저자 : Elton Stoneman

프리랜서. 도커에서 지정한 엠배서더인 도커 캡틴. blog.sixeyed.com

커리어 대부분이 마이크로소프트 엔터프라이즈 시스템 설계 및 전달에 대한 컨설팅.

목 자

1. 도커 컨테이너와 이미지 이해하기
2. 시작하기 전에
3. 도커의 기본적인 사용법
4. 도커 이미지 만들기
5. 애플리케이션 소스 코드에서 도커 이미지까지
6. 도커 허브 등 레지스트리에 이미지 공유하기
7. 도커 볼륨을 이용한 퍼시스턴트 스토리지
8. 컨테이너로 분산 애플리케이션 실행하기
9. 컨테이너 오케스트레이션을 이용한 스케일링
10. 운영 환경 투입을 위한 컨테이너 준비하기
11. 도커 컨테이너와 이미지 이해하기
12. 시작하기 전에

1.1 컨테이너가 IT 전반에 사용될 수 밖에 없는 이유

* 도커를 활용하면 product 출시 작업이 아주 쉬워지기 때문.
* 개발팀은 모든 프로젝트에서 같은 도구, 프로세스, 동일 런타임을 사용하기를 원한다. 도커를 이용하면 모든 제품의 빌드,배포,운영을 같은 도구와 방법으로 수행할 수 있다.
  1. 클라우드 환경으로 이주

1.2.1 어플리케이션의 클리우드 이주는 모든 조직에서 관심사이다.

(여기서 클라우드는 Azure, AWS, GCP 등)

무제한의 확장성, 빠른 배포, 사용한만큼 비용 지불.(저렴해?)

* + 1. 두 선택지가 존재
* PaaS : 어플리케이션의 각 컴포넌트를 클라우드의 managed service 에 옮기는 작업을 진행(beanstalk 등을 의미하나?), 특정 클라우드에 종속.

운영비는 절감.

* IaaS : 특정 클라우드에 종속되지 않는다. 운영비가 상승.
* 결론 : 도커를 사용하면 PaaS 의 장점인 운영비와 IaaS 의 장점인 이식성을 모두 누릴 수 있다. (리눅스와 AIX 등 OS 별 컨테이너가 호환이 되나?)

>>> Use Docker on IaaS

* + 1. 레거시 어플리케이션 현대화
* 모놀리식 어플리케이션을 단일 컨테이너로 이주 > 어플리케이션을 분할해 기능별로 별도의 컨테이너에 배치 (분할하기가 말처럼 쉽냐??.. 점진적으로 진행. 매니저가 의지가 있으면 가능)
  + 1. 마이크로 서비스 아키텍쳐 예제
* 다양한 프로그래밍 언어와 데이터베이스 사용.
* <https://github.com/microservices-demo> 참조.
  + 1. containerd 에 대하여.(책 외 내용)
* 컨테이너를 빌드하고, 실행하고 거기에 네트워크, 스토리지, CLI까지 제공해주는 Docker Engine이라는 패키지가 있었음.

하나의 패키지로 묶여있다보니 이를 해소하기 위해 여러 조직이 모여 **OCI(Open Container Initiative)라는 Container Runtime 표준을** 제정.

표준대로 각자 Container Runtime을 만들기 시작.

Docker에서 만든 Container Runtime 이 containerd

Kubernetes에서는 OCI 표준을 준수하는 이미지들을 실행할 수 있는 Container Runtime Interface, 이하 CRI 스펙을 제공

Red Hat, Intel, SUSE, Hyper, IBM 쪽에서도 OCI 표준에 따라 Kubernetes 전용 Container Runtime을 만들었는데 이것이 **CRI-O**

컨테이너를 실행할 때 **무거운 Docker Engine이 아닌 containerd나 CRI-O와 같은 가벼운 Container Runtime을 사용**

* 1. 도커 설치
* 우분투 기준, 도커 커뮤니티 설치.

|  |  |
| --- | --- |
| 기존 도커 삭제 | sudo apt-get remove docker docker-engine docker.io containerd runc |
| docker repo 추가를 위한 사전 작업 | sudo apt-get update  sudo apt-get install \  apt-transport-https \  ca-certificates \  curl \  gnupg-agent \  software-properties-common |
| repo 인증키 추가 | curl –fsSL <https://download.docker.com/linux/ubuntu/gpg> | gudo apt-key add - |
| 인증키 확인 | sudo apt-key fingerprint 0EBFCD88 |
| repo 추가 | sudo add-apt-repository \  “deb [arch=amd64] [https://download.docker.com/linux/ubuntu \](https://download.docker.com/linux/ubuntu%20\)  $(lsb\_release –cs) \  stable” |
| 도커 엔진 설치 | sudo apt-get update  sudo apt-get install docker-ce docker-ce-cli containerd.io |
| 도커 컴포즈 설치 | sudo curl –L [https://github.com/docker/compose/releases/download/1.27.4/docker-compose-$(uname -s)-$(uname -m)](https://github.com/docker/compose/releases/download/1.27.4/docker-compose-$(uname%20-s)-$(uname%20-m)) –o /usr/local/bin/docker-compose  sudo chmod +x /usr/local/bin/docker-compose |
| 설치 검증 | docker version  docker-compose version |
| 도커 컨테이너와 이미지 삭제 | 컨테이너 초기화  docker container rm –f $(docker container ls –aq)  이미지가 차지했던 용량 회수  docker image rm –f $(docker image ls –f referrence=`diamol/\*` -q) |

1. 도커의 기본적인 사용법

2.1 컨테이너에서 Hello, World

|  |  |
| --- | --- |
| 컨테이너 생성 후 실행 | docker container run diamol/ch02-hello-diamol   * 어플리케이션 패키지(이미지) 를 다운로드 후 실행   이미 존재하는 컨테이너를 시작할 때는 run 이 아닌 start 사용.   * docker container start web-ping   web-ping: 컨테이너 생성 시 --name 옵션에서 사용한 id |
| 컨테이너 실행과 내부 터미널 열기 | docker container run –interactive –tty diamol/base  --interactive: 컨테이너에 접속  --tty: 터미널 세션을 열기  보통 -it 축약형을 사용한다.  Docker container run -it ubuntu[enter] 라고 하면    -d 옵션 : 백그라운드 옵션  Docker run -it -d --name test2 ubuntu  => -it 옵션을 주지 않으면 ubuntu 가 기다리지 않고 Exited 상태로 종료. |
| 실행중인 컨테이너 | docker container ls |
| 종료된 컨테이너까지 모두 | docker container ls -all |
| 해당 컨테이너에서 top | docker container top [컨테이너ID 일부]  ex) docker container top f169 |
| 해당 컨테이너 로그 | docker container logs f169 |
| 컨테이너 상세 정보 | docker container inspect f169 |
| 컨테이너 상태 정보 | docker container stats f169 |
| 백그라운드에서 계속 실행 | docker container run –detach –publish 8088:80 diamol/ch02-hello-diamol-web  --detach: 백그라운드에서 실행  --publish [호스트][컨테이너]: 컨테이너 포트를 호스트 컴퓨터에 공개  호스트 8088 로 들어오면 컨테이너 80 으로 전달 |
| 컨테이너 삭제 | docker container rm a39b  : 컨테이너 종료가 아니라 삭제  실행 중인 컨테이너를 삭제할 경우 –force 옵션  모든 컨테이너 삭제 시  docker container rm --force ${docker container ls --all --quiet}  $() 는 괄호 안 명령의 출력을 다른 명령으로 전달 |
|  |  |

* application 이 실행 중일때 컨테이너도 [실행중] 상태가 된다.

application 이 종료되면 컨테이너는 [Existed] 상태가 된다.

대화식 컨테이너도 터미널을 종료하는 시점에 종료된다.

* 컨테이너가 종료되도 삭제되지 않는다.

2.2 컨테이너란?

각각의 VM 은 자신만의 운영체제를 가짐. CPU, RAM 을 많이 사용.

각 운영체제의 라이센스와 업데이트 설치 부담 발생.

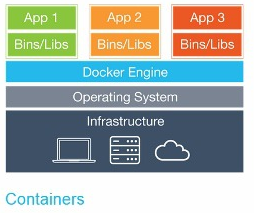
실행이 컨테이너보다 느림.

컨테이너도 VM 과 마찬가지로 외부와 독립된 환경을 제공. 각 컨테이너는 별도의 IP, Port 를 소유.

Container

Container

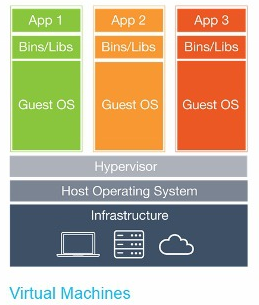
Container



VM

VM

VM



2.3. 컨테이너를 원격 컴퓨터로.(위 표 참조)

2.4 컨테이너를 웹 호스팅으로(위 표 참조)

2.5 도커가 컨테이너를 실행하는 원리

- Docker Engine : 도커의 관리 기능 담당. 로컬 이미지 캐시를 관리.

이미지 다운로드, 캐시 사용, 컨테이너,가상 네트웍 등 리소스 관리.

백그라운드로 동작.

* 도커 API : REST API. 도커 API 는 명세가 공개되어 있다. 따라서 도커 CLI 외 다른 클라이언트를 사용할 수 있다. GUI 클라이언트 등. OSS GUI 인 Portainer 가 있다. 상용은 UCP.
* 도커 CLI : 도커 API 의 클라이언트

docker Engine

운영체제

Docker Image

Cache

도커 API

3. 도커 이미지 만들기

3.0 로컬에서 도커 이미지 (구글링으로 내용 추가)

|  |  |
| --- | --- |
| 이미지 생성 | Create a new image from a container's changes  Usage: docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]] |
| 이미지 찾기 | docker search openjdlk |
| 자바 이미지 다운로드 | docker pull adoptopenjdk/openjdk11  도커 이미지는 물리적으로 여러 개의 작은 파일로 구성.  이들 각각을 이미지 레이어라고 한다. |
| Dockerfile 작성 | FROM adoptopenjdk/openjdk8 |
| 이미지 빌드 | docker image build --no-cache -f {dockerfile경로}--tag {이미지명}:{태그명} {PATH}[enter]  --no-cache : 레이어 캐시 사용안함.  PATH : build context 이다. 이미지에 포함될 리소스들이 위치.  ex) docker image build -f c:\work\Dockerfile \  --tag hmfb:0.1 C:\hmfb\git\hmfb\hmfb-zbin |
| 확인 | docker images |
| 이미지 실행 | docker run -it -p 8080:8080  --name hmfb-web-was  -e SPRING\_PROFILE=local  -e DB\_TYPE=mysql unchart/hmfb:0.5.0  -it: interative and tty  -e: 환경변수 설정 |
| 컨테이너 종료 | Ctrl+C 로 앱을 종료시키거나  docker stop |
| 이미지 |  |
|  |  |
|  |  |

3.1 도커 허브의 이미지 사용하기

https://hub.docker.com/r/diamol/ch03-web-ping

|  |  |
| --- | --- |
| 이미지 다운로드 | docker image pull diamol/ch03-web-ping |
| 이미지 실행 | docker container run –d –name web-ping diamol/ch03-web-ping  -d: --detach 의 축약  -name: 디폴트 id 가 아닌 컨테이너에 쉬운 이름을 붙인다.  해당 app은 ping client 기능을 구현했다. |
| 환경변수 지정해서 실행 | docker container run –env TARGET=google.com diamol/ch03-web-ping |
| Dockerfile 작성 | 아래 3.2 참조 |
| DockerHub repository생성 | https://hub.docker.com/repository/docker/dkkim77/unchart/general |
| docker 로그인 | docker login |
|  |  |

3.2 Dockerfile 작성하기

|  |  |
| --- | --- |
| FROM adoptopenjdk/openjdk8  ENV INSTALL\_PATH=/hmfb/git/hmfb  WORKDIR $INSTALL\_PATH  RUN echo "copy [source] [target]: source is relative path of build context. context build is [C:\hmfb\git\hmfb]"  COPY hmfb.yml $INSTALL\_PATH  COPY hmfb-web/build/libs/hmfb-web-0.0.1-SNAPSHOT.jar $INSTALL\_PATH  CMD java  -Dspring.profiles.active=$SPRING\_PROFILE  -DdbType=$DB\_TYPE  -DinstallPath=$INSTALL\_PATH  -XX:+HeapDumpOnOutOfMemoryError -jar $INSTALL\_PATH/hmfb-web-0.0.1-SNAPSHOT.jar | 모든 이미지는 다른 이미지에서 출발. 이 이미지는 adoptopenjdk/openjdk8 이미지가 시작점.  환경변수 설정  컨테이너 이미지 파일 시스템에 작업 디렉토리를 생성하고 지정. 윈도우도 ‘/’ 를 사용.  RUN: **이미지 빌드 과정**에서 필요한 커맨드를 실행하기 위해서 사용. 보통 이미지 안에 특정 소트트웨어를 설치하기 위해 사용.  RUN apk add curl  RUN pip install -r requirements.txt  로컬에 있는 개별 파일을 작업 디렉토리로 복사  로컬에 있는 jar 를 이미지 작업 디렉토리로 복사.  도커가 이미지를 실행했을 때 실행할 명령을 지정.. CMD 는 컨테이너 실행 시 수행. 이미지 빌드 시에는 수행 안됨. |

3.3 이미지 빌드하기

- ENV 에 대한 오용 사례

ENV JAVA\_OPTS=”-Dactive.profile=${PROFILE} -DdbType=${DB\_TYPE}

CMD java ${JAVA\_OPTS} ... 생략 ...

prompt> docker run -e PROFILE=local -e DB\_TYPE=mysql ... 생략...

위와 같이 사용했을 때 동작이 안됐다.

추측: CMD 는 빌드시 사용되는 명령이 아님.

이미지 생성 시 JAVA\_OPTS=”-Dactive.profile= -DdbType= “ 라는 환경변수가 생성됨. -e 옵션은 docker run 시에 주는 값임. 빌드시엔 없음.

따라서 컨테이너 실행 시 JAVA\_OPTS 는 빈 값들만 있는 환경변수이므로 -e 옵션으로 줘봐야 소용없었음.

3.4 이미지 레이어 이해하기

- 이미지는 이미지 레이어가 모인 논리적 대상

- 이미지 레이어는 **도커 엔진의 캐시에** 물리적으로 저장된 파일

- 캐시에 있는 이미지 레이어는 여러 이미지와 컨테이너에서 공유된다.

- docker image ls 에 나온 SIZE 는 공유된 os, runtime 의 용량을 포함하므로 실제 용량이라고 할 수 없다.

- docker system df 로 실제 용량을 확인.(상세하게 모르겠는데?)

|  |  |
| --- | --- |
| 이미지 사이즈 확인 | docker image ls  docker system df |
|  |  |
|  |  |
|  |  |
|  |  |

3.5 레이어 캐시를 이용한 Dockerfile 최적화

- 배포할 파일의 내용이 변경되면 캐시를 이용하지 않는다.

- 도커는 캐시를 이용하기 위해 Hash 값을 이용한다.

- 특정 레이어가 변경되어 캐시를 이용하지 않으면 그 이후 레이어도 다시 실행.

**따라서 자주 변경되는 레이어는 Dockerfile 후반부로 이동시켜야 한다**.

- CMD 인스트럭션은 FROM 뒤라면 어디에 있어도 무방.(빌드와 관련없어서?)

- ENV 는 하나로 합칠 수 있다.

ENV TARGET=”google.com” \

METHOD=”HEAD” \

INTERVAL=”3000”

4. 소스 코드에서 도커 이미지까지

4.1 빌드 서버가 필요할까?

- JENKINS : 형상 연결, 빌드, 배포, 스케줄링, 서버 재시작.

DOCKER : Git 연결, Javac, COPY, CMD [서버], 스케줄링만 안되나?

- 멀티 스테이지 빌드 작성 : JENKINS 의 PIPELINE 과 유사. 일련의 TASK 들을 연결. 어느 한 단계가 실패하면 전체 빌드가 실패한다.

|  |  |
| --- | --- |
| FROM diamol/base AS build-stage  RUN echo “first stream...” > /build.txt  FROM diamol/base AS test-stage  COPY --from=build-stage /build.txt /build.txt  RUN echo “second stream...” >> /build.txt  FROM diamol/base  COPY --from=test-stage /build.txt /build.txt  CMD cat /build.txt | FROM 절에 이름을 부여.  RUN: 실행 결과값을 레이어에 저장.  --from : build.txt 를 호스트 파일이 아니라 build-stage 단계의 결과물을 COPY. |

FROM openjdk

COPY ...

CMD java ...

test-stage

FROM junit

COPY ...

RUN java ...

build-stage

FROM maven:jdk

RUN mvn dep...

RUN mvn:package

1단계 : 의존 모듈을 내려받고 어플리케이션을 빌드한다.

2단계 : 단위 테스트를 수행한다.

3단계 : 어플리케이션을 수행한다.

4.2 자바 소스 코드 예제

- dockerfile 및 설명 생략.

- 컨테이너는 실행될 때 부여되는 가상 네트웍 내 가상 IP 를 통해 서로 통신.

|  |  |
| --- | --- |
| 도커 네트웍 생성 | docker network create nat  nat 라는 가상 네트웍 생성. |
| 컨테이너의 nat 접속 | docker container run --name iotd -d -p 8080:80 --network nat image-of-the-day |
| 도커 네트웍 정보 | docker network inspect -v nat |
| 컨테이너 상세(네트웍) 정보 | docker container inspect [컨테이너명] |

5. 도커 허브와 레지스트리에 이미지 공유

5.1 Registry, Repository, 이미지 Tag

- Registry : 대표적으로 도커 허브. 도커 엔진에 기본 설정된 Registry.

- 이미지 명 스키마: docker.io/diamol/golang:latest

docker.io : 레지스트리의 도메인.

dialmol : 작성자의 계정 이름

golang : repository 이름. 일반적으로 어플리케이션 이름.

latest : 이미지 Tag.

레지스트리와 태그는 필수는 아님. 기본값을 사용.

5.2 도커 허브에 이미지 푸시하기

|  |  |
| --- | --- |
| 도커 로그인 | docker login --username $userId |
| 업로드 사전 작업 | registry 에 push 를 위해서 이미지 참조에 계정,리파지토리가 있어야 하기 때문에 tag 를 하나 더 추가한다.  docker image tag [origin] [newTag] |
| 기존 이미지에 Tag 부여 | docker image tag unchart/hmfb dkkim77/unchart:v1 |
| 이미지 Registry 업로드 | docker image push dkkim77/checkapp:0.1 |
|  |  |

- 이미지를 push할 때 로컬에서처럼 이미지 레이어를 push 한다.

registry에 없는 레이어만 업로드된다.

- 기존의 repository 가 없으면 도커 허브에 repository 를 public 으로 생성한다.

- What's the difference between "Shared" and "Simple" tags?

도커 허브에서 "Shared" 태그는 여러 개의 이미지가 하나의 태그를 공유하는 경우에 사용됩니다. 예를 들어, "ubuntu" 태그는 여러 버전의 Ubuntu 이미지를 포함할 수 있습니다. 이 경우, "ubuntu" 태그는 여러 이미지에 대해 공유됩니다.

반면에, "Simple" 태그는 하나의 이미지만을 가리킵니다. 예를 들어, "ubuntu:18.04" 태그는 하나의 Ubuntu 18.04 이미지만을 가리킵니다.

- docker image pull python [enter] : shared tag 로 받으니 912M 가 다운로드.

5.3 나만의 도커 Registry 운영

|  |  |
| --- | --- |
| registry 다운로드 | docker pull registry  docker hub 에서 다운로드[24.15MB] |
| registry 이미지 실행 | docker container run -d -p 5000:5000 --name private-registry --restart always registry  --restart : 이미지가 종료되면 계속 재시작. |
| private 에 업로드 사전 작업 | 1. registry 컨테이너는 HTTP를 사용하지만 docker 기본 설정은 비보안 프로토콜을 사용할 수 없으므로 private registry 를 docker 설정에 추가해야 한다.  도커 설정은 C:/Program Data/docker/config/daemon.json 에 있다.  유닉스는 /etc/docker 에 위치. insecure-registries 추가.  2. 도메인을 추가한 이미지 참조 생성.  docker image tag dkkim77/unchart:0.1 127.0.0.1:5000/unchart:0.1 |
| private 에 이미지 업로드 | docker image push 127.0.0.1:5000/unchart:0.1 |

5.4 이미지 태그를 효율적으로 사용

- major.minor.patch 형태로 태그 관리

5.5 Official 이미지에서 Golden 이미지로 전환

- Offical 이미지를 기반으로 더 많은 것을 통제하고 할 때 자신의 기반 이미지 를 생성. 자신만의 Dockerfile.

6. 도커 볼륨을 이용한 Persistant Storage

6.1 컨테이너 안 데이터가 사라지는 이유

- 도커 컨테이너에도 단일 드라이브 파일 시스템이 존재.

- 컨테이너가 종료되어도 파일 시스템은 삭제되지 않는다. 컨테이너가 삭제되면 파일 시스템도 삭제된다.

컨테이너 파일 시스템은 이미지 레이어와 기록 가능 레이어로 구성.

Container2   
기록 가능 layer

Container1  
기록 가능 layer

dkkim77/checkapp

application layer

운영체제 layer

모든 컨테이너가 공유하는 이미지 레이어는 읽기 전용이고 각 컨테이너가 별도 갖는 기록 가능 레이어는 컨테이너와 같은 생애 주기를 갖는다.

도커는 이미지 레이어에 있는 파일을 copy-on-write 를 통해서 수정할 수 있다.(아래 보면 아닌듯)

>docker container run --name con1 file-display => 이미지에 있던 input.txt 내용을 출력

google.com

> echo “naver.com” > url.txt => url.txt 를 호스트에 생성

>docker container cp url.txt con1:/input.txt => 컨테이너에 url.txt 를 input.txt 로 overwrite

>docker container start --attch con1 => 재실행하면 overwrite 된 input.txt 가 출력.

naver.com

=> 이미지 레이어를 변경한 게 아닌데 저자는 그렇게 보는 것 같다.

컨테이너는 앱 update 시 밥먹듯이 삭제되고 재생성된다. 별도의 메커니즘이 필요하다.

6.2 Docker Volume

- 컨테이너를 위한 USB 메모리처럼 생각. 컨테이너에 연결 가능.

- Dockerfile 에 Volume 명령어 사용.

|  |  |
| --- | --- |
| 컨테이너 -> 호스트  파일 복사 | docker container cp 컨테이너명:파일경로 타켓경로  ex) docker container cp con1:/random/number.txt number1.txt |
| Dockerfile 에 Volume 작성 | VOLUME /data  COPY 호스트DIR /data  - COPY 호스트DIR/test.txt /data 라고 하니 아래 오류 발생.  cannot mount volume over existing file, file exists |
| Docker Client 에서 생성 | Docker volume create [volume명]  Ex) docker volume create fs\_data  ※ dockerfile 의 volume 명령과 차이 :  - 연결된 컨테이너가 삭제되더라도 유지.  - dockerfile 의 volume 은 컨테이너가 생성될 때마다 생성됨.  Volume create 문으로 생성한 것은 하나만 생성. |
| Volume 연결 | Docker container run -d -it -v fs\_data:/data … 생략 |
| Volume 내용 확인 | 리눅스: Docker container exec [컨테이너명] ls /data  윈도우: docker container exec [컨테이너명] cmd /C "dir C:\data" |
| 컨테이너 간 Volume 공유 | Docker container run -d --name newContainer --volumes-from originContainer [이미지명]  => originContainer 의 volume 을 공유한다.  확인: docker container exec newContainer ls /data |

6.3 파일 시스템 마운트

- 호스트의 스토리지를 컨테이너에 직접적으로 연결. bind mount.

호스트 컴퓨터의 디렉토리를 컨테이너 파일 시스템의 디렉토리로 마운트한다.

|  |  |
| --- | --- |
| Bind mount | source=/data/databases  target=/data  Docker container run --mount type=bind,source=$source,target=$target,readonly … 이하 생략 |

- bind mount 는 양방향으로 동작한다. 호스트에 대한 공격을 방지하기 위해 컨테이너는 최소 권한 계정으로 실행 > 호스트 컴퓨터 파일에 접근하기 위해 권한 상승이 필요.

6.4 파일 시스템 마운트의 한계점

- 이미 이미지 레이어의 파일이 컨테이너의 마운트 대상 디렉토리에도 이 파일이 있는 경우 => 이미 존재하는 대상 디렉토리에 마운트하면 마운트 원본 디렉토리가 기존 디렉토리를 대체한다. 이미지에 포함되어 있던 파일은 사용할 수 없다.

- 컨테이너에 분산 스토리지를 마운트할 계획이라면 분산 스토리지의 성능이 로 컬 스토리지와 큰 차이가 있다는 것을 고려해야.. 모든 파일 입출력이 네트워크 를 거치기 때문.

6.5 컨테이너의 파일 시스템은 어떻게 만들어지는가

-

1. 컨테이너로 분산 애플리케이션 시작하기
2. Docker Compose 로 분산 애플리케이션 실행하기

7.1 도커 컴포즈 파일의 구조

- docker compose 파일의 구성

|  |  |  |
| --- | --- | --- |
| 요소 | 설명 | 예시 |
| Version | Docker compose 파일 형식의 버전 | version: '3.7' |
| Services | 앱을 구성하는 모든 컴포넌트를 열거. 하나의 서비스를 같은 이미지로 여러 컨테이너에서 실행 가능.  서비스 이름은 컨테이너의 이름이자 도커 네트웍 상에서 컨테이너를 식별하기 위한 DNS 이름이다. | services:  pytest:  image: dkkim77/checkapp-web  ports:  - "8080:8080"  networks:  - app-net |
| Networks | 컨테이너가 연결될 모든 docker network 을 열거. external 의 의미는 nat 네트워크가 이미 존재하므로 생성하지 말라는 뜻. | networks:  app-net:  external:  name: nat |

- docker compose 실행

|  |  |
| --- | --- |
| Docker compose 실행 | > docker-compose up -d  현재 작업 디렉토리에 docker-compose.yml 파일이 존재해야 함. |
| 더 많은 옵션 | > docker-compose -f docker-compose-multicon.yml -p multiproj up -d  -f : 파일 경로 지정  -p : 프로젝트 명 지정(생략되면 실행 디렉토리 명으로 지정)  -d : dettach |

7..2 docker compose 를 사용해서 여러 컨테이너 실행하기

- 파일 예시

|  |  |
| --- | --- |
| 내용 | version: '3.7'  services:   pytest-web:  image: dkkim77/checkapp-web  ports:  - "8080:80"  networks:  - app-net   pytest:  image: dkkim77/checkapp  depends\_on:  - pytest-web networks:  app-net:  external:  name: nat |

- depends\_on 로 pytest 서비스보다 pytest-web 이 먼저 실행된다.

- 중요 : 상태가 없는 서비스일 경우 컨테이너를 늘리는 것으로 scale out 을 달 성할 수 있다.

http 요청이 들어오면 docker 가 여러 컨테이너에 분배해 준다.

|  |  |
| --- | --- |
| Scale out | > docker-compose up -d --scale pytest-web=3  컨테이너를 두 개 늘려서 처음의 세 배로 늘린다. |
| Scale out 확인 | 웹 브라우저로 여러 번 요청.  > docker-compose logs --tail=1 pytest-web  --tail : 각 컨테이너의 마지막 로그를 출력  로그를 확인해서 여러 컨테이너로 요청이 분배됐는지 확인한다. |
| Compose 정지 | > docker-compose -p multiproj stop |
| Compose 시작 | > docker-compose -p multiproj start |
| 어플리케이션 제거 | > docker-compose -p multiproj down |

7.3 도커 컨테이너 간의 통신

- 도커에 DNS 서비스가 내장. 다른 컨테이너에 접근하기 위해 DNS 이용.

도메인이 가리키는 대상이 컨테이너가 아니면 호스트에 요청을 보내 호스트가 속한 네트워크나 인터넷의 IP 주소를 조회한다.

|  |  |
| --- | --- |
| 컨테이너에서 shell 실행 | > docker container exec -it [컨테이너1 명] cmd |
| 도메인 조회 | > nslookup [컨테이너2 명] |

7.4 도커 컴포즈로 어플리케이션 설정값 지정하기

-

8장. 헬스 체크와 디펜던시 체크

8.1 Health Check 을 지원하는 도커 이미지 빌드

- 도커는 컨테이너에서 실행되는 프로세스가 비정상 종료됐거나 컨테이너가 종 료되면 알 수 있지만 어플리케이션의 정상적인 상태를 알 수는 없다.

예를 들면 이 컨테이너의 처리 용량을 초과하는 수의 요청이 들어올 경우 '503 service unavailable' 오류를 발생시킬 것이다.

- 도커는 어플리케이션의 상태가 정상인지 확인할 수 있는 정보를 dockerfile 에 넣을 수 있다.

|  |  |
| --- | --- |
| Health Check | Dockerfile 에 다음 인스트럭션을 넣고 빌드  > HEALTHCHECK CMD curl --fail http://localhost/health  => curl 의 요청이 성공하면 0을 도커에 반환.  도커는 0을 정상. 이외는 비정상으로 판단  비정상일 경우 컨테이너의 상태는 Up(unhealthy) 가 된다.  도커가 종료시키지는 않는다. |
| Interval/retry 지정 |  |
| 상태 확인 | > docker container inspect $(docker container ls --last 1 --format '{{.ID}}') |

- 왜 도커는 비정상적인 컨테이너에 대해 재시작을 하지 않는가?..

저자의 의견: 컨테이너에 보관된 데이터가 유실되고 그 시간 동안 어플리케이 션이 동작하지 않기 때문에 상황을 더 악화시킬 수 있다고 판단.

물론 헬스 체크도 계속된다. 일시적인 실패라면 컨테이너의 상태가 다시 healthy 로 돌아간다.

8.2 컨테이너 간 Dependancy 체크

- 어플리케이션 실행 전 필요한 요구 사항을 확인. 별도의 인스트럭션 (HEALTHCECK처럼) 없이 Dockerfile 에 체크 로직을 넣는다.

|  |  |
| --- | --- |
| 작성 예 | FROM diamol/dotnet-aspnet  // 관련 container 에 curl 로 요청해본다. 사용 가능한 상태면 curl 이 성공하고 후속 프로세스인 dotnet 이 실행된다.  CMD curl --fail http://number-api/rng && dotnet Numbers.Web.dll  WORKDIR /app |