

# Docker

# 를 익히다

<Beginner Level>

<b>소개말</b>	<b>4</b>
<b>1. 시작하기 전에</b>	<b>4</b>
1.1 가상화의 역사	4
1.2 가상 머신과 컨테이너란 무엇인가?	5
1.2.1 가상 머신 (Virtual Machines)	5
1.2.2 컨테이너 (Container)	6
1.2.3 VM과 컨테이너	7
1.2.4 왜 컨테이너를 사용하나요?	8
1.3 Docker란 무엇인가?	9
1.3.1 Docker는 누구를 위한 것인가?	10
1.3.2 Docker를 이용해 무엇을 하나요?	10
1.3.3 Docker를 채택한 회사	10
1.3.4 일반적인 Docker 활용 사례	12
1.3.5 Docker 아키텍처	13
1.4 컨테이너가 세계를 평정한 이유	15
1.4.1 클라우드로 애플리케이션 마이그레이션	15
1.4.2 Legacy 애플리케이션의 현대화	17
1.4.3 클라우드 네이티브 앱 구축	18
1.4.4 서버리스 및 기타	18
1.4.5 DevOps를 통한 디지털 혁신	19
1.5 이 책의 목적은?	20
1.5.1 이 책의 사용법	20
1.5.1.1 실습	20
1.5.1.2 References	20
1.6 실습 환경 만들기	20
1.6.1 도커 설치	20
1.6.1.1 Mac OS X에 Docker Desktop 설치	21
1.6.1.2 Windows 10에서 Docker Desktop 설치	21
1.6.1.3 Docker Toolbox 설치	21
1.6.1.4 Docker 설정 확인	21
1.6.2 샘플 코드 다운로드	22
<b>2. Docker 이해하기 및 Hello World 실행</b>	<b>22</b>
2.1 컨테이너에서 Hello World 실행하기	23
2.2 첫번째 컨테이너 실행 (docker run 명령어)	23
2.3 컨테이너 상태 확인하기	26
2.4 Image 검색하기	27
2.5 Image 가져오기	27
2.6 Image 목록 출력하기	28
2.7 컨테이너 생성하기	29
2.8 컨테이너 시작하기	29
2.9 컨테이너 재시작 하기	31

2.10 컨테이너에 접속하기	31
2.11 외부에서 컨테이너내의 명령 실행하기	31
2.12 컨테이너 실행 정지 하기	32
2.13 컨테이너 삭제하기	33
2.14 Image 삭제하기	34
<b>3. Docker Image 및 컨테이너 추가 정보</b>	<b>34</b>
3.1 Apache 웹 서버 실행	35
3.1.1 docker ps시 ports 정보 확인 하기	37
3.1.2 웹 사이트에 접속하기	37
3.1.3 랜덤 포트 매핑	37
3.1.4 특정 포트 매핑	38
<b>4. Docker Hub 알아보기</b>	<b>39</b>
4.1 Docker 이미지 검색	40
4.2 이미지 pull	42
4.3 이미지 목록 보기	43
4.4 컨테이너 시작	43
4.5 이미지 검색	43
<b>5. 나만의 Docker 이미지 만들기</b>	<b>43</b>
5.1 나만의 데이터 관리	43
5.2 이미지 commit	46
5.3 이미지에서 컨테이너 시작	48
5.4 Docker Hub로 이미지 Push 하기	48
<b>6. Docker 전용 Registry</b>	<b>49</b>
6.1 Registry 이미지를 다운 받기	49
6.2 로컬 Registry 실행 하기	50
6.3 로컬 Registry로 Push 하기	51
6.3.1 busybox 및 alpine 리눅스 이미지 가져오기	51
6.3.2 busybox 및 alpine 리눅스를 로컬 Registry에 Push 하기	52
<b>7. Data Volumes</b>	<b>53</b>
7.1 데이터 볼륨 마운트	54
7.2 호스트 Volume을 데이터 Volume으로 마운트	56
7.3 데이터 볼륨 컨테이너	59
<b>8. 컨테이너 연결하기</b>	<b>60</b>
<b>9. Dockerfile 작성하기</b>	<b>63</b>
<b>10. Docker Swarm 알아보기</b>	<b>65</b>
10.1 컨테이너 오케스트레이션이 필요한 이유	66
10.2 Docker 머신 생성하기	66
10.3 Swarm Cluster	68
10.3.1 Worker Node로 Join 하기	69

10.3.2 Manager Node로 Join 하기	69
10.4 Swarm에 Worker Node 추가하기	69
10.5 서비스 만들기	71
10.6 서비스 이용하기	72
10.7 확장 및 축소하기	73
10.8 노드 검사	73
10.9 노드 가용성 속성 변경	74
10.10 서비스 제거하기	76
10.11 롤링 업데이트 적용하기	76
10.12 결론	76
<b>11. Docker 사용 사례</b>	<b>76</b>
11.1 새로운 소프트웨어 사용해보기	77
11.2 데모에 적합	77
11.3 머피의 법칙 피하기	77
11.4 리눅스와 스크립트 학습하기	77
11.5 자원 사용을 효율적으로	77
11.6 마이크로 서비스	77
11.7 클라우드 업체간 포팅	78
<b>12. Docker에 더 가까이 다가서기</b>	<b>78</b>

# 소개말

안녕하세요. "도커(Docker)를 익히다."에 참여 해주셔서 감사합니다.

저는 주길재(Giljae Joo)이고 SK 주식회사 C&C의 Tech. Training그룹 DT Labs에서 일하고 있습니다.

본 강의는 단순하게 시작할 것이고 점차 복잡해질 것입니다. 마지막에서는 Docker 사용 사례에 대해서 언급할 계획입니다.

컨테이너(Container)를 처음 접하는 분이라면 본 강의 통해 즐거운 여행을 하실 수 있습니다. Docker에 대해 이미 경험이 있으신 분이라면 건너뛰시는 것이 좋습니다.

현재 시장에 있는 많은 Docker 서적은 당신이 Linux 전문가라고 가정하고 Docker에 대해서 설명을 하고 있습니다. 본 강의는 당신이 초보자라고 가정하고 작성되었습니다.

강의에 참여해주셔서 다시 한번 감사드립니다.

## 1. 시작하기 전에

Docker는 컨테이너라고 하는 가벼운 단위에서 어플리케이션을 실행하기 위한 플랫폼입니다. 컨테이너는 클라우드의 Serverless 기능에서 기업의 전략적인 계획에 이르기까지 어느 곳에서나 활용되고 있습니다.

Docker는 업계 최고의 운영자 및 개발자에게 핵심 역량이 되고 있습니다. Docker는 사람들이 가장 선호하는 최고의 기술입니다.

Docker는 배우기에 매우 간단한 기술입니다. 본 책은 완전 초보자를 대상으로 작성 되었습니다. 이 책을 통해 당신은 실제 현장에서 컨테이너가 사용되는 방식과 Docker를 배우기 전에 해결해야 할 문제 유형에 대해서 이해 해야 합니다.

### 1.1 가상화의 역사

이전에는 서비스 배포 프로세스가 느리고 고통스러운 작업이었습니다.

1. 개발자가 코드를 작성
2. 운영팀이 Bare-metal 머신에 코드가 작동할 수 있도록 라이브러리 버전, Patch 및 Compiler를 설치
3. 버그나 오류가 있는 경우 프로세스가 다시 시작되고 개발자가 이를 수정한 후 운영팀이 다시 배포

하이퍼바이저의 등장으로 이런 부분들이 개선되었지만, 하이퍼바이저는 동일한 호스트에 여러 개의 가상 머신이 존재하며 실행 중이거나 Shutdown 되어 있을 수 있습니다.

가상 머신은 코드 배포 및 버그 수정 대기 시간을 크게 줄였지만 실제 게임 체인저는 Docker 컨테이너였습니다.

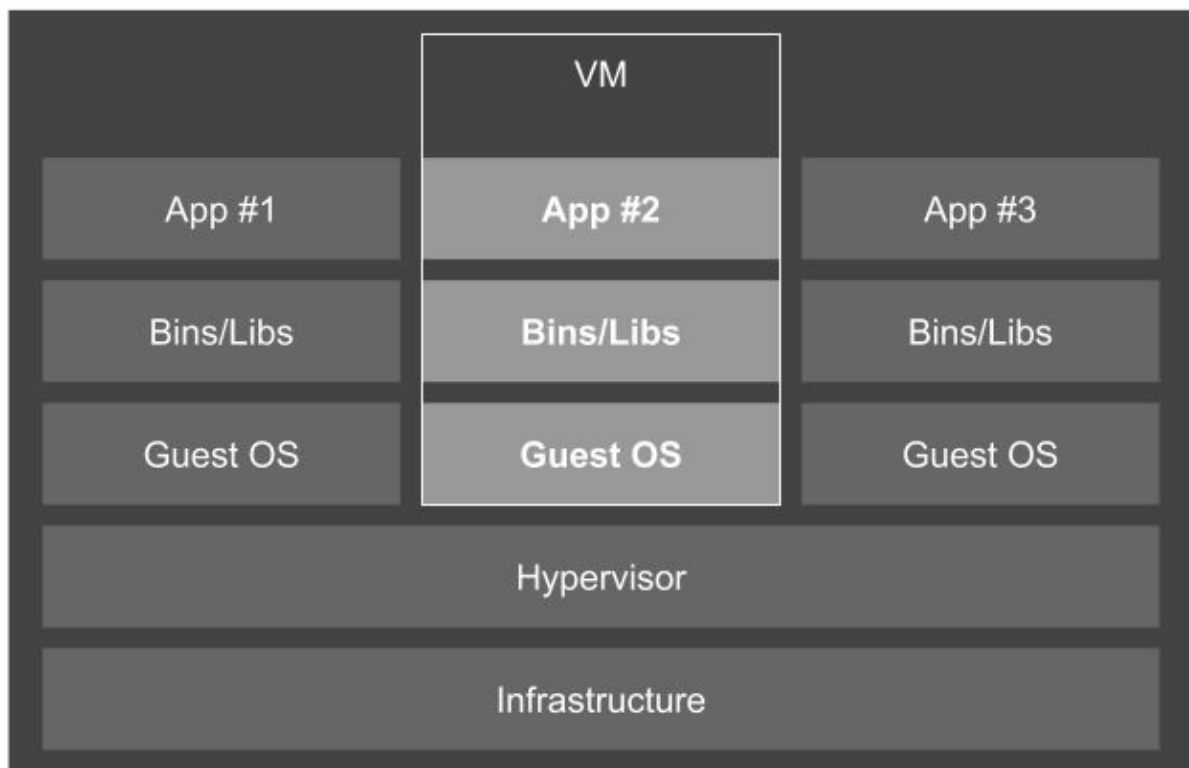
## 1.2 가상 머신과 컨테이너란 무엇인가?

가상 머신과 컨테이너는 서로 비슷한 목적을 가지고 있습니다. 가상 머신은 여러 게스트(Guest) 운영체제가 하이퍼바이저(Hypervisor)를 통해 에뮬레이션 됩니다.

컨테이너는 커널을 공유하여 여러 가상 환경을 에뮬레이션 합니다. 즉, 어플리케이션과 의존성(Dependencies)들을 독립된 단위로 묶고 격리시켜 어떤 환경 에서든 실행 가능하게 하는 것입니다. 이렇게 둘은 비슷한 역할을 수행하지만 둘간의 차이가 존재합니다.

### 1.2.1 가상 머신 (Virtual Machines)

가상 머신은 컴퓨터 에뮬레이터이며 실제 컴퓨터 처럼 프로그램들을 실행 합니다. 가상 머신은 하이퍼바이를 통해 물리적인 하드웨어 위에서 동작합니다.하이퍼바이저는 물리적인 컴퓨터 하드웨어 위에서 동작합니다.



가상 머신(VM)은 컴퓨터 시스템의 에뮬레이션입니다. 하나의 하드웨어에서 VM을 이용하여 많은 개별 컴퓨터로 보이게 실행 할 수 있습니다.

각 VM은 공유한 OS가 필요하며 하드웨어가 가상화 됩니다. 하이퍼 바이저는 하드웨어와 VM 사이에 있으며 서버를 가상화 하는데 필요합니다.

VM이 시장에서 활용되는 이유는 비용을 절감하고 효율성을 높이기 위해서 입니다. 그러나 VM은 많은 시스템 리소스를 차지 할 수 있습니다.

각 VM은 운영 체제의 전체 자원뿐만 아니라 운영 체제가 실행해야 하는 모든 하드웨어의 가상 자원을 실행 합니다. 이것은 많은 RAM과 CPU를 사용하게 됩니다.

물리적인 하드웨어에서 실행하는 것과 비교시 여전히 경제적이지만 일부 어플리케이션의 경우 과도하게 사용될 가능성이 존재합니다.

### 1.2.2 컨테이너 (Container)

컨테이너를 사용하면 VM과 같이 컴퓨터를 가상화하는 대신 OS만 가상화 됩니다.

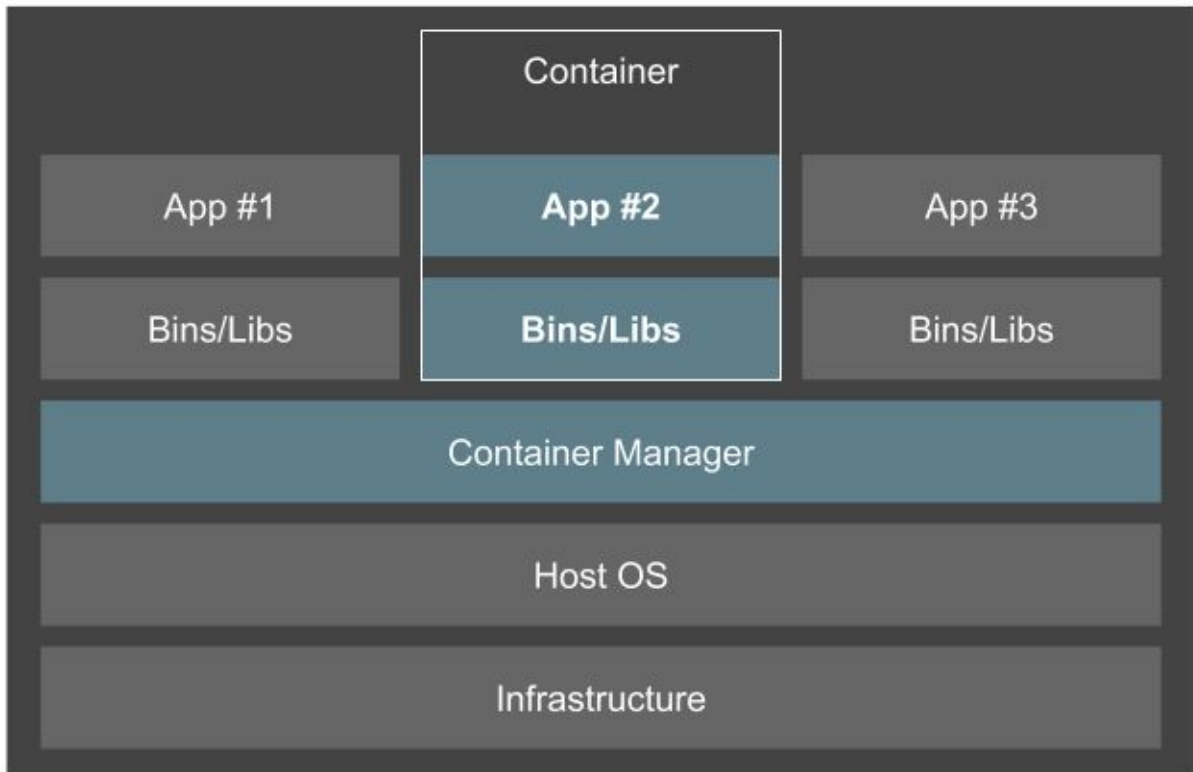
컨테이너는 물리적 서버와 호스트 OS(일반적으로 Linux 또는 Windows) 위에 존재 합니다. 각 컨테이너는 호스트 OS 커널, 바이너리 그리고 라이브러리도 읽기 전용으로 공유 합니다.

라이브러리와 같은 OS 리소스를 공유하면 운영 체제 코드를 재현해야 할 필요성이 크게 줄어들기 때문에 단일 OS위에 여러 어플리케이션을 실행 할 수 있습니다.

따라서 컨테이너는 매우 가볍고 크기는 메가 바이트에 불과하며 구동하는데 몇 초 밖에 걸리지 않습니다. VM의 경우는 실행하는데 몇 분이 걸리며 크기도 컨테이너보다 수십 배 더 큼니다.

VM과 달리 컨테이너는 운영 체제, 프로그램 및 라이브러리, 특정 프로그램을 실행하기 위한 시스템 리소스로 구성되어 있습니다.

이것이 실제로 의미하는 것은 VM을 사용할 때 보다 컨테이너가 있는 단일 서버에 어플리케이션을 2배에서 3배까지 더 넣을 수 있다는 점입니다. 또한 컨테이너를 사용하여 개발, 테스트 및 배포를 위한 일관된 운영 환경을 만들 수 있습니다.



컨테이너는 두가지의 종류가 있습니다.

- Linux 컨테이너(LXC) - Linux 컨테이너 기술은 일반적으로 LXC로 알려져 있습니다. LXC는 단일 호스트에서 여러 개의 격리된 Linux 시스템을 실행하기 위한 Linux 운영 체제 레벨의 가상화 방법입니다.
- Docker - Docker는 LXC 컨테이너를 빌드하는 프로젝트로 시작하여 컨테이너를 보다 이식 가능하고 유연하게 사용할 수 있도록 LXC에 몇가지 변경 사항을 도입하였고 향후 자체 컨테이너 런타임 환경으로 변형되었습니다. Docker는 컨테이너를 효율적으로 생성, 배포 및 실행 할 수 있는 Linux 유틸리티입니다.

### 1.2.3 VM과 컨테이너

컨테이너와 VM 모두 장/단점을 지니고 있습니다.

- VM은 서버에서 여러 어플리케이션을 실행하거나 관리해야 할 다양한 운영체제가 필요한 경우에 적합합니다.
- 최소한의 서버에서 실행되는 어플리케이션 수를 최대화 할 경우에는 컨테이너가 적합합니다.

VM	컨테이너
헤비급 선수	라이트급 선수
제한된 성능	기본 성능



각 VM은 자체 OS에서 실행	모든 컨테이너는 호스트 OS를 공유
하드웨어 수준의 가상화	OS 가상화
시작 시간 (분)	시작 시간 (초)
필요한 메모리 할당	적은 메모리 필요
완전히 격리되어 안전함	프로세스 수준의 격리

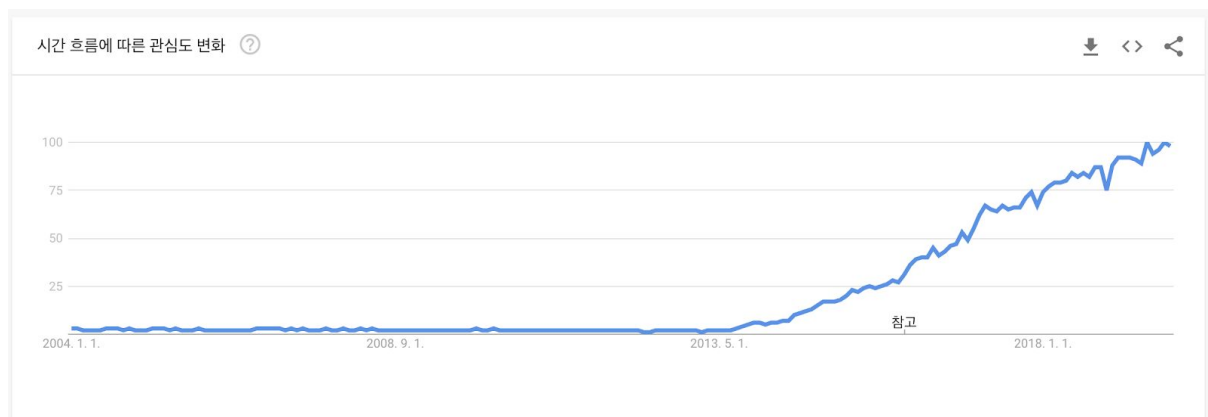
〈VM과 컨테이너의 차이점〉

현재 가상화 기술 상태에서는 VM의 유연성과 컨테이너의 최소 리소스 요구 사항이 함께 작용하여 최대 기능을 제공합니다.

#### 1.2.4 왜 컨테이너를 사용하나요?

컨테이너는 실제 실행 환경에서 어플리케이션을 추상화 할 수 있는 논리적인 패키징 메커니즘을 제공합니다. 이를 통해 개인 PC, 퍼블릭 클라우드, 데이터 센터에 관계 없이 컨테이너 기반 어플리케이션을 쉽고 일관되게 배포 할 수 있습니다. 개발자는 어디에서나 실행 할 수 있는 예측 가능한 환경을 만들 수 있습니다.

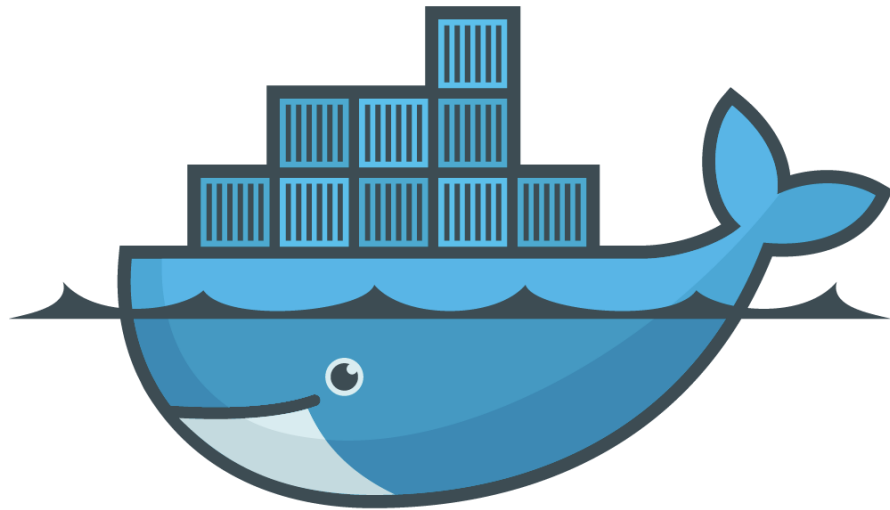
운영 관점에서 볼 때 리소스를 보다 세밀하게 제어하여 인프라의 효율성을 향상 시킬 수 있고 이는 컴퓨팅 리소스의 활용도를 높일 수 있습니다.



〈Docker에 대한 Google 트렌드〉

이러한 이점으로 인해 컨테이너(e.g. Docker)가 널리 채택되었습니다. Google, Facebook, Netflix 및 Salesforce와 같은 회사는 컨테이너를 활용하여 대규모 엔지니어링 팀의 생산성을 높이고 컴퓨팅 리소스의 활용도를 향상 시킵니다.

## 1.3 Docker란 무엇인가?



# docker

위의 그림은 Docker의 정식 로고이며 고래가 컨테이너를 싣고 항해하는 모습입니다. 이 글을 읽는 당신이 IT 분야에서 일을 하는 사람이라면 Docker에 대해서 들어보았을 것입니다. 어플리케이션을 포장해서 옮기고 컨테이너(Container)안에서 실행할 수 있게 해주는 것이 Docker입니다.

위키피디아에서는 Docker를 다음과 같이 정의 합니다.

*“도커 컨테이너는 소프트웨어의 실행에 필요한 모든 것을 포함하는 완전한 파일 시스템 안에 감싼다. 여기에는 코드, 런타임, 시스템 도구, 시스템 라이브러리 등 서버에 설치되는 무엇이든 아우른다. 이는 실행 중인 환경에 관계 없이 언제나 동일하게 실행 될 것을 보증한다. - 위키피디아”*

즉, Docker는 컨테이너를 사용하여 어플리케이션을 보다 쉽게 생성, 배포 및 실행 할 수 있도록 설계된 도구 입니다.

컨테이너를 사용하면 개발자는 라이브러리 및 기타 종속성등을 하나의 패키지로 만들어 제공할 수 있습니다. 이렇게 하면 컨테이너 덕분에 개발자는 코드 작성 및 테스트에 사용 된 환경에서 설정에 관계 없이 다른 시스템에서 어플리케이션을 실행 할 수 있습니다.

Docker는 오픈 소스입니다. 즉, 누구나 사용할 수 있고 추가 기능이 필요한 경우에는 누구나 Docker에 기여할 수 있습니다.

### 1.3.1 Docker는 누구를 위한 것인가?

Docker는 개발자와 시스템 관리자 모두에게 도움이 되도록 설계된 도구이고 DevOps (개발 + 운영) 도구 체인의 일부입니다.

개발자는 운영 환경에 대해 걱정하지 않고 코드 작성에 집중할 수 있습니다. 또한 Docker 컨테이너에서 이미 실행되도록 설계된 수천 개의 프로그램 중 하나를 애플리케이션의 일부로 사용할 수 있습니다.

Docker는 시스템 관리자에게 유연성을 제공하고 설치 공간이 작고 오버 헤드도 적기 때문에 필요한 시스템 수를 줄일 수 있습니다.

### 1.3.2 Docker를 이용해 무엇을 하나요?

아래의 시나리오를 생각해봅시다.

1. 개발자가 로컬에 코드를 작성하고 Docker 컨테이너를 사용하여 동료와 작업을 공유합니다.
2. Docker를 사용하여 어플리케이션을 테스트 환경으로 push하고 자동 혹은 수동 테스트를 진행합니다.
3. 개발자가 버그를 발견하면 개발 환경에서 버그를 수정하고 검증을 위해 테스트 환경에 재배포 할 수 있습니다.
4. 테스트가 완료되면 업데이트 된 이미지를 운영 환경으로 push 해서 문제를 해결합니다.

Docker 컨테이너는 개발자의 로컬 PC, 데이터 센터의 물리/가상 시스템, 클라우드 공급자의 환경등 여러 곳에서 실행될 수 있습니다.

Docker는 가볍고 빠릅니다. 하이퍼바이저 기반 VM대신 실행 가능하고 비용 효율적인 대안을 제공하기에 더 많은 컴퓨팅 리소스를 사용하여 비즈니스 목표를 달성 할 수 있습니다. Docker는 고밀도 환경과 적은 리소스로 더 많은 작업을 수행해야 하는 환경에 적합합니다.

### 1.3.3 Docker를 채택한 회사

Enterprise급 조직에서는 Docker를 여러 방식으로 사용하고 있습니다.

GE



GE의 어플리케이션 개발 프로세스는 수동이며 휴먼 에러등을 감안할 때 개발에서 프로덕션으로 이동하는데 평균 6주의 시간이 소모되었습니다.

이 문제는 Docker를 채택하여 한번만 빌드하고 모든 환경에서 실행할 수 있는 기능을 제공함으로써 해결되었습니다.

Docker 도입전에는 VM에서 하나의 어플리케이션을 실행 할 수 있는 VMware를 사용했지만 Docker를 사용하게 된 후 컨테이너당 평균 14개의 어플리케이션을 실행 할 수 있었습니다.

#### PayPal



PayPal은 Docker의 상용 솔루션을 사용합니다. 이를 통해 개발자 및 인프라팀은 생산성 및 민첩성 그리고 비용 효율성에서 효과를 가져왔습니다.

#### BBC News



BBS News는 매일 80000개가 넘는 온라인 뉴스를 제공합니다. 서로 다른 환경에서 26000개 이상의 작업을 순차적으로 실행하기에 작업 당 약 60분의 대기 시간이 존재했습니다.

Docker는 작업이 병렬로 실행되도록 함으로써 BBC의 시간 지연 문제를 해결했습니다. 속도와 볼륨의 근본적인 문제가 해결되었기에 개발자에게 유연성을 제공하게 되었고 지속적인 통합으로 보다 효율적이고 빠른 개발 환경 제공이 가능해졌습니다.

### 1.3.4 일반적인 Docker 활용 사례

#### 구성 단순화

Docker는 하나의 Configuration으로 모든 플랫폼에서 실행할 수 있습니다. Configuration 파일을 코드에 넣고 환경 변수를 전달하여 다른 환경에 맞출 수 있습니다. 따라서 하나의 Docker 이미지를 다른 환경에서 사용할 수 있습니다.

#### 코드 관리

Docker는 일관된 환경을 제공하여 개발 및 코딩을 훨씬 편안하게 만들어줍니다. Docker 이미지는 변경이 불가하기에 개발환경에서 운영 환경까지 어플리케이션 환경이 변경되지 않는 이점이 존재합니다.

#### 개발 생산성 향상

개발 환경을 운영 환경에 최대한 가깝게 복제할 수 있습니다. Docker를 사용하면 코드가 운영 환경의 컨테이너에서 실행될 수 있으며 VM과 달리 Docker는 오버 헤드 메모리 용량이 적기에 여러 서비스를 실행하는데 도움이 됩니다.

또한 Docker의 Shared Volume을 사용하여 호스트에서 컨테이너의 어플리케이션 코드를 사용할 수 있도록 할 수 있습니다. 이를 통해 개발자는 자신의 플랫폼 및 편집기에서 소스 코드를 편집할 수 있으며 이는 Docker내에서 실행중인 환경에 반영됩니다.

#### 어플리케이션 격리

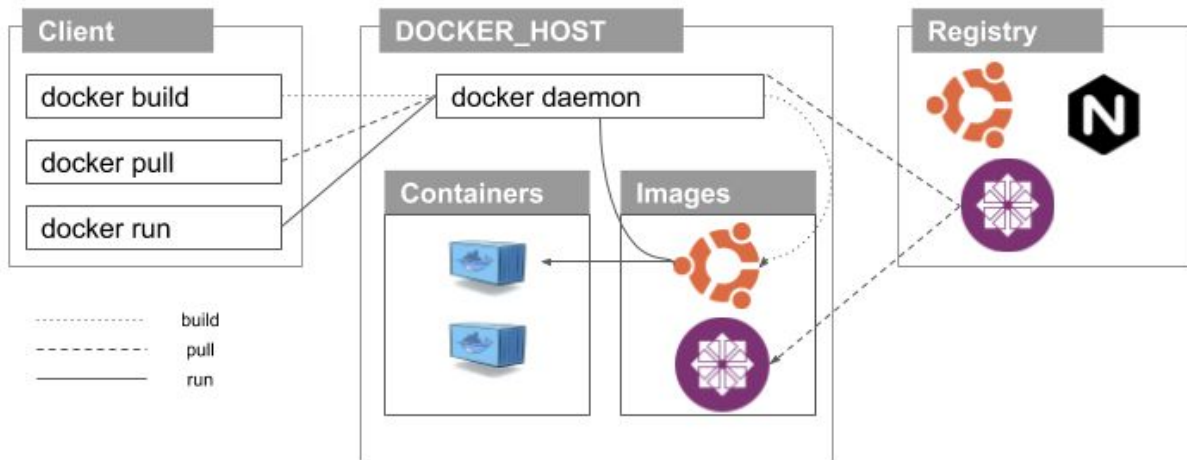
Web Server(e.g. Apache, Nginx)와 연결된 API 서버를 격리할 필요가 있는 경우가 있습니다. 이 경우 다른 컨테이너에서 API를 서버를 실행 할 수 있습니다.

## 빠른 배포

컨테이너가 OS를 부팅하지 않고 어플리케이션을 실행하기 때문에 Docker 컨테이너를 매우 빠르게 만들 수 있습니다.

### 1.3.5 Docker 아키텍처

아래의 그림은 Docker 아키텍처를 간단히 표현한 것입니다.



## 도커 엔진

Docker 시스템의 핵심입니다. Docker Engine은 클라이언트-서버 아키텍처를 따르는 어플리케이션이고 호스트 시스템에 설치됩니다. Docker Engine에는 세 가지 구성 요소가 존재합니다.

- 서버: dockerd라는 docker 데몬입니다. Docker 이미지를 만들고 관리 할 수 있습니다.
- REST API: Docker 데몬에게 무엇을 할지 지시하는 용도로 사용됩니다.
- CLI (Command Line Interface): Docker 명령을 입력하는데 사용되는 클라이언트입니다.

## Docker 클라이언트

Docker 사용자는 클라이언트를 통해 Docker와 상호 작용할 수 있습니다. docker 명령이 실행될 때 클라이언트는 이를 dockerd 데몬으로 보내어 실행합니다.

Docker API는 Docker 명령에서 사용됩니다. Docker 클라이언트는 둘 이상의 데몬과 통신 할 수 있습니다.

## Docker 레지스트리

Docker 이미지가 저장된 위치입니다. docker pull 또는 docker run 명령을 실행하면 필요한 Docker 이미지가 구성된 레지스트리에서 가져옵니다.

docker push 명령을 실행하면 docker 이미지가 구성된 레지스트리에 저장됩니다.

## Docker 객체

Docker로 작업할 때는 이미지, 컨테이너, 볼륨, 네트워크를 사용합니다. 이것들을 모두 Docker 객체라고 표현합니다.

## 이미지

Docker 이미지는 Docker 컨테이너를 만드는 설명이 포함된 읽기 전용 템플릿입니다. Docker 이미지는 Docker Hub에서 가져와서 사용하거나 기본 이미지에 추가 설명을 추가하고 새롭게 수정된 Docker 이미지를 만들 수 있습니다.

dockerfile을 사용하여 고유한 Docker 이미지를 만들 수도 있습니다. 컨테이너를 실행하기 위한 모든 지시 사항을 dockerfile에 작성하면 됩니다.

## 컨테이너

Docker 이미지를 실행하면 Docker 컨테이너가 생성됩니다. 모든 어플리케이션 및 환경은 해당 컨테이너내에서 실행됩니다. Docker API 또는 CLI를 사용하여 Docker 컨테이너를 시작, 중지, 삭제할 수 있습니다.

## 볼륨

Docker가 생성하고 Docker 컨테이너가 사용하는 영속적인 데이터는 볼륨에 저장됩니다. 이것은 docker CLI 또는 API를 통해 관리됩니다. 컨테이너에 쓰기 가능한 데이터를 유지해야 한다면 항상 볼륨을 사용하는 것이 좋습니다.

## 네트워크

Docker 네트워킹은 모든 격리된 컨테이너가 통신하는 통로입니다. Docker에는 5개의 네트워크 드라이버가 있습니다.

1. Bridge: 컨테이너의 기본 네트워크 드라이버입니다. 어플리케이션이 독립형 컨테이너 (e.g. 동일한 Docker 호스트와 통신하는 컨테이너)에서 실행될 때에 이 네트워크를 사용합니다.
2. Host: Docker 컨테이너와 Docker 호스트간의 네트워크 격리를 제거합니다. 호스트와 컨테이너 사이에 네트워크 격리가 필요하지 않을 경우에 사용됩니다.
3. Overlay: 컨테이너가 다른 Docker 호스트에서 실행 중이거나 Swarm 서비스가 여러 어플리케이션에 의해 형성될 경우에 사용됩니다.
4. None: 모든 네트워킹을 비활성화 합니다.
5. macvlan: 물리적 주소처럼 보이도록 하기위해 mac 주소를 컨테이너에 할당합니다. 네트워크 트래픽은 컨테이너간 mac 주소를 통해 라우팅 됩니다. 이 네트워크는 VM 설정을 마이그레이션 하는 동안 컨테이너를 물리적 장치처럼 보이게 하려는 경우에 사용됩니다.

## 1.4 컨테이너가 세계를 평정한 이유

구글 및 Salesforce와 같은 기업은 거의 10년동안 컨테이너를 활용해 왔습니다. 개발자 또는 관리자인 경우 컨테이너가 기술적인 관점에서 작업을 보다 쉽게 만드는데 도움이 된다고 알고 있습니다. 하지만 컨테이너는 비즈니스 관점에서도 유용합니다.

- **하드웨어 사용을 최적화 합니다.** - 컨테이너는 이미 소유한 서버를 최대한 활용할 수 있도록 도와줍니다. 새 하드웨어를 구입할 필요가 없어 비용이 절약됩니다.
- **최고의 IT 인재 유지** - 최고의 IT 인재를 유지하거나 유치하려면 그들이 관심있는 기술을 사용해야 합니다. 많은 숙련된 개발자와 관리자의 기술에는 컨테이너가 포함됩니다.
- **컨테이너는 오픈 소스** - VMware와 같은 기술과는 달리 컨테이너는 오픈 소스입니다. 이는 라이선스 비용을 줄이는데 도움이 됩니다. 또한 벤더사에 Lock-In되는 문제도 해결합니다.
- **학습 곡선 관리** - 컨테이너로 마이그레이션 하는데에는 학습이 필요합니다. 기존 어플리케이션을 컨테이너화하고 새로운 유형의 환경에서 작업하는 법을 배워야 합니다. 이는 시간이 걸립니다. 그러나 아주 오래 걸리진 않습니다. Linux 및 VM 작업 경험이 있는 대부분의 사람들은 컨테이너 패러다임을 쉽게 이해할 수 있습니다.
- **더 빠른 배포** - 컨테이너는 지속적인 Deploy 파이프 라인을 위한 이상적인 환경입니다. 따라서 어플리케이션 변경을 매우 빠르게 추진하려는 조직에 유리합니다. 이는 시장에서 경쟁 우위를 유지하는데 중요합니다.
- **배포 유연성 제공** - 컨테이너는 특정 프레임워크 또는 서버를 사용할 필요가 없습니다. 원하는 언어로 작성하고 모든 Linux 버전에서 실행할 수 있습니다. 현재 Windows에서도 Docker를 지원하지만 Production 환경에서는 사용할 수 없습니다. 이러한 유연성은 특정 프레임워크에 얽매이지 않는 비즈니스에 도움이 됩니다.
- **일관성** - 컨테이너는 테스트, 준비 및 배포 환경을 동일하게 만듭니다. 이는 안정성을 보장하고 버그가 있는 어플리케이션을 운영 환경에 Deploy할 가능성을 최소화 합니다.
- **자원 재활용** - 컨테이너를 사용하기 위해 새로운 서버를 구입하거나 새로운 클라우드 호스트로 마이그레이션 할 필요가 없습니다. 이미 소유하거나 임대한 클라우드/인프라에서 Docker 환경을 설정 할 수 있습니다.

### 1.4.1 클라우드로 애플리케이션 마이그레이션

많은 조직에서 애플리케이션을 클라우드로 옮기는 것은 중요합니다. 이는 매력적인 옵션입니다. Amazon, Microsoft, Google등의 클라우드 서비스를 이용하게 되면 서버, 디스크, 네트워크 및 전원 등에 대해서 걱정할 필요가 없습니다.

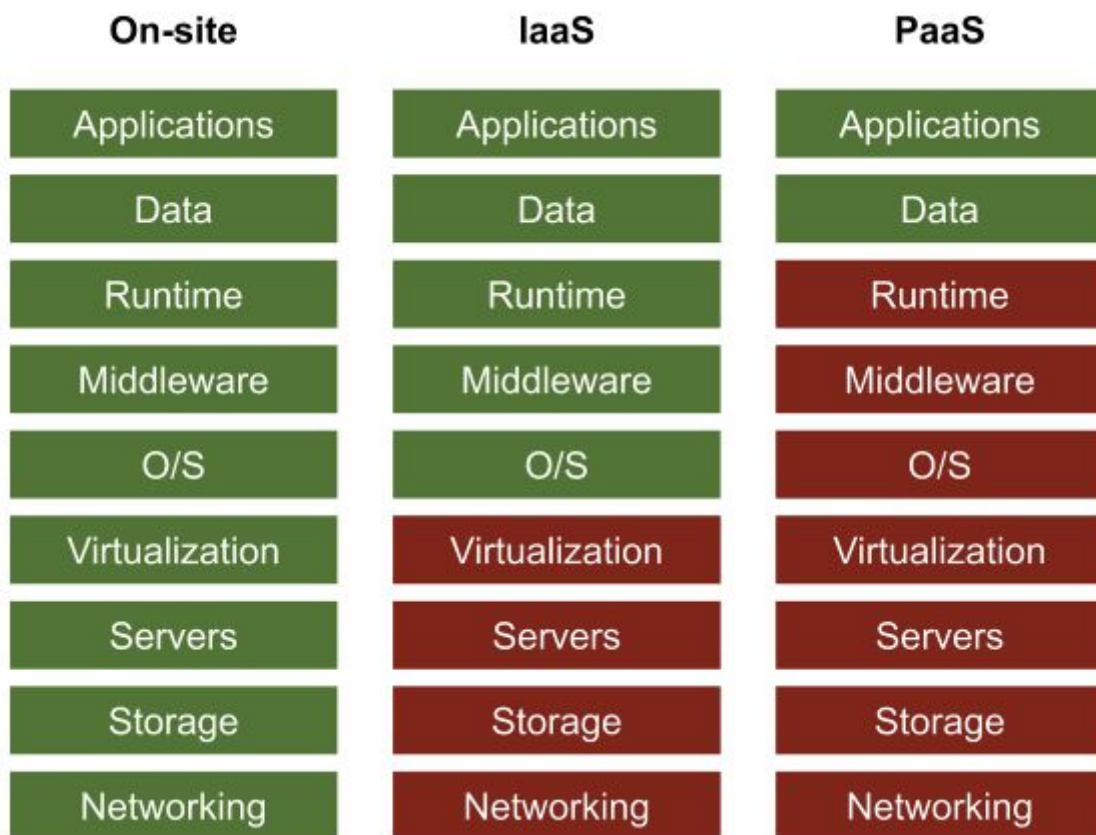
사실상 무한한 확장이 가능하고 전 세계에 퍼져있는 데이터 센터에서 애플리케이션을 호스팅 할 수 있습니다. 몇 분안에 새로운 환경에 배포하고 사용한 리소스에 대해서만 요금이 청구됩니다.



그동안에는 어플리케이션을 클라우드로 마이그레이션 하기 위한 두 가지 옵션인 IaaS(Infrastructure as a Service)와 PaaS(Platform as a Service)가 있었습니다. 하지만 지금에서는 어느 옵션도 훌륭하지 않습니다.

PaaS를 선택하고 프로젝트를 수행하여 모든 어플리케이션을 클라우드로 마이그레이션 할 수 있지만, 이는 어려운 프로젝트가 될 것이며 단일 클라우드로 고정되어 있습니다. 이것에 대한 대안은 어플리케이션의 각 구성 요소에 대해 VM을 이용하는 IaaS입니다. 하지만 이는 이식성이 가능하지만 운영 비용이 훨씬 높습니다.

아래의 그림은 어플리케이션을 클라우드로 마이그레이션시 IaaS 및 PaaS에 대해서 보여줍니다.



IaaS 옵션을 사용하여 비용이 많이드는 형태의 비효율적인 VM을 많이 실행하거나 PaaS를 사용하여 운영 비용을 낮추고 마이그레이션에 더 많은 시간을 소비하는 방법이 있습니다.

Docker는 세번째 옵션을 제공합니다. 어플리케이션의 각 부분을 컨테이너로 마이그레이션 하고 Kubernetes 서비스를 사용하거나 혹은 Docker 클러스터에서 전체 어플리케이션 컨테이너를 실행할 수 있습니다. 컨테이너에서 어플리케이션을 패키징하고 실행하는 방법에 대해서는 차후에 기술합니다.

클라우드로 마이그레이션하기전에 동일한 어플리케이션을 Docker로 마이그레이션을 하면 IaaS의 이식성에 대한 이점과 PaaS의 비용 절감 이점을 얻을 수 있습니다.

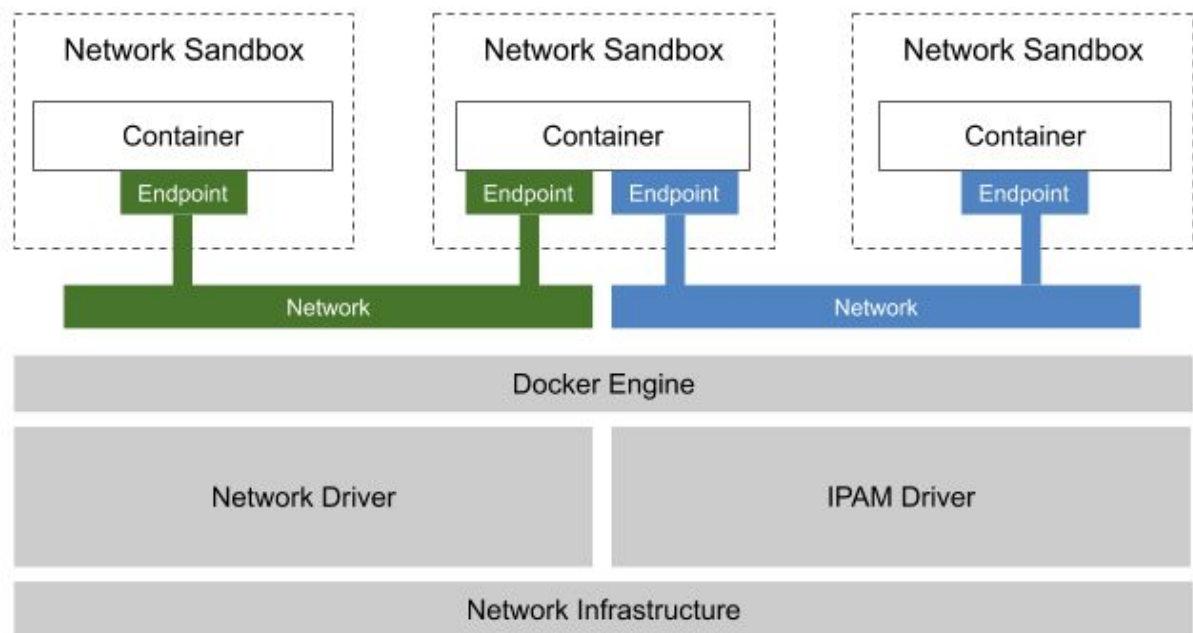
컨테이너로 마이그레이션 하는데 약간의 투자가 필요하지만, 코드를 변경할 필요가 없으며 노트북에서 서버 혹은 클라우드에 이르기까지 동일한 기술 스택을 모든 환경에서 사용하기에 동일한 방식으로 실행됩니다.

### 1.4.2 Legacy 애플리케이션의 현대화

컨테이너에서 거의 모든 애플리케이션을 실행할 수 있지만, 기존 애플리케이션이 Monolithic 형태라면 큰 가치를 얻을 수 없습니다. Monolithic은 컨테이너에서도 잘 작동하지만 민첩성을 제한하게 됩니다. 컨테이너를 사용하여 30초내에 새로운 기능을 올릴 수 있지만, 2백만줄의 코드로 작성된 Monolithic에서는 아마도 약 2주의 테스트 주기를 거쳐야 할 것입니다.

애플리케이션을 Docker로 옮기는 것은 코드를 완전히 다시 작성하지 않고도 새로운 패턴을 채택하여 아키텍처를 현대화하는 단계를 고려해야 합니다. 이 방법은 생각보다 간단합니다. 본 책에서 배울 Dockerfile 및 Docker Compose를 사용하여 애플리케이션을 단일 컨테이너로 마이그레이션 하는 것부터 시작하면 됩니다.

컨테이너는 자체 가상 네트워크에서 실행되기에 외부 환경에 노출되지 않고 서로 통신할 수 있습니다.



이는 마이크로 서비스 아키텍처의 이점을 제공 합니다. 주요 기능을 독립적으로 관리 할 수 있는 작고 격리된 단위입니다. 기능을 확장하거나 축소 할 수 있으며 요구 사항에 맞게 다양한 기술을 사용할 수 있습니다.

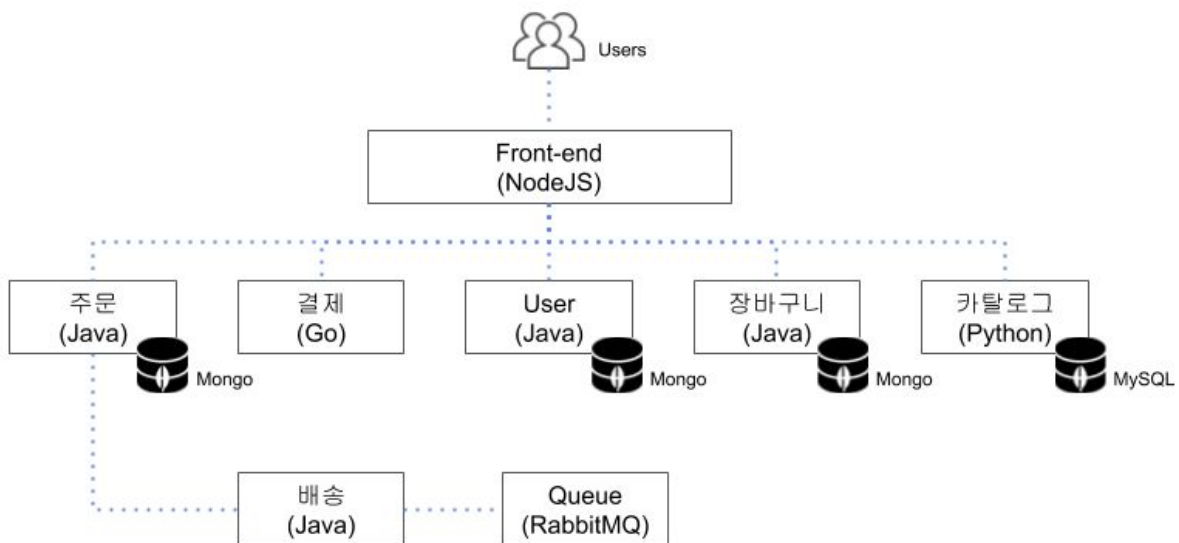
Docker를 사용하면 이전 애플리케이션 아키텍처를 현대화하는 것이 쉽습니다. 이는 앞으로 진행할 Hands-on을 통해 경험해 보실 수 있습니다.

### 1.4.3 클라우드 네이티브 앱 구축

Docker를 사용하면 기존 앱이 마이크로 서비스 형태이든 모놀리식이든 상관없이 제공할 수 있습니다. 또한 클라우드 기본 원칙을 기반으로 하는 새로운 프로젝트일 경우 Docker를 통해 좀 더 쉽게 사용될 수 있습니다.

클라우드 네이티브 컴퓨팅 재단 (CNCF)는 마이크로 서비스와 같은 어플리케이션을 배포하는데 필요한 오픈 소스 소프트웨어 스택을 정의하고 있습니다.

아래의 그림은 새로운 마이크로 서비스 어플리케이션의 일반적인 아키텍처를 보여줍니다.



각 구성 요소는 자체 데이터를 소유하고 API를 통해 노출합니다. Front-end는 모든 API 서비스를 사용하는 웹 어플리케이션입니다. 프로세스를 구성하는 어플리케이션들은 다양한 프로그래밍 언어와 다른 데이터베이스 기술을 사용합니다. 그러나 모든 구성 요소에는 Dockerfile이 포함되어 있고 전체 어플리케이션은 Docker Compose 파일에 정의 되어 있습니다.

본 책에서는 Docker를 사용하여 코드를 컴파일 하는 방법도 배우게 됩니다. 즉, 위와 같은 앱을 빌드하고 실행하기 위해서 개발 도구를 설치할 필요가 없습니다. 개발자는 Docker를 설치하고 소스 코드를 복제하며 커맨드로 전체 어플리케이션을 빌드하고 실행 할 수 있습니다.

Docker를 사용하면 타사 소프트웨어를 쉽게 가져와서 기능을 추가할 수 있습니다. Docker Hub는 컨테이너에서 실행되는 소프트웨어를 공유하는 공개 서비스입니다. CNCF는 모니터링에서 메시지 큐에 이르기까지 사용할 수 있는 오픈 소스 프로젝트 맵을 제공하고 있고 이는 Docker Hub에서 무료로 사용할 수 있습니다.

### 1.4.4 서버리스 및 기타

현재의 IT 트렌드중 중요한 것은 일관성입니다. 모든 프로젝트에 동일한 도구, 프로세스 및 런타임을 사용하려고 합니다. Docker를 사용하면 Windows에서 실행되는 .NET 모놀리스에서 Linux에서 실행되는 새로운 Go 어플리케이션까지 사용할 수 있습니다. Docker 클러스터를

구축하여 모든 앱을 실행 할 수 있으므로 동일한 방식으로 전체 애플리케이션 환경을 구축, 배포, 관리 할 수 있습니다.

기술 혁신은 업무용 어플리케이션과 분리되어서는 안됩니다. Docker에서도 동일한 도구와 기술을 계속 사용할 수 있어야 합니다. 현재 가장 흥미로운 기술중 하나가 서버리스 입니다.

서버리스는 컨테이너에 관한 것입니다. 서버리스의 목표는 개발자가 함수 코드를 작성하고 이를 서비스로 Push하여 해당 서비스가 코드를 빌드 및 패키징화 하는 것입니다. 사용자가 함수를 사용할 때 서비스는 함수의 인스턴스를 시작하여 요청을 처리합니다. 관리할 빌드 서버, 파이프라인 또는 프로덕션 서버가 없으며 모두 플랫폼에서 관리합니다.

현재 서버리스에 대한 공개 표준이 없기 때문에 AWS Lambda 기능을 가져와서 Azure에서 실행할 수 없습니다. 클라우드 제공 업체에 Lock-In되지 않는 서버리스를 원하거나 데이터 센터를 이용중인 경우에는 널리 사용되는 오픈 소스 서버리스 프레임워크인 Nuclio, OpenFaaS, Fn Project를 사용하여 Docker에서 자체 플랫폼을 호스팅 할 수 있습니다.

이 뿐만 아니라, 머신러닝, 블록체인 및 IoT와 같은 다른 주요 혁신 기술은 Docker의 일관된 패키징 및 배포 모델에서 이점을 얻을 수 있습니다. Docker Hub에 배포된 주요 프로젝트를 중에 TensorFlow 및 Hyperledger가 좋은 예입니다.

Docker가 컨테이너를 Edge 및 IoT 단말의 기본 런타임으로 만들기 위해 Arm과 파트너 관계를 맺었고 이 분야가 현재 많이 발전되고 있습니다.

#### 1.4.5 DevOps를 통한 디지털 혁신

위에서 언급한 모든 시나리오는 기술에 관한 것이지만, 많은 조직이 직면한 가장 큰 문제는 운영입니다. 특히 대기업일 경우에는 이 부분이 가장 큰 문제입니다.

일반적으로 “개발자” 및 “운영자”에게 이런 역할이 맡겨져 프로젝트를 수행하는 동안 여러 부분을 담당합니다. 일반적으로 릴리즈시에 테스트를 통해 품질을 향상시키기 위해 노력할 것이고 일년에 약 2~3개의 릴리즈만 관리 할 수 있게 됩니다.

DevOps는 단일팀이 “dev”와 “ops”를 하나의 결과물로 결합하여 전체 어플리케이션 수명 주기를 관리하게 함으로써 소프트웨어 배포 및 유지 관리에 Agility를 제공하는 것을 목표로 합니다.

DevOps는 주로 문화에 관한 것이고 대규모 릴리즈 및 소규모 릴리즈 혹은 일일 배포까지 할 수 있습니다. 그러나 Team이 사용하는 기술을 변경하지 않고서는 이렇게하기가 어렵습니다.

운영자는 Bash, Nagios, PowerShell 및 System Center와 같은 도구에 대한 배경 지식을 지니고 있습니다. 개발자는 Make, Maven, MSBuild에 대한 배경 지식을 지니고 있습니다. 컨테이너로 이동하여 DevOps로의 변화를 이끌려면 Team내에서 Dockerfile 및 Docker Compose 파일을 사용하여 동일한 언어/도구를 이용해야 합니다.

더 나아가서 CALMS라고 하는 DevOps(문화, 자동화, Lean, 메트릭 및 공유)를 구현하기 위한 강력한 프레임워크가 있습니다. Docker는 이러한 모든 곳에서 작동합니다. 자동화는 컨테이너

실행의 중심이며, 분산 된 앱은 간단한 원칙에 기반을 두고 있고, 프로덕션 앱의 메트릭과 배포 프로세스의 메트릭을 쉽게 볼 수 있으며 Docker Hub는 공유로써의 역할을 담당합니다.

## 1.5 이 책의 목적은?

지금까지 설명한 시나리오는 현재 IT 산업에서 일어나는 모든 활동을 다루고 있습니다. Docker가 이 모든 것의 열쇠라는 것을 얘기했습니다. Docker로 이런 종류의 문제를 해결하도록 하려는 것이 이 책의 목적입니다.

이 책은 Docker 사용 방법을 가르치는 것이므로 Docker 자체가 어떻게 작동하는지에 대해서는 자세히 설명하지 않습니다. 내부를 자세히 원한다면 다른 책을 보셔야 합니다.

이 책의 샘플은 모두 크로스 플랫폼이므로 Arm 프로세서를 포함하여 Windows, Mac 또는 Linux를 사용하여 작업할 수 있습니다.

### 1.5.1 이 책의 사용법

이 책은 1 시간에 각장을 볼 수 있고 약 한달안에 책 전체를 읽을 수 있도록 제작되었습니다. 매일 1 시간씩 섹션을 읽고 샘플 코드를 통해 연습을 하는 시간을 고려 했습니다.

#### 1.5.1.1 실습

각 섹션은 실습으로 끝납니다. 이 책의 소스 코드는 모두 GitLab에 있습니다. 실습 환경을 설정할때에 복제하여 이용해야 합니다.

#### 1.5.1.2 References

이 책에 언급된 내용에 대해서 더 자세히 알고싶으시면 docs.docker.com 을 보시면 됩니다. 여기에는 Docker Engine 설정 부터 Dockerfile 및 Docker Compose의 문법, Docker Swarm 미 Docker의 엔터프라이즈 제품에 대해서 언급하고 있습니다.

## 1.6 실습 환경 만들기

이제 시작합니다. 이 책과 함께 봐야 할 것은 Docker와 샘플 코드 입니다.

### 1.6.1 도커 설치

Docker Community Edition은 개발 용도로 적합합니다. 최신 버전의 Windows 10 또는 Mac OS X에서 Docker를 사용하기 위한 가장 좋은 선택은 Docker Desktop 입니다. 이전 버전은 Docker Toolbox를 사용할 수 있습니다. Docker는 주요 Linux 배포판을 위한 설치 패키지도 제공합니다. 가장 적합한 옵션을 사용하여 Docker를 설치하세요.

#### 1.6.1.1 Mac OS X에 Docker Desktop 설치

Mac용 Docker Desktop을 사용하려면 OSX Sierra 이상 버전의 OSX가 설치되어 있어야 합니다. 메뉴 막대의 왼쪽 상단에 있는 Apple 아이콘을 클릭하고 “이 Mac에 관하여”를 선택하여 버전을 확인하세요.

<https://www.docker.com/products/docker-desktop> 으로 이동하여 안정 버전을 다운로드 합니다.

다운로드하여 설치 프로그램을 실행하고 모든 기본값을 그대로 사용합니다. Docker Desktop이 실행중이면 상단 바 시계 근처의 Mac 메뉴 막대에 Docker 로고 아이콘이 표시됩니다.

#### 1.6.1.2 Windows 10에서 Docker Desktop 설치

Docker Desktop을 사용하려면 Windows 10 Professional 또는 Enterprise가 필요하기에 Windows 업데이트가 최신 상태인지 확인해야 합니다.

<https://www.docker.com/products/docker-desktop> 으로 이동하여 안정 버전을 다운로드 합니다. 다운로드 된 설치 프로그램을 실행하고 모든 기본값을 그대로 사용합니다. Docker Desktop이 실행 중이면 Windows 바 시계 근처의 작업 표시줄에 Docker 로고 아이콘이 표시됩니다.

#### 1.6.1.3 Docker Toolbox 설치

이전 버전의 Windows 또는 OS X를 사용하는 경우 Docker Toolbox를 사용할 수 있습니다. <https://docs.docker.com/toolbox> 로 이동하여 설치 방법대로 수행합니다. VirtualBox와 같은 Virtual Machine 소프트웨어를 먼저 설정해야 합니다. (Docker Desktop은 별도의 VM 관리자가 필요없습니다.)

#### 1.6.1.4 Docker 설정 확인

Docker 플랫폼을 구성하는 여러 구성 요소가 있지만 이 책에서는 Docker가 실행 중이고 Docker Compose가 설치되어 있는지만 확인하면 됩니다.

먼저 터미널창을 오픈 한 다음에 “docker version” 커맨드로 확인합니다.

```
grouq:~ giljae$ docker version
Client: Docker Engine - Community
Version:      19.03.5
API version:  1.40
Go version:   go1.12.12
Git commit:   633a0ea
Built:        Wed Nov 13 07:22:34 2019
OS/Arch:      darwin/amd64
Experimental: false
```

```
Server: Docker Engine - Community
Engine:
Version:      19.03.5
API version:  1.40 (minimum version 1.12)
Go version:   go1.12.12
Git commit:   633a0ea
Built:        Wed Nov 13 07:29:19 2019
OS/Arch:      linux/amd64
Experimental: true
containerd:
Version:      v1.2.10
GitCommit:    b34a5c8af56e510852c35414db4c1f4fa6172339
runc:
Version:      1.0.0-rc8+dev
GitCommit:    3e425f80a8c931f88e6d94a8c831b9d5aa481657
docker-init:
Version:      0.18.0
GitCommit:    fec3683
```

클라이언트와 서버의 Version 번호를 볼 수 있으면 Docker가 제대로 작동합니다. 아직 클라이언트와 서버가 무엇인지 모르셔도 됩니다. 차차 배우게 될 예정입니다.

Docker와 상호 작용하는 Docker Compose를 테스트해야 합니다. “docker-compose version”으로 확인합니다.

```
grouq:~ giljae$ docker-compose version
docker-compose version 1.24.1, build 4667896b
docker-py version: 3.7.3
CPython version: 3.6.8
OpenSSL version: OpenSSL 1.1.0j 20 Nov 2018
```

## 1.6.2 샘플 코드 다운로드

이 책의 소스 코드는 GitLab의 공개 Git 저장소에 있습니다.

[https://gitlab.com/giljae/learn\\_docker](https://gitlab.com/giljae/learn_docker)로 이동하여 “복제 또는 다운로드” 버튼을 클릭하여 소스 코드를 로컬 시스템에 다운로드하고 압축을 해제하세요.

## 2. Docker 이해하기 및 Hello World 실행

이제 Docker를 실습 할 차례입니다. 이 장에서는 Docker의 핵심 기능인 컨테이너에서 어플리케이션을 실행하는 방법에 대해서 배울 수 있습니다.

## 2.1 컨테이너에서 Hello World 실행하기

새로운 프로그래밍 언어를 배울때에는 항상 “Hello World”를 출력합니다. 이것과 동일한 방식으로 Docker를 시작해 봅시다.

Docker가 이미 설치되어 동작중이므로 터미널 창을 엽니다. Mac에서는 “iTerm or Terminal”, Linux에서는 “Bash Shell”, Windows에서는 “PowerShell”을 권장합니다.

Docker에 명령을 보내어 간단한 “Hello, World” 텍스트를 출력하는 컨테이너를 실행하라는 명령을 내립니다.

```
$ docker container run giljae/ld_ch02:0.1
```

위의 명령어를 실행하면 아래의 결과가 나옵니다.

```
grouq:~ giljae$ docker container run giljae/ld_ch02:0.1
Unable to find image 'giljae/ld_ch02:0.1' locally
0.1: Pulling from giljae/ld_ch02
e7c96db7181b: Pull complete
3c9c179c43cf: Pull complete
5204d567b517: Pull complete
Digest:
sha256:3e21bb71d4c6fa841dc78c80a096da45874b0631f543b411bf6a6520c480eb
22
Status: Downloaded newer image for giljae/ld_ch02:0.1
-----
Hello, Chapter 2!
-----
My name is:
70fef9286bd6
-----
Im running on:
Linux 4.9.184-linuxkit x86_64
-----
My address is:
inet addr:172.17.0.2 Bcast:172.17.255.255 Mask:255.255.0.0
```

Docker의 세상에 오신걸 환영 합니다.

## 2.2 첫번째 컨테이너 실행 (docker run 명령어)

터미널에서 아래의 명령어를 실행하세요.



```
$ docker pull alpine
```

pull 명령은 Docker 레지스트리에서 알파인 이미지를 가져와서 시스템에 저장합니다. 명령어를 사용하여 로컬 시스템에 저장된 모든 이미지 목록을 볼 수 있습니다. (e.g. docker images)

```
grouq:~ giljae$ docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
alpine        latest    965ea09ff2eb   7 weeks ago    5.55MB
```

이제 위의 이미지를 기반으로 Docker 컨테이너를 실행합니다. 이를 위해서 docker run 명령어를 사용합니다.

```
grouq:~ giljae$ docker run alpine ls -l
total 56
drwxr-xr-x  2 root  root    4096 Oct 21 13:39 bin
drwxr-xr-x  5 root  root    340 Dec 15 12:20 dev
drwxr-xr-x  1 root  root    4096 Dec 15 12:20 etc
drwxr-xr-x  2 root  root    4096 Oct 21 13:39 home
drwxr-xr-x  5 root  root    4096 Oct 21 13:39 lib
drwxr-xr-x  5 root  root    4096 Oct 21 13:39 media
drwxr-xr-x  2 root  root    4096 Oct 21 13:39 mnt
drwxr-xr-x  2 root  root    4096 Oct 21 13:39 opt
dr-xr-xr-x 206 root  root      0 Dec 15 12:20 proc
drwx----- 2 root  root    4096 Oct 21 13:39 root
drwxr-xr-x  2 root  root    4096 Oct 21 13:39 run
drwxr-xr-x  2 root  root    4096 Oct 21 13:39 sbin
drwxr-xr-x  2 root  root    4096 Oct 21 13:39 srv
dr-xr-xr-x 13 root  root      0 Dec 15 12:20 sys
drwxrwxrwt  2 root  root    4096 Oct 21 13:39 tmp
drwxr-xr-x  7 root  root    4096 Oct 21 13:39 usr
drwxr-xr-x 11 root  root    4096 Oct 21 13:39 var
```

디렉토리 목록들이 출력되지요? 어떻게 된 것일까요?  
docker run 명령을 실행하면,

1. docker 클라이언트는 Docker Daemon에 연결됩니다,
2. Docker Daemon은 Image(e.g. Alpine)가 로컬에 존재하는지 로컬 저장소를 확인하고 그렇지 않을 경우 Docker Store에서 Image를 다운로드 합니다. (우리는 이미 이전에 docker pull alpine으로 이미지를 다운받았기에 다운로드 하지 않습니다.)
3. Docker Daemon은 컨테이너를 만든 다음 해당 컨테이너에서 명령을 실행합니다.
4. Docker Daemon은 명령어에 대한 출력을 Docker 클라이언트로 스트리밍 합니다.

즉, docker run alpine과 리눅스 명령어인 ls -l 을 제공했으므로 Docker는 지정된 명령을 실행합니다.

아래의 명령어를 실행해봅시다.

```
grouq:~ giljae$ docker run alpine echo "Hello World!!"  
Hello World
```

이 경우 Docker 클라이언트는 echo 명령을 실행하고 종료됩니다. 느꼈듯이 이 모든 것이 꽤 빨리 일어났습니다. VM을 부팅하고 명령을 실행한 다음 종료한다고 상상해보세요. 이제 컨테이너가 빠르다는 이유를 느꼈을거라 생각합니다.

다른 명령어를 실행해봅시다.

```
grouq:~ giljae$ docker run alpine /bin/sh
```

이상합니다. 아무것도 출력되지 않습니다. 그 이유는 인터프리터 쉘은 스크립트 명령을 실행한 후에 종료됩니다. 따라서 아무것도 출력하지 않고 종료하지 않으려면 아래처럼 수행해야 합니다.

```
grouq:~ giljae$ docker run -it alpine /bin/sh  
/ #
```

alpine에서 shell을 띄운 상태에서 ls -l 명령을 실행해봅시다.

```
/ # ls -l  
total 56  
drwxr-xr-x  2 root  root    4096 Oct 21 13:39 bin  
drwxr-xr-x  5 root  root    360 Dec 15 12:32 dev  
drwxr-xr-x  1 root  root    4096 Dec 15 12:32 etc  
drwxr-xr-x  2 root  root    4096 Oct 21 13:39 home  
drwxr-xr-x  5 root  root    4096 Oct 21 13:39 lib  
drwxr-xr-x  5 root  root    4096 Oct 21 13:39 media  
drwxr-xr-x  2 root  root    4096 Oct 21 13:39 mnt  
drwxr-xr-x  2 root  root    4096 Oct 21 13:39 opt  
dr-xr-xr-x 200 root  root      0 Dec 15 12:32 proc  
drwx----- 1 root  root    4096 Dec 15 12:34 root  
drwxr-xr-x  2 root  root    4096 Oct 21 13:39 run  
drwxr-xr-x  2 root  root    4096 Oct 21 13:39 sbin  
drwxr-xr-x  2 root  root    4096 Oct 21 13:39 srv  
dr-xr-xr-x 13 root  root      0 Dec 15 12:32 sys  
drwxrwxrwt  2 root  root    4096 Oct 21 13:39 tmp  
drwxr-xr-x  7 root  root    4096 Oct 21 13:39 usr  
drwxr-xr-x 11 root  root    4096 Oct 21 13:39 var  
/ #
```

리눅스 커널 버전을 알 수 있는 uname -a 도 실행해봅시다.

```
/ # uname -a  
Linux 2e8c5e16a35f 4.9.184-linuxkit #1 SMP Tue Jul 2 22:58:16 UTC 2019 x86_64
```

자, 이제 exit 명령으로 컨테이너를 종료합니다.

```
/ # exit
```

## 2.3 컨테이너 상태 확인하기

이제 docker ps 명령에 대해서 알아보시다. docker ps 명령은 현재 실행중인 모든 컨테이너를 보여줍니다.

```
grouq:~ giljae$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			

컨테이너가 실행되고 있지 않으므로 빈 줄이 나타납니다. 옵션을 추가하여 시도해 봅시다.

```
grouq:~ giljae$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
2e8c5e16a35f	alpine	"/bin/sh"	8 minutes ago	Exited (0) 4 minutes ago
339345cae78c	alpine	"/bin/sh"	11 minutes ago	Exited (0) 11 minutes ago
75e80226ef47	alpine	"echo 'Hello Worldcl..."	14 minutes ago	Exited (0) 14 minutes ago
35b6b503cf76	alpine	"ls -l"	20 minutes ago	Exited (0) 20 minutes ago
fa32968406d4	alpine	"la -l"	21 minutes ago	Created
happy_albattani				
68fe0cfa2acd	alpine	"la -l"	21 minutes ago	Created
laughing_jackson				
8cbf74f2fa89	alpine	"la -l"	21 minutes ago	Created
heuristic_cannon				

위에 나온 목록은 실행한 모든 컨테이너의 목록입니다. Status 열은 해당 컨테이너가 몇 분전에 종료되었는지 알려줍니다.

계속해서 명령어를 수행하는 방법은 아래와 같습니다.

```
grouq:~ giljae$ docker run -it alpine /bin/sh
/ # ls
bin  etc  lib  mnt  proc  run  srv  tmp  var
dev  home media opt  root sbin sys  usr
/ # uname -a
```

```
Linux 5a2c0b3732d3 4.9.184-linuxkit #1 SMP Tue Jul 2 22:58:16 UTC 2019 x86_64
Linux
```

run 뒤에 -it를 붙이면 인터프리터 tty에 연결됩니다. 이렇게하면 컨테이너에서 원하는 만큼 명령을 실행할 수 있습니다.

docker run은 아마 가장 많이 사용하는 명령어가 될 것입니다. 더 자세한 내용은 docker run --help로 확인할 수 있습니다.

## 2.4 Image 검색하기

docker search 명령어는 Docker Hub(<https://hub.docker.com>)에서 이미지를 검색해주는 명령어입니다.

docker는 Docker Hub를 통해 이미지를 공유하고 있습니다. 유명 오픈소스 및 리눅스 배포본을 Docker Hub를 통해 구할 수 있습니다. Docker Hub에서 우분투 이미지를 검색해보도록 합시다.

```
grouq:~ giljae$ docker search ubuntu
NAME                                DESCRIPTION                                STARS
OFFICIAL      AUTOMATED
ubuntu                                Ubuntu is a Debian-based Linux operating sys...
10270         [OK]
dorowu/ubuntu-desktop-lxde-vnc      Docker image to provide HTML5
VNC interface ... 370               [OK]
rastasheep/ubuntu-ssh               Dockerized SSH service, built on top of
offi... 236                          [OK]
consol/ubuntu-xfce-vnc              Ubuntu container with "headless" VNC
session... 199                      [OK]
ubuntu-upstart                      Upstart is an event-based replacement for
th... 102                          [OK]
neurodebian                         NeuroDebian provides neuroscience research
s... 62                          [OK]
1and1internet/ubuntu-16-nginx-php-phpmyadmin-mysql-5
ubuntu-16-nginx-php-phpmyadmin-mysql-5 50 [OK]
ubuntu-debootstrap                  debootstrap --variant=minbase
--components=m... 41 [OK]
nuagebec/ubuntu                    Simple always updated Ubuntu docker
images w... 24 [OK]
i386/ubuntu                         Ubuntu is a Debian-based Linux operating
sys... 18
1and1internet/ubuntu-16-apache-php-5.6 ubuntu-16-apache-php-5.6
```

## 2.5 Image 가져오기

docker pull 명령어를 사용하여 Docker Hub에서 우분투 리눅스 이미지를 가져오도록

해보겠습니다.

```
grouq:~ giljae$ docker pull ubuntu:latest
latest: Pulling from library/ubuntu
7ddbc47eeb70: Pull complete
c1bbdc448b72: Pull complete
8c3b70e39044: Pull complete
45d437916d57: Pull complete
Digest:
sha256:6e9f67fa63b0323e9a1e587fd71c561ba48a034504fb804fd26fd8800039835
d
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
```

사용법:

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

위의 명령어에서 latest를 사용하면 최신 버전을 가져옵니다. ubuntu:14.04 로 하면 14.04 버전의 이미지를 가져옵니다.

## 2.6 Image 목록 출력하기

docker pull로 다운로드한 이미지 목록을 출력해보도록 하겠습니다.

```
grouq:~ giljae$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	latest	775349758637	6 weeks ago	64.2MB
alpine	latest	965ea09ff2eb	7 weeks ago	5.55MB

docker images 명령은 현재 시스템에 설치된 모든 이미지를 표시합니다.

각 열의 의미는 아래와 같습니다.

- REPOSITORY
  - 이미지의 저장소입니다.
- TAG
  - 이미지에 대한 논리 구분자입니다. 일반적으로 version으로 활용합니다.
- IMAGE ID
  - 이미지의 고유한 식별자입니다.
- CREATED
  - 이미지가 생성 된 후 일 수입니다.
- SIZE
  - 이미지의 크기입니다.

모든 이미지가 아닌 특정 이미지의 목록을 출력하고 싶을 경우에는 아래처럼 사용하시면 됩니다.

```
grouq:~ giljae$ docker images ubuntu
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	latest	775349758637	6 weeks ago	64.2MB
ubuntu	14.04	2c5e00d77a67	7 months ago	188MB

이렇게 하면 이름은 “ubuntu”와 같지만 tag가 다른 이미지들이 출력됩니다.

## 2.7 컨테이너 생성하기

docker run 명령어를 이용하여 이미지를 컨테이너로 생성하고 linux shell을 실행해보겠습니다.

```
grouq:~ giljae$ docker run -it ubuntu /bin/bash
root@0da605b182cd:/#
```

사용법:

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

docker run 명령어로 ubuntu 이미지를 컨테이너로 생성 한 후 ubuntu 이미지의 /bin/bash를 실행했습니다. 이미지 이름 대신에 이미지 ID를 사용해도 됩니다.

이제 Host OS와 완전히 격리된 ubuntu 컨테이너를 생성하였습니다. 리눅스 명령어를 입력하면서 테스트 해보시길 바랍니다.

exit 명령어를 입력하면 컨테이너에서 빠져나오게 되고 이렇게 되면 컨테이너가 stop 되게 됩니다.

## 2.8 컨테이너 시작하기

방금 exit로 빠져나온 컨테이너를 다시 시작해보도록 하겠습니다.

우선, 컨테이너 목록을 확인합니다.

```
grouq:~ giljae$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
0da605b182cd	ubuntu	"/bin/bash"	7 minutes ago	Exited (0)
About a minute ago		exciting_bhaskara		
5a2c0b3732d3	alpine	"/bin/sh"	58 minutes ago	Exited (0) 32 minutes ago
		eager_easley		
2e8c5e16a35f	alpine	"/bin/sh"	About an hour ago	Exited (0)
About an hour ago		elastic_diffie		
339345cae78c	alpine	"/bin/sh"	About an hour ago	Exited (0)
About an hour ago		hungry_euler		
75e80226ef47	alpine	"echo 'Hello Worldcl..."	About an hour ago	Exited (0)
About an hour ago		upbeat_einstein		

```
35b6b503cf76    alpine    "ls -l"    About an hour ago    Exited (0)
About an hour ago
fa32968406d4    alpine    "ls -l"    About an hour ago    Created
happy_albattani
68fe0cfa2acd    alpine    "ls -l"    About an hour ago    Created
laughing_jackson
8cbf74f2fa89    alpine    "ls -l"    About an hour ago    Created
heuristic_cannon
```

방금 실행한 컨테이너가 보입니다. 이제 컨테이너를 시작하도록 합니다.

```
grouq:~ giljae$ docker start exciting_bhaskara
exciting_bhaskara
```

docker ps 로 컨테이너가 시작되었는지 확인합니다.

```
grouq:~ giljae$ docker ps
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS
PORTS    NAMES
0da605b182cd    ubuntu    "/bin/bash"    8 minutes ago    Up 9 seconds
exciting_bhaskara
```

ubuntu 컨테이너가 시작된 것을 확인할 수 있습니다.

사용법:

```
docker start [OPTIONS] CONTAINER [CONTAINER...]
```

그런데 좀 이상하지 않나요?

docker start exciting\_bhaskara로 시작한걸 기억하시나요? 앞에서 컨테이너를 생성할 때 아래의 명령어로 생성을 했었습니다.

```
$ docker run -it ubuntu /bin/bash
```

이렇게 하면 docker가 임의로 컨테이너 이름을 부여하게 됩니다. 그래서

“exciting\_bhaskara”라는 컨테이너 이름이 부여되었고, 컨테이너 시작시 이 이름을 사용하게 된 것입니다. 좀 불편하지요? 그럼 어떻게 해야 할까요?

```
grouq:~ giljae$ docker run -it --name ubuntu ubuntu /bin/bash
```

--name 옵션을 사용해서 이름을 부여해주도록 합니다. 이렇게 하면 컨테이너를 시작할때 아래처럼 할 수 있습니다.

```
$ docker start ubuntu
```

docker ps 로 컨테이너가 해당 이름으로 시작되었는지 확인합니다.

```
grouq:~ giljae$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
11eb0f78c677	ubuntu	"/bin/bash"	3 minutes ago	Up About a minute

ubuntu라는 이름으로 시작된 것을 확인할 수 있습니다.

## 2.9 컨테이너 재시작 하기

운영체제 재부팅 처럼 컨테이너도 재시작을 할 수 있습니다.

```
grouq:~ giljae$ docker restart ubuntu
ubuntu
grouq:~ giljae$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
11eb0f78c677	ubuntu	"/bin/bash"	4 minutes ago	Up 2 seconds

사용법:

```
docker restart [OPTIONS] CONTAINER [CONTAINER...]
```

## 2.10 컨테이너에 접속하기

컨테이너를 구동하고 접속해보도록 하겠습니다.

```
grouq:~ giljae$ docker attach ubuntu
root@11eb0f78c677:/#
```

위처럼 구동중인 컨테이너에 접속한 것을 확인할 수 있습니다. 컨테이너 이름 대신에 컨테이너 ID를 사용해도 됩니다.

사용법:

```
docker attach [OPTIONS] CONTAINER
```

## 2.11 외부에서 컨테이너내의 명령 실행하기

간혹 컨테이너에 접속하지 않고, 외부에서 컨테이너내의 명령을 실행할 경우가 생깁니다.

```
grouq:~ giljae$ docker exec ubuntu ls -l
total 64
```



```
drwxr-xr-x 2 root root 4096 Oct 29 21:25 bin
drwxr-xr-x 2 root root 4096 Apr 24 2018 boot
drwxr-xr-x 5 root root 360 Dec 16 02:29 dev
drwxr-xr-x 1 root root 4096 Dec 15 13:50 etc
drwxr-xr-x 2 root root 4096 Apr 24 2018 home
drwxr-xr-x 8 root root 4096 May 23 2017 lib
drwxr-xr-x 2 root root 4096 Oct 29 21:25 lib64
drwxr-xr-x 2 root root 4096 Oct 29 21:25 media
drwxr-xr-x 2 root root 4096 Oct 29 21:25 mnt
drwxr-xr-x 2 root root 4096 Oct 29 21:25 opt
dr-xr-xr-x 204 root root 0 Dec 16 02:29 proc
drwx----- 1 root root 4096 Dec 15 13:50 root
drwxr-xr-x 1 root root 4096 Oct 31 22:20 run
drwxr-xr-x 1 root root 4096 Oct 31 22:20/sbin
drwxr-xr-x 2 root root 4096 Oct 29 21:25 srv
dr-xr-xr-x 13 root root 0 Dec 16 02:29 sys
drwxrwxrwt 2 root root 4096 Oct 29 21:25 tmp
drwxr-xr-x 1 root root 4096 Oct 29 21:25 usr
drwxr-xr-x 1 root root 4096 Oct 29 21:25 var
```

위의 명령어는 실행중인 컨테이너내의 명령을 실행한 결과입니다.

사용법:

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

## 2.12 컨테이너 실행 정지 하기

컨테이너를 정지하는 방법에 대해서 알아보도록 합니다. 우선, 현재 구동중인 컨테이너 목록을 출력합니다.

```
grouq:~ giljae$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
11eb0f78c677	ubuntu	"/bin/bash"	13 hours ago	Up 6 minutes

구동중인 컨테이너를 확인하고 아래의 명령으로 정지시킵니다.

```
grouq:~ giljae$ docker stop ubuntu
ubuntu
```

컨테이너 이름 대신에 컨테이너 ID를 사용해도 됩니다.

사용법:

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

ubuntu 컨테이너를 정지 했으니, docker ps로 확인합니다.

```
grouq:~ giljae$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			

현재 구동중인 컨테이너가 없습니다.

## 2.13 컨테이너 삭제하기

이미 생성된 컨테이너를 삭제하도록 하겠습니다.

```
grouq:~ giljae$ docker rm ubuntu
ubuntu
```

사용법:

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

컨테이너 이름 대신에 컨테이너 ID를 사용해도 됩니다.

컨테이너가 삭제되었는지 확인해봅니다.

```
grouq:~ giljae$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
5a2c0b3732d3	alpine	"/bin/sh"	14 hours ago	Exited (0) 14 hours ago
2e8c5e16a35f	alpine	"/bin/sh"	14 hours ago	Exited (0) 14 hours ago
339345cae78c	alpine	"/bin/sh"	14 hours ago	Exited (0) 14 hours ago
75e80226ef47	alpine	"echo 'Hello Worldcl..."	14 hours ago	Exited (0) 14 hours ago
35b6b503cf76	alpine	"ls -l"	14 hours ago	Exited (0) 14 hours ago
fa32968406d4	alpine	"la -l"	14 hours ago	Created
happy_albattani				
68fe0cfa2acd	alpine	"la -l"	14 hours ago	Created
laughing_jackson				
8cbf74f2fa89	alpine	"la -l"	14 hours ago	Created
heuristic_cannon				

ubuntu 컨테이너가 삭제된 것을 확인할 수 있습니다.

## 2.14 Image 삭제하기

하나 이상의 이미지를 삭제해보도록 하겠습니다. 이 명령어는 호스트 노드에서 하나 이상의 이미지를 제거하고 태그를 해제합니다. 이미지에 여러개의 태그가 존재하는 경우, 태그와 함께 아규먼트로 사용하면 태그만 삭제되고 태그가 유일할 경우 이미지와 태그 모두 삭제됩니다.

사용법:

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

ubuntu 이미지를 삭제하도록 하겠습니다. tag를 입력하지 않았기에 latest가 자동으로 붙습니다.

```
grouq:~ giljae$ docker rmi ubuntu
Untagged: ubuntu:latest
Untagged:
ubuntu@sha256:6e9f67fa63b0323e9a1e587fd71c561ba48a034504fb804fd26fd8800039835d
Deleted:
sha256:775349758637aff77bf85e2ff0597e86e3e859183ef0baba8b3e8fc8d3cba51c
Deleted:
sha256:4fc26b0b0c6903db3b4fe96856034a1bd9411ed963a96c1bc8f03f18ee92ac2a
Deleted:
sha256:b53837dafdd21f67e607ae642ce49d326b0c30b39734b6710c682a50a9f932bf
Deleted:
sha256:565879c6effe6a013e0b2e492f182b40049f1c083fc582ef61e49a98dca23f7e
Deleted:
sha256:cc967c529ced563b7746b663d98248bc571afdb3c012019d7f54d6c092793b8b
```

이미지가 삭제되었는지 확인합니다.

```
grouq:~ giljae$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
alpine              latest             965ea09ff2eb       7 weeks ago        5.55MB
```

ubuntu 이미지가 삭제된 것을 확인할 수 있습니다.

## 3. Docker Image 및 컨테이너 추가 정보

이번 장에서는 이미지, 컨테이너 실행, 컨테이너 네트워킹 등을 처리하는 방법에 대해서 알아보도록 합니다.

## 3.1 Apache 웹 서버 실행

이제 조금 더 자세히 Docker내에서 실행되는 애플리케이션을 어떻게 구상하는지 알아보도록 하겠습니다.

우리는 Apache Web Server를 실행해야 합니다. 이렇게 하려면 누군가에 의해 이미 생성된 Docker 이미지가 있어야겠죠?

따라서 첫 번째로 아래를 수행합니다.

```
grouq:~ giljae$ docker search httpd
```

위의 명령어에 대한 결과는 아래와 같습니다.

```
grouq:~ giljae$ docker search httpd
NAME                                DESCRIPTION                                STARS     OFFICIAL
AUTOMATED
httpd                               The Apache HTTP Server Project            2771      [OK]
centos/httpd-24-centos7             Platform for running Apache httpd 2.4 or bui... 27
centos/httpd                        26                                          [OK]
armhf/httpd                         The Apache HTTP Server Project            8
salim1983hoop/httpd24               Dockerfile running apache config          2
[OK]
dariko/httpd-rproxy-ldap            Apache httpd reverse proxy with LDAP authent... 1
[OK]
solsson/httpd-openidc               mod_auth_openidc on official httpd image, ve... 1
[OK]
lead4good/httpd-fpm                 httpd server which connects via fcgi proxy h... 1
[OK]
interlutions/httpd                 httpd docker image with debian-based config ... 0
[OK]
dockerpinata/httpd                  0
itsziget/httpd24                   Extended HTTPD Docker image based on the off... 0
[OK]
manasip/httpd                       0
manageiq/httpd_configmap_generator  Httpd Configmap Generator                0
[OK]
appertly/httpd                     Customized Apache HTTPD that uses a PHP-FPM ... 0
[OK]
izdock/httpd                        Production ready Apache HTTPD Web Server + m... 0
trollin/httpd                       0
amd64/httpd                         The Apache HTTP Server Project            0
publici/httpd                       httpd:latest                              0          [OK]
e2eteam/httpd                       0
manageiq/httpd                      Container with httpd, built on CentOS for Ma... 0
[OK]
buzzardev/httpd                    Based on the official httpd image          0          [OK]
hypoport/httpd-cgi                  httpd-cgi                                  0          [OK]
alvistack/httpd                     Docker Image Packaging for Apache           0          [OK]
ppc64le/httpd                       The Apache HTTP Server Project            0
```

httpd를 검색하니 많은 이미지 목록이 출력됩니다. 이 중에서 우리는 공식 이미지를 설치할 계획입니다.

어떻게 해야 할까요? 위에서 배운 명령어 중 docker pull을 사용합니다.

```
grouq:~ giljae$ docker pull httpd
Using default tag: latest
latest: Pulling from library/httpd
000eee12ec04: Pull complete
32b8712d1f38: Pull complete
f1ca037d6393: Pull complete
c4bd3401259f: Pull complete
51c60bde4d46: Pull complete
Digest:
sha256:ac6594daaa934c4c6ba66c562e96f2fb12f871415a9b7117724c52687080d35d
Status: Downloaded newer image for httpd:latest
docker.io/library/httpd:latest
```

httpd 이미지가 다운로드 되었습니다. docker images를 이용해서 확인합니다.

```
grouq:~ giljae$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
httpd	latest	2ae34abc2ed0	2 weeks ago	165MB
alpine	latest	965ea09ff2eb	7 weeks ago	5.55MB

httpd 이미지에 대한 자세한 내용은 [공식 문서](#)를 확인하시면 됩니다. 각 이미지에 대한 설명 페이지를 살펴보는 것은 중요합니다. 해당 페이지에 이미지 실행 방법 및 기타 구성 사항에 대해서 상세하게 작성되어 있습니다.

공식 이미지에서 제공하는 Apache 웹 서버를 컨테이너에서 실행하려면 다음을 수행해야 합니다. 아래의 명령어에는 -d 옵션을 사용하고 있습니다. (e.g -d 옵션은 컨테이너를 분리 모드로 실행) 그리고 --name 옵션으로 컨테이너 이름을 지정했습니다.

```
grouq:~ giljae$ docker run -d --name apache httpd
c55c64b2c763bf4d0e3670301a2afb2903479f7aa42ff84d01cb82fef700fe7
```

이렇게 되면 httpd 이미지를 기반으로 기본 Apache Web Server가 시작됩니다. 컨테이너를 분리 모드로 시작했기 때문에 컨테이너 ID를 다시 얻습니다.

docker ps 명령어로 httpd가 구동되었는지 확인합니다.

```
grouq:~ giljae$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			

```
c55c64b2c763    httpd    "httpd-foreground" 45 seconds ago    Up 44
seconds        80/tcp    apache
```

실행 상태입니다. NAMES에는 구동하기전 지정한 apache가 있습니다.

### 3.1.1 docker ps시 ports 정보 확인 하기

docker ps 명령어에 대한 출력결과를 보면 PORTS가 존재하는 것을 확인할 수 있습니다. 해당 값은 80/tcp 입니다. 이는 포트 80이 컨테이너에 의해 노출되고 있음을 의미합니다. (일반적으로 Apache 웹 서버의 기본 포트는 80입니다.)

### 3.1.2 웹 사이트에 접속하기

curl 또는 로컬 컴퓨터의 브라우저를 이용해 사이트를 확인해 보도록 하겠습니다. 확인을 하려면 IP주소를 알아야 합니다. IP 주소는 무엇일까요?

Docker for Mac 또는 Docker for Windows를 실행중인 경우에는 localhost 입니다. 브라우저를 띄우고 <http://localhost> 로 접속합니다.

이상합니다! 페이지가 나타나지 않습니다. 우리의 예상은 아파치 홈페이지가 브라우저에 나타날 것으로 예상했지만 실제로는 그렇지 않습니다.

Apache Web Server가 실행되는 기본 포트는 80이지만 해당 포트가 호스트측에 노출되지 않아 웹 사이트에 접근할 수 없기 때문입니다.

80 포트는 컨테이너에 의해 노출되지만 호스트에는 노출되지 않습니다. 다시 말해서, 80 포트는 컨테이너에 의해 노출된 개인 포트이지만 공개 호스트에 매핑된 포트가 없다는 의미입니다.

이것에 대한 해결책은 무엇일까요? 아래의 명령어를 이용합니다.

```
grouq:~ giljae$ docker port apache
grouq:~ giljae$
```

결과는 예상대로입니다. 호스트의 공개 포트에 대한 매핑 정보가 없습니다.

무언가를 해야 할 것 같습니다.!

### 3.1.3 랜덤 포트 매핑

가장 먼저 컨테이너를 중지하고 제거합니다. 그리고 동일한 컨테이너 이름(e.g. apache)을 사용할 수 있도록 합니다.

```
grouq:~ giljae$ docker stop apache
apache
```

```
grouq:~ giljae$ docker rm apache
apache
grouq:~ giljae$
```

이제 -P를 사용하여 httpd 컨테이너를 시작하도록 하겠습니다. -P 옵션의 기능은 “노출된 모든 포트를 임의 포트에 게시”하는 것입니다. 따라서 80 포트는 임의 포트에 매핑되게 될 것입니다. 그리고 매핑된 포트는 공용 포트가 됩니다.

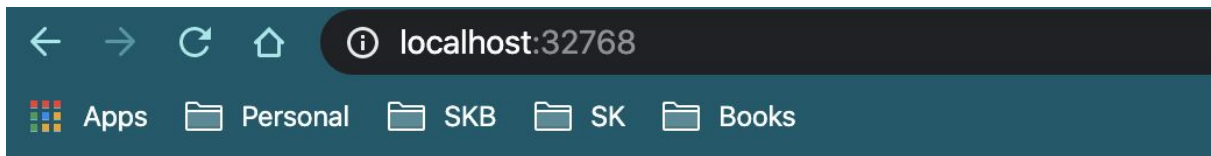
아래의 명령을 실행합니다.

```
grouq:~ giljae$ docker run -d --name apache -P httpd
959a35ffb02d613cb52e45765ca8c140f6c043deed6a6398524055f950876cd
```

port 명령을 이용해 정보를 확인합니다.

```
grouq:~ giljae$ docker port apache
80/tcp -> 0.0.0.0:32768
```

80 포트가 32768 포트로 매핑된 것을 확인할 수 있습니다. 따라서 아래의 URL로 브라우저에서 확인이 가능합니다.



## It works!

Apache 웹서버가 잘 작동하는 것을 확인할 수 있습니다.

### 3.1.4 특정 포트 매핑

32768 포트 이외의 포트를 매핑하려면 어떻게 해야 할까요? -p(소문자) 옵션을 이용해 이를 수행할 수 있습니다.

이 아규먼트의 형식은 아래와 같습니다.

```
-p HostPort : ContainerPort
e.g) -p 80:80 or -p 8080:80
```

첫 번째 아규먼트는 호스트 포트이며 80을 지정하고 있습니다. 두 번째 포트는 Apache 웹서버가 표시하는 것입니다. (e.g. 80)

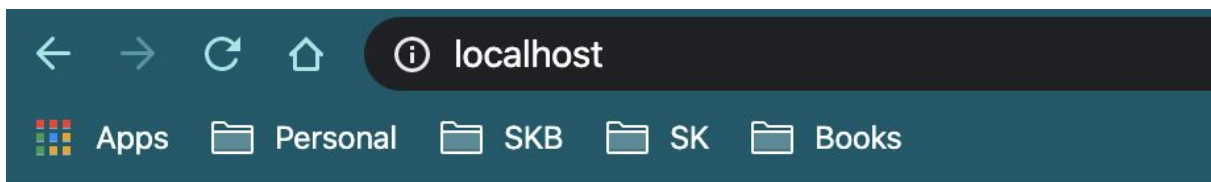
다시 확인해 봅시다.

```
grouq:~ giljae$ docker stop apache
apache
grouq:~ giljae$ docker rm apache
apache
grouq:~ giljae$ docker run -d --name apache -p 80:80 httpd
e47b69395f7570f8bfd431151f5530301a659a0af00f8219e11b10e44e836130
```

docker port 명령으로 매핑된 포트 정보를 확인합니다.

```
grouq:~ giljae$ docker port apache
80/tcp -> 0.0.0.0:80
```

이제 기본 포트(80)를 통해 웹 사이트에 접근 할 수 있습니다.



# It works!

이것으로 Apache Web Server와 관련한 섹션을 끝냅니다.

아직 궁금한 것이 많이 있습니다. HTML, CSS등을 어디에 위치시켜야 하는지 등등... 다음을 위해 남겨두도록 하지요.

## 4. Docker Hub 알아보기

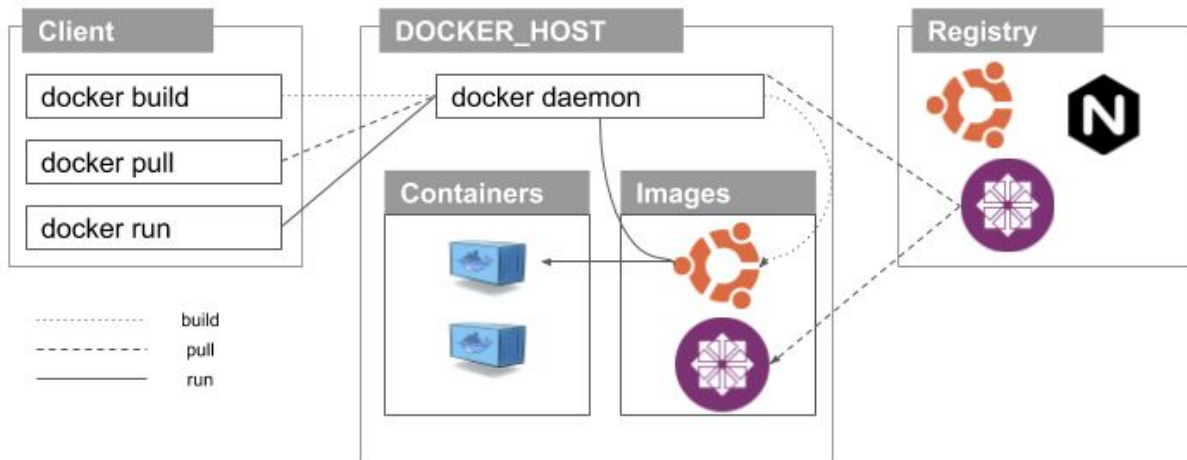
Docker 레지스트리 작업에 대해서 알아보도록 하겠습니다.

Docker 툴셋은 아래처럼 구성되어 있습니다.

1. Docker 데몬
2. Docker 클라이언트
3. Docker 허브

Docker Architecture는 아래처럼 구성 됩니다.





위의 그림에서 레지스트리 부분에 초점을 맞추겠습니다. 레지스트리는 허브라고도 합니다. 따라서 이 문서에서는 두 용어를 서로 바꿔서 사용할 수 있습니다.

Docker는 공용 Docker 이미지 목록을 찾을 수 있는 Docker Registry(Hub)라는 공용 저장소를 호스팅합니다. 많은 개발자들이 이미지를 준비해 주었기에 상당히 편리합니다. 이미지를 가져와서 컨테이너를 시작하기만 하면 되기 때문입니다.

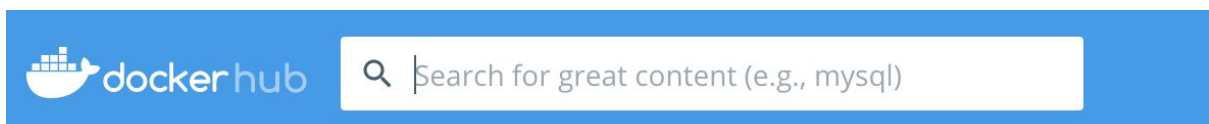
개발자에게 미치는 영향은 엄청납니다. 로컬에서 필요한 애플리케이션을 다운로드/설정하는 시간이 크게 줄어 듭니다. 예를 들어서 MySQL을 사용해야 한다고 가정 합시다. 사용 가능한 방법 중 하나는 공식 MySQL 사이트에서 설치 프로그램을 다운로드 하는 것입니다. 그리고 설치 가이드에 따라 설치하고 설정을 합니다. 이 프로세스는 시간이 오래 걸리고 오류가 발생할 여지가 존재합니다.

반면, 컨테이너는 Docker와 같은 툴체인을 사용하여 쉽게 빠르게 작업을 할 수 있습니다. 이를 위해서는 아래의 단계를 수행해야 합니다.

1. Docker Hub 방문: <https://hub.docker.com>
2. 허브에 가입하면 로컬에서 생성한 Docker 이미지를 허브에 Push할 수 있습니다. 이는 본인 뿐만아니라 다른 누군가에게도 도움이 됩니다.
3. 위의 사이트에 등록된 Docker 이미지 목록을 확인해보세요.


## 4.1 Docker 이미지 검색

Docker Hub 계정에 로그인했다고 가정합니다. (사실 검색은 계정이 없어도 동작합니다.) 아래의 검색 상자에서 이미지를 검색해보세요.



예를 들어서 Java를 입력하고 엔터를 누르면 이미지 결과 목록이 출력됩니다.

1 - 25 of 21,171 results for **Java**. [Clear search](#) Most Popular



**java**


Updated 44 minutes ago

Java is a concurrent, class-based, and object-oriented programming language.

Container Linux x86-64 Base Images Programming Languages

OFFICIAL IMAGE

10M+ Downloads 2.0K Stars




**Oracle Java 8 SE (Server JRE)** DOCKER CERTIFIED

By Oracle • Updated 2 months ago

Oracle Java 8 SE (Server JRE)

Container Docker Certified Linux x86-64 Programming Languages

VERIFIED PUBLISHER



**node**

Updated an hour ago

Node.js is a JavaScript-based platform for server-side and networking applications.

Container Linux PowerPC 64 LE ARM 386 IBM Z ARM 64 x86-64 Application Infrastructure

OFFICIAL IMAGE


10M+ Downloads 8.2K Stars

**openjdk**

10M+ 2.0K

별표와 다운로드 횟수를 확인하세요. 이것은 해당 이미지의 인기에 대해서 알려줍니다.

해당 리포지토리를 클릭하면 다양한 이미지(e.g. 태그별) 및 해당 이미지를 기반으로 컨테이너를 실행하기 위한 설명이 적혀있는 페이지가 나타납니다.



openjdk ☆  
Docker Official Images  
OpenJDK is an open-source implementation of the Java Platform, Standard Edition

10M+

Container Windows Linux x86-64 PowerPC 64 LE ARM 64 ARM 386 IBM Z Programming Languages

Official Image

Windows - x86-64 ( latest )

Copy and paste to pull this image

docker pull openjdk

View Available Tags

Description Reviews Tags

### Supported tags and respective Dockerfile links

**Note:** the description for this image is longer than the Hub length limit of 25000, so the "Supported tags" list has been trimmed to compensate. The full list can be found at <https://github.com/docker-library/docs/tree/master/openjdk/README.md>. See [docker/hub-beta-feedback#238](#) for more information.

### Quick reference

- **Where to get help:**  
the Docker Community Forums, the Docker Community Slack, or Stack Overflow
- **Where to file issues:**  
<https://github.com/docker-library/openjdk/issues>
- **Maintained by:**

위 이미지에서 최신 태그를 살펴봅니다.

Description Reviews Tags

Filter Tags

Sort by Latest

latest  
Last updated 4 days ago by doijanky

docker pull openjdk:latest

DIGEST	OS/ARCH	COMPRESSED SIZE
e318b76728d5	windows/amd64	5.52 GB
399788bfd0f2	windows/amd64	2.25 GB
dc4c55fc6df4	linux/amd64	242.16 MB

사이트에서 여러 항목을 클릭하면서 익숙해지도록 합니다.

## 4.2 이미지 pull

이제 이미지와 태그에 익숙해 졌다고 생각합니다.  
아래와 같이 docker pull 명령을 통해 Java 이미지를 가져 올 수 있습니다.

```
$ docker pull openjdk
```

또는

```
$ docker pull openjdk:latest
```

## 4.3 이미지 목록 보기

docker images 명령으로 로컬에 보유한 이미지 목록을 확인할 수 있습니다.

```
$ docker images
```

## 4.4 컨테이너 시작

로컬 이미지가 존재하기에 아래의 명령을 통해 컨테이너를 시작할 수 있습니다.

```
$ docker run <imagename>:<tag>
```

## 4.5 이미지 검색

앞에서 우리는 Docker Hub 웹 인터페이스를 통해 Docker 이미지를 검색하는 방법을 살펴 보았습니다. docker 명령 줄을 통해서도 그렇게 할 수 있습니다.

```
$ docker search <검색하고자하는 이미지명>
```

예를 들어서

```
$ docker search httpd
$ docker search openjdk
$ docker search java
$ docker search mysql
```

# 5. 나만의 Docker 이미지 만들기

이 섹션에서는 이미지를 만들기 위한 첫 단계를 설명할 것입니다. Docker Hub에 존재하는 이미지 위에 소프트웨어를 추가하여 간단하게 만들어 볼 것입니다.

이미지를 만든 후에는 이미지를 Docker Hub로 Push합니다. Docker Hub에 아직 등록하지 않은 경우에는 [링크\(https://hub.docker.com/signup\)](https://hub.docker.com/signup)에서 등록하세요.

## 5.1 나만의 데이터 관리

컨테이너와 이미지의 주요 차이점을 이해하는 것이 중요합니다.

**“이미지는 불변이고 컨테이너는 임시입니다.”**

boot2docker 유틸리티를 시작한 다음 이전에 다운로드한 기본 ubuntu 이미지를 작업을 시작하겠습니다. 먼저 우분투 최신 이미지가 있는지 확인합니다.

아래의 명령을 실행하세요.

```
$ docker images
```

docker 이미지 목록중 ubuntu:latest가 있는지 확인 합니다. 없다면 아래의 명령을 실행합니다.

```
$ docker pull ubuntu:latest
```

다음 명령을 이용하여 ubuntu:latest 이미지를 기반으로 컨테이너를 시작합니다.

```
grouq:~ giljae$ docker run -it --name mycontainer1 --rm ubuntu:latest  
root@061998964b65:/#
```

이제 다음 명령을 실행하여 인기있는 Git 소프트웨어를 컨테이너 인스턴스에 설치합니다.

```
root@061998964b65:/# apt-get update  
Get:1 http://archive.ubuntu.com/ubuntu bionic InRelease [242 kB]  
Get:2 http://security.ubuntu.com/ubuntu bionic-security InRelease [88.7 kB]  
Get:3 http://security.ubuntu.com/ubuntu bionic-security/main amd64 Packages [761 kB]  
Get:4 http://archive.ubuntu.com/ubuntu bionic-updates InRelease [88.7 kB]  
Get:5 http://archive.ubuntu.com/ubuntu bionic-backports InRelease [74.6 kB]  
Get:6 http://archive.ubuntu.com/ubuntu bionic/main amd64 Packages [1344 kB]  
Get:7 http://security.ubuntu.com/ubuntu bionic-security/multiverse amd64 Packages [6781 B]  
Get:8 http://security.ubuntu.com/ubuntu bionic-security/restricted amd64 Packages [19.2 kB]  
Get:9 http://security.ubuntu.com/ubuntu bionic-security/universe amd64 Packages [795 kB]  
Get:10 http://archive.ubuntu.com/ubuntu bionic/universe amd64 Packages [11.3 MB]  
Get:11 http://archive.ubuntu.com/ubuntu bionic/restricted amd64 Packages [13.5 kB]  
Get:12 http://archive.ubuntu.com/ubuntu bionic/multiverse amd64 Packages [186 kB]  
Get:13 http://archive.ubuntu.com/ubuntu bionic-updates/restricted amd64 Packages [32.7 kB]  
Get:14 http://archive.ubuntu.com/ubuntu bionic-updates/main amd64 Packages [1057 kB]  
Get:15 http://archive.ubuntu.com/ubuntu bionic-updates/multiverse amd64 Packages [10.5 kB]  
Get:16 http://archive.ubuntu.com/ubuntu bionic-updates/universe amd64 Packages [1322 kB]  
Get:17 http://archive.ubuntu.com/ubuntu bionic-backports/main amd64 Packages [2496 B]  
Get:18 http://archive.ubuntu.com/ubuntu bionic-backports/universe amd64 Packages [4244 B]
```

```
Fetches 17.4 MB in 10s (1823 kB/s)
Reading package lists... Done
```

업데이트가 완료되었으면 아래의 명령을 통해 git을 설치합니다.

```
root@061998964b65:/# apt-get install git
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  ca-certificates git-man krb5-locales less libasn1-8-heimdal libbsd0 libcurl3-gnutls
  libedit2 liberror-perl libexpat1 libgdbm-compat4 libgdbm5 libgssapi-krb5-2
  libgssapi3-heimdal libhcrypto4-heimdal libheimbase1-heimdal
  libheimntlm0-heimdal libhx509-5-heimdal libk5crypto3 libkeyutils1
  libkrb5-26-heimdal libkrb5-3 libkrb5support0 libldap-2.4-2 libldap-common
  libnghttp2-14 libperl5.26 libpsl5 libroken18-heimdal librtmp1 libsasl2-2
  libsasl2-modules libsasl2-modules-db libsasl2-modules-gssapi-mit libsasl2-modules-gssapi-heimdal
  libsasl2-modules-ldap libsasl2-modules-otp libsasl2-modules-sql libssh2-1 libssh2-1
  libwind0-heimdal libx11-6 libx11-data libxau6 libxcb1 libxdmcp6 libxext6 libxmuu1
  multiarch-support netbase openssh-client openssl patch perl
  perl-modules-5.26 publicsuffix xauth
Suggested packages:
  gettext-base git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk
  gitweb git-cvs git-mediawiki git-svn gdbm-l10n krb5-doc krb5-user
  libsasl2-modules-gssapi-mit | libsasl2-modules-gssapi-heimdal
  libsasl2-modules-ldap libsasl2-modules-otp libsasl2-modules-sql keychain libpam-ssh
  monkeysphere ssh-askpass ed diffutils-doc perl-doc libterm-readline-gnu-perl |
  libterm-readline-perl-perl make
The following NEW packages will be installed:
  ca-certificates git git-man krb5-locales less libasn1-8-heimdal libbsd0 libcurl3-gnutls
  libedit2 liberror-perl libexpat1 libgdbm-compat4 libgdbm5 libgssapi-krb5-2
  libgssapi3-heimdal libhcrypto4-heimdal libheimbase1-heimdal
  libheimntlm0-heimdal libhx509-5-heimdal libk5crypto3 libkeyutils1
  libkrb5-26-heimdal libkrb5-3 libkrb5support0 libldap-2.4-2 libldap-common
  libnghttp2-14 libperl5.26 libpsl5 libroken18-heimdal librtmp1 libsasl2-2
  libsasl2-modules libsasl2-modules-db libsasl2-modules-gssapi-mit libsasl2-modules-gssapi-heimdal
  libsasl2-modules-ldap libsasl2-modules-otp libsasl2-modules-sql libssh2-1 libssh2-1
  libwind0-heimdal libx11-6 libx11-data libxau6 libxcb1 libxdmcp6 libxext6 libxmuu1
  multiarch-support netbase openssh-client openssl patch perl
  perl-modules-5.26 publicsuffix xauth
0 upgraded, 54 newly installed, 0 to remove and 2 not upgraded.
Need to get 18.9 MB of archives.
After this operation, 103 MB of additional disk space will be used.
Do you want to continue? [Y/n]
```

계속 진행할꺼냐는 메시지가 나옵니다. Y로 계속 진행하세요.

Git 설치가 완료되면 잘 설치되었는지 확인합니다.

```
root@061998964b65:/# git --version
git version 2.17.1
```

이제 exit를 입력하여 컨테이너를 종료합니다. 컨테이너를 시작하는 동안 --rm 플래그를 사용했기에 컨테이너는 종료시 제거됩니다.

다시 동일한 우분투 컨테이너의 다른 인스턴스를 시작합니다.

```
$ docker run -it --name mycontainer1 --rm ubuntu:late
```

프롬프트에서 git을 입력해봅니다. git을 찾을 수 없다는 메시지가 나올 것입니다.

```
grouq:~ giljae$ docker run -it --name mycontainer1 --rm ubuntu:latest
root@f86c7d2dab7c:/# git
bash: git: command not found
```

어떻게 된 것이죠? 분명 우리는 우분투에 Git을 설치하고 그것이 존재할 줄 알았습니다. 위에서 설명했듯이 이런 방식으로 시작된 각 컨테이너 인스턴스는 독립적이며 Git이 설치되지 않은 마스터 이미지(e.g. ubuntu:latest)에 따라 달라지게 됩니다.

## 5.2 이미지 commit

Git이 설치된 우분투 버전을 사용하려면 이미지를 commit해야 합니다. 이것이 의미하는 것은 우분투 이미지를 기반으로 컨테이너를 시작한다는 것입니다. 그리고 git과 같은 소프트웨어를 추가합니다. 이것은 컨테이너 상태를 수정했음을 의미합니다. 따라서 컨테이너의 현재 상태를 이미지로 저장해야 git이 반영된 컨테이너를 시작할 수 있습니다.

이전 섹션에서 사용한 컨테이너를 exit로 종료하고 아래의 단계를 수행합니다.

이번에는 --rm flag없이 우분투 기반의 컨테이너를 시작합니다.

```
rouq:~ giljae$ docker run -it --name mycontainer1 ubuntu:latest
root@767b7e13efa8:/#
```

아래 명령을 통해 패키지를 업데이트하고 Git을 설치합니다.

```
root@767b7e13efa8:/# apt-get update
root@767b7e13efa8:/# apt-get install git
```

설치가 완료되면 아래의 명령을 통해 Git이 설치되어 있는지 확인합니다.

```
root@767b7e13efa8:/# git --version
git version 2.17.1
```

exit를 입력하여 컨테이너를 종료합니다.

아래의 명령어를 입력하여 mycontainer1을 확인합니다.

```
grouq:~ giljae$ docker ps -all
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
767b7e13efa8	ubuntu:latest	"/bin/bash"	4 minutes ago	Exited (0) 45 seconds ago
	mycontainer1			

mycontainer1은 현재 종료 상태입니다.

docker commit 명령을 통해 컨테이너 이미지를 커밋합니다.

사용법:

```
docker commit [ContainerID] [Repository [:Tag]]
```

이 경우 mycontainer1에 이름을 부여했기에 컨테이너 아이디 대신 사용할 수 있습니다. 해당 컨테이너에 이름을 부여하지 않은 경우에는 docker ps -all 명령에 표시된 컨테이너 ID를 사용해야 합니다.

Repository의 이름이 중요합니다. 마지막에는 Docker Hub로 Push할 계획이기에 권장되는 저장소 이름의 형식은 아래와 같습니다.

```
<dockerhub_username>/<repository_name>
```

이번 예제의 경우 repository\_name은 ubuntu-git으로 할 예정이기에 <dockerhub\_username>의 값을 자신의 사용자 이름으로 대체해야 합니다.

태그를 제공하지 않으면 "latest"로 표시됩니다. 저의 경우 dockerhub\_username이 giljae이기에 아래와 같이 commit합니다.

```
grouq:~ giljae$ docker commit mycontainer1 giljae/ubuntu-git
sha256:bf60f2de446a91f1a8345f6131c75603994dd53f7146f965fa7fed3a9c370ca0
```

이렇게 하면 이미지 ID가 다시 생성됩니다. 이제 Docker 이미지 목록을 확인해봅시다.

```
grouq:~ giljae$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
giljae/ubuntu-git	latest	bf60f2de446a	27 seconds ago	186MB
httpd	latest	2ae34abc2ed0	2 weeks ago	165MB
ubuntu	latest	775349758637	6 weeks ago	64.2MB
alpine	latest	965ea09ff2eb	7 weeks ago	5.55MB

목록 맨 위에 방금 commit한 이미지가 존재하는 것을 확인할 수 있습니다. 이는 리포지토리 giljae/ubuntu-git이 생성되었음을 의미합니다.



## 5.3 이미지에서 컨테이너 시작

이제 새로 생성된 docker 이미지(e.g. yourusername/ubuntu-git)에서 컨테이너를 시작할 수 있습니다.

```
grouq:~ giljae$ docker run -it --name myc1 giljae/ubuntu-git
root@179c15350d05:/#
```

위의 명령은 새로 생성한 이미지를 기반으로 컨테이너를 시작합니다. 그리고 git --version 명령어로 Git이 설치되어 있는지 확인해봅시다.

```
root@179c15350d05:/# git --version
git version 2.17.1
```

Git이 설치되어 있는 것을 확인할 수 있습니다.

## 5.4 Docker Hub로 이미지 Push 하기

Docker Hub로 새롭게 생성한 이미지를 Push하려면 아래의 단계를 거쳐야 합니다.

1. Docker Hub에서 계정을 만들어야 합니다.
2. docker login에 성공하면 Login Succeeded 메시지가 표시됩니다.

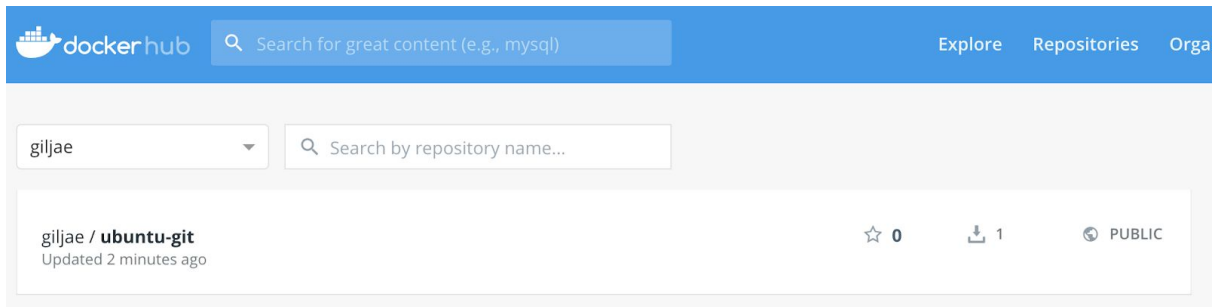
```
grouq:~ giljae$ docker login
Authenticating with existing credentials...
Login Succeeded
```

3. 아래의 명령어를 이용하여 이미지를 Docker Hub로 Push 합니다.

관련 파일을 업로드하는데 시간이 조금 걸립니다. Docker는 기본 이미지(e.g. ubuntu)가 존재하는지 여부를 감지하기에 업로드 해야하는 레이어만 똑똑하게 선택합니다. 프로세스는 다음과 같이 실행됩니다.

```
grouq:~ giljae$ docker push giljae/ubuntu-git
The push refers to repository [docker.io/giljae/ubuntu-git]
1bbb9d10563e: Pushed
e0b3afb09dc3: Mounted from library/ubuntu
6c01b5a53aac: Mounted from library/ubuntu
2c6ac8e5063e: Mounted from library/ubuntu
cc967c529ced: Mounted from library/ubuntu
latest: digest:
sha256:9a42aa82ee7b562ff51abb125a007266068042fc8f02b61e1aab16e3844930f
9 size: 1364
```

새로 만든 이미지가 Push 되었습니다. Docker Hub 웹사이트에서 저장소를 확인해봅시다.



잘 올라간것을 확인할 수 있습니다. docker search 명령을 통해 리포지토리를 검색해봅시다.

#### 연습 :

기본 이미지 위에 소프트웨어 설치하여 나만의 이미지를 만들고 Docker Hub에 Push 해봅니다.

## 6. Docker 전용 Registry

이번 섹션에서는 로컬 Docker Registry를 호스팅 할 수 있는 방법에 대해서 알아보도록 하겠습니다. 이전 섹션에서는 Docker가 호스팅하는 공개 레지스트리인 Docker Hub에 대해서 살펴보았습니다. Docker Hub는 Docker 이미지에 대해서 공개적으로 활용하는데 중요한 역할을 하지만 회사 내부의 팀 및 조직에서 사용하려면 Private Registry를 설정해야 합니다.

### 6.1 Registry 이미지를 다운 받기

Docker Hub에서 Registry를 검색합니다.

```
grouq:~ giljae$ docker search registry
```

NAME	DESCRIPTION	STARS	OFFICIAL
AUTOMATED			
registry	The Docker Registry 2.0 implementation for s...	2795	[OK]
distribution/registry	WARNING: NOT the registry official image!!! ...	58	
[OK]			
stefanscherrer/registry-windows	Containerized docker registry for Windows Se...	27	
budry/registry-arm	Docker registry build for Raspberry PI 2 and...	18	
deis/registry	Docker image registry for the Deis open sour...	12	
sixeyed/registry	Docker Registry 2.6.0 running on Windows - N...	9	
anoxis/registry-cli	You can list and delete tags from your priva...	8	[OK]
vmware/registry		6	
jc21/registry-ui	A nice web interface for managing your Docke...	5	
allingeek/registry	A specialization of registry:2 configured fo...	4	[OK]
pallet/registry-swift	Add swift storage support to the official do...	4	[OK]
goharbor/registry-photon		2	
ibmcom/registry	Docker Image for IBM Cloud private-CE (Commu...	1	
webhippie/registry	Docker images for Registry	1	[OK]
metadata/registry	Metadata Registry is a tool which helps you ...	1	
[OK]			
conjurinc/registry-oauth-server	Docker registry authn/authz server backed by...	1	

```

upmcenterprises/registry-creds          0
ghmlee/registrybot          registrybot          0          [OK]
lorieri/registry-ceph          Ceph Rados Gateway (and any other S3 compati... 0
kontena/registry          Kontena Registry          0
dwpdigital/registry-image-resource  Concourse resource type          0
gisjedi/registry-proxy          Reverse proxy of registry mirror image gisje... 0
concourse/registry-image-resource          0
convox/registry          0
zoined/registry          Private Docker registry based on registry:2 0

```

우리는 registry를 가져올 것입니다. 아래의 명령어로 registry를 가져옵니다.

```

grouq:~ giljae$ docker pull registry
Using default tag: latest
latest: Pulling from library/registry
c87736221ed0: Pull complete
1cc8e0bb44df: Pull complete
54d33bcb37f5: Pull complete
e8afc091c171: Pull complete
b4541f6d3db6: Pull complete
Digest:
sha256:8004747f1e8cd820a148fb7499d71a76d45ff66bac6a29129bdfbdc0154d14
6
Status: Downloaded newer image for registry:latest
docker.io/library/registry:latest

```

docker images 명령어로 확인합니다.

```

grouq:~ giljae$ docker images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
giljae/ubuntu-git  latest      bf60f2de446a     28 minutes ago  186MB
httpd               latest      2ae34abc2ed0     2 weeks ago     165MB
ubuntu              latest      775349758637     6 weeks ago     64.2MB
alpine              latest      965ea09ff2eb     7 weeks ago     5.55MB
registry            latest      f32a97de94e1     9 months ago    25.8MB

```

설치 된것을 확인할 수 있습니다.

## 6.2 로컬 Registry 실행 하기

Registry Docker 이미지는 컨테이너의 포트 5000에서 시작하도록 구성되어 있습니다. 따라서 호스트 포트도 5000으로 매핑합니다.

다음 명령어를 통해 Registry를 시작할 수 있습니다.

```

grouq:~ giljae$ docker run -d -p 5000:5000 --name local_registry registry
079525e3ca9d182376a829c27127fb23dcc80d19448ee38e62c7f31504851bd3

```

성공적으로 실행되면 console에 컨테이너 ID가 출력됩니다.

Registry 이미지를 기반으로 local\_registry라는 컨테이너를 생성했습니다. 컨테이너가 분리 모드로 시작되고 포트도 5000:5000 으로 매핑했습니다.

docker ps 명령어로 "local\_registry" 컨테이너가 시작되었는지 확인해봅시다.

```
grouq:~ giljae$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS
PORTS         NAMES
079525e3ca9d   registry      "/entrypoint.sh /etc..." 2 minutes ago  Up 2
minutes       0.0.0.0:5000->5000/tcp   local_registry
```

## 6.3 로컬 Registry로 Push 하기

이제 몇 개의 이미지를 가지고 로컬 Registry로 Push 해보도록 하겠습니다.

### 6.3.1 busybox 및 alpine 리눅스 이미지 가져오기

아래와 같이 두 이미지에 대해 pull 명령을 실행합니다.

#busybox

```
grouq:~ giljae$ docker pull busybox
Using default tag: latest
latest: Pulling from library/busybox
322973677ef5: Pull complete
Digest:
sha256:1828edd60c5efd34b2bf5dd3282ec0cc04d47b2ff9caa0b6d4f07a21d1c0808
4
Status: Downloaded newer image for busybox:latest
docker.io/library/busybox:latest
```

#alpine 리눅스

```
grouq:~ giljae$ docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine
Digest:
sha256:c19173c5ada610a5989151111163d28a67368362762534d8a8121ce95cf2b
d5a
Status: Image is up to date for alpine:latest
docker.io/library/alpine:latest
```

이미지가 다운로드되면 docker images 명령어로 목록에 존재하는지 확인하세요.

```
grouq:~ giljae$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
giljae/ubuntu-git	latest	bf60f2de446a	39 minutes ago	186MB
busybox	latest	b534869c81f0	12 days ago	1.22MB
httpd	latest	2ae34abc2ed0	2 weeks ago	165MB
ubuntu	latest	775349758637	6 weeks ago	64.2MB
alpine	latest	965ea09ff2eb	7 weeks ago	5.55MB
registry	latest	f32a97de94e1	9 months ago	25.8MB

이제 로컬 Registry에서 alpine 리눅스를 가져와봅시다.

```
grouq:~ giljae$ docker pull localhost:5000/alpine
Using default tag: latest
Error response from daemon: manifest for localhost:5000/alpine:latest not found:
manifest unknown: manifest unknown
```

로컬 Registry에 아무런 작업을 하지 않았기 때문에 alpine 리눅스를 가져 오려고 하면 이미지를 찾을 수 가 없다고 나옵니다. 우리는 이미 5000 포트에서 Registry 컨테이너를 시작했기 때문에 아래와 같이 작업을 해야 합니다.

특정 Registry에서 이미지를 가져오는 형식은 다음과 같습니다.

```
[REGISTRY_HOSTNAME : REGISTRY_PORT] / IMAGENAME
```

Docker Hub의 경우 [REGISTRY\_HOSTNAME : REGISTRY\_PORT] 옵션을 지정하지 않았습니니다. 그러나 로컬 Registry의 경우 Docker 클라이언트가 볼 수 있도록 지정해야 합니다.

### 6.3.2 busybox 및 alpine 리눅스를 로컬 Registry에 Push 하기

이제 다운로드 한 두 개의 이미지(busybox 및 alpine)를 로컬 Registry에 Push합니다. Docker Hub에서 직접 다운로드 한 두개의 이미지입니다.

우리는 이 이미지를 수정하지 않았습니니다. 그러나 앞에서 설명한 것처럼 수정해서 로컬 Registry로 Push 할 수도 있습니다.

이미지를 로컬 Registry로 Push하는 단계는 다음과 같이 수행됩니다.

첫 번째 단계는 이미지 또는 컨테이너를 가져와야 합니다. 이미 가져온 alpine 리눅스 이미지로 작업을 하도록 하겠습니다.

아래의 명령을 실행하여 alpine:latest 이미지에 Push할 로컬 Registry의 Tag를 지정하세요.

```
grouq:~ giljae$ docker tag alpine:latest localhost:5000/alpine:latest
```

docker images 명령을 실행하면 alpine 및 localhost:5000/alpine 이미지가 모두 표시됩니다.

```
grouq:~ giljae$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
giljae/ubuntu-git	latest	bf60f2de446a	26 hours ago	186MB
busybox	latest	b534869c81f0	13 days ago	1.22MB
httpd	latest	2ae34abc2ed0	2 weeks ago	165MB
ubuntu	latest	775349758637	6 weeks ago	64.2MB
alpine	latest	965ea09ff2eb	8 weeks ago	5.55MB
localhost:5000/alpine	latest	965ea09ff2eb	8 weeks ago	5.55MB
registry	latest	f32a97de94e1	9 months ago	25.8MB

Tag가 지정된 이미지 또는 컨테이너를 로컬 레지스트리에 Push 합니다.  
 이것은 앞에서 배운 `docker push` 명령어를 이용합니다. Tag가 지정된 `localhost:5000/alpine` 이미지만 사용하면 됩니다. 아래의 명령을 실행합니다.

```
grouq:~ giljae$ docker push localhost:5000/alpine:latest
The push refers to repository [localhost:5000/alpine]
77cae8ab23bf: Pushed
latest: digest:
sha256:e4355b66995c96b4b468159fc5c7e3540fcef961189ca13fee877798649f531
a size: 528
```

이제까지 개인 Registry를 사용하는 방법을 알아보았습니다. 이것은 클라이언트 측에서 인증을 시행하지 않는 Registry입니다. 하지만 팀/조직내에서 사용하려면 보안 Registry를 사용해야 하는 점을 명심해야 합니다.

## 7. Data Volumes

본 섹션에서는 Docker Volumes에 대해서 알아보도록 하겠습니다. Docker Volumes는 Docker 컨테이너 내에서 데이터를 관리하는 방법입니다.

지금까지는 다양한 이미지에서 컨테이너를 생성했습니다. 컨테이너는 일시적이고 컨테이너가 제거되면 사라집니다. 컨테이너 내부에서 실행되는 애플리케이션에서 사용한 데이터가 유지되려면 어떻게 해야 할까요? 예를 들어서 데이터를 생성하는 애플리케이션이 있고 파일을 작성하거나 데이터베이스에 저장하는 작업이 존재할 때, 컨테이너가 제거되어 나중에 다시 컨테이너를 시작하더라도 해당 데이터가 여전히 존재하게 하려면 어떻게 해야 할까요?

다시 말해서, 컨테이너 수명주기를 데이터와 분리하는 작업이 필요합니다. 생성 된 데이터가 컨테이너 수명주기에 의해 손상되거나 재사용이 안되는 현상을 Volumes을 통해 해결할 계획입니다.

Docker 공식 문서에서 언급하는 데이터를 관리 할 수 있는 방법은 두가지가 있습니다.

- Data Volumes
- Data volume containers

위의 두 경우에 대한 예를 살펴보도록 하겠습니다.

## 데이터 볼륨에 대해 명심해야 할 사항

- 데이터 볼륨은 컨테이너를 위해 특별히 설계된 디렉토리입니다.
- 컨테이너가 생성되면 초기화 됩니다. 기본적으로 컨테이너가 중지되어도 삭제되지 않습니다. 볼륨을 참조하는 컨테이너가 없으면 Garbage 수집도 되지 않습니다.
- 데이터 볼륨은 독립적으로 업데이트 됩니다. 컨테이너 간 데이터 볼륨 공유도 가능 합니다. 읽기 전용 모드로 mount 할 수도 있습니다.

## 7.1 데이터 볼륨 마운트

컨테이너에 데이터 볼륨을 마운트하는 가장 기본적인 작업부터 해보도록 하겠습니다. busybox 이미지를 이용하여 작업을 합니다.

컨테이너의 볼륨을 마운트하는 옵션으로 -v [/VolumeName]을 사용합니다. 아래와 같이 컨테이너를 시작하도록 합니다.

```
grouq:~ giljae$ docker run -it -v /data --name container1 busybox
```

container1이 시작되고 프롬프트가 표시됩니다. 프롬프트에서 아래와 같이 ls 명령을 실행합니다.

```
/ # ls
bin  data dev  etc  home  proc  root  sys  tmp  usr  var
```

data라는 이름의 볼륨이 표시됩니다. touch 명령어를 이용하여 file.txt 라는 파일을 생성합니다.

```
/ # cd data
/data # touch file.txt
/data # ls
file.txt
```

지금까지 한 일은 컨테이너에 볼륨 /data를 마운트하였고, 해당 디렉토리(/data)내에 파일을 생성했습니다.

이제 exit를 입력하고 터미널로 돌아가 컨테이너를 종료합니다.

```
/data # exit
grouq:~ giljae$
```

docker ps -a 를 이용하여 container1이 종료 상태인지 확인합니다.

```
grouq:~ giljae$ docker ps -a | grep container1
7882dab8ba00      busybox          "sh"              5 minutes ago    Exited (0)
About a minute ago      container1
```

이제 컨테이너를 시작할 때 docker가 수행하는 작업을 살펴보도록 하겠습니다.

```
grouq:~ giljae$ docker inspect container1
```

JSON 포맷으로 제공되고 Mounts 속성을 찾아서 살펴 봅니다. 아래는 Mounts 속성 정보입니다.

```
"Mounts": [
  {
    "Type": "volume",
    "Name":
    "bf564523eed84a97674ac033f87404c5ba3c9c25221d582bdf5e00480e05370d",
    "Source":
    "/var/lib/docker/volumes/bf564523eed84a97674ac033f87404c5ba3c9c25221d582bdf5e00480e05370d/_data",
    "Destination": "/data",
    "Driver": "local",
    "Mode": "",
    "RW": true,
    "Propagation": ""
  }
],
```

볼륨(/data)을 마운트 할때 /var/lib/... 폴더를 생성 했음을 의미합니다. 이곳에 볼륨에서 만든 모든 파일이 저장됩니다. 이전에 우리는 file.txt 파일을 만들었습니다.

또한 RW 모드가 true로 설정되어 있습니다. (e.g. 읽기 및 쓰기)

컨테이너(container1)를 다시 시작하고 file.txt가 있는지 확인해봅시다.

```
grouq:~ giljae$ docker restart container1
container1
grouq:~ giljae$ docker attach container1
/ # ls
bin data dev etc home proc root sys tmp usr var
/ # cd data
/data # ls
file.txt
```

위에서 실행한 단계는 아래와 같습니다.

1. 컨테이너(container1)를 다시 시작했습니다.
2. 실행중인 컨테이너(container1)에 연결 했습니다.
3. ls를 실행하여 볼륨(/data)가 존재하는지 확인했습니다.
4. /data 디렉토리로 이동하여 file.txt가 있는지 확인했습니다.

데이터 볼륨을 유지하는 방법에 대해서 알게 되었습니다. 이제 컨테이너를 종료하데 컨테이너를 제거해보도록 합시다.

```
/data # exit
```



```
grouq:~ giljae$ docker rm container1
container1
grouq:~ giljae$ docker volume ls
DRIVER          VOLUME NAME
local
6f7b3519d129a3333d50eda99e9dcc07f502930838b166b096c7031f376a1dd3
```

container1을 제거했지만 데이터 볼륨이 존재함을 보여줍니다. 이것은 ghost volume입니다. 아래의 도움말을 보면 컨테이너를 제거할때 볼륨도 제거하는 -v 옵션이 존재합니다.

```
grouq:~ giljae$ docker rm --help

Usage: docker rm [OPTIONS] CONTAINER [CONTAINER...]

Remove one or more containers

Options:
  -f, --force      Force the removal of a running container (uses SIGKILL)
  -l, --link       Remove the specified link
  -v, --volumes    Remove the volumes associated with the container
```

#### 연습:

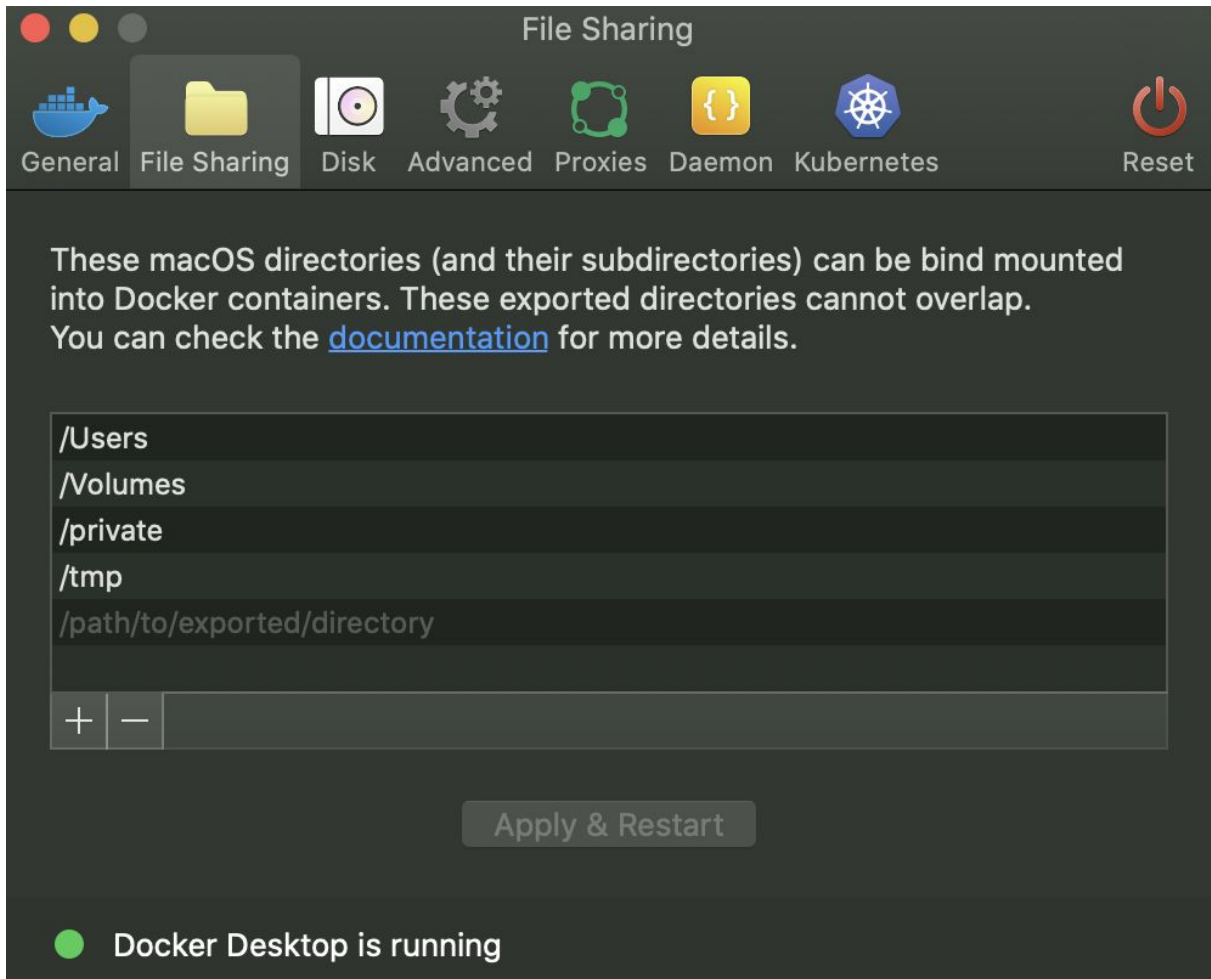
동일한 /data 볼륨으로 다른 컨테이너를 시작하면 어떻게 될까요? 파일이 여전히 존재할까요? 아니면 각 컨테이너에 고유한 파일 시스템이 있을까요? 확인해보세요.

## 7.2 호스트 Volume을 데이터 Volume으로 마운트

컨테이너에 볼륨을 마운트하는 방법을 위에서 살펴보았습니다. 다음 단계는 볼륨 마운트와 동일한 방식의 프로세스를 살펴 보는 것입니다.

Docker 컨테이너에 기존 호스트 폴더를 마운트합니다. 이것은 흥미로운 개념입니다. 외부 폴더에 존재하는 파일을 수정하고 컨테이너와 다른 컨테이너간 볼륨을 공유할 경우에 매우 유용합니다.

저는 Docker for Mac을 사용하고 있기에, 이를 기반으로 설명할 예정입니다. 메뉴바의 Docker 아이콘을 클릭하여 Preferences를 들어가면 아래의 화면을 볼 수 있습니다.



Docker 컨테이너를 시작하는 동안 호스트 볼륨을 마운트하려면 `volume -v` 옵션을 사용해야 합니다.

```
-v host_folder : container_volumename
```

호스트 볼륨으로 사용할 폴더를 생성합니다.

```
grouq:~ giljae$ mkdir container_volume
grouq:~ giljae$ cd container_volume/
grouq:container_volume giljae$ pwd
/Users/giljae/container_volume
```

그리고 busybox 컨테이너를 시작하도록 하겠습니다.

```
grouq:container_volume giljae$ docker run -it --name container1 -v
/Users/giljae/container_volume:/datavol busybox
/ #
```

위의 명령어는 호스트 폴더 `/Users/giljae/container_volume` 을 컨테이너(container1) 내부에 마운트 될 볼륨 `/datavol`에 매핑한 것입니다.

ls 명령어를 실행하여 /datavol이 마운트되어 있는지 확인합니다.

```
/ # ls
bin    dev    home   root   tmp    var
datavol etc    proc   sys    usr
```

/datavol로 이동해서 폴더 내용을 확인합니다.

```
/ # cd datavol
/datavol # ls
/datavol #
```

폴더내에 파일이 존재하지 않습니다. 호스트 폴더에서 파일을 하나 생성해보도록 합시다.

```
grouq:~ giljae$ cd container_volume/
grouq:container_volume giljae$ touch file.txt
grouq:container_volume giljae$ ls
file.txt
grouq:container_volume giljae$
```

컨테이너(container1)에 매핑된 /datavol 폴더에 생성한 파일이 존재하는지 확인합니다.

```
/datavol # ls
file.txt
/datavol #
```

호스트 폴더에서 생성한 file.txt가 존재하는 것을 확인할 수 있습니다.

#### 연습:

1. 호스트 폴더 (/Users/<username>/container\_volume)에 직접 파일을 추가 한 다음 실행중인 컨테이너에서 파일이 보이는지 확인하세요.
2. 컨테이너의 shell에서 /datavol 폴더로 이동한 다음 파일을 추가하세요. 그리고 호스트 폴더에서 추가한 파일이 보이는지 확인하세요.
3. 데이터 볼륨으로 마운트 된 호스트 폴더를 어떻게 사용할지 고민해보세요.

#### 참고:

- 데이터 볼륨으로 마운트 된 호스트 폴더를 활용하는 방법 중 하나는 예를 들어서 프로젝트를 수행중이고 컨테이너에서 Apache Web Server를 실행한다고 가정합니다.
- 컨테이너를 시작하고 웹 서버가 사용할 수 있는 호스트 폴더를 마운트할 수 있습니다. 이 경우 호스트 컴퓨터에서 해당 폴더의 파일들을 변경하면 Docker 컨테이너에 바로 반영이 됩니다.

## 7.3 데이터 볼륨 컨테이너

이제 데이터 볼륨 컨테이너를 만들어보도록 합시다. 컨테이너간에 데이터를 공유하거나 혹은 임시 컨테이너의 데이터를 사용하려는 케이스가 존재할 때, 이는 매우 유용합니다. 아래의 단계를 살펴봅시다.

1. 우선 데이터 볼륨 컨테이너를 생성합니다.
2. 다른 컨테이너를 생성하고 1단계에서 생성된 컨테이너에서 볼륨을 마운트합니다.

실제 실행해볼까요?

먼저 컨테이너(container1)를 만들고 볼륨을 마운트합니다.

```
grouq:~ giljae$ docker run -it -v /data --name container1 busybox
```

/data 폴더로 들어가서 두개의 더미 파일을 생성합니다.

```
/ # cd data
/data # touch file1.txt
/data # touch file2.txt
```

이제 다른 터미널을 열어서 docker ps로 실행중인 컨테이너를 확인합니다.

```
grouq:~ giljae$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS
PORTS         NAMES
8037b0447dba   busybox    "sh"                    3 minutes ago Up 3 minutes
container1
grouq:~ giljae$
```

실행중인 container1에서 명령어를 실행하여 /data 폴더의 파일을 확인합니다.

```
grouq:~ giljae$ docker exec container1 ls /data
file1.txt
file2.txt
```

같은 컨테이너에서 /data 볼륨을 확인했습니다. 이제 다른 컨테이너(container2)를 생성하면서 container1의 /data 볼륨을 마운트 합니다.

```
grouq:~ giljae$ docker run -it --volumes-from container1 --name container2 busybox
```

이제 ls 명령어를 실행하여 /data 폴더내의 파일을 확인합니다.

```
/ # ls /data
file1.txt file2.txt
/ #
```

container1, container2에서 동일한 데이터 볼륨을 사용했고 이를 확인했습니다.

더 자세한 사항은 데이터 볼륨에 대한 [공식 문서](#)를 확인 하시길 바랍니다.

## 8. 컨테이너 연결하기

본 섹션에서는 Docker 컨테이너를 연결하는 방법에 대해서 알아볼 것입니다. 컨테이너를 연결하면 Docker 컨테이너가 상호간 통신 할 수 있습니다.

샘플 웹 애플리케이션을 생각해봅시다. 웹서버와 데이터베이스 서버가 있을 수 있습니다. Docker 컨테이너간 연결하는 방법에 대해서 아래의 이야기를 할 것입니다.

1. 데이터베이스 서버를 실행할 Docker 컨테이너를 생성합니다.
2. 1단계에서 생성한 컨테이너에 대한 link flag를 기반으로 두 번째 컨테이너(웹 서버)를 생성합니다. 이렇게하면 링크 이름을 통해 데이터베이스 서버와 통신이 가능합니다.

이미 앞에서 소개한 네트워킹 포트를 사용하지 않고 컨테이너를 서로 연결하는 방법을 본 섹션에서 배울 예정입니다.

Docker Compose가 권장되는 방법이지만, 이번장에서는 link flag에 대해서만 언급하도록 하겠습니다.

아래의 명령어를 통해 Redis 이미지를 가져옵니다.

```
grouq:~ giljae$ docker pull redis
Using default tag: latest
latest: Pulling from library/redis
000eee12ec04: Already exists
5cc53381c195: Pull complete
48bb7bcb5fbf: Pull complete
ef8a890bb1c2: Pull complete
32ada9c6fb0d: Pull complete
76e034b0f296: Pull complete
Digest:
sha256:1eedfc017b0cd3e232878ce38bd9328518219802a8ef37fe34f58dcf591688ef
Status: Downloaded newer image for redis:latest
docker.io/library/redis:latest
```

그리고 redis 컨테이너(redis1)를 시작합니다.

```
grouq:~ giljae$ docker run -d --name redis1 redis
e7352a25969b91009a058cbe20574405c1128a131a420db930c19ccbaa52bd2e
```

docker ps 명령어로 redis 컨테이너가 생성되었는지 확인합니다.

```
grouq:~ giljae$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
e7352a25969b	redis	"docker-entrypoint.s..."	41 seconds ago	Up 41
seconds	6379/tcp	redis1		

포트 6379에서 시작된 것을 확인할 수 있습니다.

이제 busybox 컨테이너를 시작하도록 합니다.

```
grouq:~ giljae$ docker run -it --link redis1:redis --name redisclient1 busybox / #
```

새로운 flag가 보입니다. --link 입니다.

```
--link <source_container_name>:<container_alias_name>
```

우리는 <source\_container\_name>을 redis1으로 입력했습니다. 그 이유는 이전에 시작한 redis 컨테이너의 이름입니다. <container\_alias\_name>은 redis로 부여했습니다.

위에서 시작한 redisclient1 컨테이너의 /etc/hosts 파일을 확인해볼까요?

```
/ # cat /etc/hosts
127.0.0.1    localhost
::1         localhost ip6-localhost ip6-loopback
fe00::0     ip6-localnet
ff00::0     ip6-mcastprefix
ff02::1     ip6-allnodes
ff02::2     ip6-allrouters
172.17.0.2   redis e7352a25969b redis1
172.17.0.3   191153dd128d
```

위의 파일을 보시면 컨테이너 redis1이 redis 이름과 연결되어 있습니다. 이제 호스트 이름, alias로 핑을 수행해봅시다.

```
/ # ping redis
PING redis (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.151 ms
64 bytes from 172.17.0.2: seq=1 ttl=64 time=0.131 ms
64 bytes from 172.17.0.2: seq=2 ttl=64 time=0.133 ms
```

ping이 잘되는 것을 확인할 수 있습니다. 서로간 통신이 연결된 것입니다.

환경 변수도 살펴볼까요?

```
/ # env
REDIS_PORT=tcp://172.17.0.2:6379
REDIS_PORT_6379_TCP_ADDR=172.17.0.2
```

```
HOSTNAME=57855759487f
REDIS_NAME=/redisclient1/redis
SHLVL=1
REDIS_PORT_6379_TCP_PORT=6379
HOME=/root
REDIS_PORT_6379_TCP_PROTO=tcp
REDIS_ENV_REDIS_DOWNLOAD_URL=http://download.redis.io/releases/redis-5.0.7.tar.gz
REDIS_ENV_REDIS_VERSION=5.0.7
REDIS_PORT_6379_TCP=tcp://172.17.0.2:6379
TERM=xterm
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
REDIS_ENV_REDIS_DOWNLOAD_SHA=61db74eabf6801f057fd24b590232f2f337d422280fd19486eca03be87d3a82b
REDIS_ENV_GOSU_VERSION=1.11
PWD=/
```

다양한 환경 변수가 자동 생성되어 있는 것을 확인할 수 있습니다.

redis1 컨테이너를 종료하고 터미널로 돌아갑니다.

redis 이미지를 기반으로 컨테이너를 시작합니다.

```
grouq:~ giljae$ docker run -it --link redis1:redis --name client1 redis sh
```

그리고 redis 클라이언트(redis-cli)를 실행하여 다른 컨테이너에서 실행되고 있는 다른 redis 서버에 연결합니다.

```
# redis-cli -h redis
redis:6379>
```

연결 된 것을 확인할 수 있습니다. 표준 redis 명령을 실행해봅니다.

```
redis:6379> PING
PONG
redis:6379> set myvar DOCKER
OK
redis:6379> get myvar
"DOCKER"
redis:6379>
```

정상적으로 작동합니다. 이제 해당 컨테이너를 종료하고 다른 클라이언트(client2)를 시작합니다.

```
grouq:~ giljae$ docker run -it --link redis1:redis --name client2 redis sh
```

몇 가지 명령을 실행하여 연결이 잘 되었는지 확인합니다.

```
# redis-cli -h redis
redis:6379> get myvar
"DOCKER"
redis:6379>
```

지금까지 각 컨테이너를 서로 연결하는 방법을 배워보았습니다.

#### 추가 자료

컨테이너와 링크를 단일 파일로 지정하여 연결하는 매커니즘을 제공하는 [Docker Compose](#)에 대해서도 확인해보시길 바랍니다.

## 9. Dockerfile 작성하기

본 섹션에서는 Dockerfile을 통해 자체적으로 Docker 이미지를 만드는 방법에 대해서 배워보도록 하겠습니다. 컨테이너를 실행하고, 소프트웨어를 설치하고 이미지를 만들기 위해 commit을 수행하여 이미지를 만들어볼 계획입니다.

Dockerfile은 이미지를 작성하는 방법에 대한 지침을 작성한 텍스트 파일입니다. Dockerfile에서 지원하는 문법의 일부를 이번 섹션에서 배우게 될 것입니다.

이미지를 만드는 방법에 대해서 기술합니다.

1. Dockerfile을 작성합니다.
2. docker build 명령을 이용하여 1단계에서 만든 Dockerfile을 기반으로 Docker 이미지를 만듭니다.



#### 기본 명령

- FROM - 빌드 프로세스를 사용하고 시작할 기본 이미지를 정의합니다.
- RUN - 이미지에서 명령을 실행하려면 명령과 인수가 필요합니다.
- CMD - RUN 명령과 유사한 기능이지만 컨테이너가 인스턴스화 된 후에 실행됩니다.
- ENTRYPOINT - 컨테이너가 생성 될 때 이미지의 기본 어플리케이션을 대상으로 합니다.
- ADD - 파일을 소스에서 컨테이너 내부로 복사합니다.
- ENV - 환경 변수를 설정합니다.

로컬 홈디렉토리에 docker\_image라는 폴더를 생성합니다.



```
grouq:~ giljae$ mkdir docker_image
```

cd 명령을 통해 해당 디렉토리로 이동합니다.

```
grouq:~ giljae$ cd docker_image
```

vi 혹은 사용하는 편집기를 열어서 Dockerfile을 만듭니다.

```
FROM busybox:latest
MAINTAINER Giljae Joo
```

Docker 이미지는 서로 위에 쌓여진 Layer일 뿐이기에 기본 이미지부터 시작하는 것이 좋습니다. FROM은 기본 이미지를 설정하는 명령어입니다. MAINTAINER는 생성된 이미지의 작성자를 알려줍니다. FROM 명령어에서 다른 기본 이미지(e.g. ubuntu:latest)도 사용할 수 있습니다.

파일을 저장하고 터미널로 돌아갑니다. /image 디렉토리에서 아래의 명령을 실행합니다.

```
grouq:docker_image giljae$ docker build -t myimage:latest .
Sending build context to Docker daemon 2.048kB
Step 1/2 : FROM busybox:latest
--> b534869c81f0
Step 2/2 : MAINTAINER Giljae Joo
--> Running in e52ae914f0d1
Removing intermediate container e52ae914f0d1
--> c3825f344149
Successfully built c3825f344149
Successfully tagged myimage:latest
```

docker build 명령을 실행했습니다. 이 명령은 Docker 이미지를 빌드할때 사용됩니다. -t 옵션은 Docker 이미지 태그입니다. 이미지 이름과 태그를 지정할 수 있습니다. “.”은 Dockerfile의 위치를 지정합니다. Docker 빌드를 실행하는 디렉토리내에 Dockerfile이 존재하기에 현재 디렉토리를 의미하는 “.”으로 지정했습니다.

docker images 명령을 실행하여 myimage 이미지가 존재하는지 확인해봅시다.

```
grouq:docker_image giljae$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
myimage	latest	c3825f344149	About a minute ago	1.22MB
giljae/ubuntu-git	latest	bf60f2de446a	3 days ago	186MB
busybox	latest	b534869c81f0	2 weeks ago	1.22MB
httpd	latest	2ae34abc2ed0	3 weeks ago	165MB
redis	latest	dcf9ec9265e0	3 weeks ago	98.2MB
ubuntu	latest	775349758637	7 weeks ago	64.2MB
alpine	latest	965ea09ff2eb	8 weeks ago	5.55MB
localhost:5000/alpine	latest	965ea09ff2eb	8 weeks ago	5.55MB
registry	latest	f32a97de94e1	9 months ago	25.8MB

docker run 명령을 이용하여 컨테이너를 시작합니다.

```
grouq:docker_image giljae$ docker run -it myimage  
/ #
```

exit로 셸을 빠져나와서 Dockerfile을 수정하도록 합니다. 에디터로 Dockerfile을 열고 CMD 명령어 라인을 추가합니다.

```
FROM busybox:latest  
MAINTAINER Giljae Joo  
CMD ["date"]
```

다시 빌드하고 컨테이너를 시작합니다.

```
grouq:docker_image giljae$ docker build -t myimage:latest .  
Sending build context to Docker daemon 2.048kB  
Step 1/3 : FROM busybox:latest  
---> b534869c81f0  
Step 2/3 : MAINTAINER Giljae Joo  
---> Using cache  
---> c3825f344149  
Step 3/3 : CMD ["date"]  
---> Running in 45db681e73af  
Removing intermediate container 45db681e73af  
---> cfd5ea2fd982  
Successfully built cfd5ea2fd982  
Successfully tagged myimage:latest  
grouq:docker_image giljae$ docker run -it myimage  
Fri Dec 20 04:20:08 UTC 2019  
grouq:docker_image giljae$
```

셸에 접속하면 날짜가 출력되는 것을 확인할 수 있습니다.

#### 연습:

CMD 명령을 수정해봅시다. CMD ["ls","-al"]와 같은 다른 명령어로 수정하고 이미지를 다시 빌드하고 컨테이너를 시작해보세요.

셸에 접속하면 날짜가 출력되는 것을 확인할 수 있습니다.

## 10. Docker Swarm 알아보기

컨테이너 오케스트레이션 시스템은 Building -> Shipping -> Running Container의 과정을 대규모로 실행하기 위해 필요합니다. 이것을 위해 제공되는 솔루션 목록은 아래와 같습니다.

- [Kubernetes](#)

- [Docker Swarm](#)
- [Apache Mesos](#)

본 섹션에서는 컨테이너 오케스트레이션중 Docker Swarm에 대해서 언급합니다. 일반적으로는 Kubernetes를 많이 사용중이지만, 학습을 위해 Docker Swarm을 기준으로 설명합니다. Kubernetes는 “Kubernetes를 익히다.”에서 언급할 계획입니다.

## 10.1 컨테이너 오케스트레이션이 필요한 이유

수백 개의 컨테이너를 실행해야 하는 상황을 생각해봅시다. 이를 잘 관리하기 위해서는 관리를 위한 기능이 필요합니다.

- 컨테이너의 상태 체크
- 특정 Docker 이미지에 대해 고정된 컨테이너에서 시작
- 부하에 따라 컨테이너를 늘리거나 줄이는 기능
- 모든 컨테이너에서 애플리케이션 업데이트 수행
- 기타 등등

Docker Swarm을 사용하여 어떻게 해결 할 수 있는지 알아보도록 하겠습니다.

### 전제 조건

- 기본 Docker 명령에 익숙 해야 합니다.
- [Docker Toolbox](#)가 시스템에 설치되어 있어야 합니다.
- [Virtualbox](#)가 시스템에 설치되어 있어야 합니다.

## 10.2 Docker 머신 생성하기

Docker Swarm에서 노드 역할을 하는 Docker 시스템 환경을 만들 예정입니다. 여기서는 6개의 Docker 머신을 만들려고 합니다. 여기서 하나는 Manager(Leader)로 동작하고 다른 것들은 Worker Node가 됩니다. 표준 명령어를 사용하여 아래와 같이 **manager1**이라는 Docker 머신을 만듭니다.

```
grouq:~ giljae$ docker-machine create --driver virtualbox manager1
Running pre-create checks...
(manager1) Default Boot2Docker ISO is out-of-date, downloading the latest release...
(manager1) Latest release for github.com/boot2docker/boot2docker is v19.03.5
(manager1) Downloading /Users/giljae/.docker/machine/cache/boot2docker.iso from
https://github.com/boot2docker/boot2docker/releases/download/v19.03.5/boot2doc
ker.iso...
(manager1)
0%....10%....20%....30%....40%....50%....60%....70%....80%....90%....100%
Creating machine...
(manager1) Copying /Users/giljae/.docker/machine/cache/boot2docker.iso to
/Users/giljae/.docker/machine/machines/manager1/boot2docker.iso...
```

```
(manager1) Creating VirtualBox VM...
(manager1) Creating SSH key...
(manager1) Starting the VM...
(manager1) Check network to re-create if needed...
(manager1) Found a new host-only adapter: "vboxnet0"
(manager1) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this virtual
machine, run: docker-machine env manager1
```

Worker 노드도 만들도록 하겠습니다. worker1부터 worker3까지 생성합니다.

```
grouq:~ giljae$ docker-machine create --driver virtualbox worker1
grouq:~ giljae$ docker-machine create --driver virtualbox worker2
grouq:~ giljae$ docker-machine create --driver virtualbox worker3
```

생성 후에 “docker-machine ls” 명령어로 상태를 확인합니다.

```
grouq:~ giljae$ docker-machine ls
```

NAME	ACTIVE	DRIVER	STATE	URL	SWARM	DOCKER
manager1	-	virtualbox	Running	tcp://192.168.99.100:2376		v19.03.5
worker1	-	virtualbox	Running	tcp://192.168.99.101:2376		v19.03.5
worker2	-	virtualbox	Running	tcp://192.168.99.102:2376		v19.03.5
worker3	-	virtualbox	Running	tcp://192.168.99.103:2376		v19.03.5

모두 생성된 것을 확인할 수 있습니다.

manager1의 IP 주소를 알아보도록 합시다. manager1의 IP 주소를 알아내는 방법은 아래와 같습니다.

```
grouq:~ giljae$ docker-machine ip manager1
192.168.99.100
```

Docker 머신에 SSH로 접속을 할 계획입니다. SSH를 통해 해당 시스템에서 docker 명령을 실행해야 하기 때문에 SSH 사용에 익숙해져야 합니다.

사용법:

```
docker-machine ssh <머신 이름>
```

manager1 docker에 접속해보도록 합시다.

```
grouq:~ giljae$ docker-machine ssh manager1
( '>' )
/) TC (w Core is distributed with ABSOLUTELY NO WARRANTY.
(/-__-w)      www.tinycorelinux.net

docker@manager1:~$
```

## 10.3 Swarm Cluster

머신 설정이 완료되었기에 Swarm 설정을 진행 할 수 있습니다.

제일 처음에 해야 할 일은 Swarm을 초기화하는 것입니다. 우리는 manager1 머신에 SSH를 설치하고 초기화합니다. manager1에 접속한 상태에서 아래의 명령어를 입력합니다. (우리는 이미 이전에 manager1의 IP를 알아두었습니다.)

```
docker@manager1:~$ docker swarm init --advertise-addr 192.168.99.100
Swarm initialized: current node (yli8mapw893qwmavhq7at3x0t) is now a manager.
```

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

```
docker swarm join --token
SWMTKN-1-0mlmww1rp83ji0tsonyzud0ny99tcymc3yyi3fx6ot7sydabtm-1iwjz0h9pxg
4acro0sjm119q4 192.168.99.100:2377
```

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

```
docker@manager1:~$
```

위에 출력된 내용을 보면 다른 노드를 연결하려면 docker swarm join 명령을 사용하라고 언급되어 있습니다. 노드는 작업자(Worker) 또는 관리자(Leader)로 참여할 수 있습니다. 어느 시점에서든 하나의 Leader만 존재하고 다른 LEAD 노드는 현재 Leader가 아웃될 경우에 백업으로 사용됩니다.

아래의 명령어를 실행하여 Swarm 상태를 확인할 수 있습니다.

```
docker@manager1:~$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER
STATUS	ENGINE VERSION			
yli8mapw893qwmavhq7at3x0t *	manager1	Ready	Active	
Leader	19.03.5			

```
docker@manager1:~$
```

현재까지 단일노드 manager1이 존재하며 MANAGER열에 Leader 값을 가진 것을 확인할 수 있습니다.

### 10.3.1 Worker Node로 Join 하기

노드를 결합할 때 사용할 docker swarm 명령을 찾으려면 join-token <role> 명령어를 이용해야 합니다.

worker에 대한 join명령을 찾으려면 아래의 명령어를 실행합니다.

```
docker@manager1:~$ docker swarm join-token worker
To add a worker to this swarm, run the following command:

    docker swarm join --token
    SWMTKN-1-0mlmww1rp83ji0tsonyzud0ny99tcymc3yyi3fx6ot7sydabtm-1iwjz0h9pxg
    4acro0sjm119q4 192.168.99.100:2377

docker@manager1:~$
```

### 10.3.2 Manager Node로 Join 하기

관리자 노드의 Join 명령어를 찾으려면 아래의 명령어를 이용해야 합니다.

```
docker@manager1:~$ docker swarm join-token manager
To add a manager to this swarm, run the following command:

    docker swarm join --token
    SWMTKN-1-0mlmww1rp83ji0tsonyzud0ny99tcymc3yyi3fx6ot7sydabtm-46i5tmcmfqi
    3h544glpsf3qq8 192.168.99.100:2377

docker@manager1:~$
```

위의 두 가지 경우 모두 토큰이 제공되고 Manager 노드에 Join할 수 있음을 확인할 수 있습니다. (IP 주소가 MANAGER\_IP와 동일)

## 10.4 Swarm에 Worker Node 추가하기

이제 Worker로 Join하기 위해 사용할 명령어를 알게되었고, 이를 사용하여 각 Worker에서 join 명령을 실행할 수 있습니다.

위의 경우 3대의 Worker Machine이 존재합니다. (worker1/2/3)

worker1에서 다음을 수행 합니다.

- worker1 시스템으로 접속 합니다. (e.g. docker-machine ssh worker1)

- 그리고 Worker로 Join하기 위해 아래의 명령을 실행합니다.

```
docker@worker1:~$ docker swarm join --token
SWMTKN-1-0mlmww1rp83ji0tsonyzud0ny99tcymc3yyi3fx6ot7sydabtm-1iwjz0h9pxg
4acro0sjm119q4 192.168.99.100:2377
This node joined a swarm as a worker.
docker@worker1:~$
```

worker2/3에 대해서도 위와 동일하게 작업을 수행합니다.

모든 Worker Node가 Swarm에 Join한 후 manager1의 SSH 콘솔에서 아래의 명령어를 사용하여 Swarm의 상태를 확인합니다. 즉, 해당 노드가 참여하는 노드를 확인합니다.

```
docker@manager1:~$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER
yli8mapw893qwmavhq7at3x0t *	manager1	Leader	Ready	Active
op87x0jmdhvk030rjscbcjw6r	worker1	Ready	Active	
856rqgebt7da4c0yppdl6fcwl	worker2	Ready	Active	
mlve8o2r06p1hvf8bqpwwx3ab	worker3	Ready	Active	

```
docker@manager1:~$
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER
ig7y8bo5j8jrf9ovs9g970vvh	worker3	Ready	Active	

```
docker@manager1:~$
```

4개의 Node가 존재하는데 하나는 관리자(manager1)이고 다른 노드는 모두 Worker입니다. docker info 명령을 이용하여 Swarm의 세부 정보를 확인해봅시다.

```
docker@manager1:~$ docker info
~~~~~ 생략 ~~~~~
Swarm: active
NodeID: yli8mapw893qwmavhq7at3x0t
Is Manager: true
ClusterID: a9y4jlvvx2jnk346xaplgcrfp
Managers: 1
Nodes: 4
Default Address Pool: 10.0.0.0/8
SubnetSize: 24
Data Path Port: 4789
Orchestration:
  Task History Retention Limit: 5
Raft:
  Snapshot Interval: 10000
```

```

Number of Old Snapshots to Retain: 0
Heartbeat Tick: 1
Election Tick: 10
Dispatcher:
Heartbeat Period: 5 seconds
CA Configuration:
Expiry Duration: 3 months
Force Rotate: 0
Autolock Managers: false
Root Rotation In Progress: false
Node Address: 192.168.99.100
Manager Addresses:
192.168.99.100:2377
~~~~~ 생략 ~~~~~

```

위의 정보를 확인해봅시다.

- Swarm은 활성화 된 것으로 표시됩니다. 총 4개의 노드와 1개의 관리자가 있습니다.
- manager1에서 docker info 명령을 실행중이기에 “Is Manager”가 true로 표시됩니다.
- Raft 항목은 합의 알고리즘을 의미합니다. 자세한 내용은 [여기](#)를 확인하세요.

## 10.5 서비스 만들기

이제까지 우리는 Leader와 Worker를 연동했습니다. 이제 컨테이너를 실행할 차례입니다. 우리가 할 일은 관리자에게 컨테이너를 실행하도록 지시하는 것입니다. 컨테이너를 예약하고, 노드에 명령을 보내고 배포하는 작업을 처리해줍니다.

서비스를 시작하려면 아래의 순서대로 해야 합니다.

- 실행할 Docker 이미지는 무엇입니까? (여기서는 Docker Hub에서 제공되는 nginx 이미지를 실행합니다.)
- 80 포트로 서비스를 오픈 합니다.
- 시작할 컨테이너 수를 지정합니다.
- 서비스 이름을 작성합니다. (편리하게 관리하기 위함)

우선 우리가 할일은 nginx 컨테이너를 시작하는 것입니다. manager1에 SSH로 접속 한 후 아래의 명령어를 실행합니다.

```

docker@manager1:~$ docker service create --replicas 3 -p 80:80 --name web nginx
g2kc4lueo1yu1k33p4okzqxt6
overall progress: 3 out of 3 tasks
1/3: running
[=====>]
2/3: running
[=====>]
3/3: running

```



```
[=====>]
verify: Service converged
```

아래의 명령어로 서비스 상태를 확인합니다.

```
docker@manager1:~$ docker service ls
ID                NAME      MODE          REPLICAS  IMAGE      PORTS
g2kc4lueo1yu     web       replicated    3/3        nginx:latest
*:80->80/tcp
docker@manager1:~$
```

다음의 명령어로 서비스 상태와 노드 상태를 확인합니다.

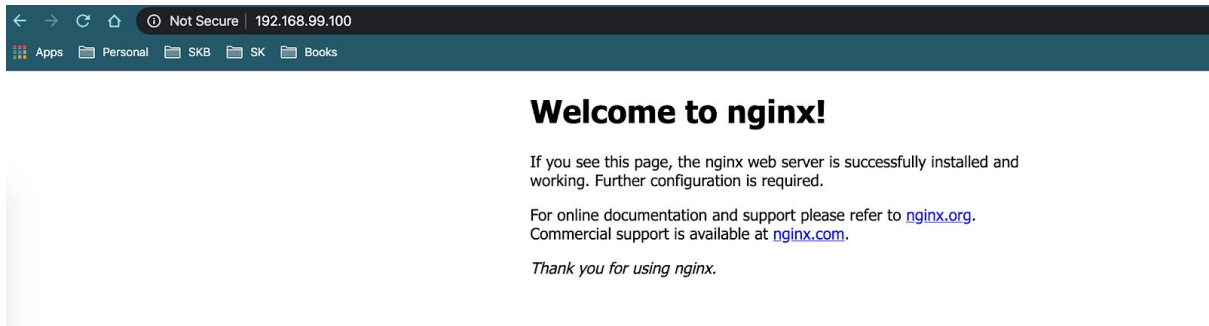
```
docker@manager1:~$ docker service ps web
ID                NAME      IMAGE          NODE          DESIRED STATE
CURRENT STATE    ERROR     PORTS
wtvu8i8qjadv     web.1     nginx:latest   manager1      Running
Running about a minute ago
z07fi174otkf     web.2     nginx:latest   worker1       Running
Running about a minute ago
8euc7m27n5up     web.3     nginx:latest   worker2       Running
Running about a minute ago
docker@manager1:~$
```

nginx 데몬이 시작된 것을 확인할 수 있습니다.

```
docker@manager1:~$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS
PORTS         NAMES
0326a579e968  nginx:latest  "nginx -g 'daemon of..." 6 minutes ago  Up 6
minutes      80/tcp        web.1.wtvu8i8qjadvx6fudmoo0e7zv
docker@manager1:~$
```

## 10.6 서비스 이용하기

Docker 머신 IP (manager1 또는 worker1/2/3)중 하나를 선택하여 브라우저에 URL (http://<machine-ip>)을 입력 후 접속해 보세요.



Nginx 페이지가 나오는 것을 확인할 수 있습니다.

## 10.7 확장 및 축소하기

서비스를 운영하다 보면 scale을 조정할 경우가 발생합니다. 이 부분은 docker service scale 명령어를 통해 지원됩니다. 현재 3개의 컨테이너가 실행 중입니다. manager1 노드에서 명령을 실행하여 아래와 같이 최대 5개까지 확장합니다.

```
docker@manager1:~$ docker service scale web=5
web scaled to 5
overall progress: 5 out of 5 tasks
1/5: running
[=====>]
2/5: running
[=====>]
3/5: running
[=====>]
4/5: running
[=====>]
5/5: running
[=====>]
verify: Service converged
docker@manager1:~$
```

규모가 5개로 조정되었습니다.

아래의 명령어를 통해 서비스 및 프로세스 작업의 상태를 확인합니다.

```
docker@manager1:~$ docker service ls
ID            NAME      MODE          REPLICAS  IMAGE      PORTS
g2kc4lueo1yu  web      replicated    5/5        nginx:latest
*:80->80/tcp
docker@manager1:~$
```

web(nginx)가 5개로 확장되었습니다.

## 10.8 노드 검사

docker node inspect 명령어를 이용해 언제든지 노드를 검사할 수 있습니다.

예를 들어서, 확인하려는 노드(e.g. manager1)의 SSH에 접속해 있는 경우에는 노드의 이름을 "self"로 사용 할 수 있습니다.

```
docker@manager1:~$ docker node inspect self
```

다른 노드를 확인하려면 노드 이름을 지정하면 됩니다.

```
docker@manager1:~$ docker node inspect worker1
```

## 10.9 노드 가용성 속성 변경

manager1 노드에서 아래의 명령어를 실행하여 노드 목록과 상태를 볼 수 있습니다.

```
docker@manager1:~$ docker node ls
ID                HOSTNAME          STATUS      AVAILABILITY    MANAGER
STATUS    ENGINE VERSION
yli8mapw893qwmavhq7at3x0t * manager1      Ready        Active
Leader      19.03.5
op87x0jmdhvk030rjscbcjw6r worker1      Ready        Active
19.03.5
856rqgebt7da4c0yppdl6fcwl worker2      Ready        Active
19.03.5
mlve8o2r06p1hvfq8bqpwwx3ab worker3      Ready        Active
19.03.5
docker@manager1:~$
```

AVAILABILITY가 Active로 되어 있는것을 알수 있습니다.

때로는 관리상의 이유로 노드를 중단시켜야 할 경우가 발생합니다. 이 의미는 AVAILABILITY를 Drain mode로 설정하는 것을 의미합니다. 우리는 노드 중 하나를 이용하여 테스트하도록 하겠습니다.

우선 웹 서비스의 프로세스 상태와 실행중인 노드를 확인합니다.

```
docker@manager1:~$ docker service ps web
ID                NAME          IMAGE          NODE          DESIRED STATE
CURRENT STATE    ERROR          PORTS
wtvu8i8qjadv     web.1         nginx:latest   manager1      Running
Running 2 hours ago
z07fi174otkf     web.2         nginx:latest   worker1      Running
Running 2 hours ago
8euc7m27n5up     web.3         nginx:latest   worker2      Running
Running 2 hours ago
uk1k8ej2195j     web.4         nginx:latest   worker3      Running
Running 10 minutes ago
tzdb0xxq3nb6     web.5         nginx:latest   worker3      Running
Running 10 minutes ago
docker@manager1:~$
```

복제본(Replica)가 5개 있는 것을 확인했습니다.

- manager1에 1개
- worker1에 1개
- worker2에 1개

- worker3에 2개

이제 worker1 노드에서 아래의 명령을 이용하여 상태를 확인합니다.

```
docker@manager1:~$ docker node ps worker1
```

ID	NAME	IMAGE	NODE	DESIRED STATE
z07fi174otkf	web.2	nginx:latest	worker1	Running

```
Running 2 hours ago
docker@manager1:~$
```

현재 Running 상태입니다. docker node inspect 명령어를 이용하여 노드의 가용성을 확인합니다.

```
docker@manager1:~$ docker node inspect worker1
```

```
~~~ 생략 ~~~
  "Spec": {
    "Labels": {},
    "Role": "worker",
    "Availability": "active"
  },
~~~ 생략 ~~~
```

Availability 속성이 “active”임을 알 수 있습니다.

이제 가용성(Availability)을 DRAIN으로 설정해보겠습니다. 해당 명령을 실행하면 Manager는 해당 노드에서 실행중인 작업을 중지하고 ACTIVE 가용성이 있는 다른 노드에서 복제본을 시작합니다.

따라서 Manager가 worker1에서 실행중인 1개의 컨테이너를 가져와서 다른 노드에 예약합니다. 이제 Availability 속성을 “drain”으로 설정해서 노드를 업데이트 해봅시다.

```
docker@manager1:~$ docker node update --availability drain worker1
```

```
worker1
docker@manager1:~$
```

프로세스 상태를 살펴봅시다.

```
docker@manager1:~$ docker service ps web
```

ID	NAME	IMAGE	NODE	DESIRED STATE
wtvu8i8qjadv	web.1	nginx:latest	manager1	Running
mofh2xf7kk6z	web.2	nginx:latest	manager1	Running
z07fi174otkf	web.2	nginx:latest	worker1	Shutdown
8euc7m27n5up	web.3	nginx:latest	worker2	Running

```
Running 2 hours ago
uk1k8ej2195j      web.4      nginx:latest    worker3      Running
Running 19 minutes ago
tzdb0xxq3nb6     web.5     nginx:latest    worker3      Running
Running 19 minutes ago
docker@manager1:~$
```

worker1의 컨테이너가 재조정되고 있음을 알 수 있습니다. 위에서는 manager1으로 예약되고 구동중입니다.

훌륭하지 않습니까?

## 10.10 서비스 제거하기

“docker service rm” 명령을 사용하여 서비스를 제거 할 수 있습니다.

```
docker@manager1:~$ docker service rm web
web
docker@manager1:~$ docker service ls
ID            NAME          MODE          REPLICAS    IMAGE          PORTS
docker@manager1:~$ docker service inspect web
[]
Status: Error: no such service: web, Code: 1
docker@manager1:~$
```

## 10.11 롤링 업데이트 적용하기

업데이트 된 Docker 이미지가 있는 경우 서비스 업데이트 명령어를 이용하여 롤링 업데이트를 적용합니다.

```
$ docker service update --image <imagename>:<version> web
```

## 10.12 결론

Docker Swarm은 매우 단순합니다. Kubernetes와 Swarm중 누가 승자일까요?  
둘 중 무엇이 적합한지 확인 후에 결정을 내리는 것이 맞지 않을까요?

# 11. Docker 사용 사례

Docker는 개발자들의 개발 환경에 많은 변화를 가져왔습니다. 평상시에 Docker와 함께 한다면 개발이 한층 더 수월해질것입니다.

## 11.1 새로운 소프트웨어 사용해보기

개발자는 항상 여러가지 소프트웨어를 테스트합니다. 그러나 소프트웨어를 다운로드하고 설정하는 작업은 번거롭습니다. Docker는 이미지를 기반으로 시작되기에 훨씬 간단하게 환경을 제공합니다.

예를 들어서 노트북에서 데이터베이스(MariaDB)를 구동시키려고 합니다. Docker는 매우 간단하게 MariaDB를 제공합니다. 로컬에 설치하게 되면 시간이 뺏기겠지요?

## 11.2 데모에 적합

다른 사람들에게 데모를 제공해야 할 때, 데모를 위한 소프트웨어 스택을 구성하는 것은 시간도 많이 걸리고 번거롭습니다. 데모할 장비가 바뀌기라도 한다면... 끔찍합니다. Docker는 소프트웨어 스택을 이미지 기반으로 패키징할 수 있기에 간편합니다.

## 11.3 머피의 법칙 피하기

개발을 하다보면 내 컴퓨터에서 동작하는 경우가 많습니다. 설정의 문제이거나 기타 다른 요인으로 인해 이런 상황들이 종종 발생합니다.

Docker를 이용하게 되면 이런 부담에서 벗어날 수 있습니다. Docker는 컨테이너 형식과 런타임을 제공하기에 우리를 행복하게 만들어줍니다.

다른 동료나 테스트팀에게 간단한 Docker 명령을 제공하는 것만으로 더 행복해질 수 있습니다.

그리고 교육을 하게 되었을때도 마찬가지입니다. Docker를 사용하여 모든 교육생의 컴퓨터에 소프트웨어를 설정하는 악몽에서 벗어날 수 있습니다. 이는 많은 시간을 절약하게 해줍니다.

## 11.4 리눅스와 스크립트 학습하기

Linux OS 및 스크립트 언어에 익숙하지 않은 사람들은 Docker를 이용해 학습 환경을 쉽게 구성할 수 있습니다.

## 11.5 자원 사용을 효율적으로

VM과 비교할 때 Docker 컨테이너가 훨씬 가볍다는 것을 우리는 배웠습니다. 조금더 자원 활용을 효율적으로 할 수 있습니다.

## 11.6 마이크로 서비스

마이크로 서비스에 대해서 들어보셨지요? Docker와 마이크로 서비스는 잘 작동합니다. Docker를 이용하면 개발팀이 더 쉽게 패키징할 수 있고 이는 테스트팀 및 배포팀에 이미지를 제공하기에

상호간 업무가 효율적으로 진행 됩니다.

## 11.7 클라우드 업체간 포팅

대부분의 클라우드 제공 업체는 Docker에 대한 전폭적인 지원을 표명했습니다. 이 의미는 다른 클라우드 제공 업체로 쉽게 이동할 수 있는 가능성을 보여줍니다. 즉, 훨씬 쉽고 단순하게 배포할 수 있습니다.

## 12. Docker에 더 가까이 다가서기

아래는 Docker를 배우는데 도움이 되는 자료의 모음입니다.

- [공식 도커 문서](#)
- [도커 사용해보기](#)
- [무료 도커 교육](#)
- [Awesome 도커](#)
- [도커 cheat sheet](#)
- [도커 사용법](#)
- [도커 공식 블로그](#)