨테이너를 관리하다 보면 실행중인 컨테이너에 들어가보거나 컨테이너의 파일을 실행하고 싶을 때가 있습니다. 컨테이너에 SSH를 설치하면 되지 않을까? 라고 생각할 수 있지만 SSH는 권장하지 않습니다.  예전에는 [nsenter](https://github.com/jpetazzo/nsenter)라는 프로그램을 이용하였는데 docker에 exec라는 명령어로 흡수되었습니다.

컨테이너 명령어를 실행하는 방법은 다음과 같습니다.

docker exec [OPTIONS] CONTAINER COMMAND [ARG...]

run 명령어와 유사해 보입니다. 차이는 run은 새로 컨테이너를 만들어서 실행하고 exec는 실행중인 컨테이너에 명령어를 내리는 정도입니다.

일단, 가볍게 실행중인 MySQL 컨테이너에 접속해보겠습니다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | docker exec -it mysql /bin/bash  *# MySQL test*  $ mysql -uroot  mysql> show databases;  +--------------------+  | Database |  +--------------------+  | information\_schema |  | mysql |  | performance\_schema |  | sys |  | wp |  +--------------------+  5 rows **in** set **(**0.00 sec**)**  mysql> quit  exit |

키보드 입력이 필요하니 run 명령어와 마찬가지로 -it 옵션을 주었고 bash 쉘로 접속하여 마치 가상머신에 들어온 것 같은 느낌이 듭니다. 접속한 이후에는 어떤 작업도 할 수 있고 컨테이너를 마음껏 건드릴 수 있습니다.

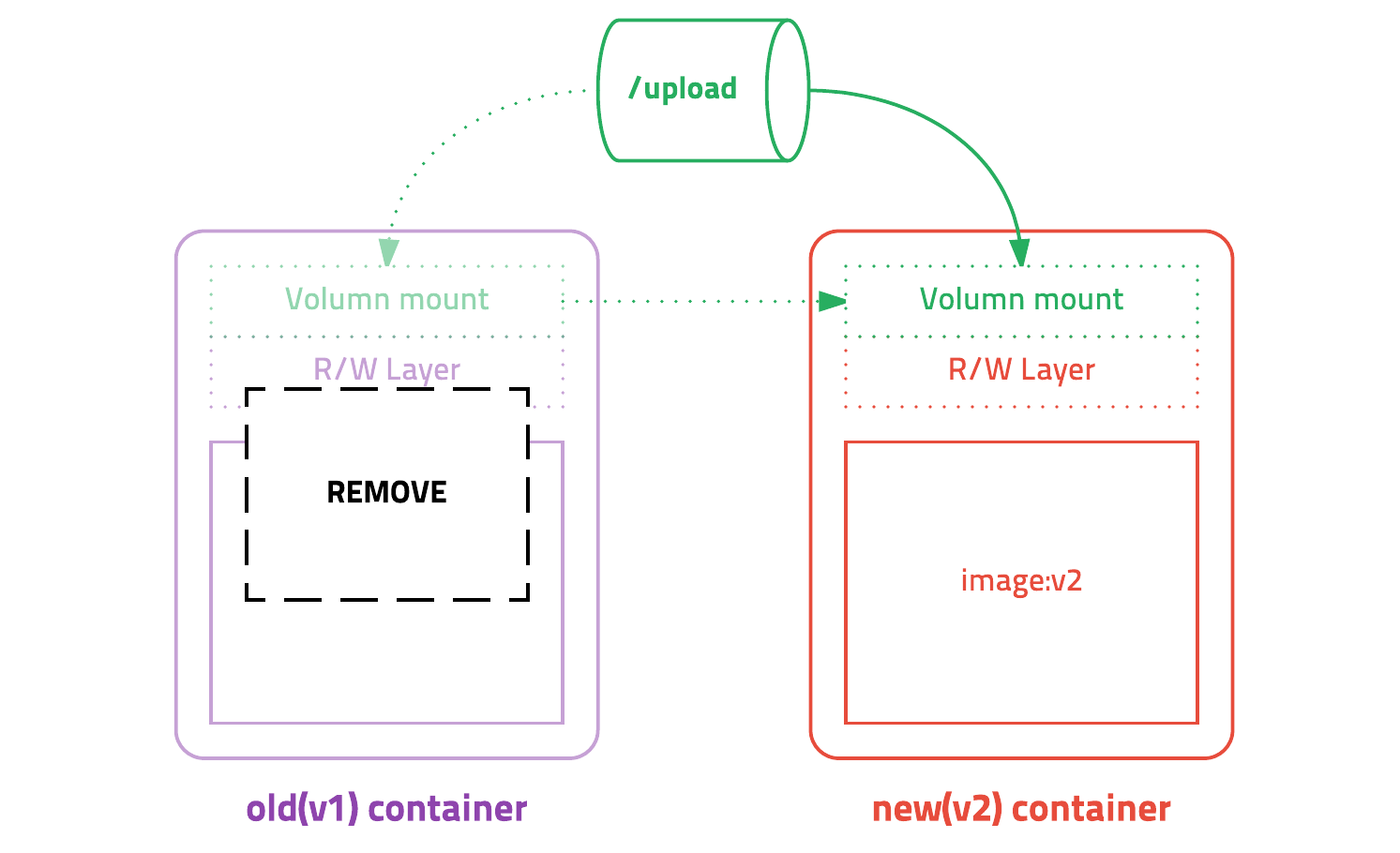
쉘로 완전한 권한을 얻는 방법말고 바로 mysql명령어를 실행 할 수도 있습니다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | docker exec -it mysql mysql -uroot  *# MySQL test*  $ mysql -uroot  mysql> show databases;  +--------------------+  | Database |  +--------------------+  | information\_schema |  | mysql |  | performance\_schema |  | sys |  | wp |  +--------------------+  5 rows **in** set **(**0.00 sec**)**  mysql> quit |

이제, 호스트 OS에 mysql을 설치하지 않아도 mysql 클라이언트를 사용할 수 있습니다. 굳이 복잡한 작업이 필요 없는 경우는 -it 옵션없이 단순하게 명령을 실행하고 종료할 수도 있습니다.

## 컨테이너 업데이트

이제 지금까지 배운 모든걸 정리해서 컨테이너를 새로운 버전으로 업데이트 하는 과정을 살펴보겠습니다.

도커 컨테이너 업데이트

도커에서 컨테이너를 업데이트 하려면 새 버전의 이미지를 다운(pull)받고 기존 컨테이너를 삭제(stop, rm) 한 후 새 이미지를 기반으로 새 컨테이너를 실행(run)하면 됩니다. 배포와 관련된 자세한 사항은 다음글에서 이야기하고 여기선 그냥 그렇구나 하고 이해합시다.

컨테이너를 삭제한다는 건 컨테이너에서 생성된 파일이 사라진다는 뜻입니다. 데이터베이스라면 그동안 쌓였던 데이터가 모두 사라진다는 것이고 웹 어플리케이션이라면 그동안 사용자가 업로드한 이미지가 모두 사라진다는 것입니다.

**이런 상황을 방지하기 위해 컨테이너 삭제시 유지해야하는 데이터는 반드시 컨테이너 내부가 아닌 외부 스토리지에 저장해야 합니다.** 가장 좋은 방법은 [AWS S3](http://docs.aws.amazon.com/ko_kr/AmazonS3/latest/dev/Welcome.html)같은 클라우드 서비스를 이용하는 것이고 그렇지 않으면 데이터 볼륨Data volumes을 컨테이너에 추가해서 사용해야 합니다. 데이터 볼륨을 사용하면 해당 디렉토리는 컨테이너와 별도로 저장되고 컨테이너를 삭제해도 데이터가 지워지지 않습니다.

데이터 볼륨을 사용하는 방법은 몇가지가 있는데 여기서는 호스트의 디렉토리를 마운트해서 사용하는 방법에 대해 알아봅니다. run명령어에서 소개한 옵션중에 -v 옵션을 드디어 사용해 보겠습니다. MySQL이라면 /var/lib/mysql디렉토리에 모든 데이터베이스 정보가 담기므로 호스트의 특정 디렉토리를 연결해주면 됩니다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | *# before*  docker run -d -p 3306:3306 \  -e MYSQL\_ALLOW\_EMPTY\_PASSWORD**=**true \  --name mysql \  mysql:5.7  *# after*  docker run -d -p 3306:3306 \  -e MYSQL\_ALLOW\_EMPTY\_PASSWORD**=**true \  --name mysql \  -v /my/own/datadir:/var/lib/mysql \ *# <- volume mount*  mysql:5.7 |

위 샘플은 호스트의 /my/own/datadir디렉토리를 컨테이너의 /var/lib/mysql디렉토리로 마운트 하였습니다. 이제 데이터베이스 파일은 호스트의 /my/own/datadir디렉토리에 저장되고 컨테이너를 삭제해도 데이터는 사라지지 않습니다. 최신버전의 MySQL 이미지를 다운받고 다시 컨테이너를 실행할 때 동일한 디렉토리를 마운트 한다면 그대로 데이터를 사용할 수 있습니다.

## Docker Compose

지금까지 도커를 커맨드라인에서 명령어로 작업했습니다. 지금은 간단한 작업만 했기 때문에 명령이 길지 않지만 컨테이너 조합이 많아지고 여러가지 설정이 추가되면 명령어가 금방 복잡해집니다.

도커는 복잡한 설정을 쉽게 관리하기 위해 [YAML](https://en.wikipedia.org/wiki/YAML)방식의 설정파일을 이용한 [Docker Compose](https://docs.docker.com/compose/)라는 툴을 제공합니다. 깊게 파고들면 은근 기능이 많고 복잡한데 이번에는 아주 가볍게 다루고 지나가도록 하겠습니다.

|  |
| --- |
| https://en.wikipedia.org/wiki/YAML  **YAML** ( **Yet Another Multicolumn Layout** )은 [브라우저 간](https://en.wikipedia.org/wiki/Cross-browser)[CSS 프레임 워크](https://en.wikipedia.org/wiki/CSS_framework) 입니다.  **YAML** ( "YAML Ai n't Markup Language" 의 [재귀 적 약어](https://en.wikipedia.org/wiki/Recursive_acronym) )은 [사람이 읽을 수있는](https://en.wikipedia.org/wiki/Human-readable)[데이터 직렬화 언어](https://en.wikipedia.org/wiki/Serialization) 입니다. 일반적으로 [구성 파일](https://en.wikipedia.org/wiki/Configuration_file) 및 데이터가 저장 또는 전송되는 응용 프로그램에 사용됩니다. YAML은 XML ( [Extensible Markup Language)](https://en.wikipedia.org/wiki/Extensible_Markup_Language) 과 동일한 많은 통신 애플리케이션을 대상으로 하지만 [SGML](https://en.wikipedia.org/wiki/SGML) 과 의도적으로 다른 최소한의 구문을 가지고 있습니다. |

### **설치하기**

Docker for Mac 또는 Docker for Windows를 설치했다면 자동으로 설치됩니다. 리눅스의 경우 다음 명령어를 입력하여 설치합니다. 그냥 설치파일 하나 다운받으면 됩니다.

|  |  |
| --- | --- |
| 1  2  3  4 | curl -L "https://github.com/docker/compose/releases/download/1.9.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose  chmod +x /usr/local/bin/docker-compose  *# test*  docker-compose version |

### **wordpress 만들기**

기존에 명령어로 만들었던 wordpress를 compose를 이용해 만들어 보겠습니다.

먼저 빈 디렉토리를 하나 만들고 docker-compose.yml파일을 만들어 설정을 입력합니다.

|  |  |
| --- | --- |
|  | version: '2' |
|  |  |
|  | services: |
|  | db: |
|  | image: mysql:5.7 |
|  | volumes: |
|  | - db\_data:/var/lib/mysql |
|  | restart: always |
|  | environment: |
|  | MYSQL\_ROOT\_PASSWORD: wordpress |
|  | MYSQL\_DATABASE: wordpress |
|  | MYSQL\_USER: wordpress |
|  | MYSQL\_PASSWORD: wordpress |
|  |  |
|  | wordpress: |
|  | depends\_on: |
|  | - db |
|  | image: wordpress:latest |
|  | volumes: |
|  | - wp\_data:/var/www/html |
|  | ports: |
|  | - "8000:80" |
|  | restart: always |
|  | environment: |
|  | WORDPRESS\_DB\_HOST: db:3306 |
|  | WORDPRESS\_DB\_PASSWORD: wordpress |
|  | volumes: |
|  | db\_data: |
|  | wp\_data: |

몇몇 생소해보이는 설정이 눈에 보이지만, 일단 실행해 봅니다.

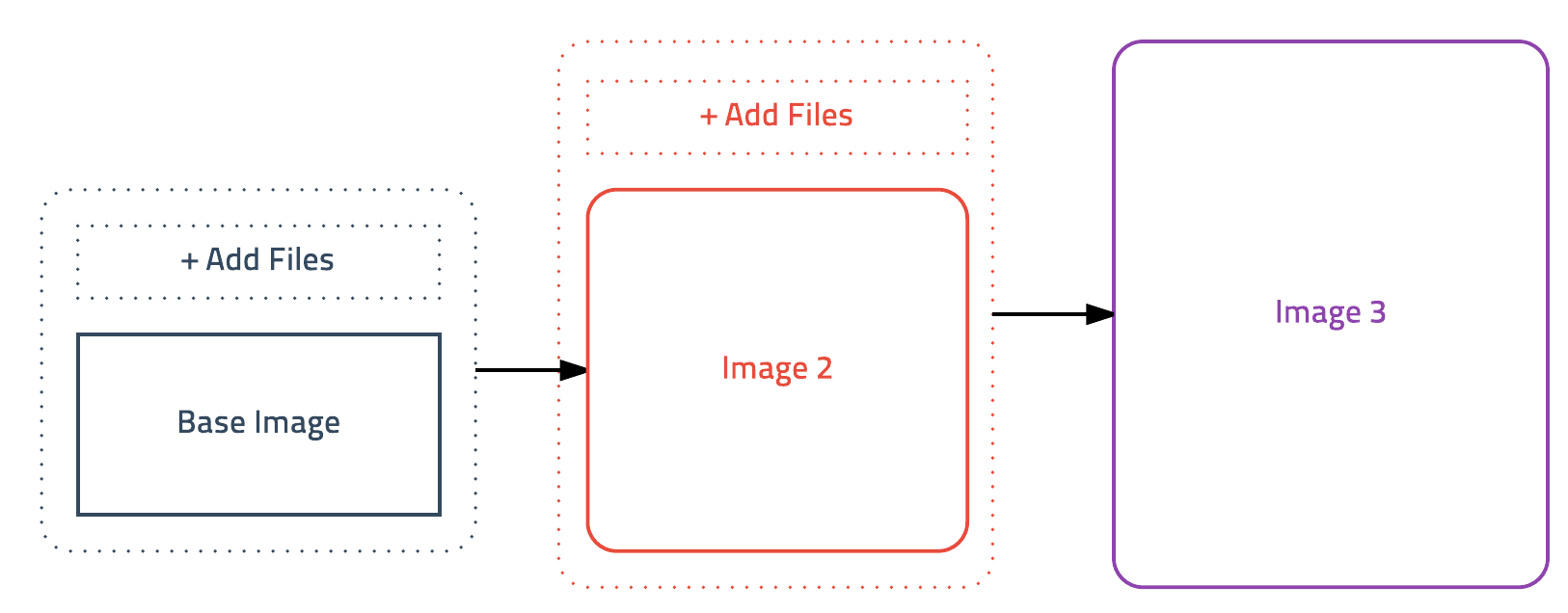
|  |  |
| --- | --- |
| 1 | docker-compose up |

와우, 아주 손쉽게 워드프레스가 만들어 졌습니다. 단지 명령어를 설정파일로 바꾼거에 불과하지만 가독성과 편리성은 훨씬 향상되었습니다.

Docker Compose의 다른 기능과 생소한 설정내용은 숙제로 남겨드립니다. 원래 개발공부라는게 왠만큼 했다고 생각하면 또 다른게 나오고 끊임없이 공부해야 하는 분야입니다.  도커에 대해 이해를 했다면 Docker Compose 또한 쉽게 사용할 수 있을 것입니다.

## 도커 이미지 만들기

도커는 이미지를 만들기 위해 **컨테이너의 상태를 그대로 이미지로 저장**하는 단순하고 무식(?)한 방법을 사용합니다.

create docker image

예를 들어, 어떤 애플리케이션을 이미지로 만든다면 리눅스만 설치된 컨테이너에 애플리케이션을 설치하고 그 상태를 그대로 이미지로 저장합니다. 가상머신의 스냅샷과 비스므리한 방식입니다.

이런 과정은 콘솔에서 명령어를 직접 입력하는 것과 별 차이가 없으므로 쉘 스크립트를 잘 알아야 하지만 좋은 샘플이 많이 [공개](https://github.com/search?utf8=%E2%9C%93&q=dockerfile%EF%BF%BC)되어 있어 잘 몰라도 크게 걱정하지 않아도 됩니다.  또한 컨테이너의 가벼운 특성과 레이어 개념을 이용하여 생성과 테스트를 빠르게 수행할 수 있습니다.

이제 [Ruby](https://www.ruby-lang.org/ko)로 만들어진 간단한 웹 애플리케이션을 도커라이징Dockerizing(=도커 이미지를 만듦)해보겠습니다.

### **Sinatra 웹 애플리케이션 샘플**



Sinatra

일단 웹 애플리케이션 소스코드를 작성해야겠죠. [Sinatra](http://www.sinatrarb.com/)라는 가벼운 웹 프레임워크를 사용하기 위해 새로운 폴더를 만들고 Gemfile과 app.rb를 만듭니다.

**Gemfile**

|  |  |
| --- | --- |
| 1  2 | source 'https://rubygems.org'  gem 'sinatra' |

**app.rb**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | require 'sinatra'  require 'socket'  get '/' **do**  Socket.**gethostname**  **end** |

ruby와 sinatra에 대해 전혀 모르더라도 Gemfile은 패키지를 관리하고 app.rb는 호스트명을 출력하는 웹 서버를 만들었다는 걸 예상할 수 있습니다.

이제 패키지를 설치하고 서버를 실행해보겠습니다.

|  |  |
| --- | --- |
| 1  2 | bundle install *# install package*  bundle exec ruby app.rb *# Run sinatra* |

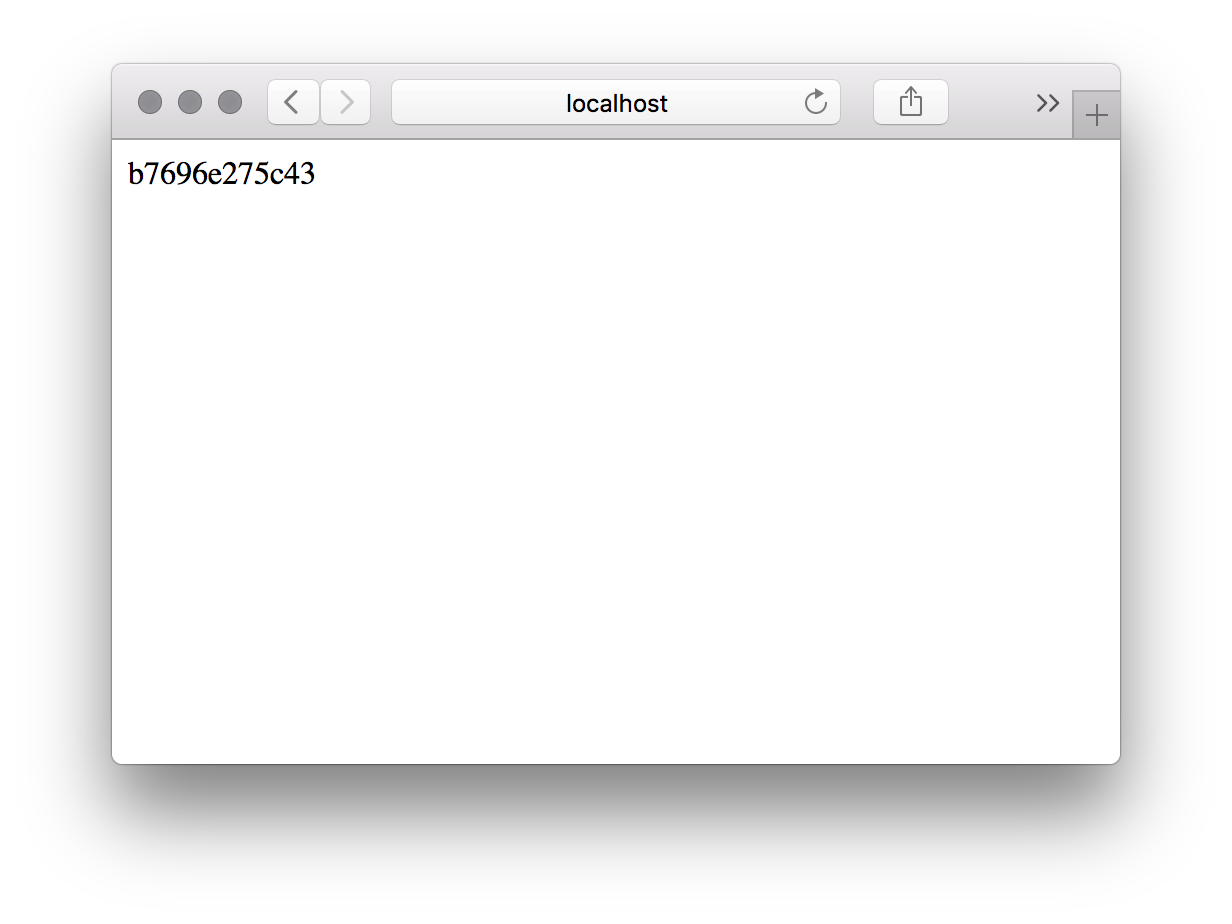
ruby가 설치되어 있지 않다고요? 도커만 있으면 문제없습니다. 다음 명령어를 실행합니다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | docker run --rm \  -p 4567:4567 \  -v $PWD:/usr/src/app \  -w /usr/src/app \  ruby \  bash -c "bundle install && bundle exec ruby app.rb -o 0.0.0.0" |

호스트의 디렉토리를 루비가 설치된 컨테이너의 디렉토리에 마운트한다음 그대로 명령어를 실행하면 로컬에 개발 환경을 구축하지 않고 도커 컨테이너를 개발환경으로 사용할 수 있습니다.

도커를 개발환경으로 사용하면 개발=테스트=운영이 동일한 환경에서 실행되는 놀라운 상황이 펼쳐집니다. 이 부분은 재미있는 내용이 많지만, 주제에서 벗어나므로 이 정도만 언급하고 다음 기회에 더 자세히 알아봅니다.

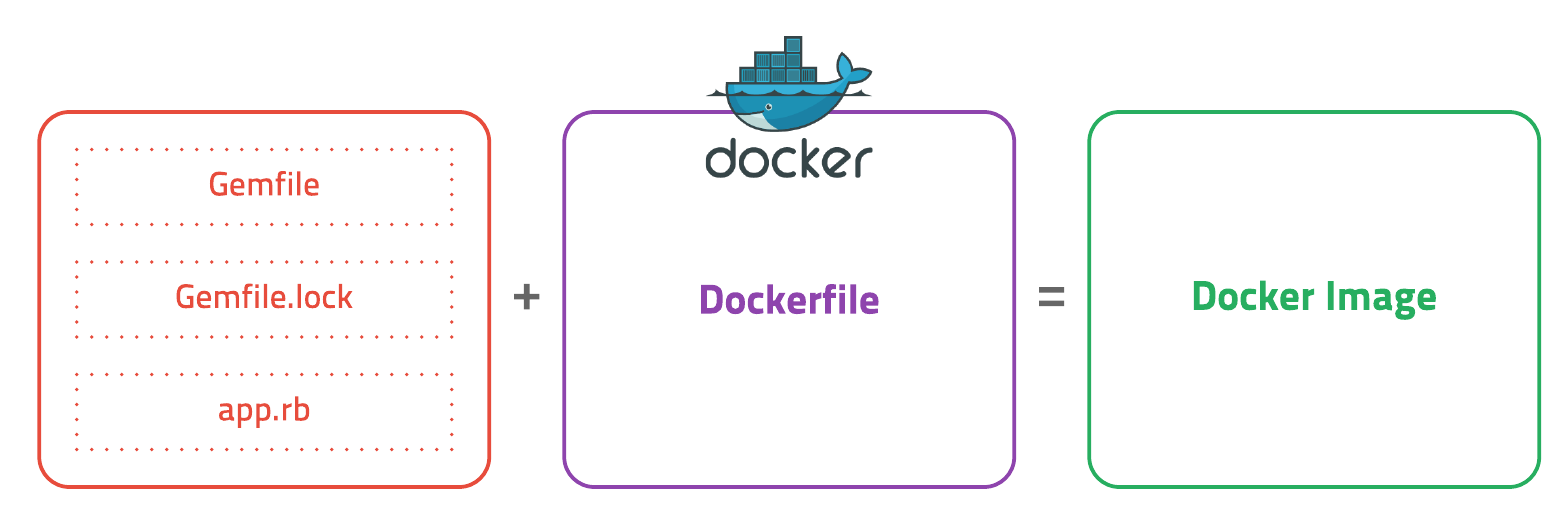
서버가 정상적으로 실행됐으면 웹 브라우저에서 테스트해봅니다. http://localhost:4567



Browser test

도커 컨테이너의 호스트명이 보입니다. 소스는 잘 작성한 것 같네요! 이제 도커 이미지를 만들 준비가 완료됐습니다.

### **Ruby Application Dockerfile**

Dockerfile

도커는 이미지를 만들기 위해 Dockerfile이라는 이미지 빌드용 DSLDomain Specific Language 파일을 사용합니다. 단순 텍스트 파일로 일반적으로 소스와 함께 관리합니다.

고급 개발자는 바로 Dockerfile을 만들 수도 있겠지만, 일반 개발자들은 일단 리눅스 서버에서 테스트로 설치해보고 안 되면 될 때까지 삽질하면서 최적의 과정을 Dockerfile로 작성해야 합니다. 우리는 초보니까 Ruby 웹 애플리케이션을 ubuntu에 배포하는 과정을 먼저 살펴보겠습니다.

| **순서** | **작업** |
| --- | --- |
| 1 | ubuntu 설치 |
| 2 | ruby 설치 |
| 3 | 소스 복사 |
| 4 | Gem 패키지 설치 |
| 5 | Sinatra 서버 실행 |

이 과정을 그대로 쉘 스크립트로 옮겨봅니다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | *# 1. ubuntu 설치 (패키지 업데이트)*  apt-get update  *# 2. ruby 설치*  apt-get install ruby  gem install bundler  *# 3. 소스 복사*  mkdir -p /usr/src/app  scp Gemfile app.rb root@ubuntu:/usr/src/app *# From host*  *# 4. Gem 패키지 설치*  bundle install  *# 5. Sinatra 서버 실행*  bundle exec ruby app.rb |

ubuntu 컨테이너를 실행하고 위 명령어를 그대로 실행하면 웹 서버를 실행할 수 있습니다. 리눅스에서 테스트가 끝났으니 이 과정을 Dockerfile로 만들면 됩니다. 아직 자세한 명령어를 배우진 않았지만 일단 만들어 봅니다. 핵심 명령어는 파일을 복사하는 COPY와 명령어를 실행하는 RUN입니다.

|  |  |
| --- | --- |
|  | # 1. ubuntu 설치 (패키지 업데이트 + 만든사람 표시) |
|  | FROM ubuntu:16.04 |
|  | MAINTAINER subicura@subicura.com |
|  | RUN apt-get -y update |
|  |  |
|  | # 2. ruby 설치 |
|  | RUN apt-get -y install ruby |
|  | RUN gem install bundler |
|  |  |
|  | # 3. 소스 복사 |
|  | COPY . /usr/src/app |
|  |  |
|  | # 4. Gem 패키지 설치 (실행 디렉토리 설정) |
|  | WORKDIR /usr/src/app |
|  | RUN bundle install |
|  |  |
|  | # 5. Sinatra 서버 실행 (Listen 포트 정의) |
|  | EXPOSE 4567 |
|  | CMD bundle exec ruby app.rb -o 0.0.0.0 |

쉘 스크립트의 내용을 거의 그대로 Dockerfile로 옮겼습니다. 차이점은 도커 빌드 중엔 키보드를 입력할 수 없기 때문에 (y/n)을 물어보는 걸 방지하기 위해 -y 옵션을 추가한 것 정도입니다.

빌드 파일을 작성했으니 이제 이미지를 만들어 봅니다.

### **Docker build**

이미지를 빌드하는 명령어는 다음과 같습니다.

docker build [OPTIONS] PATH | URL | -

생성할 이미지 이름을 지정하기 위한 -t(--tag) 옵션만 알면 빌드가 가능합니다.

Dockerfile을 만든 디렉토리로 이동하여 다음 명령어를 입력합니다.

|  |  |
| --- | --- |
| 1 | docker build -t app . |

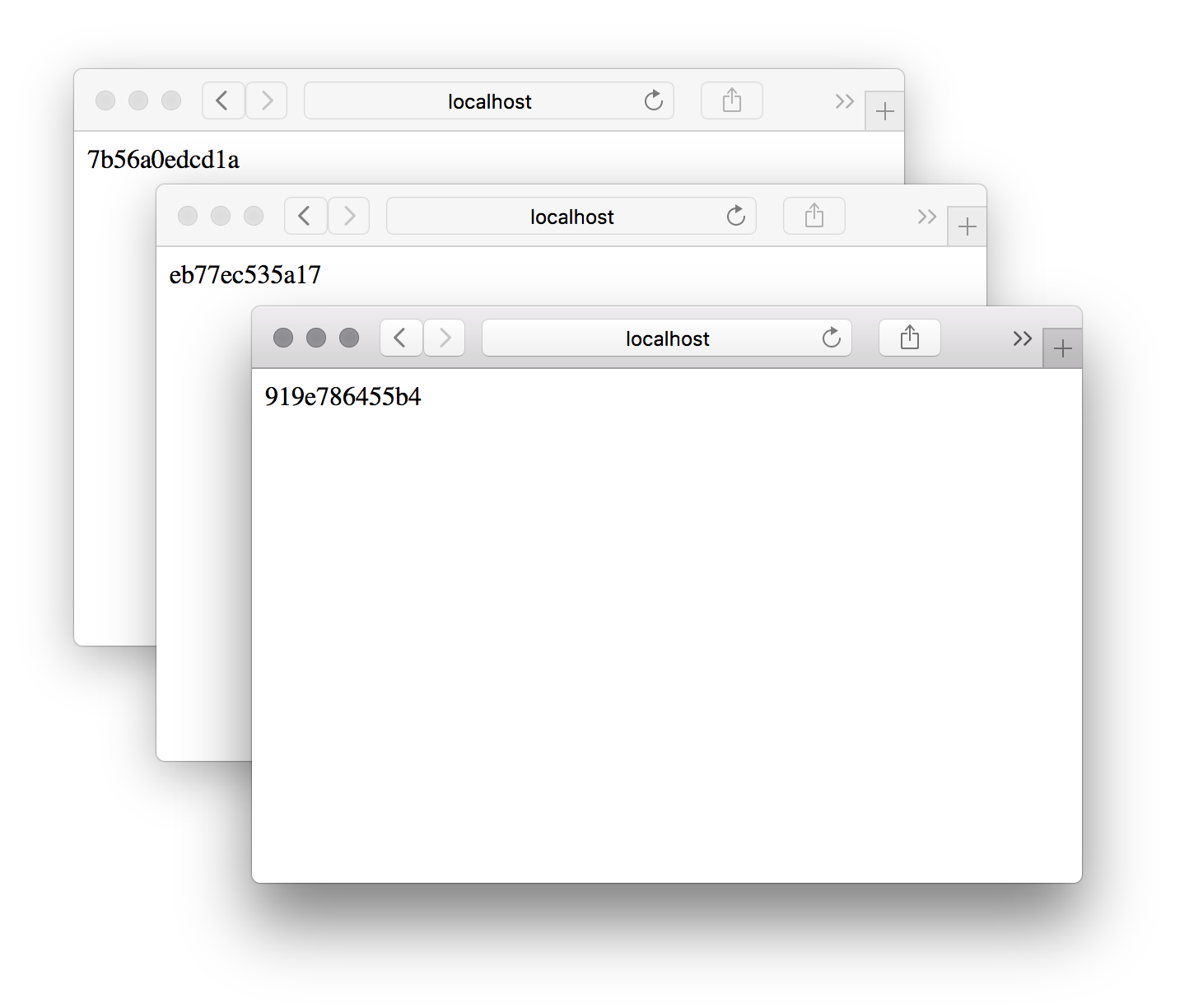
빌드 명령어를 실행하면 Dockerfile에 정의한 내용이 한 줄 한 줄 실행되는 걸 볼 수 있습니다. 실제로 명령어를 실행하기 때문에 빌드 시간이 꽤 걸리는 걸 알 수 있습니다. 최종적으로 Successfully built xxxxxxxx 메시지가 보이면 정상적으로 이미지를 생성한 것입니다.

그럼 이미지가 잘 생성되었는지 확인해보겠습니다.

|  |  |
| --- | --- |
| 1 | docker images |

드디어 첫 번째 도커 이미지를 생성했습니다! 이미지를 생성했으니 잘 동작하는지 컨테이너를 실행해보겠습니다.

|  |  |
| --- | --- |
| 1  2  3 | docker run -d -p 8080:4567 app  docker run -d -p 8081:4567 app  docker run -d -p 8082:4567 app |



Container test result

접속 성공입니다. 기분이 좋아서 호스트 네임을 출력하는  웹서버를 3개나 만들었습니다. 이미지가 잘 만들어졌네요.

### **Dockerfile 기본 명령어**

이미지를 만드는 데 사용한 Dockerfile의 기본적인 명령어를 살펴보겠습니다.

**FROM**

FROM <image>:<tag>

FROM ubuntu:16.04

베이스 이미지를 지정합니다. 반드시 지정해야 하며 어떤 이미지도 베이스 이미지가 될 수 있습니다. tag는 될 수 있으면 latest(기본값)보다 구체적인 버전(16.04등)을 지정하는 것이 좋습니다. 이미 만들어진 다양한 베이스 이미지는 [Docker hub](https://hub.docker.com/explore/)에서 확인할 수 있습니다.

**MAINTAINER**

MAINTAINER <name>

MAINTAINER subicura@subicura.com

Dockerfile을 관리하는 사람의 이름 또는 이메일 정보를 적습니다. 빌드에 딱히 영향을 주지는 않습니다.

**COPY**

COPY <src>... <dest>

COPY . /usr/src/app

파일이나 디렉토리를 이미지로 복사합니다. 일반적으로 소스를 복사하는 데 사용합니다. target디렉토리가 없다면 자동으로 생성합니다.

**ADD**

ADD <src>... <dest>

ADD . /usr/src/app

COPY명령어와 매우 유사하나 몇가지 추가 기능이 있습니다. src에 파일 대신 URL을 입력할 수 있고 src에 압축 파일을 입력하는 경우 자동으로 압축을 해제하면서 복사됩니다.

**RUN**

RUN <command>

RUN ["executable", "param1", "param2"]

RUN bundle install

가장 많이 사용하는 구문입니다. 명령어를 그대로 실행합니다. 내부적으로 /bin/sh -c 뒤에 명령어를 실행하는 방식입니다.

**CMD**

CMD ["executable","param1","param2"]

CMD command param1 param2

CMD bundle exec ruby app.rb

도커 컨테이너가 실행되었을 때 실행되는 명령어를 정의합니다. 빌드할 때는 실행되지 않으며 여러 개의 CMD가 존재할 경우 가장 마지막 CMD만 실행됩니다. 한꺼번에 여러 개의 프로그램을 실행하고 싶은 경우에는 run.sh파일을 작성하여 데몬으로 실행하거나 [supervisord](http://supervisord.org/)나 [forego](https://github.com/ddollar/forego)와 같은 여러 개의 프로그램을 실행하는 프로그램을 사용합니다.

**WORKDIR**

WORKDIR /path/to/workdir

RUN, CMD, ADD, COPY등이 이루어질 기본 디렉토리를 설정합니다. 각 명령어의 현재 디렉토리는 한 줄 한 줄마다 초기화되기 때문에 RUN cd /path를 하더라도 다음 명령어에선 다시 위치가 초기화 됩니다. 같은 디렉토리에서 계속 작업하기 위해서 WORKDIR을 사용합니다.

**EXPOSE**

EXPOSE <port> [<port>...]

EXPOSE 4567

도커 컨테이너가 실행되었을 때 요청을 기다리고 있는(Listen) 포트를 지정합니다. 여러개의 포트를 지정할 수 있습니다.

**VOLUME**

VOLUME ["/data"]

컨테이너 외부에 파일시스템을 마운트 할 때 사용합니다. 반드시 지정하지 않아도 마운트 할 수 있지만, 기본적으로 지정하는 것이 좋습니다.

**ENV**

ENV <key> <value>

ENV <key>=<value> ...

ENV DB\_URL mysql

컨테이너에서 사용할 환경변수를 지정합니다. 컨테이너를 실행할 때 -e옵션을 사용하면 기존 값을 오버라이딩 하게 됩니다.

여기까지 Dockerfile에서 가장 많이 사용하는 명령어에 대해 알아보았습니다. 모든 명령어가 궁금하신 분은 [공식문서](https://docs.docker.com/engine/reference/builder/)(https://docs.docker.com/engine/reference/builder/)를 참고하세요.

### **Build 분석**

도커는 Dockerfile을 가지고 무슨 일을 하는 걸까요? 빌드를 하면서 궁금하지 않으셨나요? 궁금하지 않으셨더라도 build로그를 보면서 하나하나 살펴봅니다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Sending build context to Docker daemon 5.12 kB <-- **(**1**)**  Step 1/10 : FROM ubuntu:16.04 <-- **(**2**)**  ---**>** f49eec89601e <-- **(**3**)**  Step 2/10 : MAINTAINER subicura@subicura.com <-- **(**4**)**  ---**>** Running **in** f4de0c750abb <-- **(**5**)**  ---**>** 4a400609ff73 <-- **(**6**)**  Removing intermediate container f4de0c750abb <-- **(**7**)**  Step 3/10 : RUN apt-get -y update <-- **(**8**)**  ...  ...  Successfully built 20369cef9829 <-- **(**9**)** |

(1) Sending build context to Docker daemon 5.12 kB

빌드 명령어를 실행한 디렉토리의 파일들을 빌드컨텍스트build context라고 하고 이 파일들을 도커 서버(daemon)로 전송합니다. 도커는 서버-클라이언트 구조이므로 도커 서버가 작업하려면 미리 파일을 전송해야 합니다.

(2) Step 1/10 : FROM ubuntu:16.04

Dockerfile을 한 줄 한 줄 수행합니다. 첫 번째로 FROM 명령어를 수행합니다. ubuntu:16.04 이미지를 다운받는 작업입니다.

(3) ---> f49eec89601e

명령어 수행 결과를 이미지로 저장합니다. 여기서는 ubuntu:16.04를 사용하기로 했기 때문에 ubuntu 이미지의 ID가 표시됩니다.

(4) Step 2/10 : MAINTAINER subicura@subicura.com

Dockerfile의 두 번째 명령어인 MAINTAINER 명령어를 수행합니다.

(5) ---> Running in f4de0c750abb

명령어를 수행하기 위해 바로 이전에 생성된 f49eec89601e 이미지를 기반으로 f4de0c750abb컨테이너를 임시로 생성하여 실행합니다.

(6) ---> 4a400609ff73

명령어 수행 결과를 이미지로 저장합니다.

(7) Removing intermediate container f4de0c750abb

명령어를 수행하기 위해 임시로 만들었던 컨테이너를 제거합니다.

(8) Step 3/10 : RUN apt-get -y update

Dockerfile의 세 번째 명령어를 수행합니다. 이전 단계와 마찬가지로 바로 전에 만들어진 이미지를 기반으로 임시 컨테이너를 만들어 명령어를 실행하고 그 결과 상태를 이미지로 만듭니다. 이 과정을 마지막 줄까지 무한 반복합니다.

(9) Successfully built 20369cef9829

최종 성공한 이미지 ID를 출력합니다.

결론적으로 도커 빌드는 임시 컨테이너 생성 > 명령어 수행 > 이미지로 저장 > 임시 컨테이너 삭제 > 새로 만든 이미지 기반 임시 컨테이너 생성 > 명령어 수행 > 이미지로 저장 > 임시 컨테이너 삭제 > … 의 과정을 계속해서 반복한다고 볼 수 있습니다. 명령어를 실행할 때마다 이미지 레이어를 저장하고 다시 빌드할 때 Dockerfile이 변경되지 않았다면 기존에 저장된 이미지를 그대로 캐시처럼 사용합니다.

이러한 레이어 개념을 잘 이해하고 있어야 최적화된 이미지를 생성할 수 있습니다.

### **도커 이미지 리팩토링**

사실 앞에서 만든 이미지는 몇 가지 최적화 문제가 있습니다. 다시 한땀 한땀 살펴보겠습니다.

**Base Image**

위에서 만든 Ruby 애플리케이션 이미지는 ubuntu를 베이스로 만들었지만 사실 훠어어얼씬 간단한 ruby 베이스 이미지가 존재합니다.  기존에 ruby를 설치했던 명령어는 ruby 이미지를 사용하는 것으로 간단하게 생략할 수 있습니다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | *# before*  FROM ubuntu:16.04  MAINTAINER subicura@subicura.com  RUN apt-get -y update  RUN apt-get -y install ruby  RUN gem install bundler  *# after*  FROM ruby:2.3  MAINTAINER subicura@subicura.com |

ruby외에도 nodejs, python, java, go등 다양한 베이스 이미지가 이미 존재합니다. 세부적인 설정이 필요하지 않다면 그대로 사용하는게 간편합니다.

**Build Cache**

조금전에 빌드한 이미지를 다시 빌드해볼까요?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Sending build context to Docker daemon 13.31 kB  Step 1/10 : FROM ubuntu:16.04  ---**>** f49eec89601e  Step 2/10 : MAINTAINER subicura@subicura.com  ---**>** Using cache  ---**>** fc41cd8ac52d  Step 3/10 : RUN apt-get -y update  ---**>** Using cache  ---**>** 61d45ce11dc6  .... |

한번 빌드한 이미지를 다시 빌드하면 굉장히 빠르게 완료되는 걸 알 수 있습니다. 이미지를 빌드하는 과정에서 각 단계를 이미지 레이어로 저장하고 다음 빌드에서 캐시로 사용합니다.

도커는 빌드할 때 Dockerfile의 명령어가 수정되었거나 추가하는 파일이 변경 되었을 때 캐시가 깨지고 그 이후 작업은 새로 이미지를 만들게 됩니다. ruby gem 패키지를 설치하는 과정은 꽤 많은 시간이 소요되는데 최대한 캐시를 이용하여 빌드 시간을 줄여야 합니다.

기존 소스에서 소스파일이 수정되면 캐시가 깨지는 부분은 다음과 같습니다.

|  |  |
| --- | --- |
| 1  2  3 | COPY . /usr/src/app *# <- 소스파일이 변경되면 캐시가 깨짐*  WORKDIR /usr/src/app  RUN bundle install *# 패키지를 추가하지 않았는데 또 인스톨하게 됨 ㅠㅠ* |

복사하는 파일이 이전과 다르면 캐시를 사용하지 않고 그 이후 명령어는 다시 실행됩니다. ruby gem 패키지를 관리하는 파일은 Gemfile이고 Gemfile은 잘 수정되지 않으므로 다음과 같이 순서를 바꿀 수 있습니다.

|  |  |
| --- | --- |
| 1  2  3  4 | COPY Gemfile**\*** /usr/src/app/ *# Gemfile을 먼저 복사함*  WORKDIR /usr/src/app  RUN bundle install *# 패키지 인스톨*  COPY . /usr/src/app *# <- 소스가 바꼈을 때 캐시가 깨지는 시점 ^0^* |

gem 설치 하는 부분을 소스 복사 이전으로 옮겼습니다. 이제 소스가 수정되더라도 매번 gem을 설치하지 않아 더욱 빠르게 빌드할 수 있습니다. 요즘 언어들은 대부분 패키지 매니저를 사용하므로 비슷한 전략으로 작성하면 됩니다.

**명령어 최적화**

이미지를 빌드할 때 불필요한 로그는 무시하는게 좋고 패키지 설치시 문서 파일도 생성할 필요가 없습니다.

|  |  |
| --- | --- |
| 1  2  3  4  5 | *# before*  RUN apt-get -y update  *# after*  RUN apt-get -y -qq update |

-qq 옵션으로 로그를 출력하지 않게 했습니다. 각종 리눅스 명령어는 보통 quite 옵션이 있으니 적절하게 적용하면 됩니다.

|  |  |
| --- | --- |
| 1  2  3  4  5 | *# before*  RUN bundle install  *# after*  RUN bundle install --no-rdoc --no-ri |

--no-doc과 --no-ri 옵션으로 필요 없는 문서를 생성하지 않아 이미지 용량도 줄이고 빌드 속도도 더 빠르게 했습니다.

**이쁘게**

명령어는 비슷한 것끼리 묶어 주는 게 보기도 좋고 레이어 수를 줄이는데 도움이 됩니다. 도커 이미지는 스토리지 엔진에 따라 레이어의 개수가 127개로 제한되어 있는 경우도 있어 너무 많은 명령어는 좋지 않습니다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | *# before*  RUN apt-get -y -qq update  RUN apt-get -y -qq install ruby  *# after*  RUN apt-get -y -qq update **&&** \  apt-get -y -qq install ruby |

**최종**

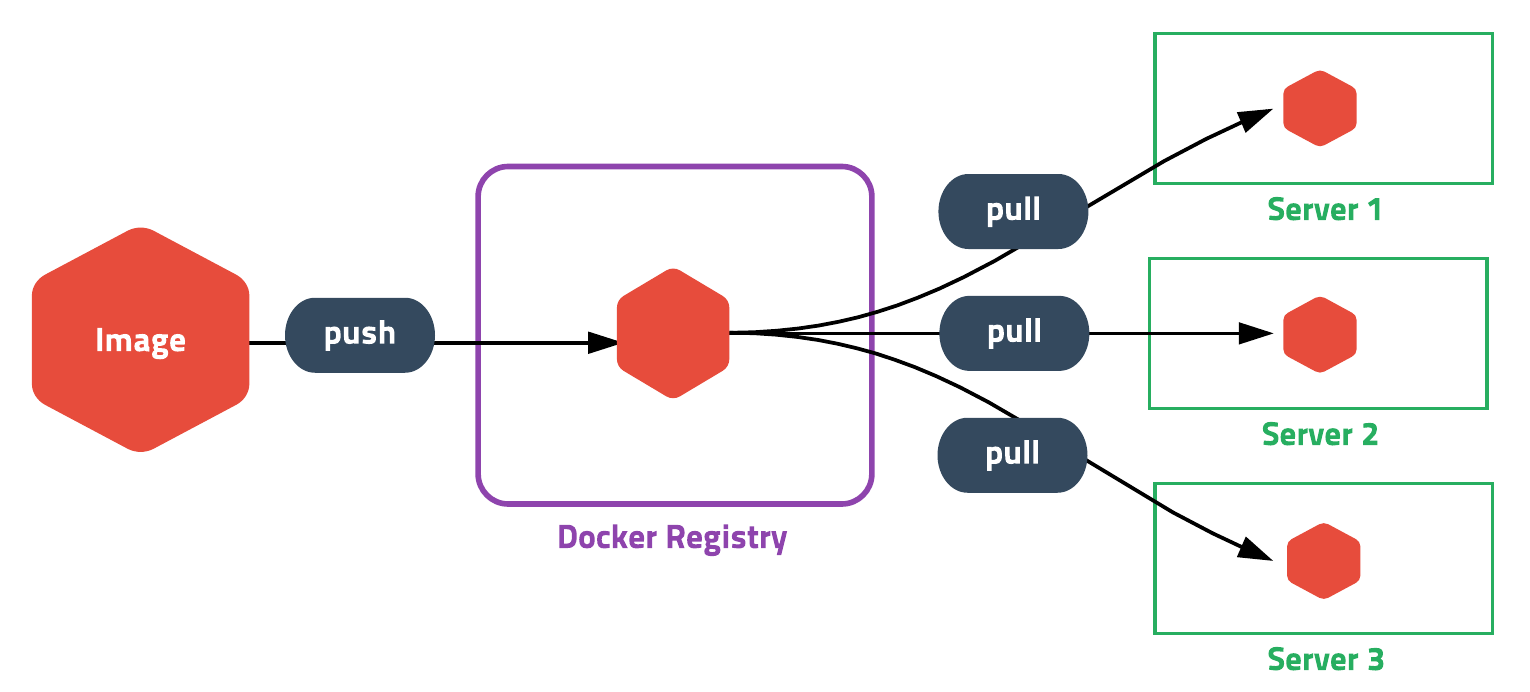
최종 결과는 다음과 같습니다.

|  |  |
| --- | --- |
|  | FROM ruby:2.3 |
|  | MAINTAINER subicura@subicura.com |
|  | COPY Gemfile\* /usr/src/app/ |
|  | WORKDIR /usr/src/app |
|  | RUN bundle install --no-rdoc --no-ri |
|  | COPY . /usr/src/app |
|  | EXPOSE 4567 |
|  | CMD bundle exec ruby app.rb -o 0.0.0.0 |

드디어 아까보다 훨씬 나은 이미지가 완성됐습니다.

이 외에도 다양한 이미지 생성팁이 있지만 일단 이 정도면 꽤 이쁘게 만들어진 것 같습니다.

## 이미지 저장소



Docker Registry

도커는 빌드한 이미지를 서버에 배포하기 위해 직접 파일을 복사하는 방법 대신 [도커 레지스트리](https://docs.docker.com/registry/)Docker Registry라는 이미지 저장소를 사용합니다. 도커 명령어를 이용하여 이미지를 레지스트리에 푸시push하고 다른 서버에서 풀pull받아 사용하는 구조입니다.

도커 레지스트리는 [오픈소스](https://github.com/docker/distribution)로 무료로 설치할 수 있고 설치형이 싫다면 도커(Docker Inc.)에서 서비스 중인 [도커 허브](https://hub.docker.com/)Docker Hub를 사용할 수 있습니다.

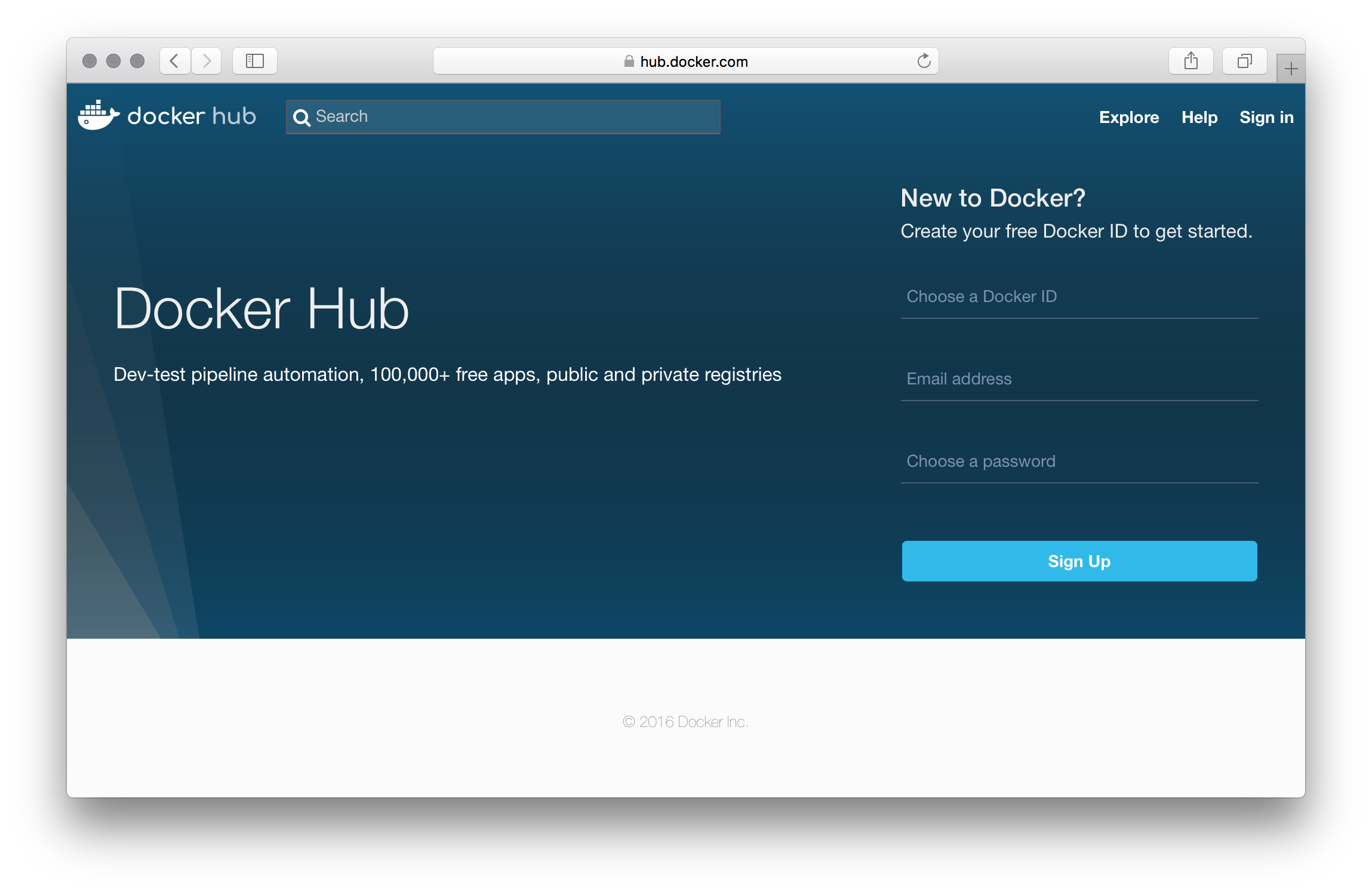
### **Docker Hub**

도커 허브는 도커에서 제공하는 기본 이미지 저장소로 ubuntu, centos, debian등의 베이스 이미지와 ruby, golang, java, python 등의 공식 이미지가 저장되어 있습니다. 일반 사용자들이 만든 이미지도 50만 개가 넘게 저장되어 있고 다운로드 횟수는 80억 회를 넘습니다.

회원가입만 하면 대용량의 이미지를 무료로 저장할 수 있고 다운로드 트래픽 또한 무료입니다. 단, 기본적으로 모든 이미지는 공개되어 누구나 접근 가능하므로 비공개로 사용하려면 유료 서비스를 이용해야 합니다. (한 개는 무료)

회원가입을 하고 앞에서 만든 Ruby 웹 애플리케이션 이미지를 저장해보겠습니다.

**회원가입**

Docker Hub

[도커 허브](https://hub.docker.com/)사이트(https://hub.docker.com/)에 접속하면 쉽게 회원가입을 할 수 있습니다. 디자인이 참 마음에 들지 않는데… 처음부터 지금까지 업데이트한 모습이 이 모양이라 앞으로도 크게 기대되지 않습니다.

**로그인**

도커에서 도커 허브 계정을 사용하려면 로그인을 해야합니다.

|  |  |
| --- | --- |
| 1 | docker login |

output:

|  |  |
| --- | --- |
| 1  2  3  4 | Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com to create one.  Username: subicura  Password:  Login Succeeded |

ID와 패스워드를 입력하면 로그인이 되고 ~/.docker/config.json에 인증정보가 저장되어 로그아웃하기 전까지 로그인 정보가 유지됩니다.

**이미지 태그**

도커 이미지 이름은 다음과 같은 형태로 구성됩니다.

[Registry URL]/[사용자 ID]/[이미지명]:[tag]

Registry URL은 기본적으로 도커 허브를 바라보고 있고 사용자 ID를 지정하지 않으면 기본값(library)을 사용합니다. 따라서 ubuntu = library/ubuntu = docker.io/library/ubuntu 는 모두 동일한 표현입니다.

도커의 tag명령어를 이용하여 기존에 만든 이미지에 추가로 이름을 지어줄 수 있습니다.

docker tag SOURCE\_IMAGE[:TAG] TARGET\_IMAGE[:TAG]

앞에서 만든 app이미지에 계정정보와 버전 정보를 추가해보겠습니다.

|  |  |
| --- | --- |
| 1 | docker tag app subicura/sinatra-app:1 |

subicura라는 ID를 사용하고 이미지 이름을 sinatra-app으로 변경했습니다. 첫 번째 버전이므로 태그는 1을 사용합니다. 이제 push명령을 이용해 도커 허브에 이미지를 전송해 봅니다.

|  |  |
| --- | --- |
| 1 | docker push subicura/sinatra-app:1 |

output:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | The push refers to a repository **[**docker.io/subicura/sinatra-app]  2adeabae7edc: Pushed  8343e5bcf528: Pushed  af3b68c8b565: Pushed  40dd6783317f: Pushed  c6ae77e29c22: Pushed  5eb5bd4c5014: Mounted from library/ubuntu  d195a7a18c70: Mounted from library/ubuntu  af605e724c5a: Mounted from library/ubuntu  59f161c3069d: Mounted from library/ubuntu  4f03495a4d7d: Mounted from library/ubuntu  1: digest: sha256:af83aca920982c1fb17f08b4aa300439470349d58d63c921f67261054a0c9467 size: 2409 |

성공적으로 이미지를 도커 허브에 푸시하였습니다. 도커 허브에 저장된 50만 개의 이미지에 새로운 이미지가 하나 추가되었습니다!

이제 어디서든 subicura/sinatra-app:1이미지를 사용할 수 있습니다.

### **Private Docker Registry**

도커 이미지를 비공개로 저장하려면 [Docker Cloud](https://www.docker.com/products/docker-cloud)를 유료($7 for 5 repos/month)로 사용하거나 레지스트리 서버를 자체적으로 구축해야합니다.

도커 레지스트리는 도커를 이용하여 쉽게 만들 수 있습니다. 도커 이미지를 저장할 서버를 도커 스스로 만들어서 도커 이미지를 관리하다니 뭔가 므-\_-흣합니다.

|  |  |
| --- | --- |
| 1  2  3  4 | docker run -d \  -v $PWD/registry:/var/lib/registry \  -p 5000:5000 \  distribution/registry:2.6.0 |

저장된 이미지는 파일로 관리되기 때문에 호스트의 디렉토리를 마운트하였습니다. (S3 저장소를 사용할 수도 있습니다) 이제 레지스트리 서버의 아이피와 포트정보를 이미지명에 추가하면 바로 사용할 수 있습니다.

|  |  |
| --- | --- |
| 1  2 | docker tag app localhost:5000/subicura/sinatra-app:1  docker push localhost:5000/subicura/sinatra-app:1 |

앞에서 만든 이름에 localhost:5000/를 추가했습니다. 레지스트리 서버에 파일이 잘 저장되었나 마운트한 디렉토리를 한번 살펴봅니다.

|  |  |
| --- | --- |
| 1 | tree registry |

**보안**

**도커 레지스트리는 일반적인 HTTP 프로토콜을 사용하여 이미지를 전송합니다. 따라서 SSL(HTTPS)을 사용하지 않으면 이미지 내용이 유출될 수 있습니다. 이런 보안 이슈 때문에 도커는 기본적으로 로컬(localhost) 서버를 제외하곤 HTTP 사용을 금지하고 있으며 이런 보안위험을 무시하려면 도커 엔진을 실행할 때 허용 옵션을 넣어야 합니다.**

관련 설정은 [문서](https://docs.docker.com/registry/insecure/)(https://docs.docker.com/registry/insecure/)를 참고하시기 바랍니다.

## 배포하기

드디어 도커 안내서의 마지막 주제, 서버관리의 꽃! 배포deploy에 대해 알아보겠습니다.

### **컨테이너 배포 방식으로**

컨테이너를 배포하는 방식은 기존에 애플리케이션을 배포하는 방식과 큰 차이가 있습니다.

기존에 애플리케이션을 배포하는 방식은 사용하는 언어, 프레임워크, 웹(or WAS)서버, 리눅스 배포판, 개발자의 취향에 따라 각각 다른 방식을 사용했습니다.

새로운 서버를 셋팅하고 한 번에 배포를 성공한다는 건 굉장히 힘든 일이었고 의존성 라이브러리가 제대로 설치되었는지 검증하기도 매우 어려웠습니다.

ftp, rsync, ant, gradle, capistrano, fabric, chef, puppet, ansible등 다양한 배포툴이 저마다의 장점을 가지고 등장하였고 배포하는 방식을 하나로 정의한다는 건 거의 불가능했습니다.

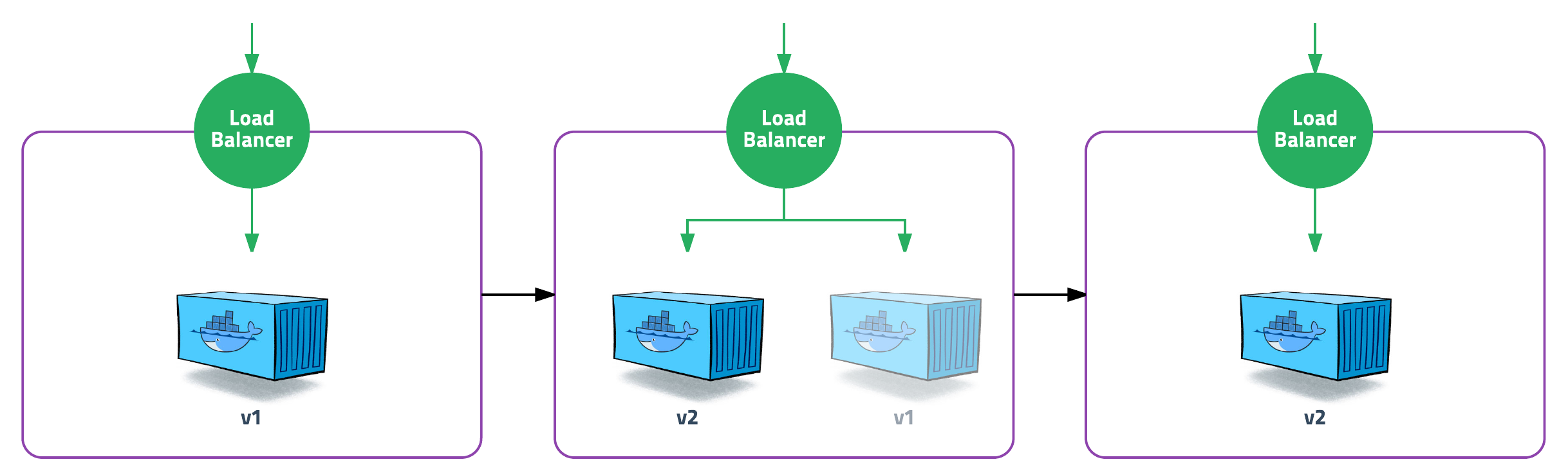
하지만, 컨테이너를 사용하면 어떤 언어, 어떤 프레임워크를 쓰든 상관없이 배포 방식이 동일해지고 과정 또한 굉장히 단순해집니다. 그냥 이미지를 다운받고 컨테이너를 실행하면 끝입니다.

음…

네, 그게 답니다.

서버에 접속해서 컨테이너를 실행할 줄 안다면 이미 배포하는 법을 알고 있는 겁니다. 참 쉽죠?

### **컨테이너 업데이트**

Update

도커를 사용하면 업데이트하는 방식도 배포와 큰 차이가 없습니다.

최신 이미지를 기반으로 새 컨테이너를 만들고 이전 컨테이너를 중지(삭제)하면 됩니다. 최신 소스를 어떻게 복사할지 서버 프로세스는 어떻게 재시작할지 고민할 필요가 없습니다. 그냥 통째로 바꿔버리는 겁니다.

단, 컨테이너를 중지하지 않고 graceful하게 샤샤샥 교체하는 방법은 아쉽지만 존재하지 않습니다.

**컨테이너를 중지하지 않고 컨테이너 내부에 접속하여 소스를 업데이트하는 방법도 “가능”은 하지만 컨테이너의 장점을 살릴 수 없는 “잘못된 패턴”입니다.**

이런 방식은 매우 단순하지만, 컨테이너가 멈추는 순간 실행 중인 프로세스가 **종료**되고 프로세스가 종료되면 고객들은 접속이 안 되고 접속이 안 되면 매출이 떨어지고 매출이 떨어지면 월급이 안나오기 때문에 무중단을 고려한 [nginx](http://nginx.org/)나 [HAProxy](http://www.haproxy.org/)같은 로드 발란서Load Balancer와 2대 이상의 컨테이너를 사용해야 합니다.

여기서는 개념만 소개하고 실제로 컨테이너를 업데이트하는 구체적인 방법은 [도커를 이용한 웹서비스 무중단 배포하기](https://subicura.com/2016/06/07/zero-downtime-docker-deployment.html)글로 대신합니다.

### **배포에 대해 더 알아보기**

도커를 이용한 배포, 그 자체는 매우 단순하지만 여러 대의 서버를 관리하고 문제없이 업데이트 하는 건 완전히 새로운 이야기입니다.

여러 대의 서버를 관리하려면 가상네트워크, 공유 파일, 로그관리, CPU나 메모리 같은 자원분배에 대해 고민해야 하고 Service Discovery에 대한 개념과 Orchestration이라는 주제에 관해 공부해야 합니다. 딱 정해진 답은 없고 현재 운영 중인 환경에 적합한 방법을 찾아야 하며 지금도 계속해서 발전하고 여러 컨퍼런스에서 활발하게 논의되는 주제입니다.

자세한 내용은 초보를 위한 안내서의 범위를 벗어나므로 더 많은 내용이 궁금하신 분은 각자 공부하는 것으로…

## 마무으리

클라우드가 발전하면서 언제든 원하는 장소에 수십 대의 서버를 클릭 한 번으로 생성하는 시대가 되었습니다. 서버는 구입하는것이 아니라 필요한 만큼 잠시 **대여**하는 개념이 되었고 오토 스케일링이라는 환상적인 기능은 부하에 따라 자동으로 서버 개수를 늘리고 줄여줍니다.

이러한 흐름 속에서 아무 문제 없이 서비스를 배포하고 운영하는 것은 사실 기적이라고 할 수 있습니다. 도커가 등장하여 복잡한 부분을 단순하게 하고 많은 문제를 해결해 주고 있지만 도커도 모든 문제를 해결해 주지는 않습니다. (대표적으로 데이터베이스처럼 stateful한 애플리케이션은 관리가 어렵습니다) 다만, 현재 시점에서 그 어떤 방법보다도 **좀 더 나은 방법**인 건 확실합니다.

초보를 위한 도커 안내서는 오늘도 서버운영으로 고통받고 있는 개발자분께 도커를 소개하고 전반적인 내용을 전달하기 위해 작성하였습니다. 길다면 길고 짧다면 짧은 글을 통해 아무쪼록 초보분들에게 많은 도움이 되었으면 좋겠습니다.

도커에 대해 궁금한 점은 [오픈컨테이너 슬랙](http://slack.opencontainer.co.kr/)으로 오시면 많은 도움을 받을 수 있습니다. 내용에 대한 다양한 피드백도 환영합니다.

https://www.redhat.com/ko/topics/containers/what-is-docker

**Docker 란?**

* IT 소프트웨어인 "Docker”는 [Linux® 컨테이너](https://www.redhat.com/ko/topics/containers)를 만들고 사용할 수 있도록 하는 컨테이너화 기술입니다.
* [오픈소스 Docker 커뮤니티](https://forums.docker.com/)(https://forums.docker.com)는 모든 사용자가 무료로 이점을 얻을 수 있도록 이러한 기술을 개선하기 위해 노력하고 있습니다.
* [Docker Inc.](https://www.docker.com/)는 Docker 커뮤니티의 활동에 기반을 두고 있으며 Docker 커뮤니티의 보안을 강화하고 개선 사항을 공유하여 커뮤니티를 발전시킵니다. 그리고 엔터프라이즈 고객을 위한 향상되고 강화된 기술을 지원합니다.

Docker를 사용하면 컨테이너를 매우 가벼운 모듈식 가상 머신처럼 다룰 수 있습니다. 또한 컨테이너를 구축, 배포, 복사하고 한 환경에서 다른 환경으로 이동하는 등 유연하게 사용할 수 있어, [애플리케이션을 클라우드에 최적화](https://www.redhat.com/ko/topics/cloud-native-apps)하도록 지원합니다.

**Docker는 어떻게 작동하나요?**

Docker기술은 Linux 커널과 함께 [Cgroups](https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html) 및 [네임스페이스](https://lwn.net/Articles/528078/)와 같은 커널의 기능을 사용하여 프로세스를 분리함으로써 독립적으로 실행될 수 있도록 합니다. 이러한 독립성은 컨테이너의 본래 목적입니다. 다시 말해서, 여러 프로세스와 플리케이션을 서로 개별적으로 실행하여 인프라를 더 효과적으로 활용하고 개별 시스템을 사용할 때와 동일한 [보안을 유지](https://www.redhat.com/ko/topics/security)할 수 있습니다.

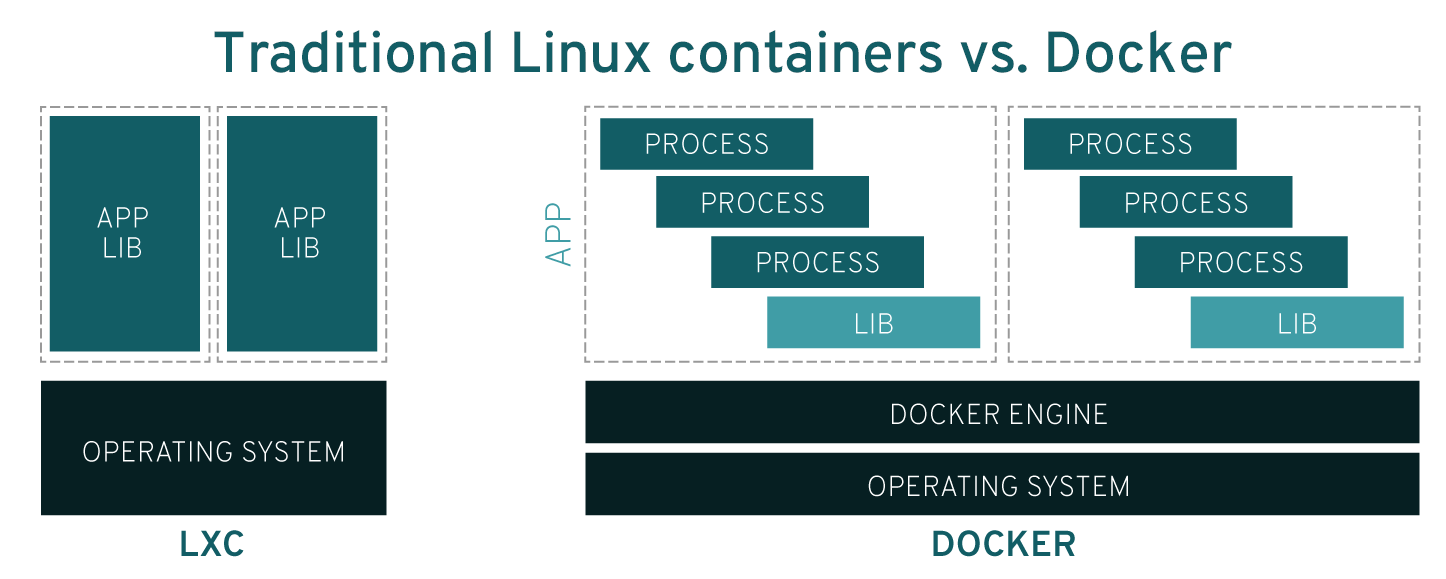
Docker를 포함한 컨테이너 툴은 이미지 기반 배포 모델을 제공하므로 여러 환경 전반에서 애플리케이션 또는 서비스를 모든 종속 항목과 손쉽게 공유할 수 있습니다. 또한 Docker는 이 컨테이너 환경 내에서 애플리케이션(또는 앱을 구성하는 결합된 프로세스) 배포를 자동화합니다.

이러한 툴은 Linux 컨테이너를 기반으로 구축되어 Docker를 사용자 친화적이고 고유하게 만들어 주므로 사용자는 그 어느 때보다도 쉽게 앱에 액세스해 빠르게 배포하고 버전 및 버전 배포를 관리할 수 있습니다.

|  |
| --- |
| Docker 컨테이너의 구성 요소 **Docker** 는 다음과 같은 기본 구성 요소와 함께 작동합니다.   * 컨테이너 – 애플리케이션 샌드 박스. 각 컨테이너는필요한 구성 데이터를 보유하는 이미지 를기반으로합니다. 이미지에서 컨테이너를 시작하면이 이미지 위에 쓰기 가능한 레이어가 추가됩니다. docker commit명령을사용하여 컨테이너를 커밋 할 때마다변경 사항을 저장하기 위해 새 이미지 레이어가 추가됩니다. * 이미지 – 컨테이너 구성의 정적 스냅 샷입니다. 이미지는 수정되지 않는 읽기 전용 레이어이며 모든 변경 사항은 쓰기 가능한 최상위 레이어에서 이루어지며 새 이미지를 생성해야만 저장할 수 있습니다. 각 이미지는 하나 이상의 상위 이미지에 따라 다릅니다. * 플랫폼 이미지 – 상위가없는 이미지입니다. 플랫폼 이미지는 컨테이너화 된 애플리케이션을 실행하는 데 필요한 런타임 환경, 패키지 및 유틸리티를 정의합니다. 플랫폼 이미지는 읽기 전용이므로 모든 변경 사항은 그 위에 쌓인 복사 된 이미지에 반영됩니다. [그림 7.1,“Docker 형식을 사용한 이미지 계층화”](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/7.0_release_notes/chap-Red_Hat_Enterprise_Linux-7.0_Release_Notes-Linux_Containers_with_Docker_Format#figu-Red_Hat_Enterprise_Linux-7.0_Release_Notes-Components_of_Docker_Containers-Image_Layering_Using_Docker_Format) 에서 이러한 스택의 예를 참조하십시오. * 레지스트리 – 이미지 저장소. 레지스트리는 다운로드 할 수있는 이미지가 포함 된 공용 또는 개인 저장소입니다. 일부 레지스트리에서는 사용자가 이미지를 업로드하여 다른 사용자가 사용할 수 있도록 할 수 있습니다. * Dockerfile – Docker 이미지에 대한 빌드 지침이있는 구성 파일입니다. Dockerfile은 빌드 절차를 자동화, 재사용 및 공유하는 방법을 제공합니다.   Docker 형식을 사용한 이미지 계층화  **그림 7.1. Docker 형식을 사용한 이미지 계층화** |

**Docker 기술이 기존의 Linux 컨테이너와 동일한가요?**

그렇지 않습니다. Docker 기술은 처음에 [LXC](https://linuxcontainers.org/) (https://linuxcontainers.org/ko/)기술을 기반으로 구축되긴 했으나, 그 이후로는 종속 관계를 벗어났습니다. 하지만 대부분의 사용자는 "기존의" Linux 컨테이너와 연결지어 생각하고 있습니다. LXC는 경량의 [가상화](https://www.redhat.com/ko/topics/virtualization) 방법으로 유용하게 사용되었지만 개발자 또는 사용자에게 우수한 경험을 제공하지는 못했습니다. Docker 기술은 컨테이너를 실행하는 기능 이상의 것을 제공하며 무엇보다도 컨테이너 생성 및 구축, 이미지 전송, 이미지 버전 관리 프로세스를 용이하게 해 줍니다.



전통적인 Linux 컨테이너는 멀티플 프로세스를 관리할 수 있는 초기화 시스템을 사용합니다. 즉, 전체 애플리케이션을 하나로 실행할 수 있습니다. Docker 기술은 애플리케이션이 개별 프로세스로 세분화되도록 하며 이를 수행할 수 있는 툴을 제공합니다. 이 세분화된 접근 방식에는 이점이 있습니다.

## Docker 컨테이너의 이점

### **모듈성**

Docker의 컨테이너화 접근 방식은 전체 애플리케이션을 분해할 필요 없이 애플리케이션의 일부를 분해하고, 업데이트 또는 복구하는 능력에 집중되어 있습니다. 사용자는 이 마이크로서비스 기반 접근 방식 외에도 [SOA(service-oriented architecture)](https://en.wikipedia.org/wiki/Service-oriented_architecture)의 작동 방식과 동일하게 멀티플 애플리케이션 사이에서 프로세스를 공유할 수 있습니다.

### **계층 및 이미지 버전 제어**

각 Docker 이미지 파일은 일련의 계층으로 이루어져 있으며 이 계층들은 단일 이미지로 결합됩니다. 이미지가 변경될 때 계층이 생성되고, 사용자가 *실행* 또는 *복사*와 같은 명령을 지정할 때마다 새 계층이 생성됩니다.

Docker는 새로운 컨테이너를 구축할 때 이러한 계층을 재사용하므로 구축 프로세스가 훨씬 더 빨라집니다. 중간 변경 사항이 이미지 사이에서 공유되므로 속도, [규모](http://developers.redhat.com/blog/2016/03/09/more-about-docker-images-size/), 효율성이 더 개선됩니다. 계층화에는 버전 관리가 내재되어 있으며 새로운 변경 사항이 발생할 때마다 내장 변경 로그가 기본적으로 적용되므로 컨테이너 이미지를 완전히 제어할 수 있습니다.

### **롤백**

계층화에서 가장 유용한 부분은 아마도 롤백 기능일 것입니다. 모든 이미지에는 계층이 있으며, 현재의 이미지 반복이 적절하지 않은 경우 이전 버전으로 롤백하면 됩니다. 이 기능은 애자일(agile) 개발 접근 방식을 지원하며 툴 관점에서 실제로 [지속적인 통합 및 연속 배포(Continuous Integration and Deployment, CI/CD)](https://www.redhat.com/ko/topics/devops/what-is-ci-cd)를 수행하는 데 도움을 줍니다.

### **신속한 배포**

새로운 하드웨어를 준비하고, 실행하고, 프로비저닝하고, 사용할 수 있게 하려면 일반적으로 며칠이 소요되었습니다. 많은 노력과 부가적인 업무가 필요하므로 부담도 상당했습니다. Docker 기반 컨테이너는 배포 시간을 몇 초로 단축할 수 있습니다. 각 프로세스에 대한 컨테이너를 생성함으로써 사용자는 유사한 프로세스를 새 앱과 빠르게 공유할 수 있습니다. 또한, 컨테이너를 추가하거나 이동하기 위해 OS를 부팅할 필요가 없으므로 배포 시간이 크게 단축됩니다. 이뿐만 아니라 배포 속도가 빨라 컨테이너에서 생성된 데이터를 비용 효율적으로 쉽게 생성하고 삭제할 수 있고 사용자는 우려를 할 필요가 없습니다.

즉, Docker 기술은 효율성을 중시하며 더 세분화되고 제어 가능한 마이크로서비스 기반 접근 방식입니다.

|  |
| --- |
| **Docker 사용의 장점**  **Docker** 는 컨테이너 관리를 위한 API, 이미지 형식 및 컨테이너 공유를위한 원격 레지스트리 사용 가능성을 제공합니다. 이 체계는 다음과 같은 이점으로 개발자와 시스템 관리자 모두에게 이익이 됩니다.   * *신속한 애플리케이션 배포* – 컨테이너에는*애플리케이션* 의 최소 ​​런타임 요구 사항이 포함되어 크기를 줄이고 신속하게 배포 할 수 있습니다. * *머신 간 이식성* – 애플리케이션 및 모든 종속 항목을 Linux 커널, 플랫폼 배포 또는 배포 모델의 호스트 버전과 독립적 인 단일 컨테이너로 번들링 할 수 있습니다. 이 컨테이너는**Docker** 를 실행하는 다른 머신으로 전송할 수 있으며호환성 문제없이 실행할 수 있습니다. * *버전 제어 및 구성 요소 재사용* – 컨테이너의 연속 버전을 추적하고 차이점을 검사하거나 이전 버전으로 롤백 할 수 있습니다. 컨테이너는 이전 레이어의 구성 요소를 재사용하므로 눈에 띄게 가볍습니다. * *공유* – 원격 저장소를 사용하여 다른 사람과 컨테이너를 공유 할 수 있습니다. Red Hat은 이러한 목적을 위해 레지스트리를 제공하며 자신 만의 개인 저장소를 구성 할 수도 있습니다. * *가벼운 설치 공간 및 최소한의 오버 헤드* – Docker 이미지는 일반적으로 매우 작기 때문에 신속한 제공이 가능하고 새로운 애플리케이션 컨테이너를 배포하는 시간이 단축됩니다. * *단순화 된 유지 관리* – Docker는 애플리케이션 종속성으로 인한 노력과 문제의 위험을 줄여줍니다. |

|  |
| --- |
| **가상머신과의 비교**  가상 머신은 모든 관련 소프트웨어 및 유지 관리 문제가있는 전체 서버를 나타냅니다. Docker 컨테이너는 애플리케이션 격리를 제공하며 최소 런타임 환경으로 구성 할 수 있습니다. Docker 컨테이너에서는 커널과 운영 체제 인프라의 일부가 공유됩니다. 가상 머신의 경우 전체 운영 체제가 포함되어야합니다.   * 컨테이너를 빠르고 쉽게 생성하거나 파괴 할 수 있습니다. 가상 머신은 전체 설치가 필요하며 실행하려면 더 많은 컴퓨팅 리소스가 필요합니다. * 컨테이너는 경량이므로 가상 머신보다 더 많은 컨테이너를 호스트 머신에서 동시에 실행할 수 있습니다. * 컨테이너는 리소스를 효율적으로 공유합니다. 가상 머신은 격리됩니다. 따라서 컨테이너에서 실행되는 응용 프로그램의 여러 변형도 매우 가볍습니다. 예를 들어, 공유 바이너리는 시스템에서 복제되지 않습니다. * 가상 머신은 계속 실행하는 동안 마이그레이션 할 수 있지만 컨테이너는 실행 중에 마이그레이션 할 수 없으며 호스트 머신에서 호스트 머신으로 이동하기 전에 중지해야합니다.   모든 사용 사례에서 컨테이너가 가상 머신을 대체하는 것은 아닙니다. 애플리케이션에 가장 적합한 것이 무엇인지 결정하려면 신중한 평가가 여전히 필요합니다. |

## Docker를 사용하는 데 제약이 있으신가요?

Docker는 단일 컨테이너 관리에 적합하도록 만들어져 있습니다. 수백 개로 세분화된 컨테이너와 컨테이너화된 앱을 점점 더 많이 사용하기 시작하면 관리와 오케스트레이션이 매우 어려워질 수 있습니다. 결국 모든 컨테이너 전체에서 네트워킹, 보안, 텔레메트리와 같은 서비스를 제공하기 위해서는 한 걸음 물러나서 컨테이너를 그룹화해야 합니다. 바로 여기에 쿠버네티스가 사용됩니다.

Docker에서는 전통적인 Linux 컨테이너에서 제공되는 것과 동일한 UNIX와 같은 기능을 사용할 수 없습니다. 여기에는 컨테이너 내에서 애플리케이션과 함께 cron 또는 syslog와 같은 프로세스를 사용할 수 있는 기능이 포함됩니다. 또한 자식 프로세스를 종료한 후에 손자 프로세스를 정리하는 작업 등 기존의 Linux 컨테이너에서 기본적으로 처리되는 작업에 대한 제약이 있습니다. 이러한 문제는 처음부터 바로 드러나지는 않지만 초기에 설정 파일을 수정하고 이러한 기능을 셋업하여 완화할 수 있습니다.

네임스페이스가 없는 기타 Linux 하위 시스템과 장치도 있습니다. 여기에는 [SELinux](https://en.wikipedia.org/wiki/Security-Enhanced_Linux), Cgroups 및 /dev/sd\* 장치가 포함됩니다. 그렇기 때문에 공격자가 이러한 하위 시스템에 대한 제어 권한을 얻을 경우 호스트가 손상됩니다. 경량을 유지하기 위해 호스트 커널을 [컨테이너와 공유할 경우 이러한 보안 취약점이 발생할 수 있습니다](https://www.redhat.com/ko/topics/security/container-security). 호스트 커널은 호스트 시스템에서 훨씬 더 확실히 분리된 가상 머신과 다릅니다.

[**Docker 컨테이너의 안전성 살펴보기**](https://opensource.com/business/14/7/docker-security-selinux)

|  |
| --- |
| https://opensource.com/business/14/7/docker-security-selinux?extIdCarryOver=true&sc\_cid=701f2000001OH7EAAW  Docker 컨테이너는 정말 안전합니까? |

[Docker 데몬](https://docs.docker.com/engine/reference/commandline/dockerd/)에도 보안 문제는 있을 수 있습니다. Docker 컨테이너를 사용하고 실행하기 위해 가장 많이 사용하는 것이 컨테이너의 퍼시스턴트 런타임인 Docker 데몬입니다. Docker 데몬을 사용하려면 루트 권한이 필요하므로 이 프로세스에 액세스할 수 있는 사람과 프로세스가 상주하는 위치에 각별한 주의를 기울여야 합니다. 예를 들어, 로컬 데몬은 웹 서버와 같은 퍼블릭 위치에 상주하는 데몬보다 공격에 대한 취약성이 낮습니다.

|  |
| --- |
| **쿠버네티스(Kubernetes)란? 개념, 성능, 사용방법 및 차이점**  Kubernetes, 또는 쿠버네티스, 또는 간단히 "큐브(kube)"는 [Linux 컨테이너](https://www.redhat.com/ko/topics/containers/whats-a-linux-container) 작업을 자동화하는 오픈소스 플랫폼을 뜻합니다. 이 플랫폼에서는 컨테이너화된 애플리케이션을 배포하고 확장하는 데 수동 프로세스가 필요하지 않습니다. 즉, Linux 컨테이너를 실행하는 호스트 그룹을 함께 클러스터링할 수 있으며 쿠버네티스를 통해 이러한 클러스터를 쉽고 효율적으로 관리할 수 있습니다. 클러스터는 [퍼블릭 클라우드](https://www.redhat.com/ko/topics/cloud-computing/what-is-public-cloud), [프라이빗 클라우드](https://www.redhat.com/ko/topics/cloud-computing/what-is-private-cloud) 또는 [하이브리드 클라우드](https://www.redhat.com/ko/topics/cloud-computing/what-is-hybrid-cloud) 전체로 호스트를 확장할 수 있습니다. 이러한 이유로 쿠버네티스는 [Apache Kafka를 통한 실시간 데이터 스트리밍](https://www.redhat.com/ko/topics/integration/what-is-apache-kafka)과 같이 신속한 확장을 요하는 [클라우드 네이티브 애플리케이션](https://www.redhat.com/ko/topics/cloud-native-apps)을 호스팅하는 데 이상적인 플랫폼입니다.  하이브리드 멀티 클라우드를 위한 쿠버네티스 통합 관리전략  <https://www.youtube.com/watch?v=GEtBjTzyHQI>  쿠버네티스는 원래 Google 엔지니어들이 개발하고 설계한 플랫폼입니다. Google은 초창기에 [Linux 컨테이너 기술에 기여(cgroups)](https://ko.wikipedia.org/wiki/Cgroups)하였으며 [Google 제품이 컨테이너에서 어떻게 작동](https://cloud.google.com/solutions/best-practices-for-operating-containers?hl=ko)하는지 대중에게 공개하였습니다. 이는 Google의 클라우드 서비스를 구동하는 기술이기도 합니다. Google은 내부 플랫폼인 [Borg](https://www.redhat.com/cms/managed-files/cl-openshift-kubernetes-primer-technichal-detail-f13237wg-201807-a4-web-ko.pdf)를 통해 일주일에 20억 개 이상의 컨테이너 배포를 생성하고 있습니다. Borg는 쿠버네티스의 전신이었으며 수년 동안 Borg를 개발하는 과정에서 축적된 경험은 쿠버네티스 기술의 주요 원동력이 되었습니다.  쿠버네티스 로고에 표시된 7개의 글자는 프로젝트의 원래 이름인 '[*Project Seven of Nine*](https://cloudplatform.googleblog.com/2016/07/from-Google-to-the-world-the-Kubernetes-origin-story.html)'을 의미합니다.  Red Hat은 쿠버네티스가 출시되기 전부터 Google과 협력하여 개발에 참여한 초기 기업 중 하나였으며 쿠버네티스 업스트림 프로젝트에서 [두 번째 주요 기여자](https://www.stackalytics.com/cncf?module=kubernetes)가 되었습니다. 2015년에 Google은 새로 설립된 [클라우드 네이티브 컴퓨팅 재단(Cloud Native Computing Foundation)](https://techcrunch.com/2015/07/21/as-kubernetes-hits-1-0-google-donates-technology-to-newly-formed-cloud-native-computing-foundation-with-ibm-intel-twitter-and-others/)에 쿠버네티스 프로젝트를 [기부](https://www.cncf.io/)했습니다.  **쿠버네티스 기능과 장점**  실제 프로덕션 애플리케이션은 여러 컨테이너에 걸쳐 있으며 이러한 컨테이너는 여러 서버 호스트에 배포되어야 합니다. [컨테이너를 위한 보안](https://www.redhat.com/ko/topics/security/container-security)은 멀티레이어 구조이며 복잡할 수 있습니다. 바로 여기에 쿠버네티스가 사용됩니다. 쿠버네티스는 이러한 워크로드를 위해 규모에 맞는 컨테이너를 배포하는 데 필요한 오케스트레이션 및 관리 기능을 제공합니다. 쿠버네티스 오케스트레이션을 사용하면 여러 컨테이너에 걸쳐 애플리케이션 서비스를 구축하고 클러스터 전체에서 컨테이너의 일정을 계획하고 이러한 컨테이너를 확장하여 컨테이너의 상태를 지속적으로 관리할 수 있습니다. 쿠버네티스를 활용하면 [IT 보안을 한층 강화](https://www.redhat.com/ko/topics/security)할 수 있습니다.  쿠버네티스는 종합적인 컨테이너 인프라를 제공할 수 있도록 네트워킹, [스토리지](https://www.redhat.com/ko/topics/data-storage), 보안, 텔레메트리, 기타 서비스와 통합해야 합니다.  Kubernetes explained - diagram  물론 이는 실제 환경에서 컨테이너를 사용하는 방식에 따라 달라집니다. Linux 컨테이너를 사용하는 가장 기본적인 방식은 컨테이너를 효율적이고 빠른 가상 머신으로 다루는 것입니다. 이를 프로덕션 환경과 여러 애플리케이션으로 확장하고 나면 개별 서비스를 제공하기 위해 같은 위치에 배치된 여러 개의 컨테이너를 함께 사용해야 한다는 것을 분명히 알 수 있습니다. 따라서 환경에서 컨테이너 수가 크게 증가하며 컨테이너가 누적됨에 따라 복잡성도 증가합니다.  쿠버네티스는 컨테이너를 "포드(pod)"로 분류하여 컨테이너 급증과 관련된 여러 가지 문제를 해결합니다. 포드는 그룹화된 컨테이너에 추상화 계층을 추가하므로 사용자가 워크로드를 예약하고 네트워킹 및 저장소와 같은 필수 서비스를 컨테이너에 제공할 수 있습니다. 쿠버네티스의 또 다른 부분을 사용해 이러한 포드 전체에서 부하를 분산하고 적합한 수의 컨테이너를 실행하여 워크로드를 지원할 수 있습니다.  쿠버네티스를 올바르게 구현하고 [Atomic Registry](http://www.projectatomic.io/), [Open vSwitch](http://openvswitch.org/), [heapster](https://github.com/kubernetes/heapster), [OAuth](https://oauth.net/), [SELinux](https://selinuxproject.org/page/Main_Page)와 같은 다른 오픈소스 프로젝트를 이용해 컨테이너 인프라의 모든 부분을 오케스트레이션할 수 있습니다. 쿠버네티스의 장점 및 주요 기능 사용자의 환경에서 쿠버네티스를 사용할 경우 얻을 수 있는 주요 이점은, 특히 [클라우드를 위한 애플리케이션 개발을 최적화](https://www.redhat.com/ko/topics/cloud-native-apps)하는 중인 경우, 쿠버네티스를 통해 물리 또는 가상 머신의 클러스트에서 컨테이너를 예약하고 실행할 수 있는 플랫폼이 확보된다는 것입니다. 더 넓게 보면, 프로덕션 환경에 컨테이너 기반 인프라를 완전히 구현해서 사용할 수 있습니다. 또한 쿠버네티스는 운영 작업 자동화와 관련이 있으므로 다른 애플리케이션 플랫폼 또는 관리 시스템에서 가능한 작업의 상당수를 컨테이너를 사용해 수행할 수 있습니다.  쿠버네티스를 사용하여 수행할 수 있는 작업은 다음과 같습니다.   * 여러 호스트에 걸쳐 컨테이너를 오케스트레이션합니다. * 하드웨어를 최대한 활용하여 엔터프라이즈 애플리케이션을 실행하는 데 필요한 리소스를 극대화합니다. * 애플리케이션 배포 및 업데이트를 제어하고 자동화합니다. * 스토리지를 장착 및 추가해 스테이트풀(stateful) 애플리케이션을 실행합니다. * 컨테이너화된 애플리케이션과 해당 리소스를 즉시 확장합니다. * 선언적으로(Declaratively) 서비스를 관리함으로써, 배포한 애플리케이션이 항상 배포 목적대로 실행되도록 합니다. * 자동 배치, 자동 재시작, 자동 복제, 자동 확장을 사용해 애플리케이션 상태 확인과 셀프 복구를 수행합니다.   그러나 쿠버네티스는 다른 프로젝트를 사용해 이러한 오케스트레이션된 서비스 전체를 제공합니다. 다른 오픈소스 프로젝트를 추가하면 쿠버네티스의 기능을 완전히 실현할 수 있습니다. 이를 위해서는 다음과 같은 요소가 필요합니다.   * 레지스트리: Atomic Registry 또는 도커 레지스트리(Docker Registry)와 같은 프로젝트 * 네트워킹: OpenvSwitch 및 인텔리전트 엣지 라우팅과 같은 프로젝트 * 텔레메트리: heapster, kibana, hawkular, elastic과 같은 프로젝트 * 보안: 멀티테넌시(Multi-tenancy) 계층을 갖춘 LDAP, SELinux, RBAC, OAUTH와 같은 프로젝트 * 자동화: 설치 및 클러스터 라이프 사이클 관리를 위한 Ansible 플레이북 추가 * 서비스: 인지도 높은 애플리케이션 패턴을 사용하여 사전 생성된 풍부한 콘텐츠로 구성된 카탈로그  쿠버네티스 기술과 방식 다른 기술과 마찬가지로 쿠버네티스는 초보자가 기술을 이해하고 습득하는 데 장애가 될 수 있는 기술 용어들이 있습니다. 쿠버네티스를 이해할 수 있도록 좀 더 일반적인 용어로 분류해서 설명해보겠습니다.  **마스터:** 쿠버네티스 노드를 제어하는 머신입니다. 여기에서 모든 태스크 할당이 시작됩니다.  **노드:** 할당된 태스크를 요청대로 수행하는 시스템입니다. 쿠버네티스 마스터가 이러한 노드를 제어합니다.  **포드:** 단일 노드에 배포된 하나 이상의 컨테이너 그룹입니다. 포드에 있는 모든 컨테이너는 IP 주소, IPC, 호스트 이름, 기타 리소스를 공유하며 포드는 기본 컨테이너에서 네트워크와 스토리지를 추상화합니다 이렇게 하면 클러스터에서 컨테이너를 더 쉽게 이동할 수 있습니다.  **복제 컨트롤러:** 이 컨트롤러는 클러스터에서 실행되어야 하는 동일한 포드 사본의 개수를 제어합니다.  **서비스:** 포드에서 작업 정의를 분리합니다 쿠버네티스 서비스 프록시는 클러스터에서 다른 위치로 이동한 경우든 교체된 경우든 서비스 요청을 적절한 포드로 자동 수신합니다.  **Kubelet:** 이 서비스는 노드에서 실행되며 컨테이너 매니페스트를 읽고, 정의된 컨테이너가 시작되어 실행 중인지 확인합니다  **kubectl:** 쿠버네티스의 명령줄 설정 툴입니다. 쿠버네티스 설치와 사용방법 쿠버네티스는 오픈소스입니다. 기업에서 일반적으로 사용하는 시스템은 물론이고, 이 기술을 지원하는 정형화된 구조는 없습니다. 프로덕션을 실행하는 단계에서 쿠버네티스를 구현하는 데 문제가 발생한다면 곤란해지며 귀사의 고객 또한 마찬가지일 것입니다.  여기서 바로 [Red Hat OpenShift](https://www.redhat.com/ko/technologies/cloud-computing/openshift)가 필요합니다. OpenShift는 기업의 쿠버네티스 뿐만 아니라 다양한 분야에도 사용할 수 있습니다. OpenShift에는 레지스트리, 네트워킹, 텔레메트리, 보안, 자동화, 서비스 등을 비롯한 부가 기술이 모두 내장되어 있어서 기업이 강력하고 실효성 있는 쿠버네티스를 구축할 수 있습니다. OpenShift를 사용하면 개발자는 확장성, 제어, 오케스트레이션을 통해 컨테이너화된 새 애플케이션을 구축하고 호스팅하고 클라우드에 배포함으로써 우수한 아이디어를 새로운 비즈니스에 신속하게 연결할 수 있습니다.  무엇보다도 중요한 것은, 오픈소스 분야의 세계적인 리더인 Red Hat이 OpenShift를 지원하고 개발한다는 사실입니다. 쿠버네티스가 귀사의 인프라에 적합한지 알아보세요 Kubernetes diagram  쿠버네티스는 운영 체제(예: [Red Hat Enterprise Linux Atomic Host](https://www.redhat.com/ko/technologies/linux-platforms/old-enterprise-linux))를 기반으로 실행되며, 노드에서 실행되는 컨테이너의 포드와 상호작용합니다. 쿠버네티스 마스터는 관리자(또는 DevOps팀)으로부터 명령을 전달받고 해당 지침을 하위 노드에 이어서 전달합니다. 이 핸드오프는 여러 서비스와 연계하여 태스크에 가장 적합한 노드를 자동으로 결정합니다 그 후 리소스를 할당하고 해당 노드에 포드를 지정하여 요청된 작업을 수행합니다.  인프라 측면에서 보면 컨테이너를 관리하는 방식에는 거의 변화가 없습니다. 컨테이너를 더 높은 수준에서 제어하므로 각각의 개별 컨테이너 또는 노드를 세분화하여 관리하지 않고도 더 효과적으로 제어할 수 있습니다. 몇 가지 작업이 필요하긴 하나, 대부분은 쿠버네티스 마스터를 할당하고 노드와 포드를 정의하는 것입니다. **도커(Docker)는 어떨까요?** [Docker](https://www.redhat.com/ko/topics/containers/what-is-docker) 기술은 원래 목적대로 계속 사용되고 있습니다. 쿠버네티스가 노드에 대해 포드를 예약하면 해당 노드의 kubelet이 지정된 컨테이너를 실행하도록 Docker에 명령합니다. 그런 다음 kubelet은 Docker로부터 이러한 컨테이너의 상태를 계속해서 수집하고 마스터에서 해당 정보를 집계합니다. Docker는 해당 노드에 컨테이너를 풀링하고 이러한 노드를 평소와 같이 시작하고 중지합니다. 차이가 있다면 관리자가 모든 컨테이너의 모든 노드에서 작업을 직접 수행하는 것이 아니라 자동화된 시스템이 이러한 작업을 Docker에 요청한다는 것입니다. |

|  |
| --- |
| <https://cloud.google.com/blog/products/containers-kubernetes/from-google-to-the-world-the-kubernetes-origin-story> (구글 한글 모드로 번역이 약간 어색하네요^^)  **Google 에서 세계로 : Kubernetes 기원이야기**  쿠 버네 티스가 어떻게 탄생했는지에 대한 이야기가 여기서 시작됩니다. 2013 년 여름, 우리는 기술 인프라 책임자이자 Google의 가장 중요한 네트워크 혁신의 주요 설계자 인 Urs Holzle과 함께했습니다. 우리는 그에게 오픈 소스 컨테이너 관리 시스템을 구축하겠다는 아이디어를 제시했습니다. 그러나 잘되지 않았습니다. 아니면 우리는 생각했습니다.  우리가 그 회의에서 어떻게 끝났는지 정말로 이해하려면 약간의 백업이 필요합니다. 수년 동안 Google은 Google 검색, Gmail 및 YouTube와 같은 집약적 인 온라인 서비스를 강화하기 위해 최고의 네트워크 인프라를 조용히 구축해 왔습니다. 우리는 해야 했기때문에 모든 것을 처음부터 만들었고 초기에는 예산이 부족했습니다. 서버에서 가능한 모든 성능을 발휘하기 위해 10 년 전에 컨테이너를 실험하기 시작했습니다. 우리는 수십만 개의 작업을 실행하고 컴퓨팅을 훨씬 더 효율적으로 만드는 [Borg](http://research.google.com/pubs/pub43438.html) 라는 클러스터 관리 시스템을 구축하여 데이터 센터를 높은 활용도로 운영 할 수 있습니다.  나중에 이 동일한 인프라를 사용하여 [Google Cloud Platform](https://cloud.google.com/) 을 제공 하므로 누구나 컴퓨팅 요구 사항에 사용할 수 있습니다. 하지만 서비스로서의 인프라 플랫폼인 [Google Compute Engine을](https://cloud.google.com/compute/) 출시 하면서 흥미로운 문제를 발견했습니다. 고객은 많은 CPU에 대해 비용을 지불했지만 VM을 실행하고 있기 때문에 사용률이 매우 낮았습니다. 이에 대한 내부 솔루션이 있다는 것을 알고 있었습니다. 또한 컨테이너가 컴퓨팅의 미래라는 것을 알았습니다. 컨테이너는 확장 가능하고 휴대 가능하며 더 효율적입니다. 컨테이너 시스템 Docker는 이미 실행 중이며 훌륭하다고 생각했습니다. 하지만 Google에서 수년간의 시행 착오를 통해 알게 된 트릭은 훌륭한 컨테이너 관리 시스템이었습니다. 그것이 우리가 만들고 싶었던 것입니다.  우리는 전에 거절 당했지만 포기하지 않았습니다. 좋은 아이디어는 일반적으로 Google에서 이기고 우리는 이것이 좋은 아이디어라고 확신했습니다. 우리는 아이디어를 제시하기 위해 우리의 말을 들어 줄 사람을 만났습니다. 전환점은 운명적인 셔틀 라이드 였는데, 클라우드 부사장이자 Urs의 핵심 전략가 중 한 명인 Eric Brewer 옆에 앉아있었습니다. 나는 에릭에게 아이디어를 설명 할 수있는 중단없는 시간을 가졌고 그는 확신했다. 얼마 지나지 않아 우리는 Urs로부터 청신호를 받았습니다.  Borg 테마에 따라 Project [Seven of Nine](https://en.wikipedia.org/wiki/Seven_of_Nine) 이라는 이름을 지정했습니다 . (참고 : 원래 이름에 대한 경의로 [Kubernetes 로고](https://github.com/kubernetes/kubernetes/blob/master/logo/logo.pdf) 에 7면이 있는 [것도이 때문](https://github.com/kubernetes/kubernetes/blob/master/logo/logo.pdf) 입니다.) 우리는 [Borg](https://queue.acm.org/detail.cfm?id=2898444) 와 그 후속 제품 의 설계 및 배포를 통해 Google에서 컨테이너 관리에 대해 배운 모든 것을 통합 한 무언가를 만들고 싶었습니다. , Omega — 우아하고 간단하며 사용하기 쉬운 UI와 결합되었습니다. 3 개월 만에 공유 할 준비가 된 프로토 타입이 생겼습니다.  우리는 항상 오픈 소싱 Kubernetes가 올바른 방법이라고 믿었으며 프로젝트에 많은 이점을 가져 왔습니다. 첫째, 피드백 루프는 본질적으로 즉각적이었습니다. 문제가 있거나 제대로 작동하지 않는 것이 있으면 즉시 알 수 있었습니다. 그러나 가장 중요한 것은 많은 훌륭한 엔지니어들과 함께 작업 할 수 있었으며, 이들 중 상당수는 컨테이너 배포로 혜택을 받을 기업의 요구 사항을 실제로 이해하고 있었습니다 ( 초기 기여자의 관점 은 [Kubernetes 블로그](http://blog.kubernetes.io/2016/07/happy-k8sbday-1.html) 를 참조하십시오). 그것은 선순환이었습니다. 재능있는 엔지니어들의 작업은 프로젝트에 더 많은 관심을 가져 왔고 개선과 사용률을 더욱 증가 시켰습니다.  쿠버네티스 로고 |

<https://kubernetes.io/ko/docs/home/>

**쿠버네티스란 무엇인가?**

쿠버네티스는 컨테이너화된 워크로드와 서비스를 관리하기 위한 이식할 수 있고, 확장 가능한 오픈소스 플랫폼으로, 선언적 구성과 자동화를 모두 지원한다. 쿠버네티스는 크고 빠르게 성장하는 생태계를 가지고 있다. 쿠버네티스 서비스, 지원 그리고 도구들은 광범위하게 제공된다.

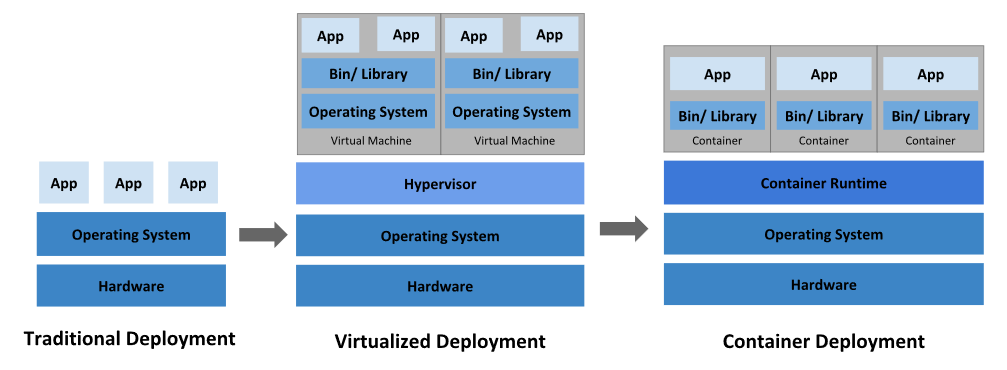
이 페이지에서는 쿠버네티스 개요를 설명한다.

쿠버네티스는 컨테이너화된 워크로드와 서비스를 관리하기 위한 이식성이 있고, 확장가능한 오픈소스 플랫폼이다. 쿠버네티스는 선언적 구성과 자동화를 모두 용이하게 해준다. 쿠버네티스는 크고, 빠르게 성장하는 생태계를 가지고 있다. 쿠버네티스 서비스, 기술 지원 및 도구는 어디서나 쉽게 이용할 수 있다.

쿠버네티스란 명칭은 키잡이(helmsman)나 파일럿을 뜻하는 그리스어에서 유래했다. 구글이 2014년에 쿠버네티스 프로젝트를 오픈소스화했다. 쿠버네티스는 프로덕션 워크로드를 대규모로 운영하는 [15년 이상의 구글 경험](https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/)과 커뮤니티의 최고의 아이디어와 적용 사례가 결합되어 있다.

## 여정 돌아보기

시간이 지나면서 쿠버네티스가 왜 유용하게 되었는지 살펴보자.



**전통적인 배포 시대:** 초기 조직은 애플리케이션을 물리 서버에서 실행했었다. 한 물리 서버에서 여러 애플리케이션의 리소스 한계를 정의할 방법이 없었기에, 리소스 할당의 문제가 발생했다. 예를 들어 물리 서버 하나에서 여러 애플리케이션을 실행하면, 리소스 전부를 차지하는 애플리케이션 인스턴스가 있을 수 있고, 결과적으로는 다른 애플리케이션의 성능이 저하될 수 있었다. 이에 대한 해결책은 서로 다른 여러 물리 서버에서 각 애플리케이션을 실행하는 것이 있다. 그러나 이는 리소스가 충분히 활용되지 않는다는 점에서 확장 가능하지 않았으므로, 물리 서버를 많이 유지하기 위해서 조직에게 많은 비용이 들었다.

**가상화된 배포 시대:** 그 해결책으로 가상화가 도입되었다. 이는 단일 물리 서버의 CPU에서 여러 가상 시스템 (VM)을 실행할 수 있게 한다. 가상화를 사용하면 VM간에 애플리케이션을 격리하고 애플리케이션의 정보를 다른 애플리케이션에서 자유롭게 액세스 할 수 없으므로, 일정 수준의 보안성을 제공할 수 있다.

가상화를 사용하면 물리 서버에서 리소스를 보다 효율적으로 활용할 수 있으며, 쉽게 애플리케이션을 추가하거나 업데이트할 수 있고 하드웨어 비용을 절감할 수 있어 더 나은 확장성을 제공한다. 가상화를 통해 일련의 물리 리소스를 폐기 가능한(disposable) 가상 머신으로 구성된 클러스터로 만들 수 있다.

각 VM은 가상화된 하드웨어 상에서 자체 운영체제를 포함한 모든 구성 요소를 실행하는 하나의 완전한 머신이다.

**컨테이너 개발 시대:** 컨테이너는 VM과 유사하지만 격리 속성을 완화하여 애플리케이션 간에 운영체제(OS)를 공유한다. 그러므로 컨테이너는 가볍다고 여겨진다. VM과 마찬가지로 컨테이너에는 자체 파일 시스템, CPU 점유율, 메모리, 프로세스 공간 등이 있다. 기본 인프라와의 종속성을 끊었기 때문에, 클라우드나 OS 배포본에 모두 이식할 수 있다.

컨테이너는 다음과 같은 추가적인 혜택을 제공하기 때문에 인기가 있다.

* 기민한 애플리케이션 생성과 배포: VM 이미지를 사용하는 것에 비해 컨테이너 이미지 생성이 보다 쉽고 효율적임.
* 지속적인 개발, 통합 및 배포: 안정적이고 주기적으로 컨테이너 이미지를 빌드해서 배포할 수 있고 (이미지의 불변성 덕에) 빠르고 효율적으로 롤백할 수 있다.
* 개발과 운영의 관심사 분리: 배포 시점이 아닌 빌드/릴리스 시점에 애플리케이션 컨테이너 이미지를 만들기 때문에, 애플리케이션이 인프라스트럭처에서 분리된다.
* 가시성은 OS 수준의 정보와 메트릭에 머무르지 않고, 애플리케이션의 헬스와 그 밖의 시그널을 볼 수 있다.
* 개발, 테스팅 및 운영 환경에 걸친 일관성: 랩탑에서도 클라우드에서와 동일하게 구동된다.
* 클라우드 및 OS 배포판 간 이식성: Ubuntu, RHEL, CoreOS, 온-프레미스, 주요 퍼블릭 클라우드와 어디에서든 구동된다.
* 애플리케이션 중심 관리: 가상 하드웨어 상에서 OS를 실행하는 수준에서 논리적인 리소스를 사용하는 OS 상에서 애플리케이션을 실행하는 수준으로 추상화 수준이 높아진다.
* 느슨하게 커플되고, 분산되고, 유연하며, 자유로운 마이크로서비스: 애플리케이션은 단일 목적의 머신에서 모놀리식 스택으로 구동되지 않고 보다 작고 독립적인 단위로 쪼개져서 동적으로 배포되고 관리될 수 있다.
* 리소스 격리: 애플리케이션 성능을 예측할 수 있다.
* 자원 사용량: 리소스 사용량: 고효율 고집적.

## 쿠버네티스가 왜 필요하고 무엇을 할 수 있나

컨테이너는 애플리케이션을 포장하고 실행하는 좋은 방법이다. 프로덕션 환경에서는 애플리케이션을 실행하는 컨테이너를 관리하고 가동 중지 시간이 없는지 확인해야 한다. 예를 들어 컨테이너가 다운되면 다른 컨테이너를 다시 시작해야 한다. 이 문제를 시스템에 의해 처리한다면 더 쉽지 않을까?

**그것이 쿠버네티스가 필요한 이유이다! 쿠버네티스는 분산 시스템을 탄력적으로 실행하기 위한 프레임 워크를 제공한다. 애플리케이션의 확장과 장애 조치를 처리하고, 배포 패턴 등을 제공한다. 예를 들어, 쿠버네티스는 시스템의 카나리아 배포를 쉽게 관리 할 수 있다.**

쿠버네티스는 다음을 제공한다.

* **서비스 디스커버리와 로드 밸런싱** 쿠버네티스는 DNS 이름을 사용하거나 자체 IP 주소를 사용하여 컨테이너를 노출할 수 있다. 컨테이너에 대한 트래픽이 많으면, 쿠버네티스는 네트워크 트래픽을 로드밸런싱하고 배포하여 배포가 안정적으로 이루어질 수 있다.
* **스토리지 오케스트레이션** 쿠버네티스를 사용하면 로컬 저장소, 공용 클라우드 공급자 등과 같이 원하는 저장소 시스템을 자동으로 탑재 할 수 있다.
* **자동화된 롤아웃과 롤백** 쿠버네티스를 사용하여 배포된 컨테이너의 원하는 상태를 서술할 수 있으며 현재 상태를 원하는 상태로 설정한 속도에 따라 변경할 수 있다. 예를 들어 쿠버네티스를 자동화해서 배포용 새 컨테이너를 만들고, 기존 컨테이너를 제거하고, 모든 리소스를 새 컨테이너에 적용할 수 있다.
* **자동화된 빈 패킹(bin packing)** 컨테이너화된 작업을 실행하는데 사용할 수 있는 쿠버네티스 클러스터 노드를 제공한다. 각 컨테이너가 필요로 하는 CPU와 메모리(RAM)를 쿠버네티스에게 지시한다. 쿠버네티스는 컨테이너를 노드에 맞추어서 리소스를 가장 잘 사용할 수 있도록 해준다.
* **자동화된 복구(self-healing)** 쿠버네티스는 실패한 컨테이너를 다시 시작하고, 컨테이너를 교체하며, '사용자 정의 상태 검사'에 응답하지 않는 컨테이너를 죽이고, 서비스 준비가 끝날 때까지 그러한 과정을 클라이언트에 보여주지 않는다.
* **시크릿과 구성 관리** 쿠버네티스를 사용하면 암호, OAuth 토큰 및 SSH 키와 같은 중요한 정보를 저장하고 관리 할 수 있다. 컨테이너 이미지를 재구성하지 않고 스택 구성에 시크릿을 노출하지 않고도 시크릿 및 애플리케이션 구성을 배포 및 업데이트 할 수 있다.

## 쿠버네티스가 아닌 것

쿠버네티스는 전통적인, 모든 것이 포함된 Platform as a Service(PaaS)가 아니다. 쿠버네티스는 하드웨어 수준보다는 컨테이너 수준에서 운영되기 때문에, PaaS가 일반적으로 제공하는 배포, 스케일링, 로드 밸런싱과 같은 기능을 제공하며, 사용자가 로깅, 모니터링 및 알림 솔루션을 통합할 수 있다. 하지만, 쿠버네티스는 모놀리식(monolithic)이 아니어서, 이런 기본 솔루션이 선택적이며 추가나 제거가 용이하다. 쿠버네티스는 개발자 플랫폼을 만드는 구성 요소를 제공하지만, 필요한 경우 사용자의 선택권과 유연성을 지켜준다.

쿠버네티스는:

* 지원하는 애플리케이션의 유형을 제약하지 않는다. 쿠버네티스는 상태 유지가 필요 없는(stateless) 워크로드, 상태 유지가 필요한(stateful) 워크로드, 데이터 처리를 위한 워크로드를 포함해서 극단적으로 다양한 워크로드를 지원하는 것을 목표로 한다. 애플리케이션이 컨테이너에서 구동될 수 있다면, 쿠버네티스에서도 잘 동작할 것이다.
* 소스 코드를 배포하지 않으며 애플리케이션을 빌드하지 않는다. 지속적인 통합과 전달과 배포, 곧 CI/CD 워크플로우는 조직 문화와 취향에 따를 뿐만 아니라 기술적인 요구사항으로 결정된다.
* 애플리케이션 레벨의 서비스를 제공하지 않는다. 애플리케이션 레벨의 서비스에는 미들웨어(예, 메시지 버스), 데이터 처리 프레임워크(예, Spark), 데이터베이스(예, MySQL), 캐시 또는 클러스터 스토리지 시스템(예, Ceph) 등이 있다. 이런 컴포넌트는 쿠버네티스 상에서 구동될 수 있고, 쿠버네티스 상에서 구동 중인 애플리케이션이 [Open Service Broker](https://openservicebrokerapi.org/) 와 같은 이식 가능한 메커니즘을 통해 접근할 수도 있다.
* 로깅, 모니터링 또는 경보 솔루션을 포함하지 않는다. 개념 증명을 위한 일부 통합이나, 메트릭을 수집하고 노출하는 메커니즘을 제공한다.
* 기본 설정 언어/시스템(예, Jsonnet)을 제공하거나 요구하지 않는다. 선언적 명세의 임의적인 형식을 목적으로 하는 선언적 API를 제공한다.
* 포괄적인 머신 설정, 유지보수, 관리, 자동 복구 시스템을 제공하거나 채택하지 않는다.
* 추가로, 쿠버네티스는 단순한 오케스트레이션 시스템이 아니다. 사실, 쿠버네티스는 오케스트레이션의 필요성을 없애준다. 오케스트레이션의 기술적인 정의는 A를 먼저 한 다음, B를 하고, C를 하는 것과 같이 정의된 워크플로우를 수행하는 것이다. 반면에, 쿠버네티스는 독립적이고 조합 가능한 제어 프로세스들로 구성되어 있다. 이 프로세스는 지속적으로 현재 상태를 입력받은 의도한 상태로 나아가도록 한다. A에서 C로 어떻게 갔는지는 상관이 없다. 중앙화된 제어도 필요치 않다. 이로써 시스템이 보다 더 사용하기 쉬워지고, 강력해지며, 견고하고, 회복력을 갖추게 되며, 확장 가능해진다.

**쿠버네티스 컴포넌트**

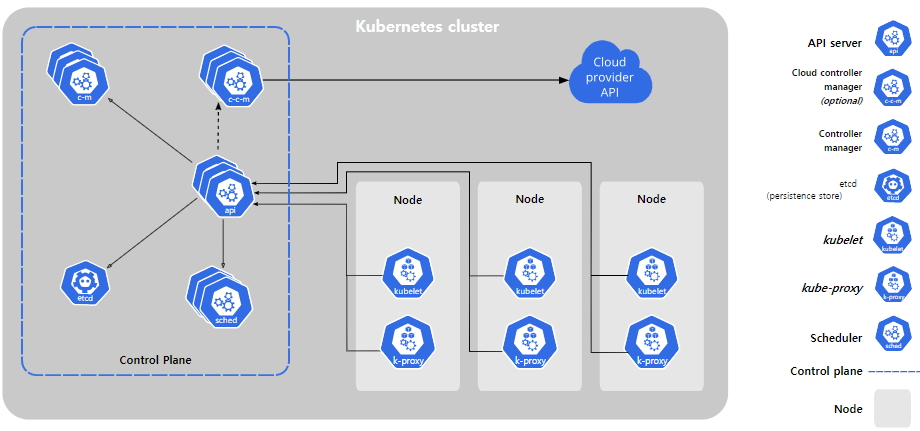
쿠버네티스를 배포하면 클러스터를 얻는다.

쿠버네티스 클러스터는 컨테이너화된 애플리케이션을 실행하는 [노드](https://kubernetes.io/ko/docs/concepts/architecture/nodes/)라고 하는 워커 머신의 집합. 모든 클러스터는 최소 한 개의 워커 노드를 가진다.

워커 노드는 애플리케이션의 구성요소인 [파드](https://kubernetes.io/ko/docs/concepts/workloads/pods/)를 호스트한다. [컨트롤 플레인](https://kubernetes.io/ko/docs/reference/glossary/?all=true#term-control-plane)은 워커 노드와 클러스터 내 파드를 관리한다. 프로덕션 환경에서는 일반적으로 컨트롤 플레인이 여러 컴퓨터에 걸쳐 실행되고, 클러스터는 일반적으로 여러 노드를 실행하므로 내결함성과 고가용성이 제공된다.

이 문서는 완전히 작동하는 쿠버네티스 클러스터를 갖기 위해 필요한 다양한 컴포넌트들에 대해 요약하고 정리한다.

여기에 모든 컴포넌트가 함께 있는 쿠버네티스 클러스터 다이어그램이 있다.



## 컨트롤 플레인 컴포넌트

컨트롤 플레인 컴포넌트는 클러스터에 관한 전반적인 결정(예를 들어, 스케줄링)을 수행하고 클러스터 이벤트(예를 들어, 디플로이먼트의 replicas 필드에 대한 요구 조건이 충족되지 않을 경우 새로운 [파드](https://kubernetes.io/ko/docs/concepts/workloads/pods/)를 구동시키는 것)를 감지하고 반응한다.

컨트롤 플레인 컴포넌트는 클러스터 내 어떠한 머신에서든지 동작할 수 있다. 그러나 간결성을 위하여, 구성 스크립트는 보통 동일 머신 상에 모든 컨트롤 플레인 컴포넌트를 구동시키고, 사용자 컨테이너는 해당 머신 상에 동작시키지 않는다. 다중-마스터-VM 설치 예제를 보려면 [고가용성 클러스터 구성하기](https://kubernetes.io/docs/admin/high-availability/)를 확인해본다.

### **kube-apiserver**

API 서버는 쿠버네티스 API를 노출하는 쿠버네티스 [컨트롤 플레인](https://kubernetes.io/ko/docs/reference/glossary/?all=true#term-control-plane) 컴포넌트이다. API 서버는 쿠버네티스 컨트롤 플레인의 프론트 엔드이다.

쿠버네티스 API 서버의 주요 구현은 [kube-apiserver](https://kubernetes.io/docs/reference/generated/kube-apiserver/) 이다. kube-apiserver는 수평으로 확장되도록 디자인되었다. 즉, 더 많은 인스턴스를 배포해서 확장할 수 있다. 여러 kube-apiserver 인스턴스를 실행하고, 인스턴스간의 트래픽을 균형있게 조절할 수 있다.

|  |
| --- |
| https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/  Kubernetes API 서버는 포드, 서비스, 복제 컨트롤러 등을 포함하는 api 개체에 대한 데이터를 검증하고 구성합니다. API 서버는 REST 작업을 서비스하고 다른 모든 구성 요소가 상호 작용하는 클러스터의 공유 상태에 대한 프런트 엔드를 제공합니다. |

### **etcd**

모든 클러스터 데이터를 담는 쿠버네티스 뒷단의 저장소로 사용되는 일관성·고가용성 키-값 저장소.

쿠버네티스 클러스터에서 etcd를 뒷단의 저장소로 사용한다면, 이 데이터를 [백업](https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/#backing-up-an-etcd-cluster)하는 계획은 필수이다.

etcd에 대한 자세한 정보는, 공식 [문서](https://etcd.io/docs)를 참고한다. (https://etcd.io/docs/)

### **kube-scheduler**

[노드](https://kubernetes.io/ko/docs/concepts/architecture/nodes/)가 배정되지 않은 새로 생성된 [파드](https://kubernetes.io/ko/docs/concepts/workloads/pods/) 를 감지하고, 실행할 노드를 선택하는 컨트롤 플레인 컴포넌트.

스케줄링 결정을 위해서 고려되는 요소는 리소스에 대한 개별 및 총체적 요구 사항, 하드웨어/소프트웨어/정책적 제약, 어피니티(affinity) 및 안티-어피니티(anti-affinity) 명세, 데이터 지역성, 워크로드-간 간섭, 데드라인을 포함한다.

### **kube-controller-manager**

[컨트롤러](https://kubernetes.io/ko/docs/concepts/architecture/controller/)를 구동하는 마스터 상의 컴포넌트.

논리적으로, 각 [컨트롤러](https://kubernetes.io/ko/docs/concepts/architecture/controller/)는 개별 프로세스이지만, 복잡성을 낮추기 위해 모두 단일 바이너리로 컴파일되고 단일 프로세스 내에서 실행된다.

이들 컨트롤러는 다음을 포함한다.

* 노드 컨트롤러: 노드가 다운되었을 때 통지와 대응에 관한 책임을 가진다.
* 레플리케이션 컨트롤러: 시스템의 모든 레플리케이션 컨트롤러 오브젝트에 대해 알맞은 수의 파드들을 유지시켜 주는 책임을 가진다.
* 엔드포인트 컨트롤러: 엔드포인트 오브젝트를 채운다(즉, 서비스와 파드를 연결시킨다.)
* 서비스 어카운트 & 토큰 컨트롤러: 새로운 네임스페이스에 대한 기본 계정과 API 접근 토큰을 생성한다.

### **cloud-controller-manager**

클라우드별 컨트롤 로직을 포함하는 쿠버네티스 [컨트롤 플레인](https://kubernetes.io/ko/docs/reference/glossary/?all=true#term-control-plane) 컴포넌트이다. 클라우드 컨트롤러 매니저를 통해 클러스터를 클라우드 공급자의 API에 연결하고, 해당 클라우드 플랫폼과 상호 작용하는 컴포넌트와 클러스터와 상호 작용하는 컴포넌트를 분리할 수 있다.

cloud-controller-manager는 클라우드 제공자 전용 컨트롤러만 실행한다. 자신의 사내 또는 PC 내부의 학습 환경에서 쿠버네티스를 실행 중인 경우 클러스터에는 클라우드 컨트롤러 매니저가 없다.

kube-controller-manager와 마찬가지로 cloud-controller-manager는 논리적으로 독립적인 여러 컨트롤 루프를 단일 프로세스로 실행하는 단일 바이너리로 결합한다. 수평으로 확장(두 개 이상의 복제 실행)해서 성능을 향상시키거나 장애를 견딜 수 있다.

다음 컨트롤러들은 클라우드 제공 사업자의 의존성을 가질 수 있다.

* 노드 컨트롤러: 노드가 응답을 멈춘 후 클라우드 상에서 삭제되었는지 판별하기 위해 클라우드 제공 사업자에게 확인하는 것
* 라우트 컨트롤러: 기본 클라우드 인프라에 경로를 구성하는 것
* 서비스 컨트롤러: 클라우드 제공 사업자 로드밸런서를 생성, 업데이트 그리고 삭제하는 것

## 노드 컴포넌트

노드 컴포넌트는 동작 중인 파드를 유지시키고 쿠버네티스 런타임 환경을 제공하며, 모든 노드 상에서 동작한다.

### **kubelet**

클러스터의 각 [노드](https://kubernetes.io/ko/docs/concepts/architecture/nodes/)에서 실행되는 에이전트. Kubelet은 [파드](https://kubernetes.io/ko/docs/concepts/workloads/pods/)에서 [컨테이너](https://kubernetes.io/ko/docs/concepts/containers/)가 확실하게 동작하도록 관리한다.

Kubelet은 다양한 메커니즘을 통해 제공된 파드 스펙(PodSpec)의 집합을 받아서 컨테이너가 해당 파드 스펙에 따라 건강하게 동작하는 것을 확실히 한다. Kubelet은 쿠버네티스를 통해 생성되지 않는 컨테이너는 관리하지 않는다.

### **kube-proxy**

kube-proxy는 클러스터의 각 [노드](https://kubernetes.io/ko/docs/concepts/architecture/nodes/)에서 실행되는 네트워크 프록시로, 쿠버네티스의 [서비스](https://kubernetes.io/docs/concepts/services-networking/service/) 개념의 구현부이다.

[kube-proxy](https://kubernetes.io/ko/docs/reference/command-line-tools-reference/kube-proxy/)는 노드의 네트워크 규칙을 유지 관리한다. 이 네트워크 규칙이 내부 네트워크 세션이나 클러스터 바깥에서 파드로 네트워크 통신을 할 수 있도록 해준다.

|  |
| --- |
| 쿠버네티스 네트워크 프록시는 각 노드에서 실행된다. 이는 각 노드의 쿠버네티스 API에 정의된 서비스를 반영하며 단순한 TCP, UDP 및 SCTP 스트림 포워딩 또는 라운드 로빈 TCP, UDP 및 SCTP 포워딩을 백엔드 셋에서 수행 할 수 있다. 서비스 클러스트 IP 및 포트는 현재 서비스 프록시에 의해 열린 포트를 지정하는 Docker-links-compatible 환경 변수를 통해 찾을 수 있다. 이러한 클러스터 IP에 클러스터 DNS를 제공하는 선택적 애드온이 있다. 유저는 apiserver API로 서비스를 생성하여 프록시를 구성해야 한다. |

kube-proxy는 운영 체제에 가용한 패킷 필터링 계층이 있는 경우, 이를 사용한다. 그렇지 않으면, kube-proxy는 트래픽 자체를 포워드(forward)한다.

### **컨테이너 런타임**

컨테이너 런타임은 컨테이너 실행을 담당하는 소프트웨어이다.

쿠버네티스는 여러 컨테이너 런타임을 지원한다. [도커(Docker)](https://docs.docker.com/engine/), [containerd](https://containerd.io/docs/), [CRI-O](https://cri-o.io/#what-is-cri-o) 그리고 [Kubernetes CRI (컨테이너 런타임 인터페이스)](https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/container-runtime-interface.md)를 구현한 모든 소프트웨어.

|  |
| --- |
| <https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/container-runtime-interface.md> CRI는 무엇입니까? CRI ( Container Runtime Interface )는 노드에서 kubelet과 통합 할 컨테이너 런타임 에 대한 사양 / 요구 사항 (추가 예정), [protobuf API](https://git.k8s.io/kubernetes/staging/src/k8s.io/cri-api/pkg/apis/runtime/v1alpha2/api.proto) 및 [라이브러리](https://git.k8s.io/kubernetes/pkg/kubelet/server/streaming) 로 구성됩니다. CRI API는 현재 알파 버전이며 CRI-Docker 통합은 Kubernetes 1.7 이상부터 기본적으로 사용됩니다. 왜 CRI를 개발합니까? CRI가 존재하기 전에 컨테이너 런타임 (예 : docker, rkt)은 kubelet에서 내부 고수준 인터페이스를 구현하여 kubelet과 통합되었습니다. 통합을 위해서는 kubelet의 내부를 이해하고 기본 Kubernetes 저장소에 기여해야했기 때문에 런타임의 진입 장벽이 높았습니다. 더 중요한 점은 새로 추가 될 때마다 기본 Kubernetes 저장소에서 상당한 유지 관리 오버 헤드가 발생하기 때문에 확장되지 않는다는 것입니다.  Kubernetes는 확장 성을 목표로합니다. CRI는 플러그 형 컨테이너 런타임을 활성화하고 더 건강한 생태계를 구축하기위한 작지만 중요한 단계입니다. CRI를 사용하는 방법? [CRI 설치 설명서를](https://kubernetes.io/docs/setup/cri/) 참조하십시오 .  ( https://kubernetes.io/docs/setup/production-environment/container-runtimes/)  CRI는 아직 어리 며 API를 개선하기 위해 개발자의 피드백을 적극적으로 통합하고 있습니다. 이전 버전과의 호환성을 유지하기 위해 노력하고 있지만 개발자는 가끔씩 API를 중단하는 변경을 예상해야합니다. Kubelet은 현재 CRI를 사용합니까? 예, Kubelet은 rktnetes 통합을 사용하는 경우를 제외하고 항상 CRI를 사용합니다.  이전의 CRI 이전 Docker 통합은 1.7에서 제거되었습니다. 사양, 설계 문서 및 제안 [CRI에](https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/) 대한 Kubernetes 1.5 [블로그 게시물](https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/) 은 일반적인 소개입니다.  아래는 CRI 사양 / 요구 사항, 설계 문서 및 제안의 혼합 목록입니다. API에 대한 더 많은 문서를 추가하기 위해 노력하고 있습니다.   * [원래 제안](https://github.com/kubernetes/kubernetes/blob/release-1.5/docs/proposals/container-runtime-interface-v1.md) * [네트워킹](https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/kubelet-cri-networking.md) * [컨테이너 메트릭](https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/cri-container-stats.md) * [실행 / 연결 / 포트 포워드 스트리밍 요청](https://docs.google.com/document/d/1OE_QoInPlVCK9rMAx9aybRmgFiVjHpJCHI9LrfdNM_s/edit?usp=sharing) * [컨테이너 stdout / stderr 로그](https://github.com/kubernetes/kubernetes/blob/release-1.5/docs/proposals/kubelet-cri-logging.md)  CRI 런타임  * [크리오](https://github.com/cri-o/cri-o) * [rktlet](https://github.com/kubernetes-retired/rktlet) * [프라 크티](https://github.com/kubernetes/frakti) * [cri-containerd](https://github.com/containerd/cri) * [특이점 -cri](https://github.com/sylabs/singularity-cri) |

## 애드온

애드온은 쿠버네티스 리소스([데몬셋](https://kubernetes.io/ko/docs/concepts/workloads/controllers/daemonset), [디플로이먼트](https://kubernetes.io/ko/docs/concepts/workloads/controllers/deployment/) 등)를 이용하여 클러스터 기능을 구현한다. 이들은 클러스터 단위의 기능을 제공하기 때문에 애드온에 대한 네임스페이스 리소스는 kube-system 네임스페이스에 속한다.

선택된 일부 애드온은 아래에 설명하였고, 사용 가능한 전체 확장 애드온 리스트는 [애드온](https://kubernetes.io/ko/docs/concepts/cluster-administration/addons/)을 참조한다.

|  |
| --- |
| <https://kubernetes.io/ko/docs/concepts/cluster-administration/addons/>  이 페이지는 사용 가능한 일부 애드온과 관련 설치 지침 링크를 나열한다. 네트워킹과 네트워크 폴리시  * [ACI](https://www.github.com/noironetworks/aci-containers)는 Cisco ACI로 통합 컨테이너 네트워킹 및 네트워크 보안을 제공한다. * [Calico](https://docs.projectcalico.org/latest/introduction/)는 네트워킹 및 네트워크 폴리시 제공자이다. Calico는 유연한 네트워킹 옵션을 지원하므로 BGP 유무에 관계없이 비-오버레이 및 오버레이 네트워크를 포함하여 가장 상황에 맞는 옵션을 선택할 수 있다. Calico는 동일한 엔진을 사용하여 서비스 메시 계층(service mesh layer)에서 호스트, 파드 및 (이스티오(istio)와 Envoy를 사용하는 경우) 애플리케이션에 대한 네트워크 폴리시를 적용한다. * [Canal](https://github.com/tigera/canal/tree/master/k8s-install)은 Flannel과 Calico를 통합하여 네트워킹 및 네트워크 폴리시를 제공한다. * [Cilium](https://github.com/cilium/cilium)은 L3 네트워크 및 네트워크 폴리시 플러그인으로 HTTP/API/L7 폴리시를 투명하게 시행할 수 있다. 라우팅 및 오버레이/캡슐화 모드를 모두 지원하며, 다른 CNI 플러그인 위에서 작동할 수 있다. * [CNI-Genie](https://github.com/Huawei-PaaS/CNI-Genie)를 사용하면 쿠버네티스는 Calico, Canal, Flannel, Romana 또는 Weave와 같은 CNI 플러그인을 완벽하게 연결할 수 있다. * [Contiv](https://contiv.github.io/)는 다양한 유스케이스와 풍부한 폴리시 프레임워크를 위해 구성 가능한 네트워킹(BGP를 사용하는 네이티브 L3, vxlan을 사용하는 오버레이, 클래식 L2 그리고 Cisco-SDN/ACI)을 제공한다. Contiv 프로젝트는 완전히 [오픈소스](https://github.com/contiv)이다. [인스톨러](https://github.com/contiv/install)는 kubeadm을 이용하거나, 그렇지 않은 경우에 대해서도 설치 옵션을 모두 제공한다. * [Contrail](https://www.juniper.net/us/en/products-services/sdn/contrail/contrail-networking/)은 [Tungsten Fabric](https://tungsten.io/)을 기반으로 하며, 오픈소스이고, 멀티 클라우드 네트워크 가상화 및 폴리시 관리 플랫폼이다. Contrail과 Tungsten Fabric은 쿠버네티스, OpenShift, OpenStack 및 Mesos와 같은 오케스트레이션 시스템과 통합되어 있으며, 가상 머신, 컨테이너/파드 및 베어 메탈 워크로드에 대한 격리 모드를 제공한다. * [Flannel](https://github.com/coreos/flannel/blob/master/Documentation/kubernetes.md)은 쿠버네티스와 함께 사용할 수 있는 오버레이 네트워크 제공자이다. * [Knitter](https://github.com/ZTE/Knitter/)는 쿠버네티스 파드에서 여러 네트워크 인터페이스를 지원하는 플러그인이다. * [Multus](https://github.com/Intel-Corp/multus-cni)는 쿠버네티스에서 SRIOV, DPDK, OVS-DPDK 및 VPP 기반 워크로드 외에 모든 CNI 플러그인(예: Calico, Cilium, Contiv, Flannel)을 지원하기 위해 쿠버네티스에서 다중 네트워크 지원을 위한 멀티 플러그인이다. * [OVN-Kubernetes](https://github.com/ovn-org/ovn-kubernetes/)는 Open vSwitch(OVS) 프로젝트에서 나온 가상 네트워킹 구현인 [OVN(Open Virtual Network)](https://github.com/ovn-org/ovn/)을 기반으로 하는 쿠버네티스용 네트워킹 제공자이다. OVN-Kubernetes는 OVS 기반 로드 밸런싱과 네트워크 폴리시 구현을 포함하여 쿠버네티스용 오버레이 기반 네트워킹 구현을 제공한다. * [OVN4NFV-K8S-Plugin](https://github.com/opnfv/ovn4nfv-k8s-plugin)은 OVN 기반의 CNI 컨트롤러 플러그인으로 클라우드 네이티브 기반 서비스 기능 체인(Service function chaining(SFC)), 다중 OVN 오버레이 네트워킹, 동적 서브넷 생성, 동적 가상 네트워크 생성, VLAN 공급자 네트워크, 직접 공급자 네트워크와 멀티 클러스터 네트워킹의 엣지 기반 클라우드 등 네이티브 워크로드에 이상적인 멀티 네티워크 플러그인이다. * [NSX-T](https://docs.vmware.com/en/VMware-NSX-T/2.0/nsxt_20_ncp_kubernetes.pdf) 컨테이너 플러그인(NCP)은 VMware NSX-T와 쿠버네티스와 같은 컨테이너 오케스트레이터 간의 통합은 물론 NSX-T와 PKS(Pivotal 컨테이너 서비스) 및 OpenShift와 같은 컨테이너 기반 CaaS/PaaS 플랫폼 간의 통합을 제공한다. * [Nuage](https://github.com/nuagenetworks/nuage-kubernetes/blob/v5.1.1-1/docs/kubernetes-1-installation.rst)는 가시성과 보안 모니터링 기능을 통해 쿠버네티스 파드와 비-쿠버네티스 환경 간에 폴리시 기반 네트워킹을 제공하는 SDN 플랫폼이다. * [Romana](https://romana.io/)는 [네트워크폴리시 API](https://kubernetes.io/ko/docs/concepts/services-networking/network-policies/)도 지원하는 파드 네트워크용 Layer 3 네트워킹 솔루션이다. Kubeadm 애드온 설치에 대한 세부 정보는 [여기](https://github.com/romana/romana/tree/master/containerize)에 있다. * [Weave Net](https://www.weave.works/docs/net/latest/kubernetes/kube-addon/)은 네트워킹 및 네트워크 폴리시를 제공하고, 네트워크 파티션의 양면에서 작업을 수행하며, 외부 데이터베이스는 필요하지 않다. |

### **DNS**

여타 애드온들이 절대적으로 요구되지 않지만, 많은 예시에서 필요로 하기 때문에 모든 쿠버네티스 클러스터는 [클러스터 DNS](https://kubernetes.io/ko/docs/concepts/services-networking/dns-pod-service/)를 갖추어야만 한다.

클러스터 DNS는 구성환경 내 다른 DNS 서버와 더불어, 쿠버네티스 서비스를 위해 DNS 레코드를 제공해주는 DNS 서버다.

쿠버네티스에 의해 구동되는 컨테이너는 DNS 검색에서 이 DNS 서버를 자동으로 포함한다.

### **웹 UI (대시보드)**

[대시보드](https://kubernetes.io/ko/docs/tasks/access-application-cluster/web-ui-dashboard/)는 쿠버네티스 클러스터를 위한 범용의 웹 기반 UI다. 사용자가 클러스터 자체뿐만 아니라, 클러스터에서 동작하는 애플리케이션에 대한 관리와 문제 해결을 할 수 있도록 해준다.

### **컨테이너 리소스 모니터링**

[컨테이너 리소스 모니터링](https://kubernetes.io/ko/docs/tasks/debug-application-cluster/resource-usage-monitoring/)은 중앙 데이터베이스 내의 컨테이너들에 대한 포괄적인 시계열 매트릭스를 기록하고 그 데이터를 열람하기 위한 UI를 제공해 준다.

### **클러스터-레벨 로깅**

[클러스터-레벨 로깅](https://kubernetes.io/ko/docs/concepts/cluster-administration/logging/) 메커니즘은 검색/열람 인터페이스와 함께 중앙 로그 저장소에 컨테이너 로그를 저장하는 책임을 진다.

**노드**

쿠버네티스는 컨테이너를 파드내에 배치하고 노드 에서 실행함으로 워크로드를 구동한다. 노드는 클러스터에 따라 가상 또는 물리적 머신일 수 있다. 각 노드는 [컨트롤 플레인](https://kubernetes.io/ko/docs/reference/glossary/?all=true#term-control-plane)에 의해 관리되며 [파드](https://kubernetes.io/ko/docs/concepts/workloads/pods/)를 실행하는 데 필요한 서비스를 포함한다.

일반적으로 클러스터에는 여러 개의 노드가 있으며, 학습 또는 리소스가 제한되는 환경에서는 하나만 있을 수도 있다.

노드의 [컴포넌트](https://kubernetes.io/ko/docs/concepts/overview/components/#%EB%85%B8%EB%93%9C-%EC%BB%B4%ED%8F%AC%EB%84%8C%ED%8A%B8)에는 [kubelet](https://kubernetes.io/docs/reference/generated/kubelet), [컨테이너 런타임](https://kubernetes.io/docs/setup/production-environment/container-runtimes) 그리고 [kube-proxy](https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/)가 포함된다.

## 관리

[API 서버](https://kubernetes.io/ko/docs/concepts/overview/components/#kube-apiserver)에 노드를 추가하는 두가지 주요 방법이 있다.

1. 노드의 kubelet으로 컨트롤 플레인에 자체 등록
2. 사용자 또는 다른 사용자가 노드 오브젝트를 수동으로 추가

노드 오브젝트 또는 노드의 kubelet으로 자체 등록한 후 컨트롤 플레인은 새 노드 오브젝트가 유효한지 확인한다. 예를 들어 다음 JSON 매니페스트에서 노드를 만들려는 경우이다.

{

**"kind"**: "Node",

**"apiVersion"**: "v1",

**"metadata"**: {

**"name"**: "10.240.79.157",

**"labels"**: {

**"name"**: "my-first-k8s-node"

}

}

}

쿠버네티스는 내부적으로 노드 오브젝트를 생성한다(표시한다). 쿠버네티스는 kubelet이 노드의 metadata.name 필드와 일치하는 API 서버에 등록이 되어있는지 확인한다. 노드가 정상이면(필요한 모든 서비스가 실행중인 경우) 파드를 실행할 수 있게 된다. 그렇지 않으면, 해당 노드는 정상이 될때까지 모든 클러스터 활동에 대해 무시된다.

**참고:**

쿠버네티스는 유효하지 않은 노드 오브젝트를 유지하고, 노드가 정상적인지 확인한다.

상태 확인을 중지하려면 사용자 또는 [컨트롤러](https://kubernetes.io/ko/docs/concepts/architecture/controller/)에서 노드 오브젝트를 명시적으로 삭제해야 한다.

노드 오브젝트의 이름은 유효한 [DNS 서브도메인 이름](https://kubernetes.io/ko/docs/concepts/overview/working-with-objects/names/#dns-%EC%84%9C%EB%B8%8C%EB%8F%84%EB%A9%94%EC%9D%B8-%EC%9D%B4%EB%A6%84)이어야 한다.

### **노드에 대한 자체-등록**

kubelet 플래그 --register-node는 참(기본값)일 경우, kubelet 은 API 서버에 스스로 등록을 시도할 것이다. 이는 대부분의 배포판에 의해 이용되는, 선호하는 패턴이다.

자체-등록에 대해, kubelet은 다음 옵션과 함께 시작된다.

* --kubeconfig - apiserver에 스스로 인증하기 위한 자격증명에 대한 경로.
* --cloud-provider - 자신에 대한 메터데이터를 읽기 위해 어떻게 [클라우드 제공자](https://kubernetes.io/ko/docs/reference/glossary/?all=true#term-cloud-provider)와 소통할지에 대한 방법.
* --register-node - 자동으로 API 서버에 등록.
* --register-with-taints - 주어진 [테인트(taint)](https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/) 리스트(콤마로 분리된 <key>=<value>:<effect>)를 가진 노드 등록.

register-node가 거짓이면 동작 안 함.

* --node-ip - 노드의 IP 주소.
* --node-labels - 클러스터에 노드를 등록할 때 추가 할 [레이블](https://kubernetes.io/ko/docs/concepts/overview/working-with-objects/labels)([NodeRestriction admission plugin](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#noderestriction)에 의해 적용되는 레이블 제한 사항 참고).
* --node-status-update-frequency - 얼마나 자주 kubelet이 마스터에 노드 상태를 게시할 지 정의.

[Node authorization mode](https://kubernetes.io/docs/reference/access-authn-authz/node/)와 [NodeRestriction admission plugin](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#noderestriction)이 활성화 되면, kubelets 은 자신의 노드 리소스를 생성/수정할 권한을 가진다.

#### 수동 노드 관리

[kubectl](https://kubernetes.io/docs/user-guide/kubectl-overview/)을 사용해서 노드 오브젝트를 생성하고 수정할 수 있다.

노드 오브젝트를 수동으로 생성하려면 kubelet 플래그를 --register-node=false 로 설정한다.

--register-node 설정과 관계 없이 노드 오브젝트를 수정할 수 있다. 예를 들어 기존 노드에 레이블을 설정하거나, 스케줄 불가로 표시할 수 있다.

파드의 노드 셀렉터와 함께 노드의 레이블을 사용해서 스케줄링을 제어할 수 있다. 예를 들어, 사용 가능한 노드의 하위 집합에서만 실행되도록 파드를 제한할 수 있다.

노드를 스케줄 불가로 표시하면 스케줄러가 해당 노드에 새 파드를 배치할 수 없지만, 노드에 있는 기존 파드에는 영향을 미치지 않는다. 이는 노드 재부팅 또는 기타 유지보수 준비 단계에서 유용하다.

노드를 스케줄 불가로 표시하려면 다음을 실행한다.

kubectl cordon $NODENAME

**참고:** [데몬셋(DaemonSet)](https://kubernetes.io/ko/docs/concepts/workloads/controllers/daemonset)에 포함되는 일부 파드는 스케줄 불가 노드에서 실행될 수 있다. 일반적으로 데몬셋은 워크로드 애플리케이션을 비우는 경우에도 노드에서 실행되어야 하는 노드 로컬 서비스를 제공한다.

## 노드 상태

노드의 상태는 다음의 정보를 포함한다.

* [주소](https://kubernetes.io/ko/docs/concepts/architecture/nodes/#addresses)
* [컨디션](https://kubernetes.io/ko/docs/concepts/architecture/nodes/#condition)
* [용량과 할당가능](https://kubernetes.io/ko/docs/concepts/architecture/nodes/#capacity)
* [정보](https://kubernetes.io/ko/docs/concepts/architecture/nodes/#info)

kubectl 을 사용해서 노드 상태와 기타 세부 정보를 볼수 있다.

kubectl describe node <insert-node-name-here>

출력되는 각 섹션은 아래에 설명되어있다.

### **주소**

이 필드의 용법은 클라우드 제공사업자 또는 베어메탈 구성에 따라 다양하다.

* HostName: 노드의 커널에 의해 알려진 호스트명이다. --hostname-override 파라미터를 통해 치환될 수 있다.
* ExternalIP: 일반적으로 노드의 IP 주소는 외부로 라우트 가능 (클러스터 외부에서 이용 가능) 하다 .
* InternalIP: 일반적으로 노드의 IP 주소는 클러스터 내에서만 라우트 가능하다.

### **컨디션**

conditions 필드는 모든 Running 노드의 상태를 기술한다. 컨디션의 예로 다음을 포함한다.

| **노드 컨디션** | **설명** |
| --- | --- |
| Ready | 노드가 상태 양호하며 파드를 수용할 준비가 되어 있는 경우 True, 노드의 상태가 불량하여 파드를 수용하지 못할 경우 False, 그리고 노드 컨트롤러가 마지막 node-monitor-grace-period (기본값 40 기간 동안 노드로부터 응답을 받지 못한 경우) Unknown |
| DiskPressure | 디스크 사이즈 상에 압박이 있는 경우, 즉 디스크 용량이 넉넉치 않은 경우 True, 반대의 경우 False |
| MemoryPressure | 노드 메모리 상에 압박이 있는 경우, 즉 노드 메모리가 넉넉치 않은 경우 True, 반대의 경우 False |
| PIDPressure | 프로세스 상에 압박이 있는 경우, 즉 노드 상에 많은 프로세스들이 존재하는 경우 True, 반대의 경우 False |
| NetworkUnavailable | 노드에 대해 네트워크가 올바르게 구성되지 않은 경우 True, 반대의 경우 False |

**참고:** 커맨드 라인 도구를 사용해서 코드화된 노드의 세부 정보를 출력하는 경우 조건에는 SchedulingDisabled 이 포함된다. SchedulingDisabled 은 쿠버네티스 API의 조건이 아니며, 대신 코드화된 노드는 사양에 스케줄 불가로 표시된다.

노드 컨디션은 JSON 오브젝트로 표현된다. 예를 들어, 다음 응답은 상태 양호한 노드를 나타낸다.

"conditions": [

{

**"type"**: "Ready",

**"status"**: "True",

**"reason"**: "KubeletReady",

**"message"**: "kubelet is posting ready status",

**"lastHeartbeatTime"**: "2019-06-05T18:38:35Z",

**"lastTransitionTime"**: "2019-06-05T11:41:27Z"

}

]

ready 컨디션의 상태가 pod-eviction-timeout ([kube-controller-manager](https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/)에 전달된 인수) 보다 더 길게 Unknown 또는 False로 유지되는 경우, 노드 상에 모든 파드는 노드 컨트롤러에 의해 삭제되도록 스케줄 된다. 기본 축출 타임아웃 기간은 **5분** 이다. 노드에 접근이 불가할 때와 같은 경우, apiserver는 노드 상의 kubelet과 통신이 불가하다. apiserver와의 통신이 재개될 때까지 파드 삭제에 대한 결정은 kubelet에 전해질 수 없다. 그 사이, 삭제되도록 스케줄 되어진 파드는 분할된 노드 상에서 계속 동작할 수도 있다.

노드 컨트롤러가 클러스터 내 동작 중지된 것을 확신할 때까지는 파드를 강제로 삭제하지 않는다. 파드가 Terminating 또는 Unknown 상태로 있을 때 접근 불가한 노드 상에서 동작되고 있는 것을 보게 될 수도 있다. 노드가 영구적으로 클러스터에서 삭제되었는지에 대한 여부를 쿠버네티스가 기반 인프라로부터 유추할 수 없는 경우, 노드가 클러스터를 영구적으로 탈퇴하게 되면, 클러스터 관리자는 손수 노드 오브젝트를 삭제해야 할 수도 있다. 쿠버네티스에서 노드 오브젝트를 삭제하면 노드 상에서 동작중인 모든 파드 오브젝트가 apiserver로부터 삭제되어 그 이름을 사용할 수 있는 결과를 낳는다.

노드 수명주기 컨트롤러는 자동으로 컨디션을 나타내는 [테인트(taints)](https://kubernetes.io/ko/docs/concepts/scheduling-eviction/taint-and-toleration/)를 생성한다. 스케줄러는 파드를 노드에 할당 할 때 노드의 테인트를 고려한다. 또한 파드는 노드의 테인트를 극복(tolerate)할 수 있는 톨러레이션(toleration)을 가질 수 있다.

|  |
| --- |
| **테인트(Taints)와 톨러레이션(Tolerations)** [노드 어피니티](https://kubernetes.io/ko/docs/concepts/scheduling-eviction/assign-pod-node/#%EC%96%B4%ED%94%BC%EB%8B%88%ED%8B%B0-affinity-%EC%99%80-%EC%95%88%ED%8B%B0-%EC%96%B4%ED%94%BC%EB%8B%88%ED%8B%B0-anti-affinity)는 [노드](https://kubernetes.io/ko/docs/concepts/architecture/nodes/) 셋을 (기본 설정 또는 어려운 요구 사항으로) 끌어들이는 [파드](https://kubernetes.io/ko/docs/concepts/workloads/pods/)의 속성이다. 테인트 는 그 반대로, 노드가 파드 셋을 제외할 수 있다.  톨러레이션 은 파드에 적용되며, 파드를 일치하는 테인트가 있는 노드에 스케줄되게 하지만 필수는 아니다.  테인트와 톨러레이션은 함께 작동하여 파드가 부적절한 노드에 스케줄되지 않게 한다. 하나 이상의 테인트가 노드에 적용된다. 이것은 노드가 테인트를 용인하지 않는 파드를 수용해서는 안 되는 것을 나타낸다. |

자세한 내용은 [컨디션별 노드 테인트하기](https://kubernetes.io/ko/docs/concepts/scheduling-eviction/taint-and-toleration/#%EC%BB%A8%EB%94%94%EC%85%98%EB%B3%84-%EB%85%B8%EB%93%9C-%ED%85%8C%EC%9D%B8%ED%8A%B8%ED%95%98%EA%B8%B0)를 참조한다.

### **용량과 할당가능**

노드 상에 사용 가능한 리소스를 나타낸다. 리소스에는 CPU, 메모리 그리고 노드 상으로 스케줄 되어질 수 있는 최대 파드 수가 있다.

용량 블록의 필드는 노드에 있는 리소스의 총량을 나타낸다. 할당가능 블록은 일반 파드에서 사용할 수 있는 노드의 리소스 양을 나타낸다.

노드에서 [컴퓨팅 리소스 예약](https://kubernetes.io/docs/tasks/administer-cluster/reserve-compute-resources/#node-allocatable)하는 방법을 배우는 동안 용량 및 할당가능 리소스에 대해 자세히 읽어보자.

### **정보**

커널 버전, 쿠버네티스 버전 (kubelet과 kube-proxy 버전), (사용하는 경우) Docker 버전, OS 이름과 같은노드에 대한 일반적인 정보를 보여준다. 이 정보는 Kubelet에 의해 노드로부터 수집된다.

### **노드 컨트롤러**

노드 [컨트롤러](https://kubernetes.io/ko/docs/concepts/architecture/controller/)는 노드의 다양한 측면을 관리하는 쿠버네티스 컨트롤 플레인 컴포넌트이다.

노드 컨트롤러는 노드가 생성되어 유지되는 동안 다양한 역할을 한다. 첫째는 등록 시점에 (CIDR 할당이 사용토록 설정된 경우) 노드에 CIDR 블럭을 할당하는 것이다.

두 번째는 노드 컨트롤러의 내부 노드 리스트를 클라우드 제공사업자의 사용 가능한 머신 리스트 정보를 근거로 최신상태로 유지하는 것이다. 클라우드 환경에서 동작 중일 경우, 노드상태가 불량할 때마다, 노드 컨트롤러는 해당 노드용 VM이 여전히 사용 가능한지에 대해 클라우드 제공사업자에게 묻는다. 사용 가능하지 않을 경우, 노드 컨트롤러는 노드 리스트로부터 그 노드를 삭제한다.

세 번째는 노드의 동작 상태를 모니터링 하는 것이다. 노드 컨트롤러는 노드가 접근 불가할 경우 (즉 노드 컨트롤러가 어떠한 사유로 하트비트 수신을 중지하는 경우, 예를 들어 노드 다운과 같은 경우이다.) NodeStatus의 NodeReady 컨디션을 ConditionUnknown으로 업데이트 하는 책임을 지고, 노드가 계속 접근 불가할 경우 나중에 노드로부터 (정상적인 종료를 이용하여) 모든 파드를 축출시킨다. (ConditionUnknown을 알리기 시작하는 기본 타임아웃 값은 40초 이고, 파드를 축출하기 시작하는 값은 5분이다.) 노드 컨트롤러는 매 --node-monitor-period 초 마다 각 노드의 상태를 체크한다.

#### 하트비트

쿠버네티스 노드에서 보내는 하트비트는 노드의 가용성을 결정하는데 도움이 된다.

하트비트의 두 가지 형태는 NodeStatus 와 [리스(Lease) 오브젝트](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.20/#lease-v1-coordination-k8s-io)이다. 각 노드에는 kube-node-lease 라는 [네임스페이스](https://kubernetes.io/ko/docs/concepts/overview/working-with-objects/namespaces) 에 관련된 리스 오브젝트가 있다. 리스는 경량 리소스로, 클러스터가 확장될 때 노드의 하트비트 성능을 향상 시킨다.

kubelet은 NodeStatus 와 리스 오브젝트를 생성하고 업데이트 할 의무가 있다.

* kubelet은 상태가 변경되거나 구성된 상태에 대한 업데이트가 없는 경우, NodeStatus 를 업데이트 한다. NodeStatus 의 기본 업데이트 주기는 5분이다(연결할 수 없는 노드의 시간 제한인 40초 보다 훨씬 길다).
* kubelet은 10초마다 리스 오브젝트를 생성하고 업데이트 한다(기본 업데이트 주기). 리스 업데이트는 NodeStatus 업데이트와는 독립적으로 발생한다. 리스 업데이트가 실패하면 kubelet에 의해 재시도하며 7초로 제한된 지수 백오프를 200 밀리초에서 부터 시작한다.

#### 안정성

대부분의 경우, 노드 컨트롤러는 초당 --node-eviction-rate(기본값 0.1)로 축출 비율을 제한한다. 이 말은 10초당 1개의 노드를 초과하여 파드 축출을 하지 않는다는 의미가 된다.

노드 축출 행위는 주어진 가용성 영역 내 하나의 노드가 상태가 불량할 경우 변화한다. 노드 컨트롤러는 영역 내 동시에 상태가 불량한 노드의 퍼센티지가 얼마나 되는지 체크한다(NodeReady 컨디션은 ConditionUnknown 또는 ConditionFalse 다.). 상태가 불량한 노드의 일부가 최소 --unhealthy-zone-threshold 기본값 0.55) 가 되면 축출 비율은 감소한다. 클러스터가 작으면 (즉 --large-cluster-size-threshold 노드 이하면 - 기본값 50) 축출은 중지되고, 그렇지 않으면 축출 비율은 초당 --secondary-node-eviction-rate(기본값 0.01)로 감소된다. 이 정책들이 가용성 영역 단위로 실행되어지는 이유는 나머지가 연결되어 있는 동안 하나의 가용성 영역이 마스터로부터 분할되어 질 수도 있기 때문이다. 만약 클러스터가 여러 클라우드 제공사업자의 가용성 영역에 걸쳐 있지 않으면, 오직 하나의 가용성 영역만 (전체 클러스터) 존재하게 된다.

노드가 가용성 영역들에 걸쳐 퍼져 있는 주된 이유는 하나의 전체 영역이 장애가 발생할 경우 워크로드가 상태 양호한 영역으로 이전되어질 수 있도록 하기 위해서이다. 그러므로, 하나의 영역 내 모든 노드들이 상태가 불량하면 노드 컨트롤러는 --node-eviction-rate 의 정상 비율로 축출한다. 코너 케이스란 모든 영역이 완전히 상태불량 (즉 클러스터 내 양호한 노드가 없는 경우) 한 경우이다. 이러한 경우, 노드 컨트롤러는 마스터 연결에 문제가 있어 일부 연결이 복원될 때까지 모든 축출을 중지하는 것으로 여긴다.

또한, 노드 컨트롤러는 파드가 테인트를 허용하지 않을 때 NoExecute 테인트 상태의 노드에서 동작하는 파드에 대한 축출 책임을 가지고 있다. 추가로, 노드 컨틀로러는 연결할 수 없거나, 준비되지 않은 노드와 같은 노드 문제에 상응하는 [테인트](https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/)를 추가한다. 이는 스케줄러가 비정상적인 노드에 파드를 배치하지 않게 된다.

**주의:** kubectl cordon 은 노드를 'unschedulable'로 표기하는데, 이는 서비스 컨트롤러가 이전에 자격 있는 로드밸런서 노드 대상 목록에서 해당 노드를 제거하기에 사실상 cordon 된 노드에서 들어오는 로드 밸런서 트래픽을 제거하는 부작용을 갖는다.

### **노드 용량**

노드 오브젝트는 노드 리소스 용량에 대한 정보(예: 사용 가능한 메모리의 양과 CPU의 수)를 추적한다. 노드의 [자체 등록](https://kubernetes.io/ko/docs/concepts/architecture/nodes/#%EB%85%B8%EB%93%9C%EC%97%90-%EB%8C%80%ED%95%9C-%EC%9E%90%EC%B2%B4-%EB%93%B1%EB%A1%9D)은 등록하는 중에 용량을 보고한다. [수동](https://kubernetes.io/ko/docs/concepts/architecture/nodes/#%EC%88%98%EB%8F%99-%EB%85%B8%EB%93%9C-%EA%B4%80%EB%A6%AC)으로 노드를 추가하는 경우 추가할 때 노드의 용량 정보를 설정해야 한다.

쿠버네티스 [스케줄러](https://kubernetes.io/docs/reference/generated/kube-scheduler/)는 노드 상에 모든 노드에 대해 충분한 리소스가 존재하도록 보장한다. 스케줄러는 노드 상에 컨테이너에 대한 요청의 합이 노드 용량보다 더 크지 않도록 체크한다. 요청의 합은 kubelet에서 관리하는 모든 컨테이너를 포함하지만, 컨테이너 런타임에 의해 직접적으로 시작된 컨 테이너는 제외되고 kubelet의 컨트롤 범위 밖에서 실행되는 모든 프로세스도 제외된다.

**참고:** 파드 형태가 아닌 프로세스에 대해 명시적으로 리소스를 확보하려면, [시스템 데몬에 사용할 리소스 예약하기](https://kubernetes.io/docs/tasks/administer-cluster/reserve-compute-resources/#system-reserved)을 본다.

## 노드 토폴로지

**FEATURE STATE:** Kubernetes v1.16 [alpha]

TopologyManager [기능 게이트(feature gate)](https://kubernetes.io/ko/docs/reference/command-line-tools-reference/feature-gates/)를 활성화 시켜두면, kubelet이 리소스 할당 결정을 할 때 토폴로지 힌트를 사용할 수 있다. 자세한 내용은 [노드의 컨트롤 토폴로지 관리 정책](https://kubernetes.io/docs/tasks/administer-cluster/topology-manager/)을 본다.

## 그레이스풀(Graceful) 노드 셧다운

**FEATURE STATE:** Kubernetes v1.20 [alpha]

GracefulNodeShutdown [기능 게이트](https://kubernetes.io/ko/docs/reference/command-line-tools-reference/feature-gates/)를 활성화한 경우 kubelet은 노드 시스템 종료를 감지하고 노드에서 실행 중인 파드를 종료한다. Kubelet은 노드가 종료되는 동안 파드가 일반 [파드 종료 프로세스](https://kubernetes.io/ko/docs/concepts/workloads/pods/pod-lifecycle/#pod-termination)를 따르도록 한다.

GracefulNodeShutdown 기능 게이트가 활성화되면 kubelet은 [systemd inhibitor locks](https://www.freedesktop.org/wiki/Software/systemd/inhibit/)를 사용하여 주어진 기간 동안 노드 종료를 지연시킨다. 종료 중에 kubelet은 두 단계로 파드를 종료시킨다.

1. 노드에서 실행 중인 일반 파드를 종료시킨다.
2. 노드에서 실행 중인 [중요(critical) 파드](https://kubernetes.io/docs/tasks/administer-cluster/guaranteed-scheduling-critical-addon-pods/#marking-pod-as-critical)를 종료시킨다.

그레이스풀 노드 셧다운 기능은 두 개의 [KubeletConfiguration](https://kubernetes.io/docs/tasks/administer-cluster/kubelet-config-file/) 옵션으로 구성된다.

* ShutdownGracePeriod:
  + 노드가 종료를 지연해야 하는 총 기간을 지정한다. 이것은 모든 일반 및 [중요 파드](https://kubernetes.io/docs/tasks/administer-cluster/guaranteed-scheduling-critical-addon-pods/#marking-pod-as-critical)의 파드 종료에 필요한 총 유예 기간에 해당한다.
* ShutdownGracePeriodCriticalPods:
  + 노드 종료 중에 [중요 파드](https://kubernetes.io/docs/tasks/administer-cluster/guaranteed-scheduling-critical-addon-pods/#marking-pod-as-critical)를 종료하는 데 사용되는 기간을 지정한다. 이는 ShutdownGracePeriod보다 작아야 한다.

예를 들어 ShutdownGracePeriod=30s, ShutdownGracePeriodCriticalPods=10s 인 경우 kubelet은 노드 종료를 30 초까지 지연시킨다. 종료하는 동안 처음 20(30-10) 초는 일반 파드의 유예 종료에 할당되고, 마지막 10 초는 [중요 파드](https://kubernetes.io/docs/tasks/administer-cluster/guaranteed-scheduling-critical-addon-pods/#marking-pod-as-critical)의 종료에 할당된다.