**Docker 소개**

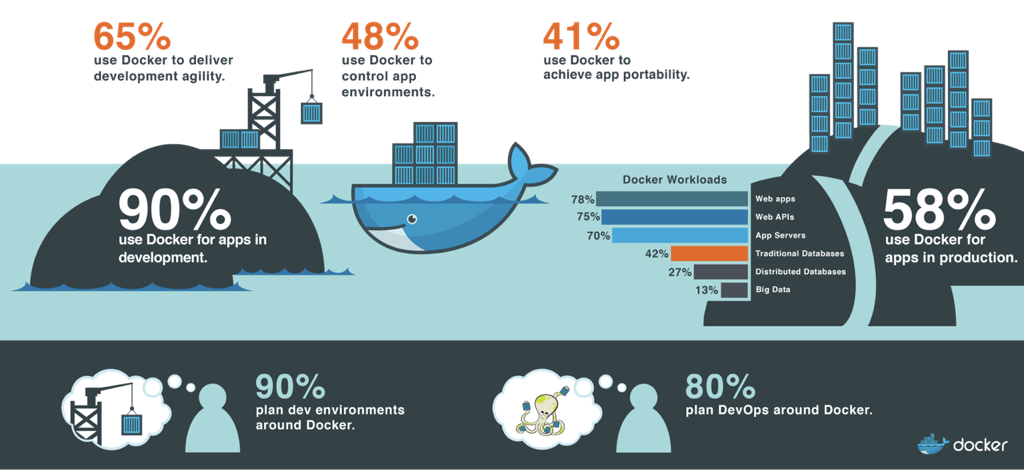
**주식회사 티시스 IT사업본부**

## 도커의 역사

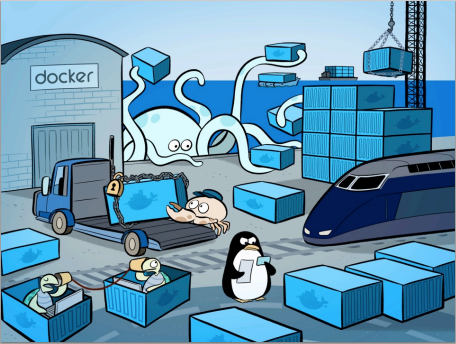


도커는 2013년 3월 산타클라라에서 열린 Pycon Conference에서 dotCloud의 창업자인 Solomon Hykes가 [The future of Linux Containers](https://youtu.be/wW9CAH9nSLs) 라는 세션을 발표하면서 처음 세상에 알려졌습니다.

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.

 2016년 설문조사에서 90%가 개발에 사용중이고 80%가 DevOps에 사용할 예정이며 58%가 운영환경에서 사용중이라고 합니다. 2014년 도커 서울 밋업을 시작할 때만 해도 대부분의 사람들이 도커를 잘 모르고 개념도 이해하지 못했는데 이제는 거의 모르는 사람이 없을 정도로 널리 쓰이고 있습니다.

## 도커란?



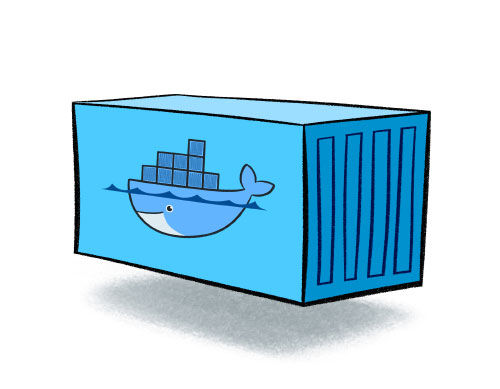
도커는 **컨테이너 기반의 오픈소스 가상화 플랫폼**입니다.

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 큐 등 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.

**컨테이너(Container)**



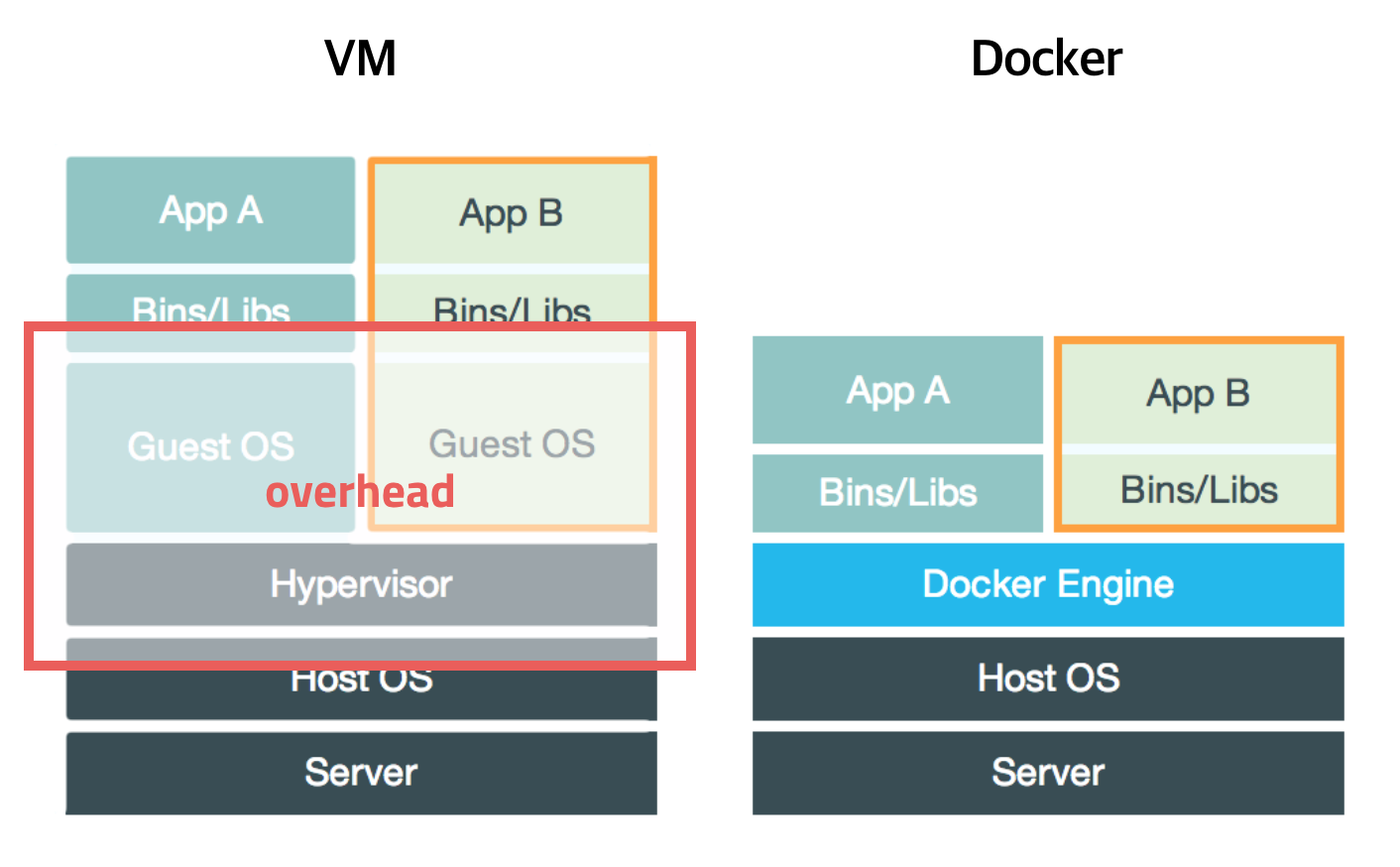
컨테이너는 격리된 공간에서 프로세스가 동작하는 기술입니다. 가상화 기술의 하나지만 기존방식과는 차이가 있습니다.

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 운영환경에서 VM 단위의 문제가 발생 시 복구 시간이 오래 걸립니다.

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.

(현재 회사에서 주로 사용하는 Nutanix HCI는 가상화 이미지를 복제 정책에 따라 여러 노드에 복사하여 VM 단위의 장애를 좀더 효율적으로 대응 할 수 있지만 확장을 위해 반드시 Nutanix를 사용해야 하는 제약사항과 서비스에서 실제 사용 할 수 있는 공간에 대해서 고민이 필요하다고 생각합니다).



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 거의 손실이 없습니다.

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

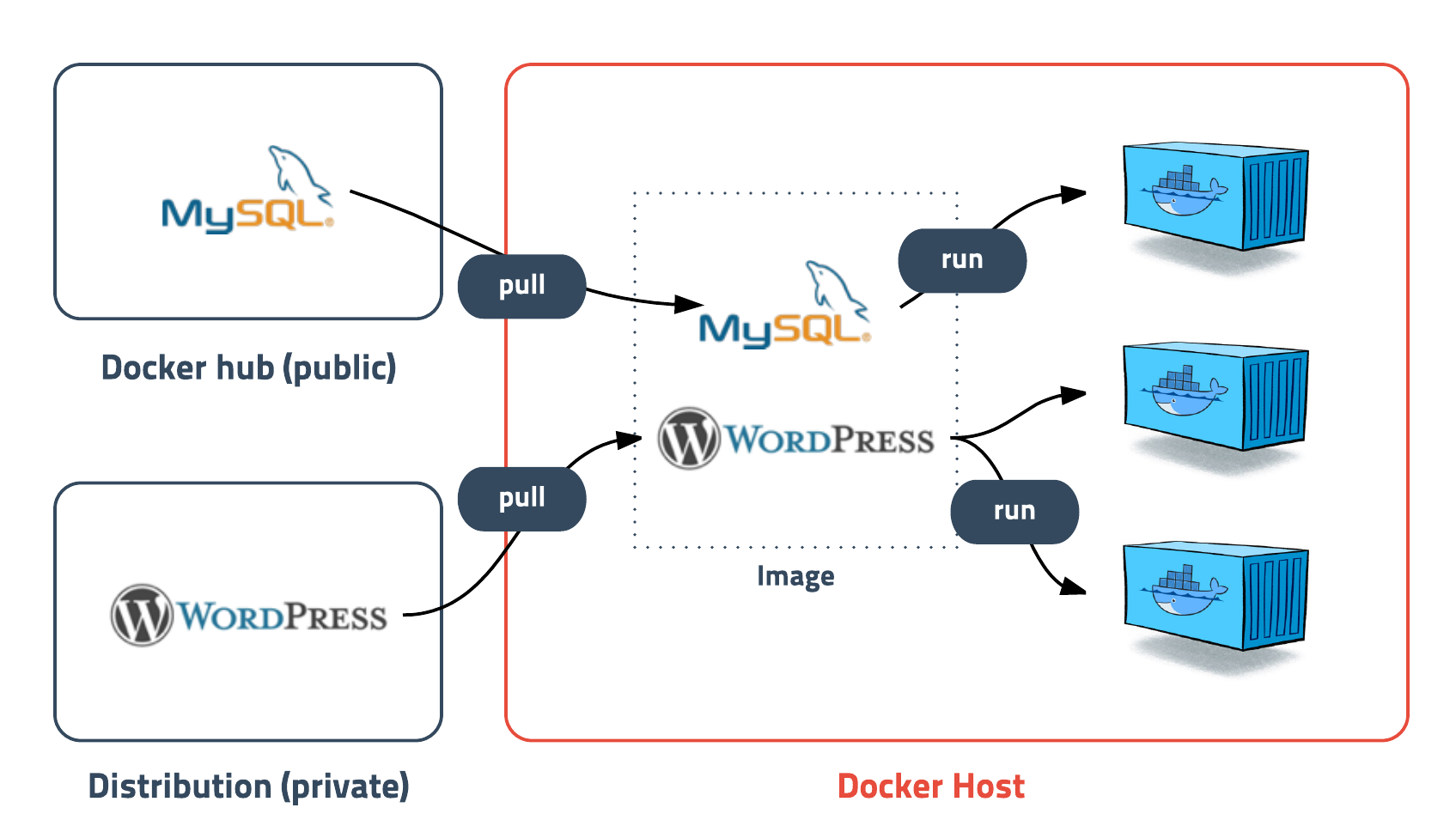
하나의 서버에 여러개의 컨테이너를 실행하면 서로 영향을 미치지 않고 독립적으로 실행되어 마치 가벼운 VMVirtual Machine을 사용하는 느낌을 줍니다. 실행중인 컨테이너에 접속하여 명령어를 입력할 수 있고 apt-get이나 yum으로 패키지를 설치할 수 있으며 사용자도 추가하고 여러개의 프로세스를 백그라운드로 실행할 수도 있습니다. CPU나 메모리 사용량을 제한할 수 있고 호스트의 특정 포트와 연결하거나 호스트의 특정 디렉토리를 내부 디렉토리인 것처럼 사용할 수도 있습니다.

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)**

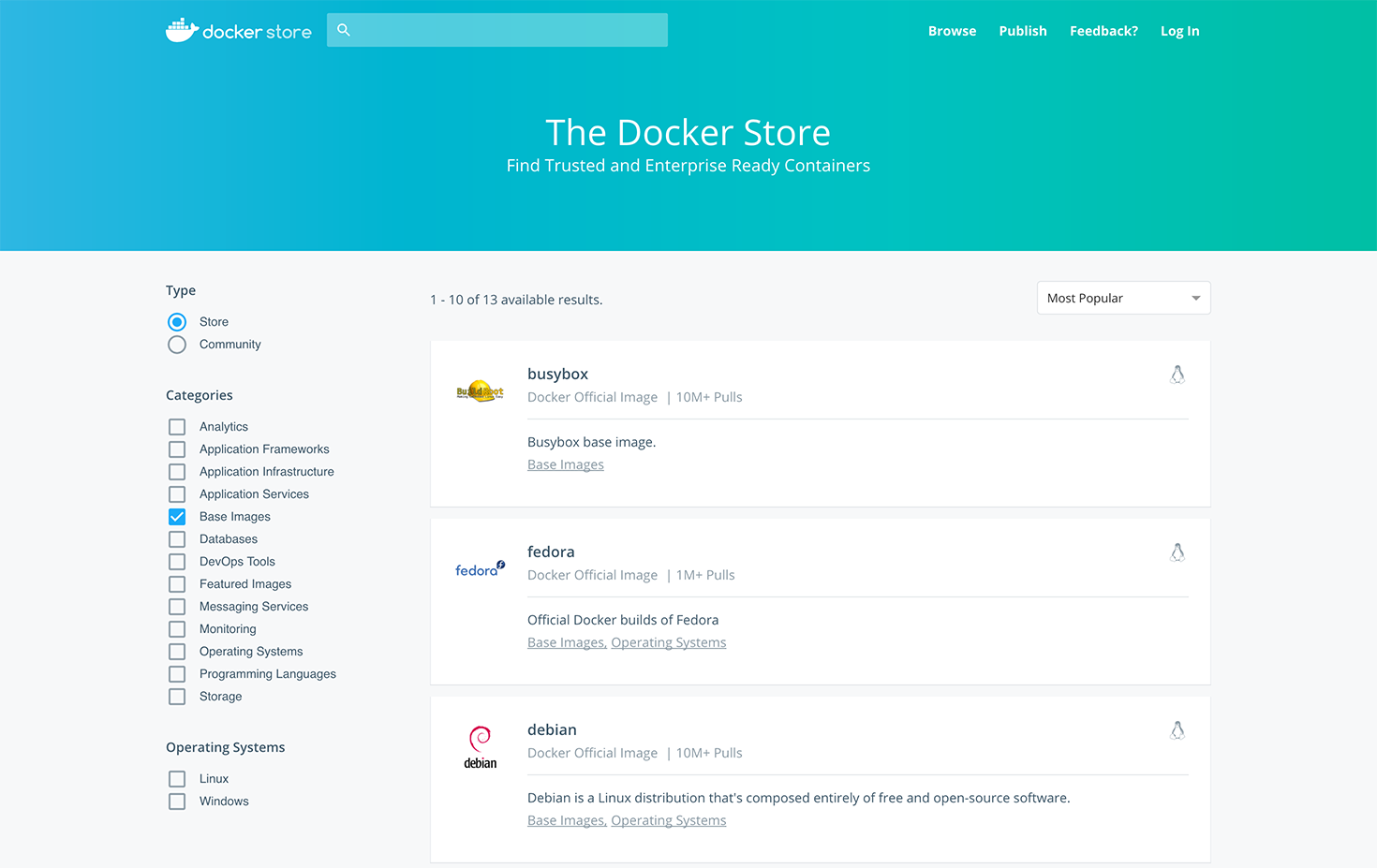


도커에서 가장 중요한 개념은 컨테이너와 함께 이미지라는 개념입니다.

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 모든 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.



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

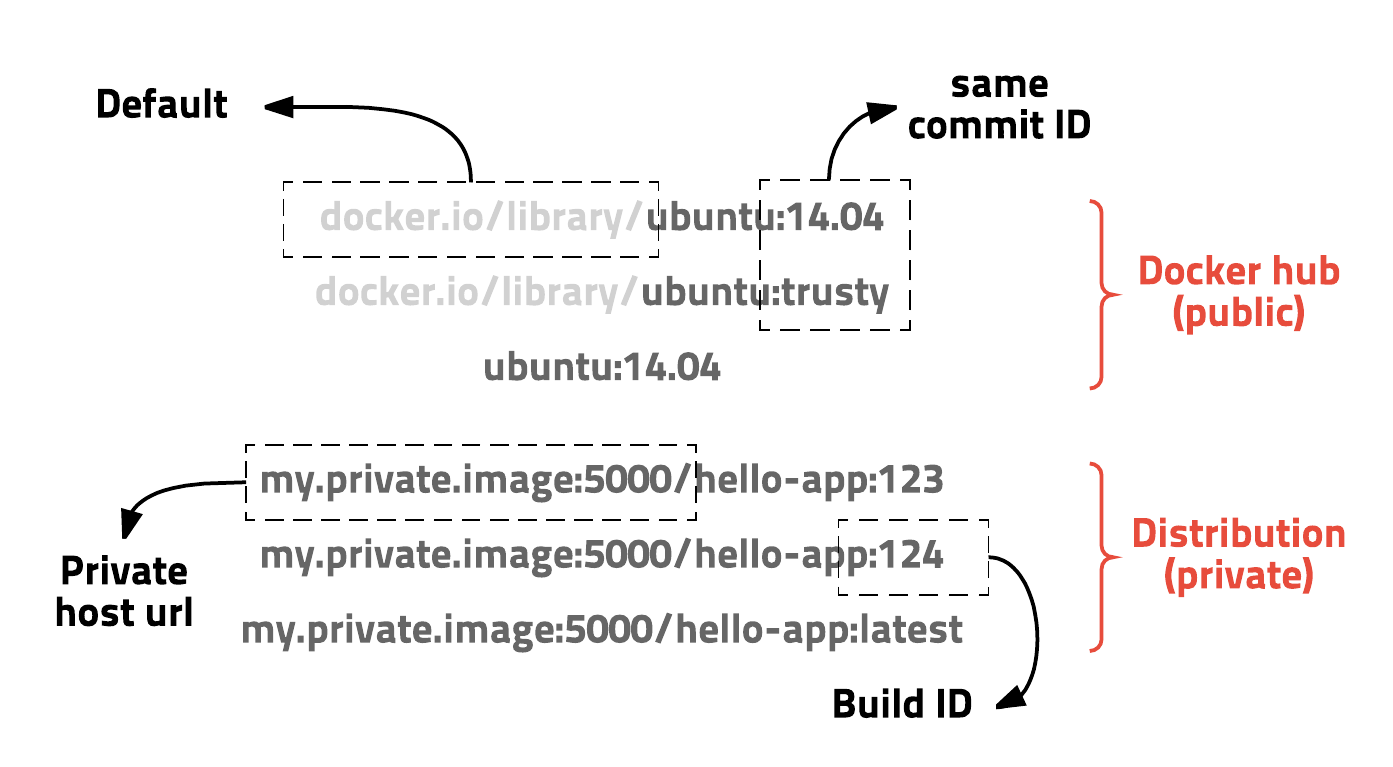
도커 이미지는 컨테이너를 실행하기 위한 모든 정보를 가지고 있기 때문에 보통 용량이 수백메가MB에 이릅니다. 처음 이미지를 다운받을 땐 크게 부담이 안되지만 기존 이미지에 파일 하나 추가했다고 수백메가를 다시 다운받는다면 매우 비효율적일 수 밖에 없습니다.

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



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.

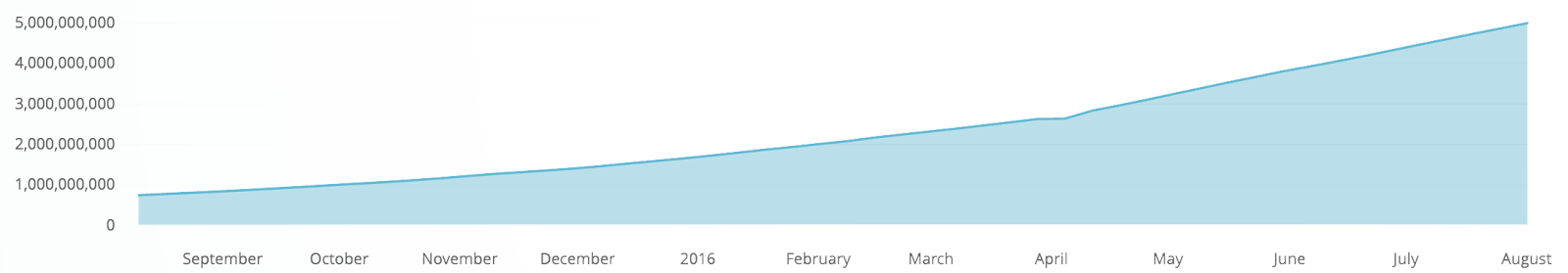
### Dockerfile

|  |  |
| --- | --- |
|  | # vertx/vertx3 debian version |
|  | FROM subicura/vertx3:3.3.1 |
|  | MAINTAINER chungsub.kim@purpleworks.co.kr |
|  |  |
|  | ADD build/distributions/app-3.3.1.tar / |
|  | ADD config.template.json /app-3.3.1/bin/config.json |
|  | ADD docker/script/start.sh /usr/local/bin/ |
|  | RUN ln -s /usr/local/bin/start.sh /start.sh |
|  |  |
|  | EXPOSE 8080 |
|  | EXPOSE 7000 |
|  |  |
|  | CMD ["start.sh"] |

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



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 잘 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.

## 도커 설치하기

도커는 리눅스 컨테이너 기술이므로 macOS나 windows에 설치할 경우 가상머신에 설치가 됩니다. 리눅스 컨테이너 말고 윈도우즈 컨테이너라는 것도 존재하지만 여기서는 다루지 않습니다.

### Linux

리눅스에 도커를 설치하는 방법은 자동 설치 스크립트를 이용하는 것이 가장 쉽습니다. 다음 명령어를 입력하면 root 권한을 요구하고 잠시 기다리면 설치가 완료됩니다. 음.. 참 쉽죠?

|  |  |
| --- | --- |
|  | curl -fsSL https://get.docker.com/ | sudo sh |

**sudo 없이 사용하기**

docker는 기본적으로 root권한이 필요합니다. root가 아닌 사용자가 sudo없이 사용하려면 해당 사용자를 docker그룹에 추가합니다.

|  |  |
| --- | --- |
|  | sudo usermod -aG docker $USER *# 현재 접속중인 사용자에게 권한주기*  sudo usermod -aG docker your-user *# your-user 사용자에게 권한주기* |

사용자가 로그인 중이라면 다시 로그인 후 권한이 적용됩니다.

**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 리눅스는 [링크](https://docs.docker.com/engine/installation/)로 대신하겠습니다.

### Docker for Mac / Docker for Windows

도커를 맥이나 윈도우즈에 설치하려면 [Docker for mac](https://docs.docker.com/docker-for-mac) 또는 [Docker for windows](https://docs.docker.com/docker-for-windows)를 설치하면 됩니다. 파일을 다운받고 설치하고 재부팅하면 대부분 문제없이 완료됩니다. 소소한 옵션들이 있는데 특별히 건드릴 부분은 없으나 한번 살펴보고 적절하게 설정하시면 됩니다. (windows는 공유 드라이브를 선택해주세요)



마치 네이티브스럽게 설치된 것 같지만 도커는 리눅스 컨테이너이므로 실제로는 가상머신에 설치가 되었습니다. 사용자는 가상머신을 사용한다는 느낌이 전혀 안드는데 그런부분을 굉장히 신경써서 설계하였습니다. 예를 들면, 포트를 연결하기 위해 도커 컨테이너의 특정 포트를 가상머신에 연결하고 다시 mac이나 windows의 포트와 연결해야 합니다. 디렉토리를 연결한다면 디렉토리를 가상머신과 공유하고 그 디렉토리를 다시 컨테이너와 연결해야 합니다. 이런 한단계 추가적으로 거쳐야하는 부분을 자연스럽게 처리해줍니다.

docker for mac은 [xhyve](https://github.com/mist64/xhyve)라는 macOS에서 제공하는 가상환경을 이용하고 docker for windows는 [Hyper-V](https://en.wikipedia.org/wiki/Hyper-V)기능을 이용합니다. 따라서 OS가 최신버전이 아니면 동작하지 않을 수 있습니다.

### 가상머신에 설치하기

이런저런 이유로 Docker for … 를 사용하지 못하는 경우 [Docker machine](https://docs.docker.com/machine/)을 이용할 수 있는데 처음 도커를 공부하는 경우에는 Virtual Box나 VMware같은 가상머신에 리눅스를 설치하고 리눅스에 접속하여 도커를 사용하는 방법을 권장합니다. 처음부터 Docker machine을 사용하면 환경이 약간 혼란스러울 수 있습니다.

### 설치확인하기

설치가 완료되었다면 정상적으로 설치되었는지 도커 명령어를 입력해 확인해 봅니다.

|  |  |
| --- | --- |
| 1 | docker version |

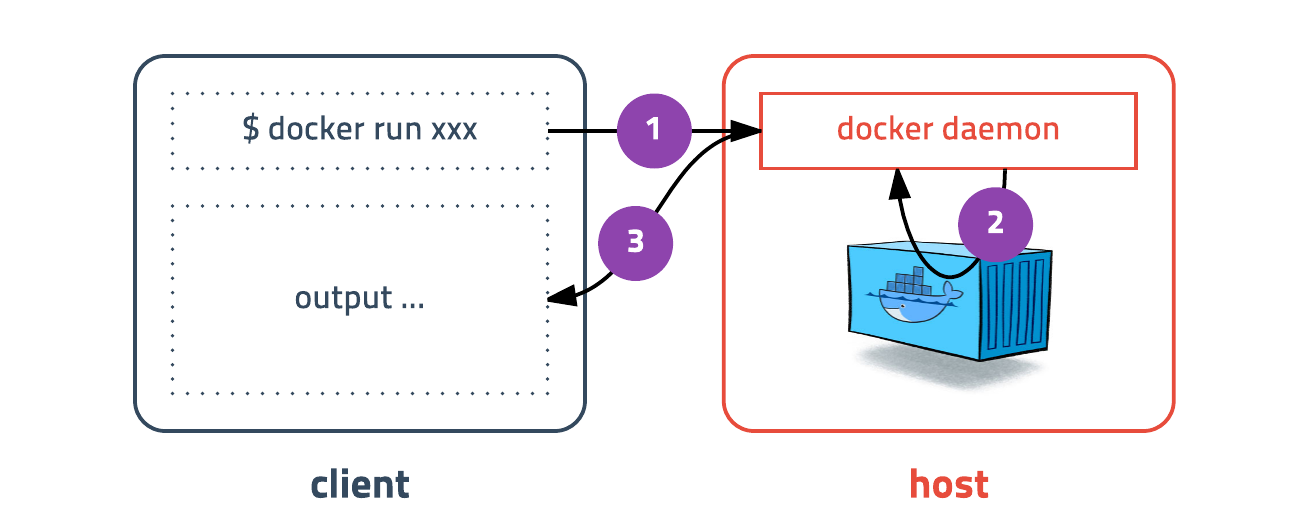
output:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | Client:  Version: 1.12.6  API version: 1.24  Go version: go1.6.4  Git commit: 78d1802  Built: Wed Jan 11 00:23:16 2017  OS/Arch: darwin/amd64  Server:  Version: 1.12.6  API version: 1.24  Go version: go1.6.4  Git commit: 78d1802  Built: Wed Jan 11 00:23:16 2017  OS/Arch: linux/amd64 |

Client와 Server 정보가 정상적으로 출력되었다면 설치가 완료된 것 입니다.

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.

혹시, 특이한 부분을 찾으셨나요? 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.



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

시작은 가볍게 ubuntu 16.04 컨테이너를 생성하고 컨테이너 내부에 들어가 봅니다.

|  |  |
| --- | --- |
| 1 | docker run ubuntu:16.04 |

run명령어를 사용하면 사용할 이미지가 저장되어 있는지 확인하고 없다면 다운로드(pull)를 한 후 컨테이너를 생성(create)하고 시작(start) 합니다.

위 예제는 ubuntu:16.04 이미지를 다운받은 적이 없기 때문에 이미지를 다운로드 한 후 컨테이너가 실행되었습니다. 컨테이너는 정상적으로 실행됐지만 뭘 하라고 명령어를 전달하지 않았기 때문에 컨테이너는 생성되자마자 종료됩니다. 컨테이너는 프로세스이기 때문에 실행중인 프로세스가 없으면 컨테이너는 종료됩니다.

이번에는 /bin/bash 명령어를 입력해서 ubuntu:16.04 컨테이너를 실행해 보겠습니다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | docker run --rm -it ubuntu:16.04 /bin/bash  *# in container*  $ cat /etc/issue  Ubuntu 16.04.1 LTS \n \l  $ ls  bin dev home lib64 mnt proc run srv tmp var  boot etc lib media opt root sbin sys usr |

컨테이너 내부에 들어가기 위해 bash 쉘을 실행하고 키보드 입력을 위해 -it 옵션을 줍니다. 추가적으로 프로세스가 종료되면 컨테이너가 자동으로 삭제되도록 --rm 옵션도 추가하였습니다.

이번에는 바로 전에 이미지를 다운 받았기 때문에 이미지를 다운로드 하는 화면 없이 바로 실행되었습니다. cat /etc/issue와 ls를 실행해보면 ubuntu 리눅스인걸 알 수 있습니다. 뭔가 가벼운 가상머신 같나요?

exit로 bash 쉘을 종료하면 컨테이너도 같이 종료됩니다.

도커를 이용하여 가장 기본적인 컨테이너를 순식간에 만들어 보았습니다.

### redis container

2번째 컨테이너는 redis 입니다. [redis](https://redis.io/)는 메모리기반의 다양한 기능을 가진 스토리지입니다. 6379 포트로 통신하며 telnet 명령어로 테스트해 볼 수 있습니다. redis 컨테이너는 detached mode(백그라운드 모드)로 실행하기 위해 -d 옵션을 추가하고 -p 옵션을 추가하여 컨테이너의 포트를 호스트의 포트로 연결해보겠습니다.

-d옵션이 없다면 프로세스가 foreground로 실행되어 아무키도 입력할 수 없게 됩니다. 컨테이너를 종료하려면 ctrl + c를 입력해 주세요.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | docker run -d -p 1234:6379 redis  *# redis test*  $ telnet localhost 1234  set mykey hello  +OK  get mykey  $5  hello |

-d 옵션을 주었기 때문에 컨테이너를 실행하자마자 컨테이너의 ID(5dff91d2…)를 보여주고 바로 쉘로 돌아왔습니다. 컨테이너는 종료된 것이 아니라 백그라운드 모드로 동작하고 있고 컨테이너 ID를 이용하여 컨테이너를 제어할 수 있습니다. -p 옵션을 이용하여 호스트의 1234포트를 컨테이너의 6379포트로 연결하였고 localhost의 1234포트로 접속하면 하면 redis를 사용할 수 있습니다.

테스트 결과 redis에 접속하여 새로운 키를 저장하고 불러오는데 성공했습니다. 실행이 간단한건 물론이고 호스트의 포트만 다르게 하면 하나의 서버에 여러개의 redis 서버를 띄우는 것도 매우 간단합니다.

### MySQL 5.7 container

3번째 실행할 컨테이너는 MySQL 서버입니다. 가장 흔하게 사용하는 데이터베이스인데 이번에는 -e 옵션을 이용하여 환경변수를 설정하고 --name 옵션을 이용하여 컨테이너에 읽기 어려운 ID 대신 쉬운 이름을 부여할 예정입니다.

--name옵션을 생략하면 도커가 자동으로 이름을 지어 줍니다. 이름은 유명한 과학자나 해커의 이름과 수식어를 조합하여 랜덤으로 [생성](https://github.com/docker/docker/blob/master/pkg/namesgenerator/names-generator.go)합니다. (ex - boring\_wozniak) 우리나라 과학자 [장영실](https://github.com/docker/docker/issues/8996)도 등록되어 있습니다.

[MySQL Docker hub](https://hub.docker.com/_/mysql/) 페이지에 접속하면 간단한 사용법과 환경변수에 대한 설명이 있습니다. 여러가지 설정값이 있는데 패스워드 없이 root계정을 만들기 위해 MYSQL\_ALLOW\_EMPTY\_PASSWORD 환경변수를 설정합니다. 그리고 컨테이너의 이름은 mysql로 할당하고 백그라운드 모드로 띄우기 위해 -d 옵션을 줍니다. 포트는 3306포트를 호스트에서 그대로 사용하겠습니다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | docker run -d -p 3306:3306 \  -e MYSQL\_ALLOW\_EMPTY\_PASSWORD**=**true \  --name mysql \  mysql:5.7  *# MySQL test*  $ mysql -h127.0.0.1 -uroot  mysql> show databases;  +--------------------+  | Database |  +--------------------+  | information\_schema |  | mysql |  | performance\_schema |  | sys |  +--------------------+  4 rows **in** set **(**0.00 sec**)**  mysql> quit |

바로 MySQL서버가 실행되었습니다. 이번 테스트는 호스트 OS에 MySQL 클라이언트가 설치되어 있어야 합니다. 추후에 실행중인 MySQL 도커 컨테이너에 접속하여 클라이언트를 실행해 보도록 하겠습니다.

처음 접속 시도시 에러가 난 화면은 MySQL 서버가 최초로 실행되면서 준비작업을 하기 때문에 발생하는 에러입니다. 컨테이너를 실행하면 백그라운드에서 MySQL 서버를 띄우는 시간이 필요하기 때문에 잠시 후에 다시 시도 했을 때 정상적으로 접속된 걸 확인할 수 있습니다.

### WordPress container

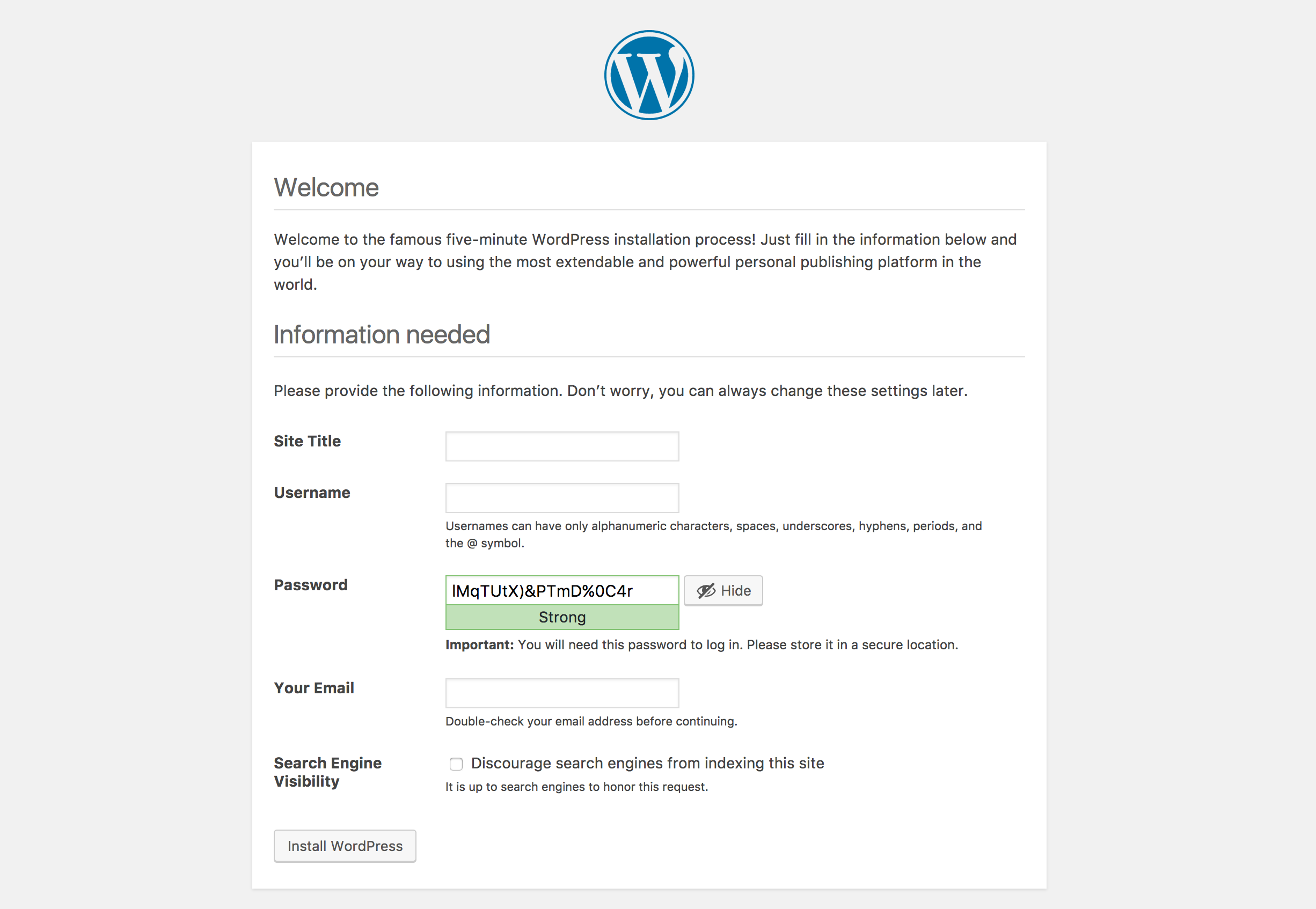
이번에는 블로그 엔진으로 유명한 [워드프레스](https://wordpress.com/)를 실행합니다. 워드프레스는 데이터베이스가 필요하기 때문에 조금 복잡한 형태를 띄지만 크게 어렵지 않습니다. 바로 전에 생성했던 MySQL 컨테이너에 워드프레스 데이터베이스를 만들고 WordPress 컨테이너를 실행할 때 --link 옵션을 이용하여 MySQL 컨테이너를 연결하겠습니다.

--link 옵션은 환경변수와 IP정보를 공유하는데 링크한 컨테이너의 IP정보를 /etc/hosts에 자동으로 입력하므로 워드프레스 컨테이너가 MySQL 데이터베이스의 정보를 알 수 있게 됩니다.

먼저, 워드프레스용 데이터베이스를 생성하고 워드프레스 컨테이너를 실행합니다. 호스트의 8080포트를 컨테이너의 80포트로 연결하고 MySQL 컨테이너와 연결한 후 각종 데이터베이스 설정 정보를 환경변수로 입력합니다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | *# create mysql database*  $ mysql -h127.0.0.1 -uroot  create database wp CHARACTER SET utf8;  grant all privileges on wp.**\*** to wp@'%' identified by 'wp';  flush privileges;  quit  *# run wordpress container*  docker run -d -p 8080:80 \  --link mysql:mysql \  -e WORDPRESS\_DB\_HOST**=**mysql \  -e WORDPRESS\_DB\_NAME**=**wp \  -e WORDPRESS\_DB\_USER**=**wp \  -e WORDPRESS\_DB\_PASSWORD**=**wp \  wordpress |

컨테이너가 제대로 실행되었는지 웹 브라우저로 확인해봅니다.



워드프레스가 실행되었습니다! 단지 이미지를 다운받고 적절한 환경변수를 입력하여 컨테이너를 실행했을 뿐입니다. 워드프레스 컨테이너 내부는 apache2와 php가 설치되어 있지만 추상화되어 있어 실행과정에선 드러나지 않습니다.

--link 옵션은 deprecated 되어 곧 사용할 수 없습니다. 대신 Docker network 기능을 이용해야 하지만 쉬운 이해를 돕기 위해 사용하였습니다.

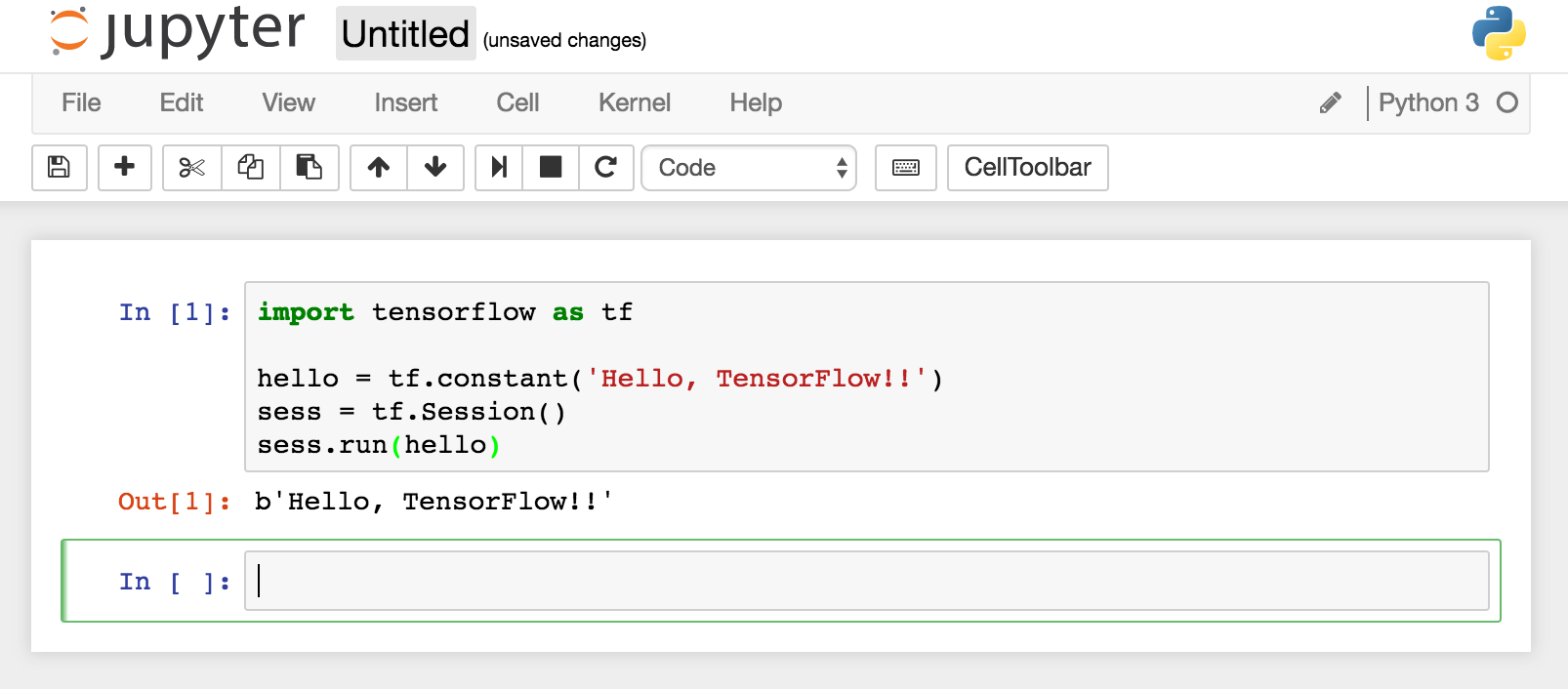
이제, 원하는 서비스가 있다면 이미지를 찾거나 직접 만들고, 어디서나 손쉽게 서비스를 실행할 수 있습니다.

### tensorflow

마지막으로 이렇게 활용할 수 있다라는 예제로 tensorflow를 실행보도록 하겠습니다. [tensorflow](https://www.tensorflow.org/)는 손쉽게 머신러닝을 할 수 있는 툴입니다. tensorflow는 python으로 만들어져 python과 관련 패키지를 설치해야 합니다. 이번에 설치하는 이미지는 python과 함께 numpy, scipy, pandas, jupyter, scikit-learn, gensim, BeautifulSoup4, Tensorflow가 설치되어 있습니다. 뭔가 복잡해 보이지만 도커라면 손쉽게 실행해 볼 수 있습니다.

|  |  |
| --- | --- |
| 1 | docker run -d -p 8888:8888 -p 6006:6006 teamlab/pydata-tensorflow:0.1 |

설치된 파일이 많아 다운로드 하는데 시간이 좀 걸립니다. 컨테이너가 실행되면 웹 브라우저에서 jupyter에 접속하여 머신러닝을 시작해 봅시다!



성공적으로 tensorflow 테스트를 마쳤습니다. 여기까지 ubuntu, MySQL, redis, Wordpress, tensorflow를 실행해 보았습니다. 가상머신을 이용해서 동일한 작업을 했다면 컴퓨터가 엄청나게 버벅이기 시작했겠지만 컨테이너 기반의 도커를 이용하여 매우 가볍게 실행하고 있습니다. 내부 구조나 설치과정은 자세히 모르지만, 간단한 도커 명령어로 여러개의 서비스를 순식간에 실행하고 사용할 수 있습니다.

## 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.

|  |  |
| --- | --- |
| 1 | docker ps |

output:

|  |  |
| --- | --- |
| 1  2  3  4  5 | CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  6a1d027b604f teamlab/pydata-tensorflow:0.1 "/opt/start" About a minute ago Up About a minute 0.0.0.0:6006->6006/tcp, 22/tcp, 0.0.0.0:8888->8888/tcp desperate\_keller  52a516f87ceb wordpress "docker-entrypoint.sh" 8 minutes ago Up 8 minutes 0.0.0.0:8080->80/tcp happy\_curran  2e2c569115b9 mysql:5.7 "docker-entrypoint.sh" 9 minutes ago Up 9 minutes 0.0.0.0:3306->3306/tcp mysql  56341072b515 redis "docker-entrypoint.sh" 16 minutes ago Up 9 minutes 0.0.0.0:1234->6379/tcp furious\_tesla |

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.

|  |  |
| --- | --- |
| 1 | docker ps -a |

output:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  6a1d027b604f teamlab/pydata-tensorflow:0.1 "/opt/start" 2 minutes ago Up 2 minutes 0.0.0.0:6006->6006/tcp, 22/tcp, 0.0.0.0:8888->8888/tcp desperate\_keller  52a516f87ceb wordpress "docker-entrypoint.sh" 9 minutes ago Up 9 minutes 0.0.0.0:8080->80/tcp happy\_curran  2e2c569115b9 mysql:5.7 "docker-entrypoint.sh" 10 minutes ago Up 10 minutes 0.0.0.0:3306->3306/tcp mysql  56341072b515 redis "docker-entrypoint.sh" 18 minutes ago Up 10 minutes 0.0.0.0:1234->6379/tcp furious\_tesla  e1a00c5934a7 ubuntu:16.04 "/bin/bash" 32 minutes ago Exited (0) 32 minutes ago berserk\_visvesvaraya |

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.

|  |  |
| --- | --- |
| 1  2  3 | docker ps *# get container ID*  docker stop **${**TENSORFLOW\_CONTAINER\_ID**}**  docker ps -a *# show all containers* |

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 |

output:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | REPOSITORY TAG IMAGE ID CREATED SIZE  wordpress latest b1fe82b15de9 43 hours ago 400.2 MB  redis latest 45c3ea2cecac 44 hours ago 182.9 MB  mysql 5.7 f3694c67abdb 46 hours ago 400.1 MB  ubuntu 16.04 104bec311bcd 4 weeks ago 129 MB  teamlab/pydata-tensorflow 0.1 7bdf5d7e0191 6 months ago 3.081 GB |

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**}** |

output:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | Untagged: teamlab/pydata-tensorflow:0.1  Untagged: teamlab/pydata-tensorflow@sha256:cb5e036afc0aa647a6fe1f384475275aeed946c751de5209d84282c986589b7f  Deleted: sha256:7bdf5d7e0191a6133a385ac69ec6a07be46b08cf5b5e6b826a89b9b47aadabe5  Deleted: sha256:9d72165f240630813a39e2f802a75f45caa2bc230881fa73c2f620c4b04686b5  Deleted: sha256:5c90ba76fff96b155b8a9446f2fb42d2ae04566832f3929df41dc52c0ae462ae  Deleted: sha256:1e23db6b02673d34c54fe87ec958ae8f2e310ca4aa911c7f8488b6c7988bfba0  Deleted: sha256:759641367a6788b55361f9eeedcbc356f8e464ef794a883b5621fba6e23c6b18  Deleted: sha256:b5d017cd48c0d8461230b5ab0aed68e27a4018996344470db3c4b5adba10b49f  Deleted: sha256:470cdb158a12a416ffebde8d299fdbde28d56da062c701bf7a51d6484f777e97  Deleted: sha256:d136c3aea28597fadb667a084a6e5701e287fd36f03cc9555b0e5bca2e674f9b  Deleted: sha256:e62e6cce767baec3711dc1ec8cfcd76f68b88a6ca9ffcd2f3d22345048a0f8d7  Deleted: sha256:8e0761516c1c1ad792c9e67d1541d574905f067cb3b67e19a3af58ca7389eee9  Deleted: sha256:ad45f6539a6ce95216c9990d26b4b2e44c0b50637c1f0949b6965610b023fd97  Deleted: sha256:ebe200261801abef54e60699c098def3a4bd39a7b4833f164b99e23a88e8a98a  Deleted: sha256:698c310f0ec8a1f607a59f15a6affa7d9d21e21d9eaf6eb65b6ab6f33fcb62b4  Deleted: sha256:e5da0b4fd6d0143d7953e47a244a9be16919d1f4018e128e82fba6df967e790e  Deleted: sha256:7d8ffbfe2c1a55581e08b81f34506a746022e5beeb85f5848154b0b7d41208cf  Deleted: sha256:854cd556e616f5e5b21cd69f1e7c0d225c9b28eb5cf653aa4aa0937930298d72  Deleted: sha256:10e6f47febf1912a4d7a08b7652e360905a6ba8b76f188fd167c8d6afa09951a  Deleted: sha256:737ee41ad49072c9ba8301a17751e16d69c1045df74ac77f4ae05f849b08faec  Deleted: sha256:62b01cf4b1440085b0b9311372eb8acb509581faf655b2837a2e234c12fdfd24  Deleted: sha256:7dcdbe5fc9a2290d596e3bab8052c3141a69a451a29a98bd12678a7139891094  Deleted: sha256:8a9f069492b445dc889f9639572dc289f7c3fc1ac4d597a7468614fa2c624091  Deleted: sha256:34a63dcc0c5574597918a8e9df80e364f9fec88c0f4e25a0e5187acd241773ad  Deleted: sha256:af6fa43db2e379d3f0f3f36d61a0ece782133314c678d7223872eb7158f102ba  Deleted: sha256:efbafc7aa3a87241c10bf0c495672cf4a7b0cc2d23db06120ae1e13528316f82  Deleted: sha256:ea0c616be3659f71afce45eff2e28e51e04b25f1ae58b306521e6877f3adf2df  Deleted: sha256:f2dccb8902ea3d85aa9a954af894f59100fc00fa78c09b213c76dcf2727e3c3e  Deleted: sha256:3fe0adbd3614d696ded87bcb47aed81f49b113770a3778e276cd88abe9010b9f  Deleted: sha256:b88d260c0bcff3b4434a39ba33f58a2f9807400d664185057edd393133630242  Deleted: sha256:4de08f41d77237b2dbc0660036fe8352fdbacf6e079623d318003ed7e833a40a  Deleted: sha256:f32c8f02713c719181c2bf32be13dbd333932411d5fc3e0b6cfb780da8bb8124  Deleted: sha256:985d69571ed5d6d7e69bf58232fd8f30f8cd4ac77f41122a80fdab56347ec22d  Deleted: sha256:8509bc9969877f784fa6e704098cbcf77d694596644f389c5a2fefe5267878ab  Deleted: sha256:0ad5b9ed74fe2eee856b368d426cdd9335a6a6ecec0c4c705b5153c219a82692  Deleted: sha256:b159c847c5b9b899c23eba59395a4d7b7b275a8c686191ed6c4ef21ceadad8a2  Deleted: sha256:89e9f76552fd1a8d249c1e133e4c8964fe1f9104bb708db1e0ca6f6171ee4c22  Deleted: sha256:6d31e1827ca8f5037077314b2403eabd82590f9eae8baf956a2c2a819db68d4e  Deleted: sha256:7f4734de8e3dd402f10030f06bcb8781129b1eb6a25c58811a76d015f2a0982f |

이미지가 삭제되었습니다. 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**}** |

output:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | WordPress not found in /var/www/html - copying now...  Complete! WordPress has been successfully copied to /var/www/html  AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.17.0.4. Set the 'ServerName' directive globally to suppress this message  AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.17.0.4. Set the 'ServerName' directive globally to suppress this message  [Thu Jan 19 16:10:16.507735 2017] [mpm\_prefork:notice] [pid 1] AH00163: Apache/2.4.10 (Debian) PHP/5.6.29 configured -- resuming normal operations  [Thu Jan 19 16:10:16.507776 2017] [core:notice] [pid 1] AH00094: Command line: 'apache2 -D FOREGROUND'  172.17.0.1 - - [19/Jan/2017:16:11:54 +0000] "GET / HTTP/1.1" 302 379 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  172.17.0.1 - - [19/Jan/2017:16:11:54 +0000] "GET /wp-admin/install.php HTTP/1.1" 200 2513 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  172.17.0.1 - - [19/Jan/2017:16:12:01 +0000] "GET /wp-includes/css/buttons.min.css?ver=4.7.1 HTTP/1.1" 200 1698 "http://localhost:8080/wp-admin/install.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  172.17.0.1 - - [19/Jan/2017:16:12:01 +0000] "GET /wp-includes/js/zxcvbn-async.min.js?ver=1.0 HTTP/1.1" 200 582 "http://localhost:8080/wp-admin/install.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  172.17.0.1 - - [19/Jan/2017:16:12:01 +0000] "GET /wp-admin/css/install.min.css?ver=4.7.1 HTTP/1.1" 200 2329 "http://localhost:8080/wp-admin/install.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  172.17.0.1 - - [19/Jan/2017:16:12:01 +0000] "GET /wp-includes/js/jquery/jquery-migrate.min.js?ver=1.4.1 HTTP/1.1" 200 4366 "http://localhost:8080/wp-admin/install.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  172.17.0.1 - - [19/Jan/2017:16:12:01 +0000] "GET /wp-includes/js/jquery/jquery.js?ver=1.12.4 HTTP/1.1" 200 34120 "http://localhost:8080/wp-admin/install.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  172.17.0.1 - - [19/Jan/2017:16:12:01 +0000] "GET /wp-admin/js/password-strength-meter.min.js?ver=4.7.1 HTTP/1.1" 200 825 "http://localhost:8080/wp-admin/install.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  172.17.0.1 - - [19/Jan/2017:16:12:01 +0000] "GET /wp-includes/js/wp-util.min.js?ver=4.7.1 HTTP/1.1" 200 924 "http://localhost:8080/wp-admin/install.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  172.17.0.1 - - [19/Jan/2017:16:12:01 +0000] "GET /wp-admin/js/user-profile.min.js?ver=4.7.1 HTTP/1.1" 200 2590 "http://localhost:8080/wp-admin/install.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  172.17.0.1 - - [19/Jan/2017:16:12:01 +0000] "GET /wp-includes/js/underscore.min.js?ver=1.8.3 HTTP/1.1" 200 6173 "http://localhost:8080/wp-admin/install.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  172.17.0.1 - - [19/Jan/2017:16:12:01 +0000] "GET /wp-includes/css/dashicons.min.css?ver=4.7.1 HTTP/1.1" 200 28951 "http://localhost:8080/wp-admin/install.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  172.17.0.1 - - [19/Jan/2017:16:12:01 +0000] "GET /wp-admin/images/wordpress-logo.svg?ver=20131107 HTTP/1.1" 200 1810 "http://localhost:8080/wp-admin/install.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  172.17.0.1 - - [19/Jan/2017:16:12:02 +0000] "GET /wp-includes/js/zxcvbn.min.js HTTP/1.1" 200 330747 "http://localhost:8080/wp-admin/install.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  172.17.0.1 - - [19/Jan/2017:16:12:02 +0000] "GET /favicon.ico HTTP/1.1" 200 229 "http://localhost:8080/wp-admin/install.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  ::1 - - [19/Jan/2017:16:12:08 +0000] "OPTIONS \* HTTP/1.0" 200 126 "-" "Apache/2.4.10 (Debian) PHP/5.6.29 (internal dummy connection)"  ::1 - - [19/Jan/2017:16:12:09 +0000] "OPTIONS \* HTTP/1.0" 200 126 "-" "Apache/2.4.10 (Debian) PHP/5.6.29 (internal dummy connection)"  ::1 - - [19/Jan/2017:16:12:10 +0000] "OPTIONS \* HTTP/1.0" 200 126 "-" "Apache/2.4.10 (Debian) PHP/5.6.29 (internal dummy connection)" |

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

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

output:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | 172.17.0.1 - - [19/Jan/2017:16:12:01 +0000] "GET /wp-includes/js/wp-util.min.js?ver=4.7.1 HTTP/1.1" 200 924 "http://localhost:8080/wp-admin/install.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  172.17.0.1 - - [19/Jan/2017:16:12:01 +0000] "GET /wp-admin/js/user-profile.min.js?ver=4.7.1 HTTP/1.1" 200 2590 "http://localhost:8080/wp-admin/install.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  172.17.0.1 - - [19/Jan/2017:16:12:01 +0000] "GET /wp-includes/js/underscore.min.js?ver=1.8.3 HTTP/1.1" 200 6173 "http://localhost:8080/wp-admin/install.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  172.17.0.1 - - [19/Jan/2017:16:12:01 +0000] "GET /wp-includes/css/dashicons.min.css?ver=4.7.1 HTTP/1.1" 200 28951 "http://localhost:8080/wp-admin/install.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  172.17.0.1 - - [19/Jan/2017:16:12:01 +0000] "GET /wp-admin/images/wordpress-logo.svg?ver=20131107 HTTP/1.1" 200 1810 "http://localhost:8080/wp-admin/install.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  172.17.0.1 - - [19/Jan/2017:16:12:02 +0000] "GET /wp-includes/js/zxcvbn.min.js HTTP/1.1" 200 330747 "http://localhost:8080/wp-admin/install.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  172.17.0.1 - - [19/Jan/2017:16:12:02 +0000] "GET /favicon.ico HTTP/1.1" 200 229 "http://localhost:8080/wp-admin/install.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12"  ::1 - - [19/Jan/2017:16:12:08 +0000] "OPTIONS \* HTTP/1.0" 200 126 "-" "Apache/2.4.10 (Debian) PHP/5.6.29 (internal dummy connection)"  ::1 - - [19/Jan/2017:16:12:09 +0000] "OPTIONS \* HTTP/1.0" 200 126 "-" "Apache/2.4.10 (Debian) PHP/5.6.29 (internal dummy connection)"  ::1 - - [19/Jan/2017:16:12:10 +0000] "OPTIONS \* HTTP/1.0" 200 126 "-" "Apache/2.4.10 (Debian) PHP/5.6.29 (internal dummy connection)" |

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.

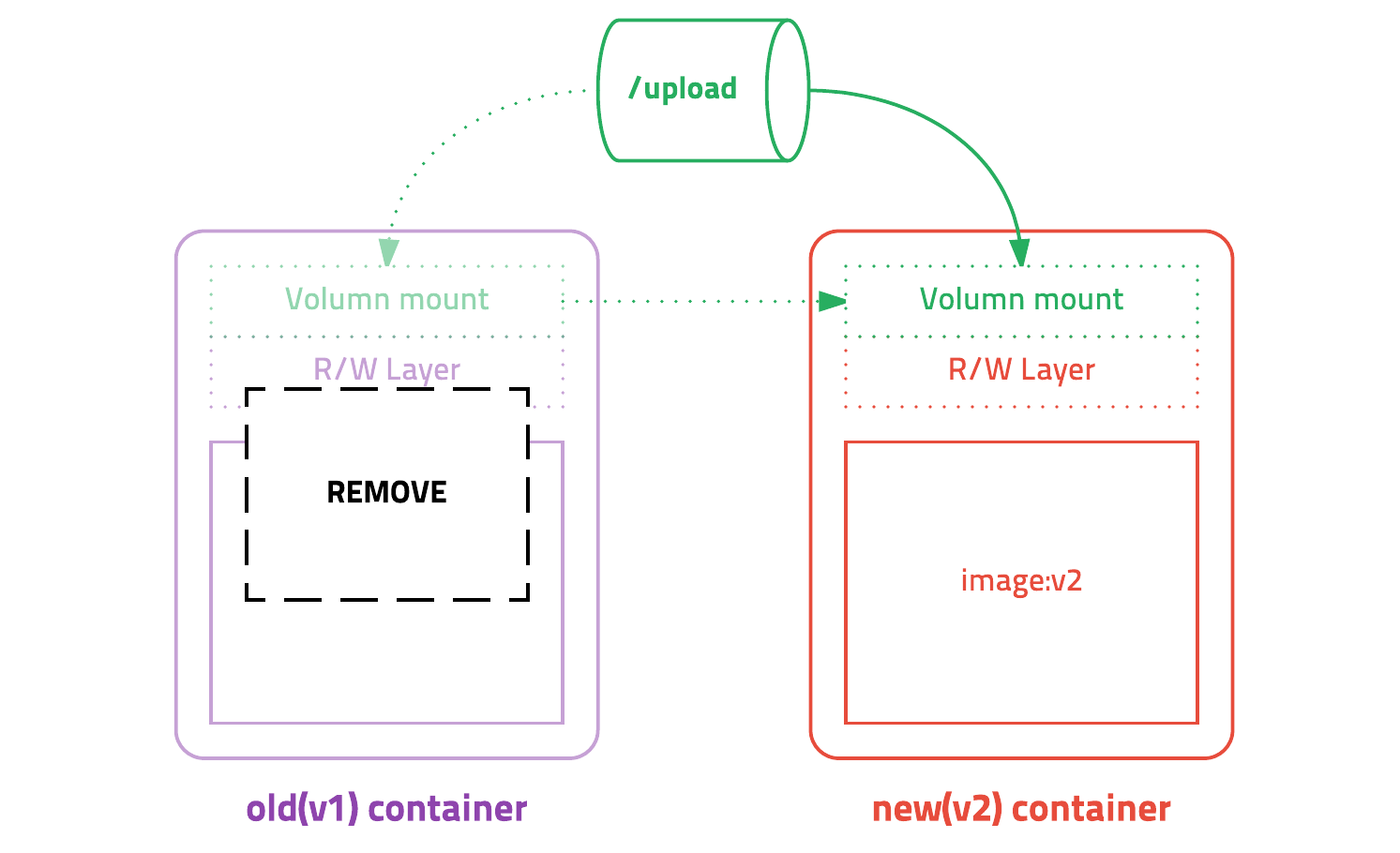
쉘로 완전한 권한을 얻는 방법말고 바로 mysql명령어를 실행 할 수도 있습니다.

|  |  |
| --- | --- |
| 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.

 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.

### 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: |

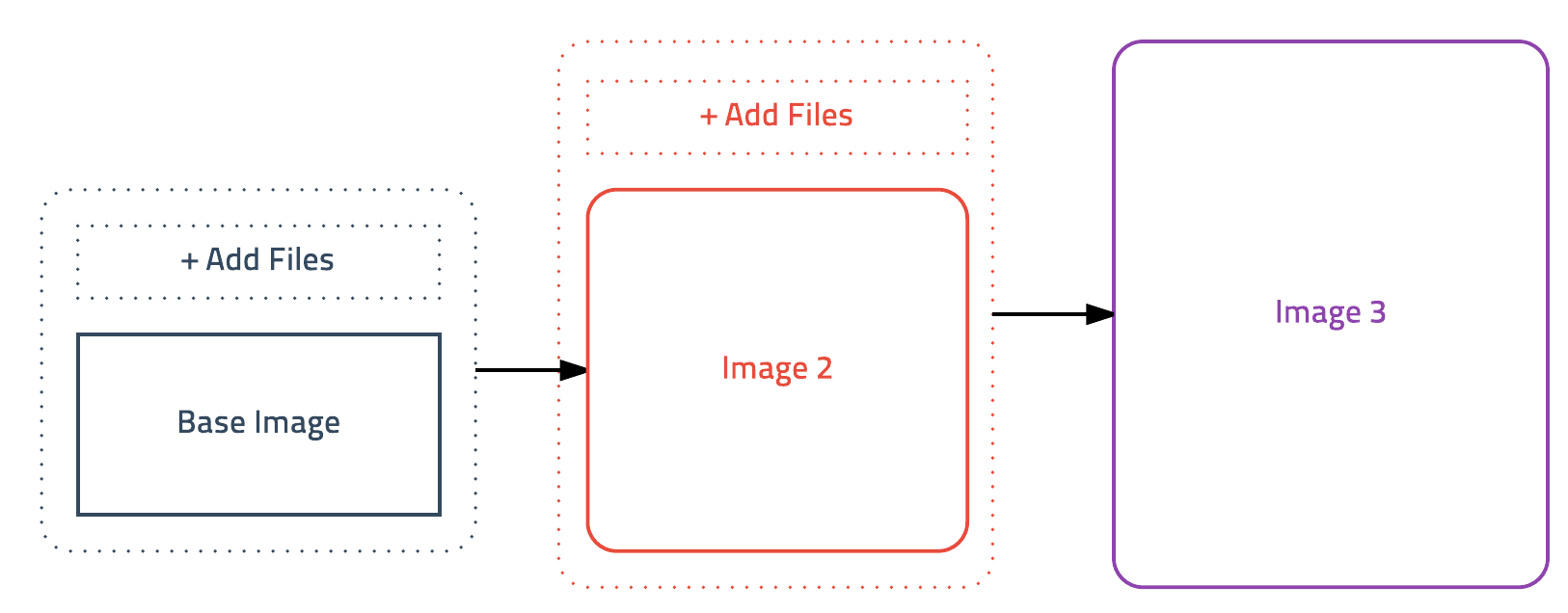
몇몇 생소해보이는 설정이 눈에 보이지만, 일단 실행해 봅니다.

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

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

## 도커 이미지 만들기

도커는 이미지를 만들기 위해 **컨테이너의 상태를 그대로 이미지로 저장**하는 방법을 사용합니다.



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

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* |

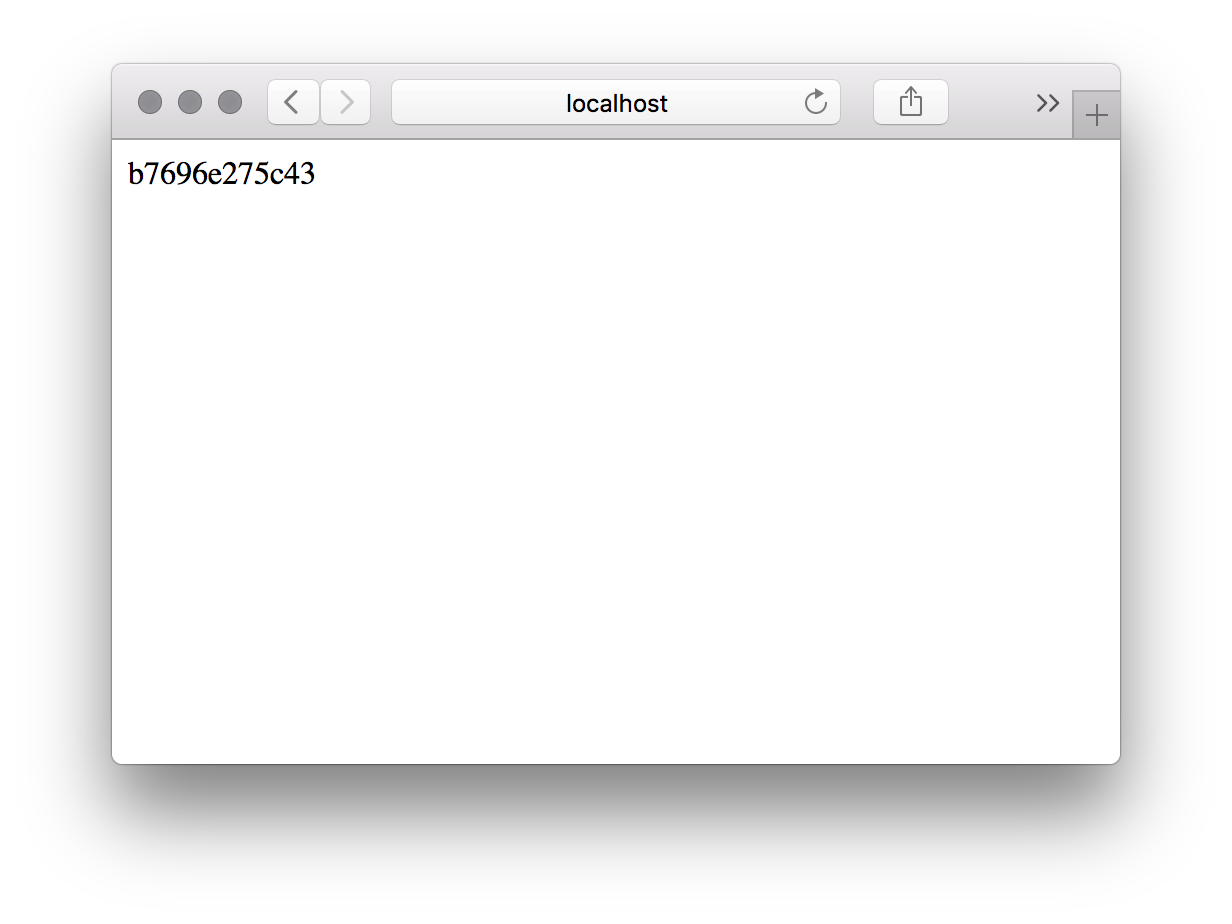
ruby가 설치되어 있지 않다고요? 도커만 있으면 문제없습니다. 다음 명령어를 실행합니다.

|  |  |
| --- | --- |
| 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" |

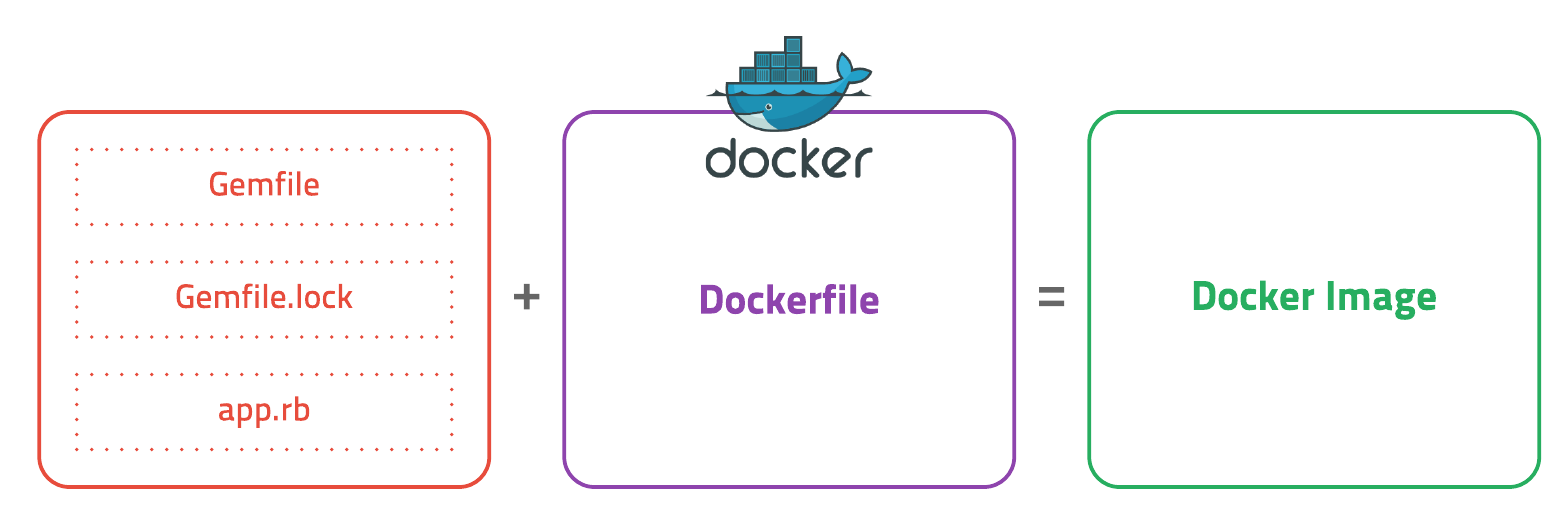
호스트의 디렉토리를 루비가 설치된 컨테이너의 디렉토리에 마운트한다음 그대로 명령어를 실행하면 로컬에 개발 환경을 구축하지 않고 도커 컨테이너를 개발환경으로 사용할 수 있습니다.

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

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

### Ruby Application 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 . |

output:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | Sending build context to Docker daemon 22.02 kB  Step 1/10 : FROM ubuntu:16.04  ---> f49eec89601e  Step 2/10 : MAINTAINER subicura@subicura.com  ---> Running in 06f20ac1017d  ---> fc41cd8ac52d  Removing intermediate container 06f20ac1017d  Step 3/10 : RUN apt-get -y update  ---> Running in c35738e75a51  Get:1 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]  Get:2 http://archive.ubuntu.com/ubuntu xenial-updates InRelease [102 kB]  Get:3 http://archive.ubuntu.com/ubuntu xenial-security InRelease [102 kB]  Get:4 http://archive.ubuntu.com/ubuntu xenial/main Sources [1103 kB]  Get:5 http://archive.ubuntu.com/ubuntu xenial/restricted Sources [5179 B]  ... 생략 ...  Step 9/10 : EXPOSE 4567  ---> Running in 9c514a1f0c8e  ---> c5ce4376282e  Removing intermediate container 9c514a1f0c8e  Step 10/10 : CMD bundle exec ruby app.rb -o 0.0.0.0  ---> Running in 1f7a9ba8d63c  ---> 54d239c00f11  Removing intermediate container 1f7a9ba8d63c  Successfully built 54d239c00f11 |

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.

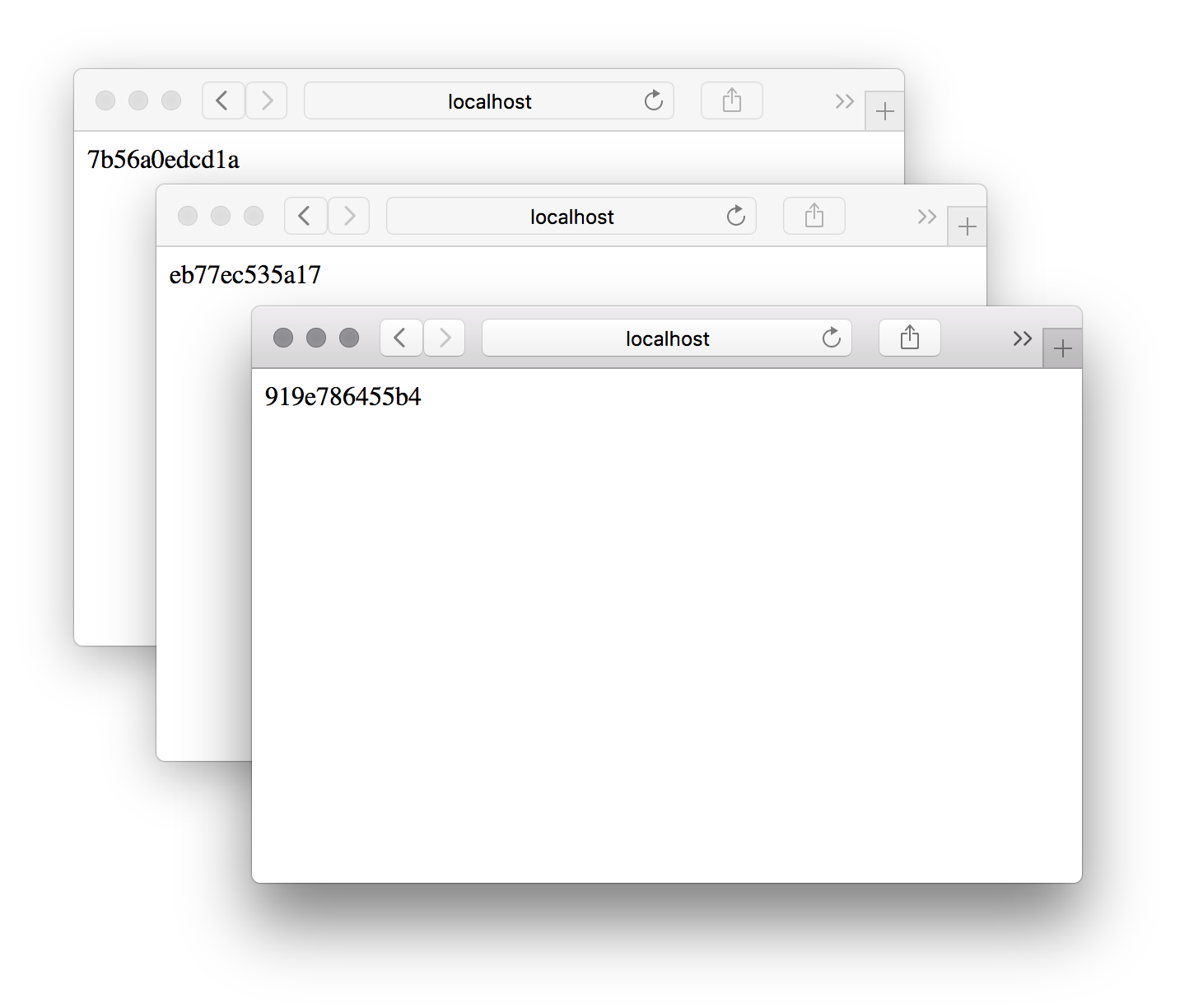
|  |  |
| --- | --- |
| 1 | docker images |

output:

|  |  |
| --- | --- |
| 1  2  3 | REPOSITORY TAG IMAGE ID CREATED SIZE  app latest 54d239c00f11 4 minutes ago 209 MB  ubuntu 16.04 f49eec89601e 2 weeks ago 129 MB |

와..와우! 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 |



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/)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.

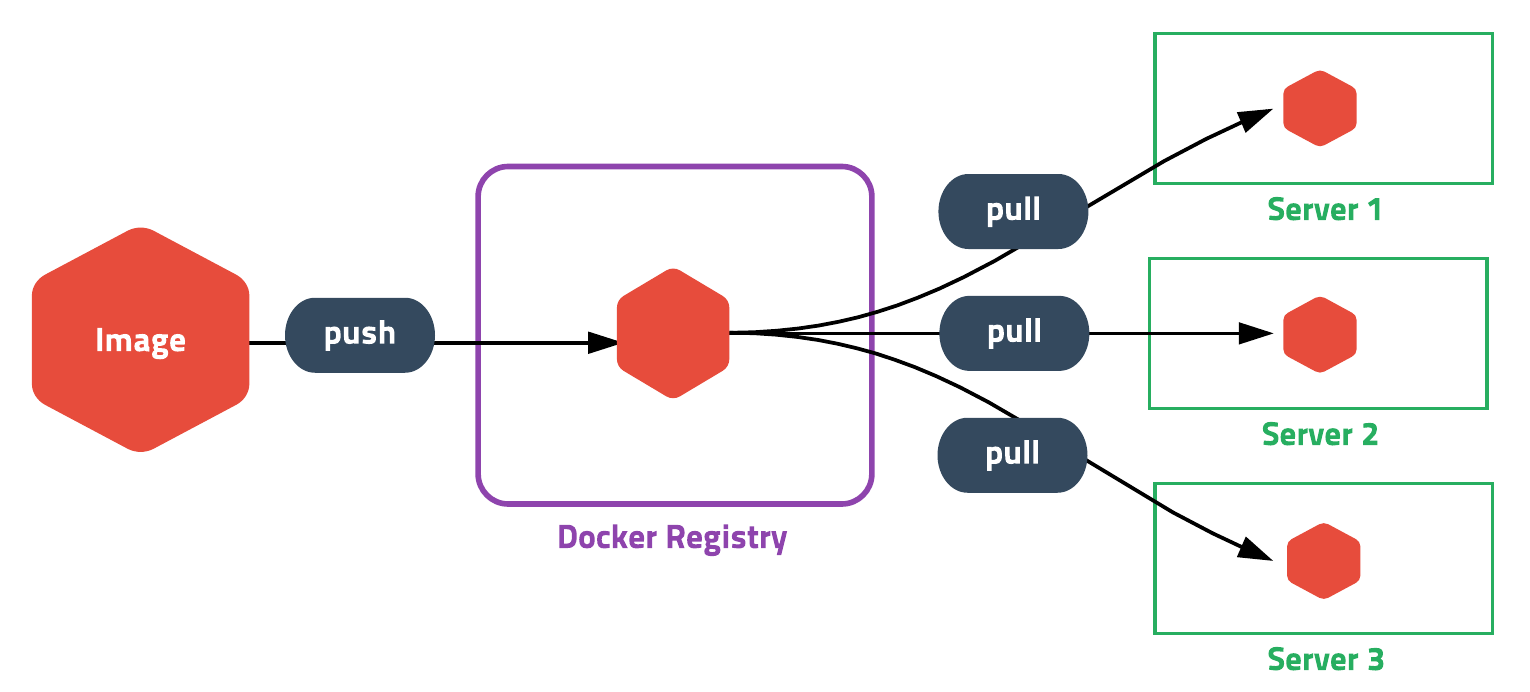
|  |  |
| --- | --- |
|  | FROM ruby:2.3 |
|  | MAINTAINER subicura@subicura.com |
|  | COPY Gemfile\* /usr/src/app/ |
|  | WORKDIR /usr/src/app |
|  | RUN bundle install --no-rdoc --no-ri |
|  | COPY . /usr/src/app |
|  | EXPOSE 4567 |
|  | CMD bundle exec ruby app.rb -o 0.0.0.0 |

[**view raw**](https://gist.github.com/subicura/877bced4293c68769302141f51e4b766/raw/f91e9b8ae3ddb699684464ae8cf3c78e1e7df471/Dockerfile)[**Dockerfile**](https://gist.github.com/subicura/877bced4293c68769302141f51e4b766#file-dockerfile) hosted with ❤ by [**GitHub**](https://github.com/)

드디어 아까보다 훨씬 나은 이미지가 완성됐습니다.

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

## 이미지 저장소



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.

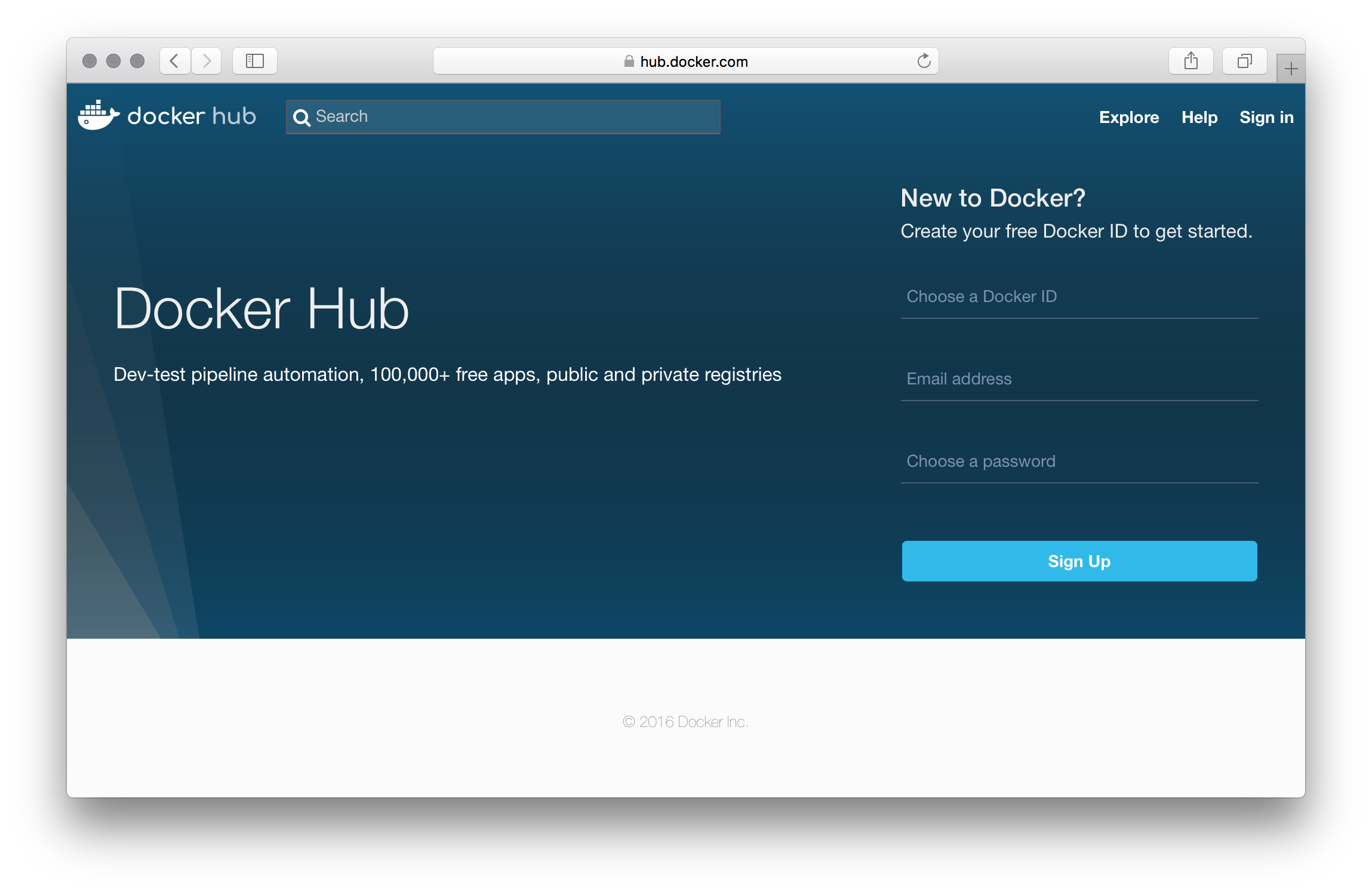
### Docker Hub

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**



[도커 허브](https://hub.docker.com/)사이트에 접속하면 쉽게 회원가입을 할 수 있습니다.

**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 |

output:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85 | registry  └── docker  └── registry  └── v2  ├── blobs  │   └── sha256  │   ├── 05  │   │   └── 05e91de6d378244d3b4dcfbb978548e47b2c22d9918638444c0f9e640d2bf819  │   │   └── data  │   ├── 1b  │   │   └── 1bacd3c8ccb1f15609a10bd4a403831d0ec0b354438ddbf644c95c5d54f8eb13  │   │   └── data  │   ├── 3e  │   │   └── 3e309f550af7bb4e3177e94e498938a24c20cf7404d90da4b24bd586d6376ab2  │   │   └── data  │   ├── 5b  │   │   └── 5b8ee2811f46dea888057c8a2eb1fcd43838bb2545588f7e108d462d0836e217  │   │   └── data  │   ├── 76  │   │   └── 761e0e61cc35ced6b08ce33b1db84a3e92a4843f908476d4cfc52f99b2b7e78a  │   │   └── data  │   ├── 86  │   │   └── 869d5d3f92f8bbdcaa1023ce719df7e337d34b2a13954cde7954c6b1327297e9  │   │   └── data  │   ├── 97  │   │   └── 976dd3af6adc358c852b97651cef1ceaade821a25f3bf566de3a8ea8b7d6bc6a  │   │   └── data  │   ├── a2  │   │   └── a2037218c9af7a6913308a60127a320272fd6d58405692a5be48ce976ace6c6d  │   │   └── data  │   ├── e2  │   │   └── e2d7e96004fdd10e671372a8b5e861bae78bcf878e062d13f6c470311936d500  │   │   └── data  │   ├── f4  │   │   └── f40e1388890a7ca2b5180e2fae69aa982926307cdfa98c8c6a2dde01c0464587  │   │   └── data  │   ├── f8  │   │   └── f8a4e25b40ce0ecc925cfb0b2ec7eec0a06948eda173e943c228cd51f78b40c5  │   │   └── data  │   └── fd  │   └── fdeb3bd5d1b49c094875dcacccb743b0bc913c7de80ba783f2f1bbda6b1a3b55  │   └── data  └── repositories  └── subicura  └── app  ├── \_layers  │   └── sha256  │   ├── 1bacd3c8ccb1f15609a10bd4a403831d0ec0b354438ddbf644c95c5d54f8eb13  │   │   └── link  │   ├── 3e309f550af7bb4e3177e94e498938a24c20cf7404d90da4b24bd586d6376ab2  │   │   └── link  │   ├── 5b8ee2811f46dea888057c8a2eb1fcd43838bb2545588f7e108d462d0836e217  │   │   └── link  │   ├── 761e0e61cc35ced6b08ce33b1db84a3e92a4843f908476d4cfc52f99b2b7e78a  │   │   └── link  │   ├── 869d5d3f92f8bbdcaa1023ce719df7e337d34b2a13954cde7954c6b1327297e9  │   │   └── link  │   ├── 976dd3af6adc358c852b97651cef1ceaade821a25f3bf566de3a8ea8b7d6bc6a  │   │   └── link  │   ├── a2037218c9af7a6913308a60127a320272fd6d58405692a5be48ce976ace6c6d  │   │   └── link  │   ├── e2d7e96004fdd10e671372a8b5e861bae78bcf878e062d13f6c470311936d500  │   │   └── link  │   ├── f40e1388890a7ca2b5180e2fae69aa982926307cdfa98c8c6a2dde01c0464587  │   │   └── link  │   ├── f8a4e25b40ce0ecc925cfb0b2ec7eec0a06948eda173e943c228cd51f78b40c5  │   │   └── link  │   └── fdeb3bd5d1b49c094875dcacccb743b0bc913c7de80ba783f2f1bbda6b1a3b55  │   └── link  ├── \_manifests  │   ├── revisions  │   │   └── sha256  │   │   └── 05e91de6d378244d3b4dcfbb978548e47b2c22d9918638444c0f9e640d2bf819  │   │   └── link  │   └── tags  │   └── latest  │   ├── current  │   │   └── link  │   └── index  │   └── sha256  │   └── 05e91de6d378244d3b4dcfbb978548e47b2c22d9918638444c0f9e640d2bf819  │   └── link  └── \_uploads  56 directories, 26 files |

이미지가 레이어별로 이쁘게 저장된 걸 확인할 수 있습니다. 이렇게 개인 저장소를 만드는 법은 매우 간단합니다. 이제 내부적으로 이미지를 관리하고 여러 서버에 배포할 수 있습니다.

**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/)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.

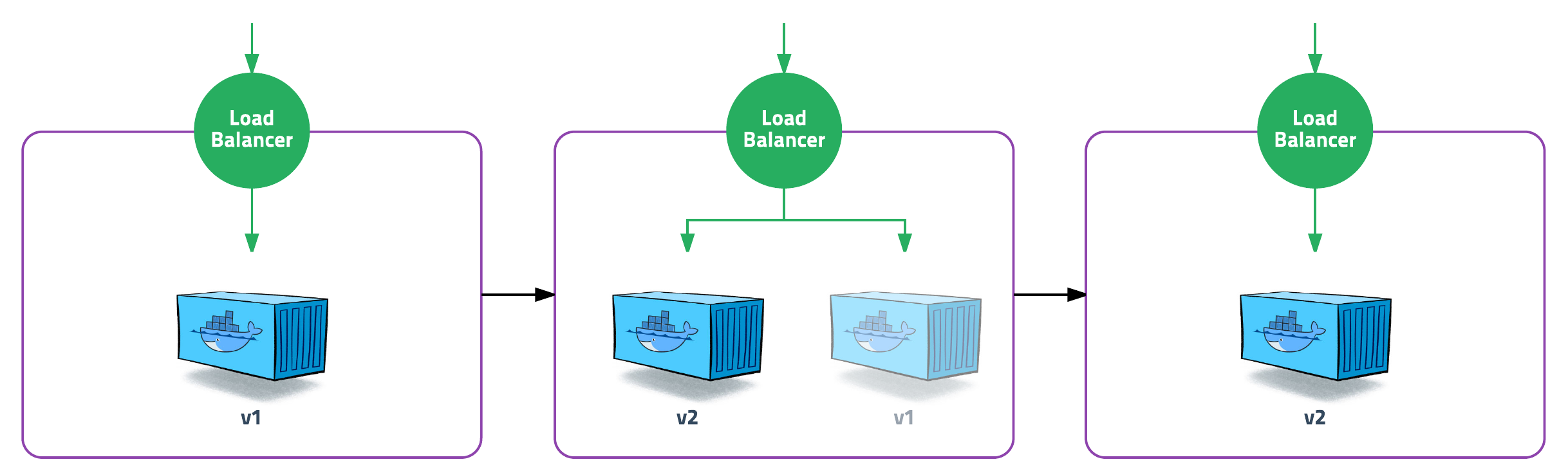
기존에 애플리케이션을 배포하는 방식은 사용하는 언어, 프레임워크, 웹(or WAS)서버, 리눅스 배포판, 개발자의 취향에 따라 각각 다른 방식을 사용했습니다.

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.

### 컨테이너 업데이트



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 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.

## 기타

본 문서는 인터넷 접속이 불가능한 인원을 위해 파일로 구성한 것이고 원래 글에 대한 저작권 문제가 발생 될 수 있으니 외부 공유는 제한합니다.

원글 내용은 <https://subicura.com/2017/02/10/docker-guide-for-beginners-create-image-and-deploy.html> 에서 확인 하시면 됩니다.