Docker 좀더 활용하기

목차

1. Docker 개인 저장소 구축하기

Docker 저장소 서버는 Docker 레지스트리(registry) 서버라고 부릅니다. docker push 명령으로 레지스트리 서버에 이미지를 올리고, docker pull 명령으로 이미지를 받을 수 있습니다.

기존에 실행되고 있는 Docker 데몬을 정지한 뒤 --insecure-registry 옵션을 사용하여 Docker 데몬을 실행

```
$ sudo service docker stop
$ sudo docker -d --insecure-registry localhost:5000
```

기존에 실행되고 있는 Docker 데몬을 정지한 뒤 --insecure-registry 옵션을 사용하여 Docker 데몬을 실행

/etc/init.d/docker

```
DOCKER_OPTS=--insecure-registry localhost:5000
```

1. Docker 개인 저장소 구축하기

1. 로컬에 이미지 데이터 저장

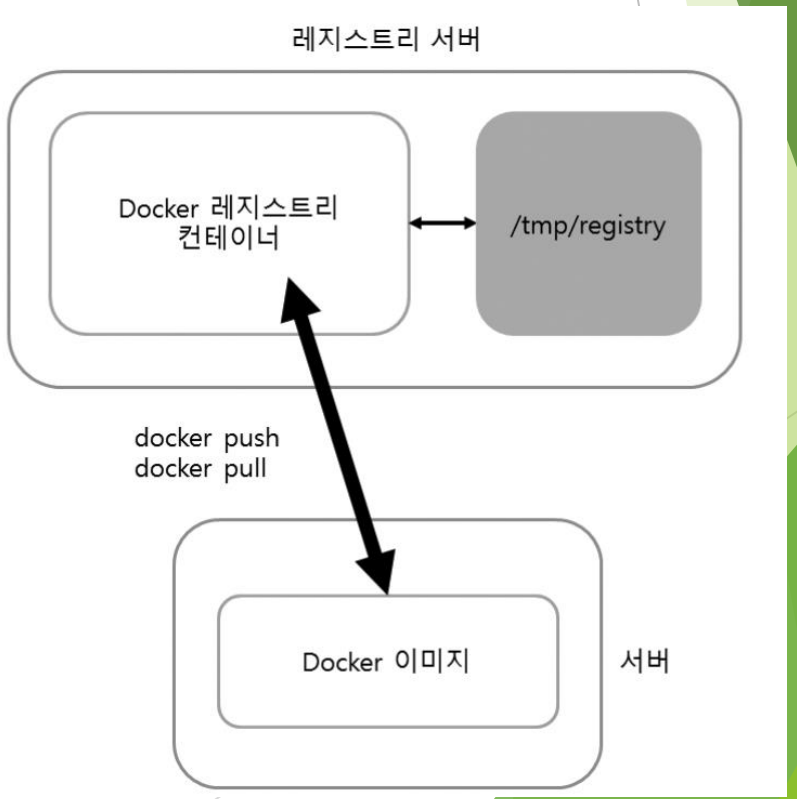
Docker 레지스트리 서버도 Docker Hub를 통해 Docker 이미지로 제공됩니다. 먼저 Docker 레지스트리 이미지를 받습니다.

```
$ sudo docker pull registry:latest
```

registry:latest 이미지를 컨테이너로 실행합니다.

```
$ sudo docker run -d -p 5000:5000 --name hello-registry \
    -v /tmp/registry:/tmp/registry \
    registry
```

✓ 이렇게 실행하면 이미지 파일은
호스트의 /tmp/registry 디렉터리에 저장됩니다.



1. Docker 개인 저장소 구축하기

2. push 명령으로 이미지 올리기

```
$ sudo docker tag hello:0.1 localhost:5000/hello:0.1
$ sudo docker push localhost:5000/hello:0.1
```

- ✓ 태그를 생성하는 명령은 docker tag 〈이미지 이름〉:〈태그〉 〈Docker 레지스트리 URL〉/〈이미지 이름〉:〈태그〉 형식
- ✓ 이미지를 올리는 명령은 docker push 〈Docker 레지스트리 URL〉/〈이미지 이름〉:〈태그〉 형식
- ✓ 개인 저장소에 이미지를 올릴 때는 태그를 먼저 생성해야 합니다.
- ✓ docker tag 명령으로 hello:0.1 이미지를 localhost:5000/hello:0.1 태그로 생성합니다.
- ✓ 그리고 docker push 명령으로 localhost:5000/hello:0.1 이미지를 개인 저장소에 올립니다(태그를 생성했으므로 실제로는 hello:0.1 이미지가 올라갑니다).

다른 서버에서 개인 저장소(Docker 레지스트리 서버)에 접속하여 이미지를 받아올 수 있습니다. 개인 저장소 서버 IP 주소가 192.168.0.39라면 다음과 같이 명령을 실행합니다.

```
$ sudo docker pull 192.168.0.39:5000/hello:0.1
```

1. Docker 개인 저장소 구축하기

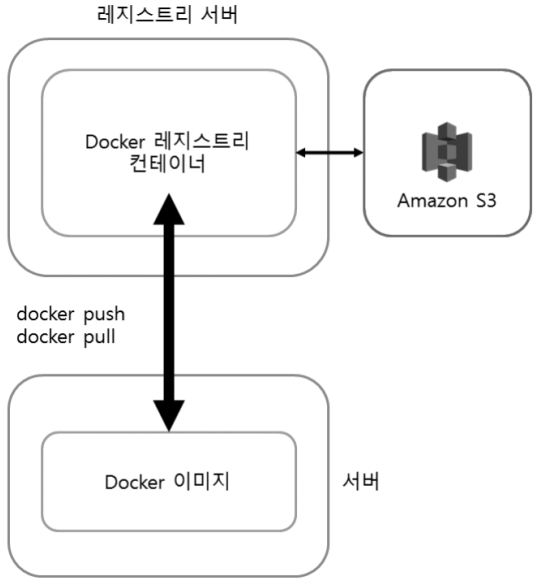
3. Amazon S3에 이미지 데이터 저장

이번에는 AWS의 S3에 이미지를 저장하는 방법입니다. 먼저 Docker 레지스트리 이미지를 받습니다.

```
$ sudo docker run -d -p 5000:5000 --name s3-registry \
    -e SETTINGS_FLAVOR=s3 \
    -e AWS_BUCKET=examplebucket10 \
    -e STORAGE_PATH=/registry \
    -e AWS_KEY=AKIABCDEFGHIJKLMNOPQ \
    -e AWS_SECRET=sF4321ABCDEFGHIJKLMNOPqrstuvwxyz21345Afc \
    registry
```

S3를 사용하려면 -e 옵션으로 설정을 해주어야 합니다.

- SETTINGS_FLAVOR: 이미지 저장 방법입니다. s3을 설정합니다.
- AWS_BUCKET: 이미지 데이터를 저장할 S3 버킷 이름입니다. 예제에서는 examplebucket10을 설정했습니다. 여러분이 생성한 S3 버킷 이름을 입력합니다.
- STORAGE_PATH: 이미지 데이터 저장 경로입니다. /registry를 설정합니다.
- AWS_KEY: AWS 액세스 키를 설정합니다.
- AWS_SECRET: AWS 시크릿 키를 설정합니다.



1. Docker 개인 저장소 구축하기

4. 기본 인증 사용하기

Docker 레지스트리에는 로그인 기능이 없습니다. 따라서 Nginx의 기본 인증(Basic Authentication) 기능을 사용해야 합니다. 또한, HTTP 프로토콜에서는 인증을 지원하지 않으므로 반드시 HTTPS 프로토콜을 사용해야 합니다.
먼저 /etc/hosts 파일을 편집하여 테스트용 도메인을 추가합니다. 이 파일은 root 권한으로 수정해야 합니다. 도메인을 구입하지 않았을 때는 이 부분을 반드시 설정해주어야 하며, 도메인을 구입하여 DNS를 설정했다면 이 부분은 건너뛰어도 됩니다.

/etc/hosts

```
127.0.0.1       localhost
127.0.1.1       ubuntu
<레지스트리 서버 IP 주소>    registry.example.com

# The following lines are desirable for IPv6 capable hosts
::1     localhost ip6-localhost ip6-loopback
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

여러분의 레지스트리 서버 IP 주소를 **registry.example.com**으로 설정합니다. 이 책에서는 **registry.example.com**을 기준으로 설명하겠습니다.

이제 SSL 사설 인증서(Self Signed)를 생성하겠습니다. SSL 공인 인증서를 구입했다면 이 부분은 건너뛰어도 됩니다.

다음 명령을 입력하여 개인 키 파일을 생성합니다.

```
$ openssl genrsa -out server.key 2048
```

1. Docker 개인 저장소 구축하기

4. 기본 인증 사용하기

인증서 서명 요청(Certificate signing request) 파일을 생성합니다.

•Country Name: 국가 코드입니다. 대문자로 KO를 입력합니다.
•State or Province Name: 주 또는 도입니다. 자신의 상황에 맞게 입력합니다.
•Locality Name: 도시입니다. 자신의 상황에 맞게 입력합니다.
•Organization Name: 회사 이름을 입력합니다.
•Organizational Unit Name: 조직 이름을 입력합니다.
•Common Name: Docker 레지스트리를 실행하는 서버의 도메인입니다. 이 부분을 정확하게 입력하지 않으면 인증서를 사용해도 로그인할 때 에러가 발생합니다. /etc/hosts 파일에 설정한대로 registry.example.com를 입력합니다.
•Email Address: 이메일 주소입니다.

```
$ openssl req -new -key server.key -out server.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:KO
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:Seoul
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Example Company
Organizational Unit Name (eg, section) []:Example Company
Common Name (e.g. server FQDN or YOUR name) []:registry.example.com
Email Address []:exampleuser@example.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:<아무것도 입력하지 않음>
An optional company name []:<아무것도 입력하지 않음>
```

1. Docker 개인 저장소 구축하기

4. 기본 인증 사용하기

서버 인증서 파일을 생성합니다.

```
$ openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
```

이제 server.crt 인증서 파일을 시스템에 설치를 해주어야 합니다(인증서 파일을 설치하지 않으려면 `--insecure-registry` 옵션을 사용해야 합니다. 이 부분은 뒤에 따로 설명하겠습니다).

우분투

```
$ sudo cp server.crt /usr/share/ca-certificates/
$ echo "server.crt" | sudo tee -a /etc/ca-certificates.conf
$ sudo update-ca-certificates
```

CentOS

```
$ sudo cp server.crt /etc/pki/ca-trust/source/anchors/
$ sudo update-ca-trust enable
$ sudo update-ca-trust extract
```

/etc/hosts에 도메인을 추가하고, 인증서 파일을 설치했으면 Docker 서비스를 재시작합니다. Docker 서비스를 재시작해야 추가된 도메인과 설치된 인증서가 적용됩니다.

```
$ sudo service docker restart
```

Docker 레지스트리에 접속할 다른 시스템에도 **server.crt** 인증서 파일을 복사하여 같은 방식으로 설치를 하고 Docker 서비스를 재시작합니다. 그리고 도메인을 구입하지 않았다면 **/etc/hosts**에 레지스트리 서버(registry.example.com)의 IP 주소를 설정합니다.

1. Docker 개인 저장소 구축하기

4. 기본 인증 사용하기

사용자 계정과 비밀번호를 저장할 .htpasswd 파일을 생성해야 합니다. 먼저 다음 패키지를 설치합니다.

우분투

```
$ sudo apt-get install apache2-utils
```

CentOS

```
$ sudo yum install httpd-tools
```

`htpasswd` 명령으로 **.htpasswd** 파일을 생성하고 **hellouser**라는 예제 사용자를 추가합니다. 비밀번호 입력 부분에는 사용할 비밀번호를 입력합니다.

```
$ htpasswd -c .htpasswd hellouser
New password:<비밀번호 입력>
Re-type new password:<비밀번호 입력>
Adding password for user hellouser
```

1. Docker 개인 저장소 구축하기

4. 기본 인증 사용하기

nginx.conf

```
worker_processes  1;

events {
    worker_connections  1024;
}

http {
    server {
        listen       443;
        server_name  registry.example.com;

        ssl on;
        ssl_certificate /etc/server.crt;
        ssl_certificate_key /etc/server.key;

        proxy_set_header Host           $http_host;
        proxy_set_header X-Real-IP      $remote_addr;
        proxy_set_header Authorization  "";

        client_max_body_size 0;

        chunked_transfer_encoding on;

        location / {
            proxy_pass          http://docker-registry:5000;
            proxy_set_header    Host  $host;
            proxy_read_timeout  900;

            auth_basic            "Restricted";
            auth_basic_user_file  .htpasswd;
        }
    }
}
```

- server_name: Docker 레지스트리 서버의 도메인을 설정합니다. 여기서는 registry.example.com을 설정합니다.
- ssl_certificate, ssl_certificate_key: /etc/server.crt, /etc/server.key를 설정합니다.
- proxy_pass: 리버스 프록시 설정입니다. Docker 레지스트리 컨테이너와 포트인 docker-registry:5000을 설정합니다.
- auth_basic: 인증 설정입니다. “Restricted”를 설정하여 기본 인증을 사용합니다.
- auth_basic_user_file: 사용자 정보가 저장된 .htpasswd 파일을 설정합니다.

1. Docker 개인 저장소 구축하기

4. 기본 인증 사용하기

다음 명령을 실행하여 Docker 레지스트리 컨테이너를 먼저 생성합니다. 컨테이너의 이름은 **nginx.conf** 파일에서 설정한 대로 **docker-registry**로 지정합니다.

```
$ sudo docker run -d --name docker-registry \
    -v /tmp/registry:/tmp/registry \
    registry:0.8.1
```

- Nginx를 통해서 사용자 인증을 하고 이미지를 주고 받을 것이기 때문에 Docker 레지스트리는 5000번 포트를 외부에 노출하지 않습니다.

1. Docker 개인 저장소 구축하기

4. 기본 인증 사용하기

Nginx 공식 이미지 1.7.5 버전으로 컨테이너를 생성하고 **docker-registry** 컨테이너와 연결합니다.

```
$ sudo docker run -d --name nginx-registry \
    -v ~/nginx.conf:/etc/nginx/nginx.conf \
    -v ~/.htpasswd:/etc/nginx/.htpasswd \
    -v ~/server.key:/etc/server.key \
    -v ~/server.crt:/etc/server.crt \
    --link docker-registry:docker-registry \
    -p 443:443 \
    nginx:1.7.5
```

- -v 옵션으로 nginx.conf 파일은 컨테이너의 /etc/nginx/nginx.conf로 연결합니다. 마찬가지로 .htpasswd 파일도 /etc/nginx/.htpasswd로 연결합니다. server.key, server.crt 파일은 앞에서 설정한 것처럼 /etc 디렉터리 아래에 연결합니다.
    - 저는 사용자 계정 디렉터리에 설정 파일과 인증서 파일을 생성했으므로 ~/nginx.conf처럼 ~가 붙습니다. 여러분이 생성한 설정 파일과 인증서 파일의 경로를 절대 경로로 지정합니다.
- --link docker-registry:docker-registry 옵션으로 앞에서 생성한 docker-registry 컨테이너를 docker-registry 별칭으로 연결합니다. 이렇게하면 nginx.conf의 proxy_pass 설정으로 Docker 레지스트리에 트래픽을 보낼 수 있습니다.
- -p 443:443 옵션으로 443번 포트를 외부에 노출합니다.

1. Docker 개인 저장소 구축하기

4. 기본 인증 사용하기

`docker login` 명령으로 **https://registry.example.com**에 로그인합니다. Username과 Password에는 **htpasswd** 명령으로 생성한 사용자와 비밀번호를 입력합니다. 이메일은 입력하지 않아도 됩니다.

```
$ sudo docker login https://registry.example.com
Username: hellouser
Password: <비밀번호 입력>
Email:
Login Succeeded
```

`docker login <Docker 레지스트리 URL>` 형식입니다.

1. Docker 개인 저장소 구축하기

4. 기본 인증 사용하기

이제 앞에서 만든 **hello:0.1** 이미지를 개인 저장소에 올려보겠습니다.

```
$ sudo docker tag hello:0.1 registry.example.com/hello:0.1
$ sudo docker push registry.example.com/hello:0.1
The push refers to a repository [registry.example.com/hello] (len: 1)
Sending image list
Pushing repository registry.example.com/hello (1 tags)
511136ea3c5a: Image successfully pushed
bfb8b5a2ad34: Image successfully pushed
c1f3bdbd8355: Image successfully pushed
897578f527ae: Image successfully pushed
9387bcc9826e: Image successfully pushed
809ed259f845: Image successfully pushed
96864a7d2df3: Image successfully pushed
ba3b051655b4: Image successfully pushed
11ae3a0f7f28: Image successfully pushed
e75f421fc19c: Image successfully pushed
507149de4094: Image successfully pushed
ce11bd8322f9: Image successfully pushed
bb5b14a7e9b8: Image successfully pushed
548b98b8a152: Image successfully pushed
e376f7a8e74c: Image successfully pushed
e2626c81818f: Image successfully pushed
7a06b68da607: Image successfully pushed
Pushing tag for rev [7a06b68da607] on {https://registry.example.com/v1/repositories/hello/tags/0.1}
```

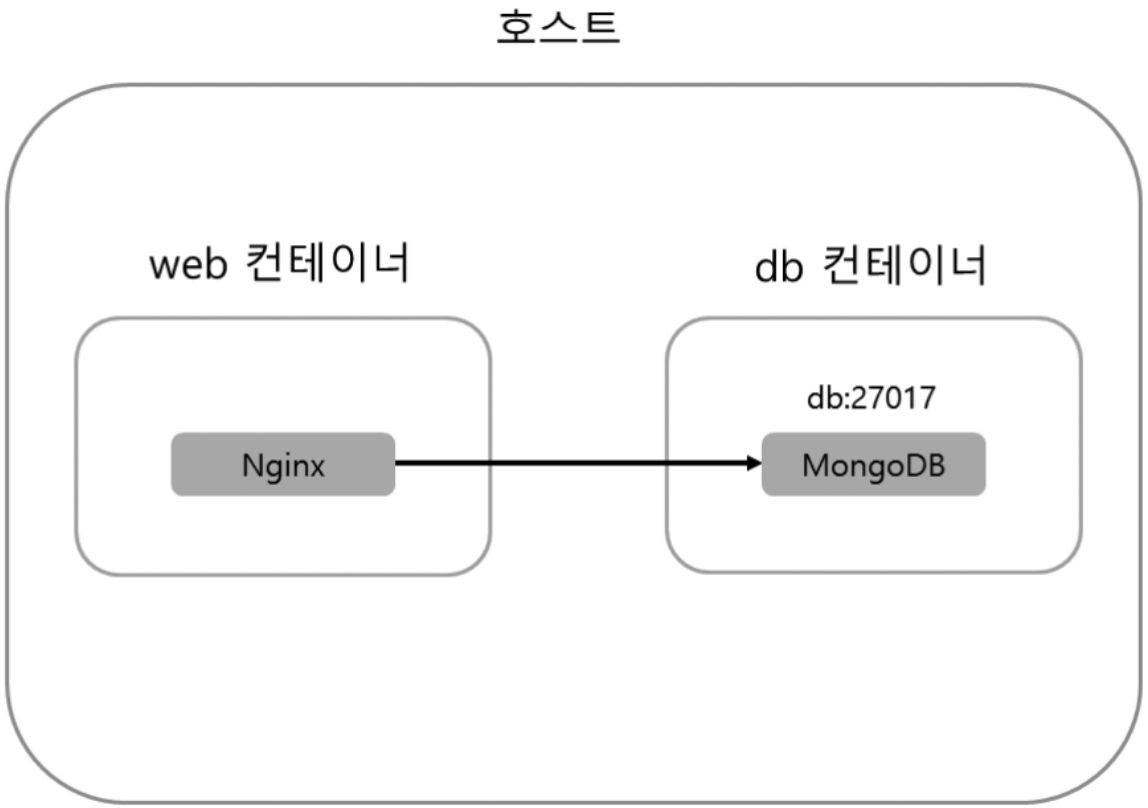
이미지의 태그는 `<Docker 레지스트리 URL>/<이미지 이름>:<태그>` 형식으로 생성합니다. 우리는 **registry.example.com**으로 설정했으므로 **registry.example.com/hello:0.1**이 됩니다.

다른 서버에서는 다음 명령을 실행하여 **registry.example.com**에 저장된 이미지를 받을 수 있습니다.

```
$ sudo docker pull registry.example.com/hello:0.1
```

2. Docker 컨테이너 연결하기

Docker로 이미지를 생성할 때 웹 서버, DB 등 필요한 프로그램을 모두 설치할 수도 있지만 보통 프로그램별로 이미지를 생성합니다. 이렇게 프로그램별로 이미지를 생성하고, 컨테이너를 생성했을 때 옆에 있는 컨테이너에 접속할 일이 많습니다. 예를 들면 웹 서버는 DB에 연결하여 데이터를 주고 받아야 합니다.



2. Docker 컨테이너 연결하기

Docker 컨테이너끼리 연결할 때는 `docker run` 명령에서 `--link` 옵션을 사용합니다. 먼저 DB 이미지를 컨테이너로 실행합니다. 이번에는 MongoDB를 사용해보겠습니다(docker run 명령은 로컬에 이미지가 없으면 자동으로 이미지를 받아옵니다).

```
$ sudo docker run --name db -d mongo
```

DB 컨테이너 이름은 **db**로 설정하였습니다.

이제 web 컨테이너를 생성하면서 **db** 컨테이너와 연결합니다. 웹 서버로 사용할 컨테이너는 **nginx** 이미지로 생성하겠습니다.

```
$ sudo docker run --name web -d -p 80:80 --link db:db nginx
```

2. Docker 컨테이너 연결하기

docker run 명령에서 연결 옵션은 `--link <컨테이너 이름>:<별칭>` 형식입니다.

컨테이너 목록을 출력합니다.

```
$ sudo docker ps
CONTAINER ID  IMAGE         COMMAND               CREATED      STATUS        PORTS               NAMES
3971618834cd  nginx:latest  nginx                 About a min  Up About a m  0.0.0.0:80->80/tcp  web
8d4031106c57  mongo:2.6     /usr/src/mongo/docke  4 minutes a  Up 4 minutes  27017/tcp           db,web/db
```

db 컨테이너와 **web** 컨테이너가 연결되었습니다. **web/db**라고 표시되는데 **web** 컨테이너에서 **db** 컨테이너에 접속할 수 있다는 것입니다.

이제 **web** 컨테이너 안에서 `db:27017` 주소로 **db** 컨테이너의 MongoDB에 접속할 수 있습니다.

```
mongodb://db:27017/exampledb
```

컨테이너 안에서 다른 컨테이너에 접속할 때는 `<별칭>:<포트 번호>` 형식으로 사용합니다.

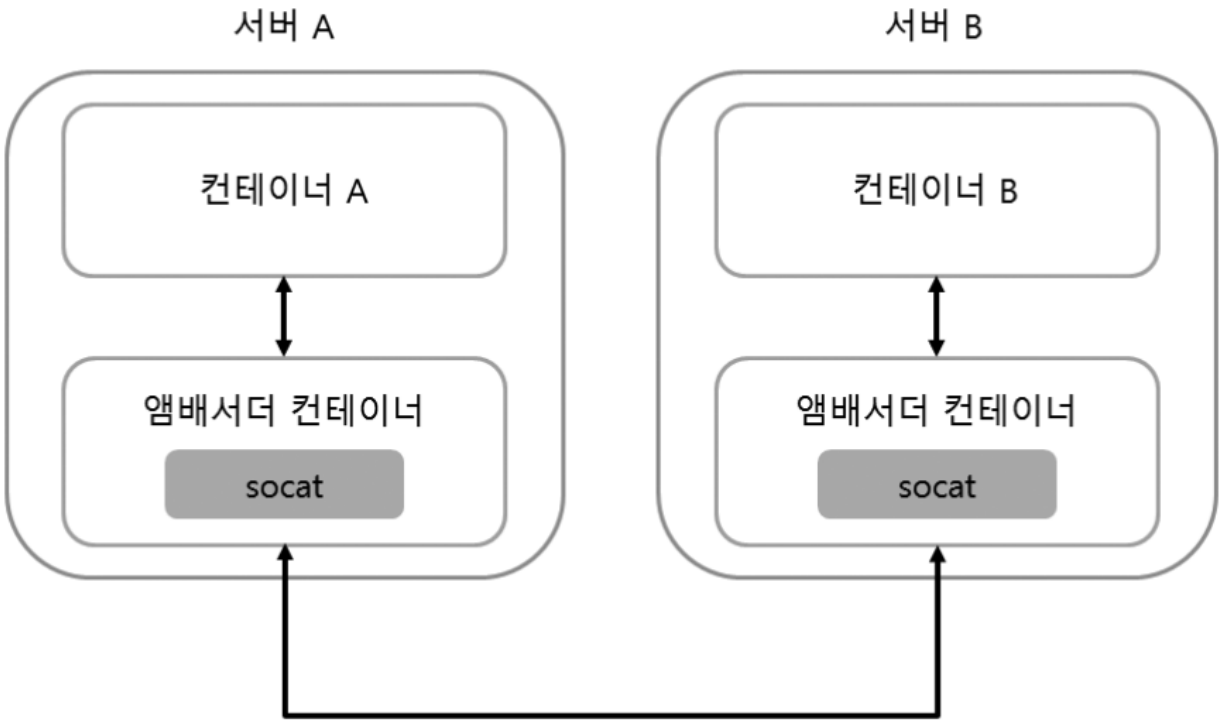
별칭과 IP 주소

다음과 같이 `docker inspect` 명령으로 web 컨테이너의 세부 정보에서 `hosts` 파일 경로를 구한 뒤 `cat` 명령으로 내용을 살펴봅니다(`는 탭키 위에 있는 문자입니다).

```
$ cat `sudo docker inspect -f "{{ .HostsPath }}" web`
172.17.0.13     aa1982fed33e
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
172.17.0.12     db
```

3. 다른 서버의 Docker 컨테이너 연결하기

앞에서 설명한 - link 옵션은 같은 서버의 컨테이너끼리 연결하는 옵션입니다. 이번에는 앰배서더 컨테이너(Ambassador Container)라는 것을 이용하여 다른 서버에 있는 컨테이너에 연결해보겠습니다.
앰배서더 컨테이너는 특별한 컨테이너가 아닌 그냥 일반적인 Docker 컨테이너입니다. 앰배서더 컨테이너는 socat이라는 프로그램을 이용하여 TCP 연결을 다른 곳으로 전달하도록 구성되어 있습니다.



3. 다른 서버의 Docker 컨테이너 연결하기

앰배서더 컨테이너의 Dockerfile을 보면 상당히 복잡하게 보이지만 생각보다 간단합니다. `docker run` 명령을 실행할 때 전달한 환경 변수를 이용하여 socat을 실행하는 셸 스크립트입니다.

```
CMD env | grep _TCP= | \
    sed 's/.*_PORT_\([0-9]*\)_TCP=tcp:\/\/\(.*\):\(.*\)/socat \
    TCP4-LISTEN:\1,fork,reuseaddr TCP4:\2:\3 \&/'  \
    | sh && top
```

`docker run` 명령에서 `--link` 옵션을 사용하거나 `-e EXAMPLE_PORT_1234_TCP=tcp://192.168.0.10:1234`라고 설정해 주면 다음과 같이 환경 변수에 포트 정보가 설정됩니다.

`env` 명령으로 환경 변수를 출력하고, `grep` 명령으로 *_TCP=*를 포함하는 문자열을 찾습니다. 그리고 `sed` 명령으로 정규 표현식을 사용하여 문자열에서 포트 번호와 IP 주소를 추출합니다. 그 뒤 추출한 포트 번호와 IP 주소를을 이용하여 socat 명령을 실행합니다.

```
EXAMPLE_PORT=tcp://192.168.0.10:1234
EXAMPLE_PORT_1234_TCP_ADDR=192.168.0.10
EXAMPLE_NAME=/example_ambassador/example
HOSTNAME=0cf479687cb0
EXAMPLE_PORT_1234_TCP_PORT=1234
HOME=/
EXAMPLE_PORT_1234_TCP_PROTO=tcp
EXAMPLE_PORT_1234_TCP=tcp://192.168.0.10:1234
TERM=xterm
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
```

위 예제 환경에서는 `socat` 명령으로 로컬의 TCP 프로토콜 **1234**번 포트를 **192.168.0.10**의 **1234**번 포트로 데이터를 전달하도록 설정합니다. 이러한 구조를 앰배서더 패턴(Ambassador Pattern)이라 합니다.

3. 다른 서버의 Docker 컨테이너 연결하기

먼저 Redis 서버로 쓸 컴퓨터에서 Redis 컨테이너를 생성합니다. 이 서버의 IP 주소는 192.168.0.10이라 하겠습니다.

```
$ sudo docker pull redis:latest
$ sudo docker run -d --name redis redis:latest
```

`--name redis` 옵션으로 컨테이너 이름을 **redis**로 지정합니다.

Redis 컨테이너를 위한 앰배서더 컨테이너를 생성합니다. svendowideit는 앰배서더 패턴을 만든 사람 이름입니다.

```
$ sudo docker run -d --link redis:redis --name redis_ambassador \
    -p 6379:6379 svendowideit/ambassador
```

- `-d` 옵션으로 컨테이너를 백그라운드로 실행합니다.
- `--link redis:redis` 옵션으로 **redis** 컨테이너를 **redis** 별칭으로 연결합니다.
- `--name redis_ambassador` 옵션으로 컨테이너 이름을 **redis_ambassador**로 지정합니다.
- `-p 6379:6379` 옵션으로 컨테이너의 **6379**번 포트와 호스트의 **6379** 연결하고 외부에 노출합니다.
- Docker Hub에 있는 **svendowideit/ambassador** 이미지를 받은 뒤 컨테이너로 생성합니다(docker run 명령은 로컬에 이미지가 없으면 자동으로 이미지를 받아옵니다).

3. 다른 서버의 Docker 컨테이너 연결하기

이제 Redis 클라이언트를 사용할 컴퓨터에서 앰배서더 컨테이너를 생성합니다. Redis 서버의 IP 주소는 **192.168.0.10**입니다.

```
$ sudo docker run -d --name redis_ambassador --expose 6379 \
    -e REDIS_PORT_6379_TCP=tcp://192.168.0.10:6379 svendowideit/ambassador
```

- `-d` 옵션으로 컨테이너를 백그라운드로 실행합니다.
- `--name redis_ambassador` 옵션으로 컨테이너 이름을 **redis_ambassador**로 지정합니다.
- `--expose 6379` 옵션으로 다른 컨테이너에서 6379번 포트에 연결할 수 있도록 설정합니다. 즉 Redis 클라이언트가 이 **redis_ambassador** 컨테이너의 6379 포트에 접속하게 됩니다. `--expose` 옵션은 `-p` 옵션과는 달리 호스트의 포트를 외부에 노출하지 않습니다.
- `-e REDIS_PORT_6379_TCP=tcp://192.168.0.10:6379` 옵션으로 IP 주소와 포트를 설정하여 다른 서버에 있는 **redis_ambassador** 컨테이너에 연결합니다.
- Docker Hub에 있는 svendowideit/ambassador 이미지 받은 뒤 컨테이너로 생성합니다.

Docker Hub에 Redis 클라이언트만 들어있는 컨테이너도 있습니다. 이 컨테이너를 이용하여 **redis_ambassador** 컨테이너에 접속합니다.

```
$ sudo docker run -i -t --rm --link redis_ambassador:redis relateiq/redis-cli
```

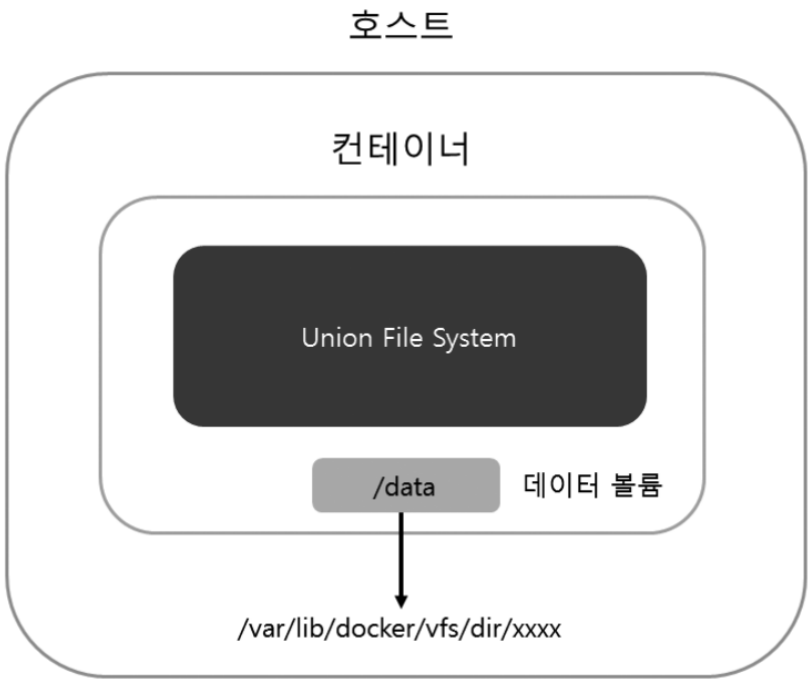
- `-i -t` 옵션으로 콘솔에서 입출력을 할 수 있도록 설정합니다.
- `--rm` 옵션으로 컨테이너를 실행만 하고 컨테이너 자체는 삭제합니다. Redis 클라이언트처럼 1회성으로 사용할 때 편리합니다.
- `--link redis_ambassador:redis` 옵션으로 **redis_ambassador** 컨테이너를 **redis** 별칭으로 연결합니다.
- Docker Hub에 있는 **relateiq/redis-cli** 이미지를 받은 뒤 컨테이너로 생성합니다.

Redis 클라이언트가 실행되면 `ping` 명령을 입력합니다. 결과로 **PONG**이 출력되면 다른 서버의 Redis 컨테이너에 정상적으로 연결된 것입니다.

```
redis 172.17.0.4:6379> ping
PONG
redis 172.17.0.4:6379>
```

4. Docker 데이터 볼륨 사용하기

Docker 데이터 볼륨은 데이터를 컨테이너가 아닌 호스트에 저장하는 방식입니다. 따라서 데이터볼륨은 컨테이너끼리 데이터를 공유할 때 활용할 수 있습니다.
Docker 컨테이너 안의 파일 변경 사항은 Union File System에 의해 관리됩니다. 하지만 데이터 볼륨은 Union File System을 통하지 않고 바로 호스트에 저장됩니다. 따라서 docker commit 명령을 통해 이미지로 생성해도 데이터 볼륨의 변경 사항은 이미지에 포함되지 않습니다.



4. Docker 데이터 볼륨 사용하기

다음 명령을 입력하면 컨테이너 안의 **/data** 디렉터리가 데이터 볼륨으로 설정됩니다. 컨테이너의 Bash 셸이 실행되면 **/data** 디렉터리로 이동한 뒤 **hello**라는 빈 파일을 생성합니다. 그리고 **exit**를 입력하여 Bash 셸에서 빠져나옵니다.

```
$ sudo docker run -i -t --name hello-volume -v /data ubuntu /bin/bash
root@019265a6920f:/# cd /data
root@019265a6920f:/data# touch hello
root@019265a6920f:/data# exit
```

데이터 볼륨 옵션은 `-v <컨테이너 디렉터리>` 형식입니다.

`docker inspect` 명령으로 **hello-volume** 컨테이너의 데이터 볼륨 경로를 확인합니다.

```
$ sudo docker inspect -f "{{ .Volumes }}" hello-volume
map[/data:/var/lib/docker/vfs/dir/0e3cacb43f11d42b4a6f186198e3e2c812a8ff62c0ab7472d18e1a9735093ae1]
```

`ls` 명령으로 앞에서 알아낸 디렉터리(/var/lib/docker/vfs/dir/xxxx)안의 파일 목록을 출력합니다. 이 디렉터리는 컨테이너를 생성할 때마다 함께 생성됩니다.

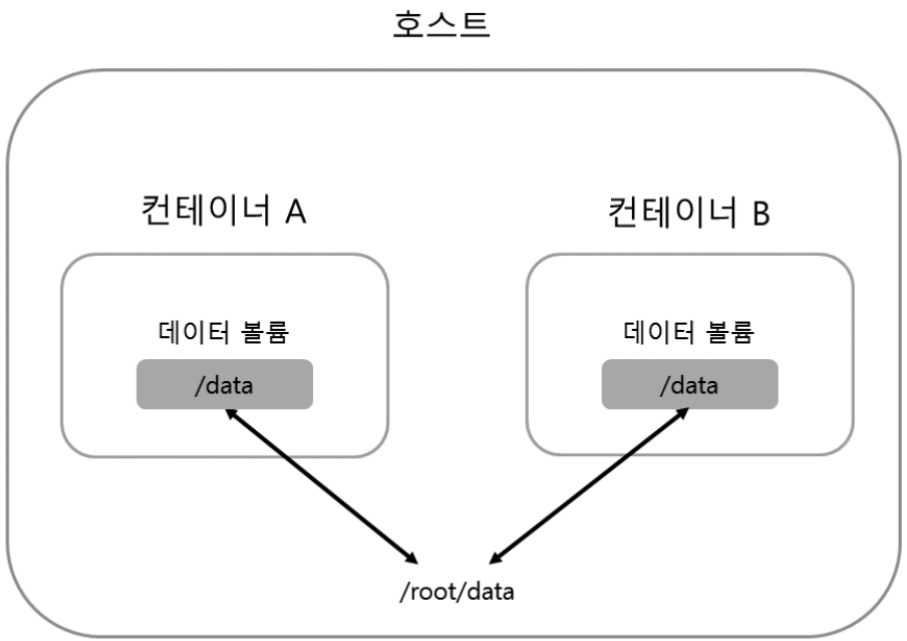
```
$ sudo ls /var/lib/docker/vfs/dir/0e3cacb43f11d42b4a6f186198e3e2c812a8ff62c0ab7472d18e1a9735093ae1
hello
```

앞에서 생성한 **hello** 파일이 보입니다. 이 디렉터리에 파일을 생성하면 컨테이너 안에서도 사용할 수 있습니다.

4. Docker 데이터 볼륨 사용하기

다음 명령을 실행하여 컨테이너를 생성하고 데이터 볼륨을 설정합니다. 컨테이너의 Bash 셸이 실행되면 **/data** 디렉터리로 이동한 뒤 **world**라는 빈 파일을 생성합니다. 그리고 **exit**를 입력하여 Bash 셸에서 빠져나옵니다.

```
$ sudo docker run -i -t --name hello-volume1 -v /root/data:/data ubuntu /bin/bash
root@f7baf3abefee:/# cd /data
root@f7baf3abefee:/data# touch world
root@f7baf3abefee:/data# exit
```



4. Docker 데이터 볼륨 사용하기

데이터 볼륨 옵션은 `-v <호스트 디렉터리>:<컨테이너 디렉터리>` 형식입니다. 여기서는 호스트의 **/root/data** 디렉터리를 Docker 컨테이너의 **/data** 디렉터리에 연결합니다.

/root/data 디렉터리의 파일 목록을 출력합니다.

```
$ sudo ls /root/data
world
```

앞에서 생성한 **world** 파일이 보입니다.

이제 두 번째 컨테이너를 생성합니다. 컨테이너의 Bash 셸이 실행되면 **/data** 디렉터리의 파일 목록을 출력합니다.

```
$ sudo docker run -i -t --name hello-volume2 -v /root/data:/data ubuntu /bin/bash
root@af5a7bdb3e5a:/# ls /data
world
```

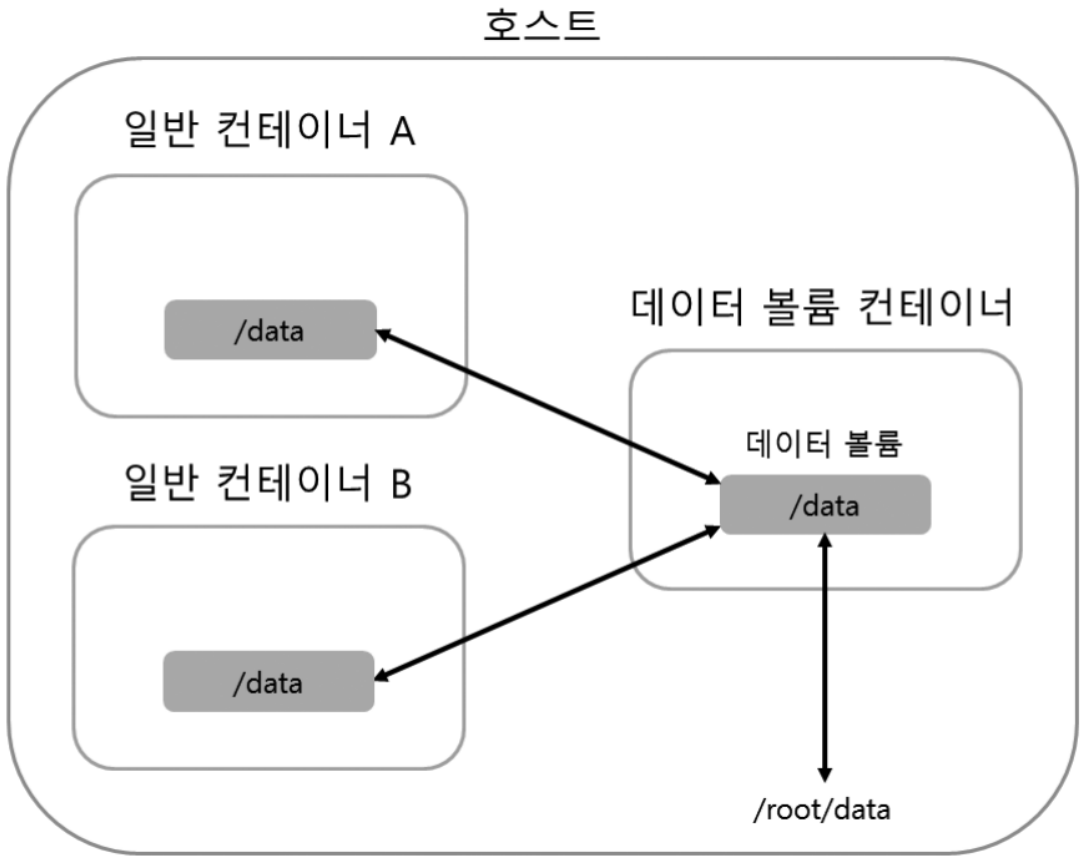
앞에서 생성한 **world** 파일이 **hello-volume2** 파일에서도 보입니다. **/data** 디렉터리에 파일을 생성하면 호스트 및 **hello-volume1** 컨테이너에서도 사용할 수 있습니다. 이렇게 데이터 볼륨 설정을 통해 컨테이너끼리 데이터를 공유할 수 있습니다.

디렉터리뿐만 아니라 호스트의 파일 하나만 컨테이너에 연결할 수도 있습니다.

```
$ sudo docker run -i -t --name hello-volume -v /root/hello.txt:/root/hello.txt \
    ubuntu /bin/bash
```

5. Docker 데이터 볼륨 컨테이너 사용하기

앞에서 데이터 볼륨을 사용하는 방법을 알아보았습니다. 데이터 볼륨 컨테이너는 데이터 볼륨을 설정한 컨테이너를 뜻합니다. 일반 컨테이너에서 데이터 볼륨 컨테이너를 연결하면 데이터 볼륨 컨테이너 안의 데이터 볼륨 디렉터리에 접근할 수 있습니다.



5. Docker 데이터 볼륨 컨테이너 사용하기

다음 명령을 입력하여 데이터 볼륨 컨테이너를 생성합니다(컨테이너 이름이 중복되면 기존 컨테이너는 `docker rm` 명령으로 삭제합니다). 컨테이너의 Bash 셸이 실행되면 **/data** 디렉터리로 이동한 뒤 **hello2**라는 빈 파일을 생성합니다. 그리고 **Ctrl+P**, **Ctrl+Q**를 차례대로 입력하여 컨테이너를 정지하지 않고 Bash 셸에서 빠져나옵니다.

```
$ sudo docker run -i -t --name hello-volume -v /root/data:/data ubuntu /bin/bash
root@c9779e329513:/# cd /data
root@c9779e329513:/data# touch hello2
```

일반 컨테이너를 생성하면서 방금 생성한 **hello-volume** 데이터 볼륨 컨테이너를 연결합니다. 컨테이너의 Bash 셸이 실행되면 **/data** 디렉터리의 파일 목록을 출력합니다.

```
$ sudo docker run -i -t --volumes-from hello-volume --name hello ubuntu /bin/bash
root@c85aaf93b14e:/# ls /data
hello2
```

데이터 볼륨 컨테이너를 연결하는 옵션은 `--volumes-from <데이터 볼륨 컨테이너>` 형식입니다.

이제 데이터 볼륨 컨테이너에서 생성한 hello2 파일이 보입니다(호스트의 **/root/data**에 연결했기 때문에 앞에서 생성한 다른 파일들이 보일 수도 있습니다).

지금은 일반 컨테이너를 하나만 연결했지만 데이터 볼륨 컨테이너에 일반 컨테이너를 여러 개 연결해도 됩니다.

다음 명령처럼 **/data** 디렉터리를 호스트의 특정 디렉터리에 연결하지 않아도 데이터 볼륨 컨테이너로 사용할 수 있습니다.

```
$ sudo docker run -i -t --name hello-volume -v /data ubuntu /bin/bash
```

6. Docker 베이스 이미지 생성하기

보통 Dockerfile로 이미지를 생성할 때 Docker Hub에서 제공하는 공식 이미지를 기반으로 생성합니다. 이번에는 나만의 베이스 이미지를 생성하는 방법을 알아보겠습니다.

1. 우분투 베이스 이미지 생성하기

우분투 리눅스 베이스 이미지를 생성해보겠습니다. 우분투 리눅스용 부트스트랩 바이너리 파일을 받아와야 하기 때문에 우분투 리눅스가 설치된 호스트에서 진행합니다.

먼저 부트스트랩 도구인 `debootstrap`을 설치합니다.

```
$ sudo apt-get install debootstrap
```

debootstrap으로 우분투 **trusty(14.04)** 바이너리 파일을 받습니다. **trusty**는 우분투 리눅스의 코드네임입니다.

```
$ sudo debootstrap trusty trusty
```

`deboostrap <코드네임> <디렉터리>` 형식입니다.

바이너리 파일을 다 받았으면 `docker import` 명령으로 베이스 이미지를 생성합니다.

```
$ sudo tar -C trusty -c . | sudo docker import - trusty
```

`tar -C trusty -c .`는 **trusty** 디렉터리의 내용을 파일 하나로 합쳐서 stdout으로 출력하는 명령입니다. 이 출력 내용을 | (파이프)를 통해 `docker import` 명령에 전달합니다.

6. Docker 베이스 이미지 생성하기

1. 우분투 베이스 이미지 생성하기

`docker import <URL 또는 -> <이미지 이름>:<태그>` 형식입니다. 다음과 같이 인터넷에 있는 파일을 사용할 수 있습니다. |를 통해 데이터를 넘겨받는다면 -를 지정합니다.

```
$ sudo docker import http://example.com/trusty.tgz trusty
```

이미지 목록을 출력합니다.

```
$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
trusty              latest              5857e0393148        5 seconds ago       228.3 MB
```

trusty 이미지가 생성되었습니다. **trusty** 이미지로 컨테이너를 생성합니다. 컨테이너의 Bash 셸이 실행되면 **/etc/lsb-release** 파일의 내용을 확인해봅니다.

```
$ sudo docker run -i -t --name hello trusty /bin/bash
root@158ea15ee10c:/# cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=14.04
DISTRIB_CODENAME=trusty
DISTRIB_DESCRIPTION="Ubuntu 14.04 LTS"
```

코드네임은 **trusty**, 릴리스 버전은 **14.04**로 표시됩니다.

6. Docker 베이스 이미지 생성하기

2. CentOS 베이스 이미지 생성하기

이번에는 CentOS 베이스 이미지를 생성해보겠습니다. CentOS용 부트스트랩 바이너리 파일을 받아와야 하기 때문에 CentOS가 설치된 호스트에서 진행합니다. 배포판 버전은 CentOS 6.5 기준입니다.

먼저 부트스트랩 도구인 `febootstrap`을 설치합니다.

```
$ sudo yum install febootstrap
```

febootstrap으로 **CentOS 6.5** 바이너리 파일을 받습니다.

```
$ sudo febootstrap -u http://ftp.kaist.ac.kr/CentOS/6.5/updates/x86_64/ \
    centos65 centos65 http://ftp.kaist.ac.kr/CentOS/6.5/os/x86_64/
```

`feboostrap <옵션> <저장소> <디렉터리> <미러 URL>` 형식입니다.

바이너리 파일을 다 받았으면 `docker import` 명령으로 베이스 이미지를 생성합니다.

```
$ sudo tar -C centos65 -c . | sudo docker import - centos65
```

`tar -C centos65 -c .`는 `centos65` 디렉터리의 내용을 파일 하나로 합쳐서 stdout으로 출력하는 명령입니다. 이 출력 내용을 | (파이프)를 통해 `docker import` 명령에 전달합니다.

6. Docker 베이스 이미지 생성하기

2. CentOS 베이스 이미지 생성하기

이번에는 CentOS 베이스 이미지를 생성해보겠습니다. CentOS용 부트스트랩 바이너리 파일을 받아와야 하기 때문에 CentOS가 설치된 호스트에서 진행합니다. 배포판 버전은 CentOS 6.5 기준입니다.

`docker import <tar 파일 URL 또는 -> <이미지 이름>:<태그>` 형식입니다. 다음과 같이 인터넷에 있는 파일을 사용할 수 있습니다. |(파이프)를 통해 데이터를 넘겨받는다면 -를 지정합니다.

```
$ sudo docker import http://example.com/centos65.tgz centos65
```

이미지 목록을 출력합니다.

```
$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
centos65            latest              8da697bd579e        8 minutes ago       429.9 MB
```

centos65 이미지가 생성되었습니다. **centos65** 이미지로 컨테이너를 생성합니다. 컨테이너의 Bash 셸이 실행되면 **/etc/centos-release** 파일의 내용을 확인해봅니다.

```
$ sudo docker run -i -t --name hello centos65 /bin/bash
bash-4.1# cat /etc/centos-release
CentOS release 6.5 (Final)
```

배포판 버전이 **CentOS release 6.5 (Final)**로 표시됩니다.

6. Docker 베이스 이미지 생성하기

3. 빈 베이스 이미지 생성하기

아무것도 들어있지 않은 베이스 이미지를 생성하는 방법입니다. Docker에서는 빈 베이스 이미지를 scratch 이미지라고 부릅니다.

/dev/null 장치를 이용하여 빈 tar 파일을 만들어서 `docker import` 명령에 전달합니다.

```
$ tar cv --files-from /dev/null | sudo docker import - scratch
```

scratch 이미지는 안에 아무것도 없기 때문에 컨테이너로 생성이 되지 않습니다. 여기서 Dockerfile을 작성하여 여러분이 만든 실행 파일을 넣으면 됩니다.

간단하게 C언어로 만든 프로그램을 scratch 이미지에 넣어보겠습니다. 먼저 **hello** 디렉터리를 생성한 뒤 **hello** 디렉터리로 이동합니다.

```
$ mkdir hello
$ cd hello
```

다음 내용을 **hello.c**로 저장합니다.

hello.c

```c
#include <stdio.h>

int main ()
{
    printf("Hello Docker\n");
    return 0;
}
```

hello.c 파일을 컴파일하여 실행 파일로 만듭니다. **scratch** 이미지에는 아무 라이브러리도 없으므로 반드시 정적(static) 바이너리로 컴파일해야 합니다.

```
~/hello$ gcc hello.c -static -o hello
```

6. Docker 베이스 이미지 생성하기

3. 빈 베이스 이미지 생성하기

아무것도 들어있지 않은 베이스 이미지를 생성하는 방법입니다. Docker에서는 빈 베이스 이미지를 scratch 이미지라고 부릅니다.

다음 내용을 Dockerfile로 저장합니다.

Dockerfile

```
FROM scratch
ADD ./hello /hello
CMD ["/hello"]
```

scratch 이미지를 기반으로 새로운 이미지를 생성합니다.

- FROM: 어떤 이미지를 기반으로 할지 설정합니다. Docker 이미지는 기존에 만들어진 이미지를 기반으로 생성합니다. `<이미지 이름>:<태그>` 형식입니다. 여기서는 앞에서 만든 scratch 이미지를 설정합니다.
- ADD: 이미지에 포함할 파일을 설정합니다. `<로컬 경로> <이미지 경로>` 형식입니다. 앞에서 **hello.c** 파일을 컴파일하여 생성한 **hello** 파일을 설정합니다.
- CMD: 컨테이너가 시작되었을 때 실행할 실행 파일 또는 스크립트입니다. **hello** 파일이 실행되도록 설정합니다.

`docker build` 명령으로 이미지를 생성합니다.

```
~/hello$ sudo docker build --tag hello:0.1 .
```

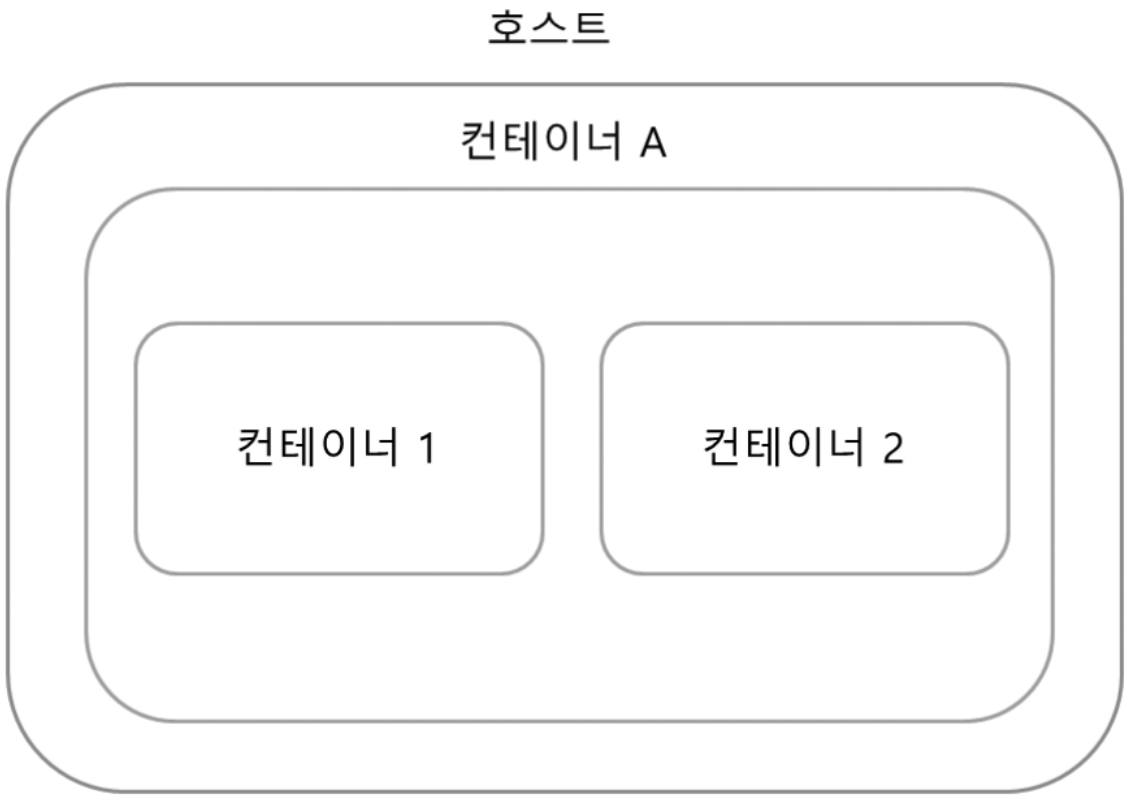
이제 **scratch** 이미지를 이용해서 만든 **hello:0.1** 이미지를 컨테이너로 생성합니다.

```
$ sudo docker run --rm hello:0.1
Hello Docker
```

Hello Docker가 출력되면 실행 파일이 정상적으로 실행된 것입니다.

7. Docker 안에서 Docker 실행하기

Docker 컨테이너 안에서 Docker를 실행하는 방법입니다. 복잡하게 왜 Docker 컨테이너 안에서 Docker를 실행할까요? 예를 들면 Jenkins나 CruiseControl과 같은 빌드 자동화 시스템을 이용해서 Docker 이미지를 생성할 때 활용할 수 있습니다. Jenkins, CruiseControl 환경 자체도 Docker 이미지로 만들면 Docker 컨테이너 안에서 Docker를 실행할 수 있어야 합니다.



7. Docker 안에서 Docker 실행하기

먼저 GitHub에서 Dockerfile과 Bash 스크립트를 받습니다.

```
$ git clone https://github.com/pyrasis/dind.git
```

dind 디렉터리로 이동한 뒤 `docker build` 명령으로 이미지를 생성합니다.

```
~$ cd dind
~/dind$ sudo docker build --tag dind .
```

잠시 기다리면 이미지가 생성됩니다. 다음 명령을 실행하여 **dind** 이미지로 컨테이너를 생성합니다.

```
~/dind$ sudo docker run -i -t --privileged dind
root@ee112b504b98:/# 2014/08/16 17:20:23 docker daemon: 1.1.2 d84a070; execdriver: native; graphdriver:
[48756c49] +job initserver()
[48756c49.initserver()] Creating server
[48756c49] +job serveapi(unix:///var/run/docker.sock)
2014/08/16 17:20:23 Listening for HTTP on unix (/var/run/docker.sock)
[48756c49] +job init_networkdriver()
[48756c49.init_networkdriver()] creating new bridge for docker0
[48756c49.init_networkdriver()] getting iface addr
[48756c49] -job init_networkdriver() = OK (0)
2014/08/16 17:20:23 WARNING: Your kernel does not support cgroup swap limit.
Loading containers: : done.
[48756c49.initserver()] Creating pidfile
[48756c49.initserver()] Setting up signal traps
[48756c49] -job initserver() = OK (0)
[48756c49] +job acceptconnections()
[48756c49] -job acceptconnections() = OK (0)

root@ee112b504b98:/#
```

여기서 `--privileged` 옵션이 중요합니다. 이 옵션은 컨테이너 안에서 호스트의 리눅스 커널 기능을 모두 사용할 수 있도록 해줍니다.

7. Docker 안에서 Docker 실행하기

Docker in Docker는 실험적인 기능이기 때문에 로그를 출력하도록 설정되어 있습니다. 로그를 출력하지 않으려면 다음과 같이 `-e LOG=file` 옵션을 사용하면 됩니다.

```
$ sudo docker run -i -t --privileged -e LOG=file dind
```

이제 Docker 컨테이너 안에서 Docker를 실행해보겠습니다. 다음 명령을 입력하여 busybox를 실행해봅니다.

```
root@ee112b504b98:/# sudo docker run -i -t busybox:latest /bin/sh
Unable to find image 'busybox:latest' locally
Pulling repository busybox
a9eb17255234: Download complete
511136ea3c5a: Download complete
42eed7f1bf2a: Download complete
120e218dd395: Download complete
/ #
```

이렇게 호스트 → **dind** 컨테이너 → **busybox** 컨테이너 순서로 실행이 되었습니다.