

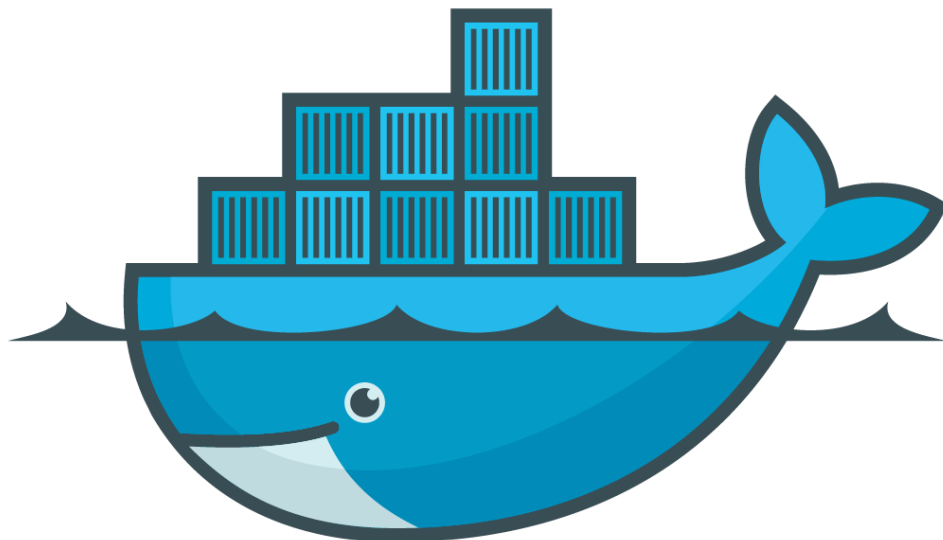
초보를 위한 도커 안내서 - 이미지 만들고 배포하기

SERIES 3/3

subicura on 10 Feb 2017

좋아요 171개

공유하기



docker

docker logo

이 글은 초보를 위한 도커 안내서 - 설치부터 배포까지 시리즈의 마지막 글입니다. 지난 글에서 도커를 설치하고 컨테이너를 실행해 보았으니 이번엔 이미지를 만들고 서버에 배포해보도록 하겠습니다.

[초보를 위한 도커 안내서 - 도커란 무엇인가?](#) SERIES 1/3

- [초보를 위한 도커 안내서 - 설치하고 컨테이너 실행하기](#) SERIES 2/3
- [초보를 위한 도커 안내서 - 이미지 만들고 배포하기 ✓](#) SERIES 3/3

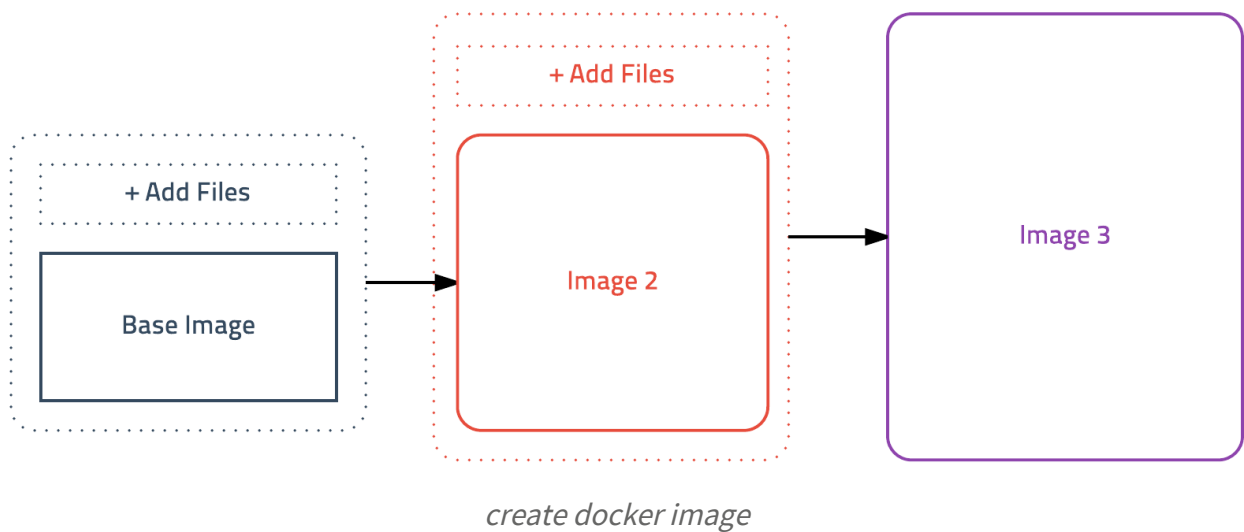


광고 innorix.com

더보기 ▼

도커 이미지 만들기

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



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

입니다.

이런 과정은 콘솔에서 명령어를 직접 입력하는 것과 별 차이가 없으므로 셸 스크립트를 잘 알아야 하지만 좋은 샘플이 많이 공개되어 있어 잘 몰라도 크게 걱정하지 않아도 됩니다. ~~복불만~~ 또한 컨테이너의 가벼운 특성과 레이어 개념을 이용하여 생성과 테스트를 빠르게 수행할 수 있습니다.

이제 Ruby로 만들어진 간단한 웹 애플리케이션을 도커라이징(Dockerizing=도커 이미지를 만듦)해보겠습니다.

Sinatra 웹 애플리케이션 샘플



Sinatra

일단 웹 애플리케이션 소스코드를 작성해야겠죠. Sinatra라는 가벼운 웹 프레임워크를 사용하기 위해 새로운 폴더를 만들고 `Gemfile` 과 `app.rb` 를 만듭니다.

Gemfile

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

app.rb

```

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

```

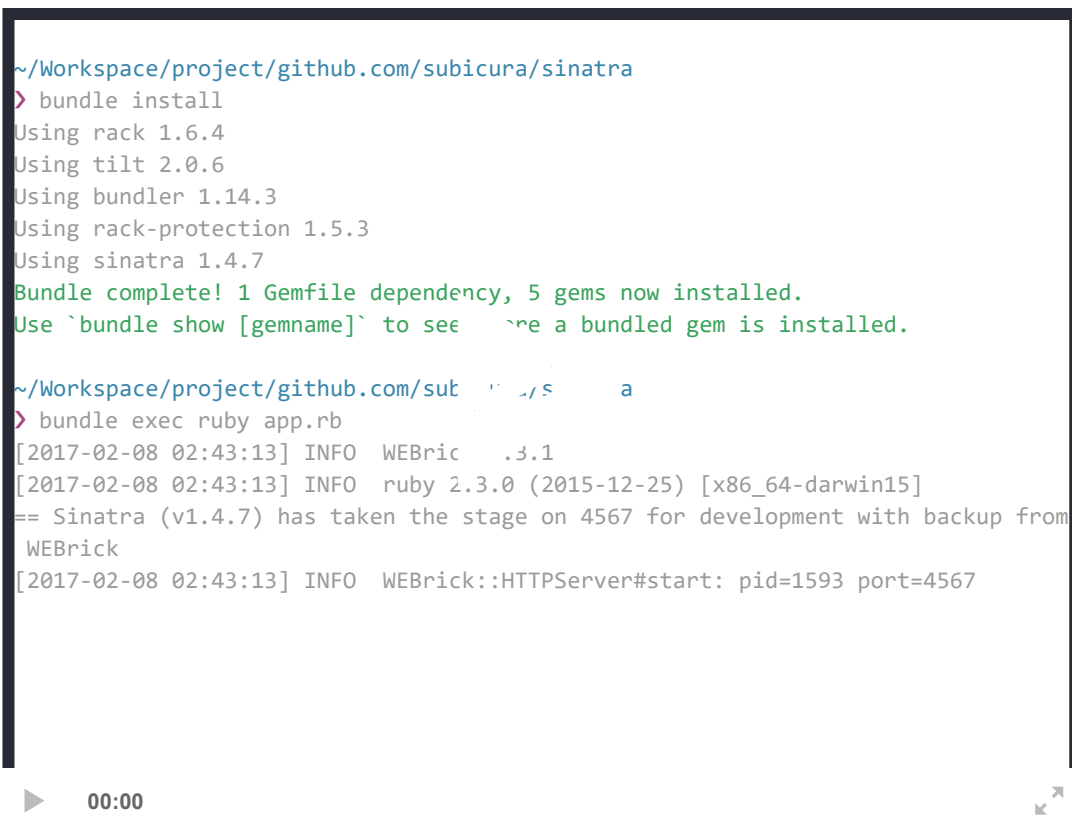
ruby와 sinatra에 대해 전혀 모르더라도 `Gemfile` 은 패키지를 관리하고 `app.rb` 는 호스트 명을 출력하는 웹 서버를 만들었다는 걸 예상할 수 있습니다.

이제 패키지를 설치하고 서버를 실행해보겠습니다.

```

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

```



```

~/Workspace/project/github.com/subicura/sinatra
> bundle install
Using rack 1.6.4
Using tilt 2.0.6
Using bundler 1.14.3
Using rack-protection 1.5.3
Using sinatra 1.4.7
Bundle complete! 1 Gemfile dependency, 5 gems now installed.
Use `bundle show [gemname]` to see where a bundled gem is installed.

~/Workspace/project/github.com/subicura/sinatra
> bundle exec ruby app.rb
[2017-02-08 02:43:13] INFO WEBrick 1.3.1
[2017-02-08 02:43:13] INFO ruby 2.3.0 (2015-12-25) [x86_64-darwin15]
== Sinatra (v1.4.7) has taken the stage on 4567 for development with backup from WEBrick
[2017-02-08 02:43:13] INFO WEBrick::HTTPServer#start: pid=1593 port=4567

```

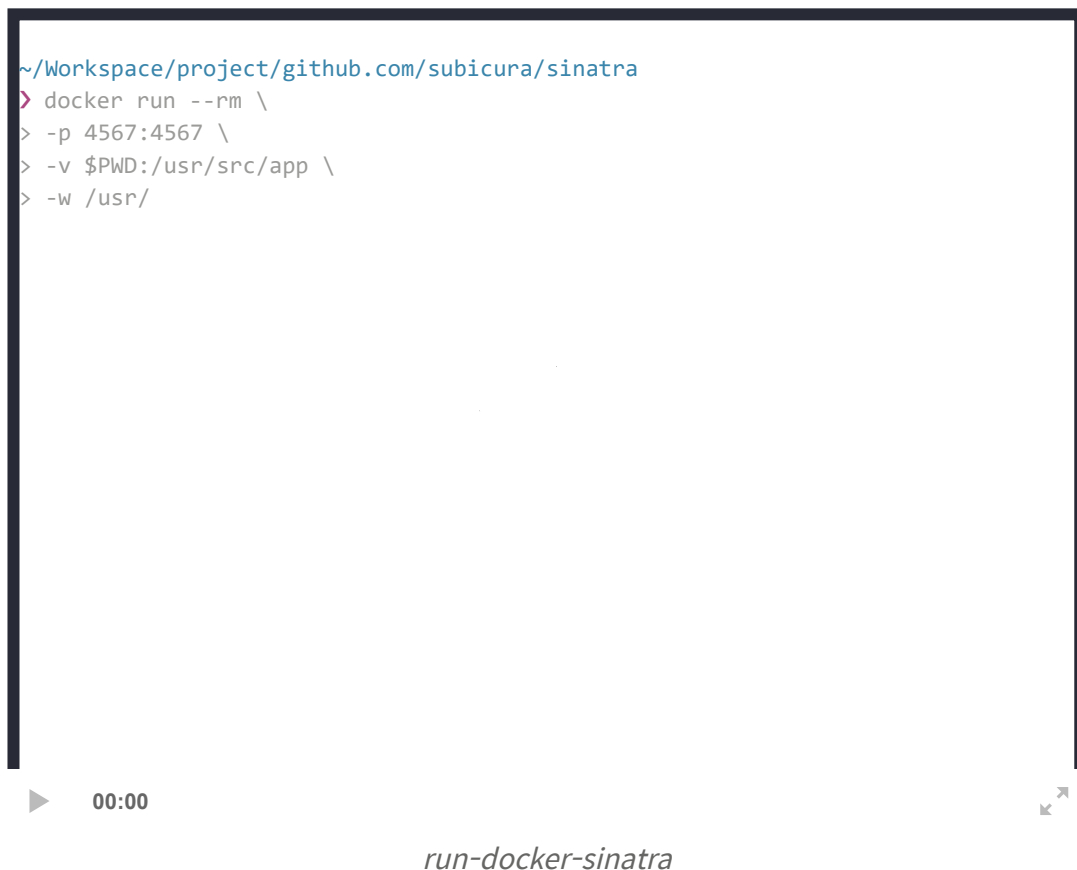
run-sinatra

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

```

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

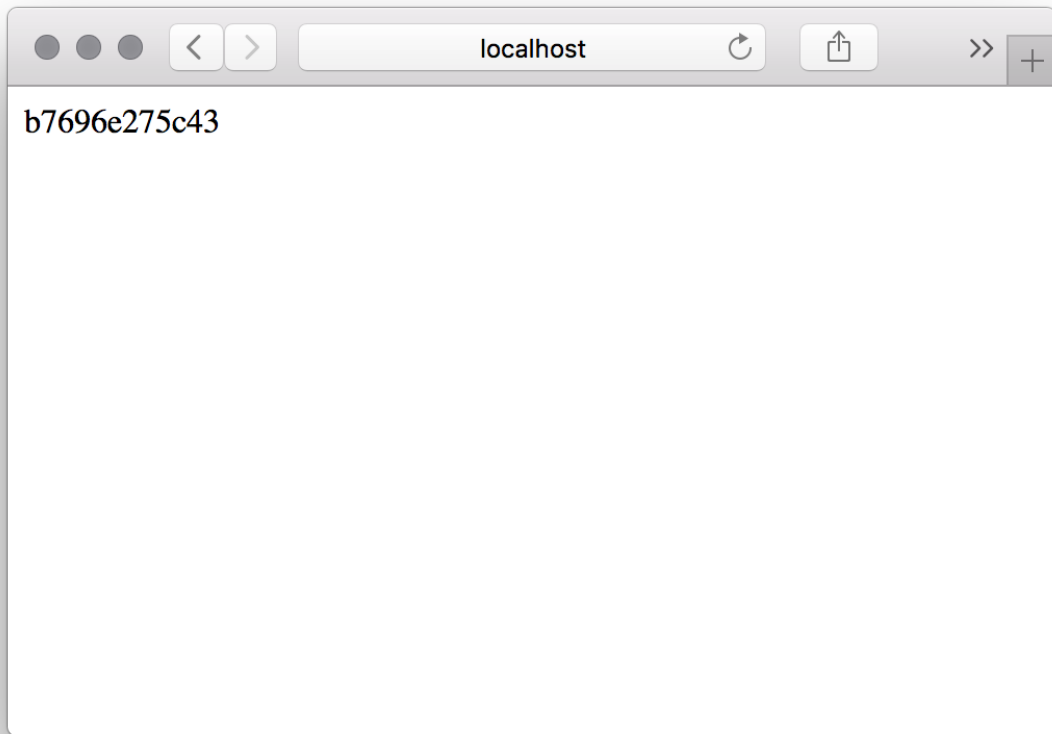
```



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

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

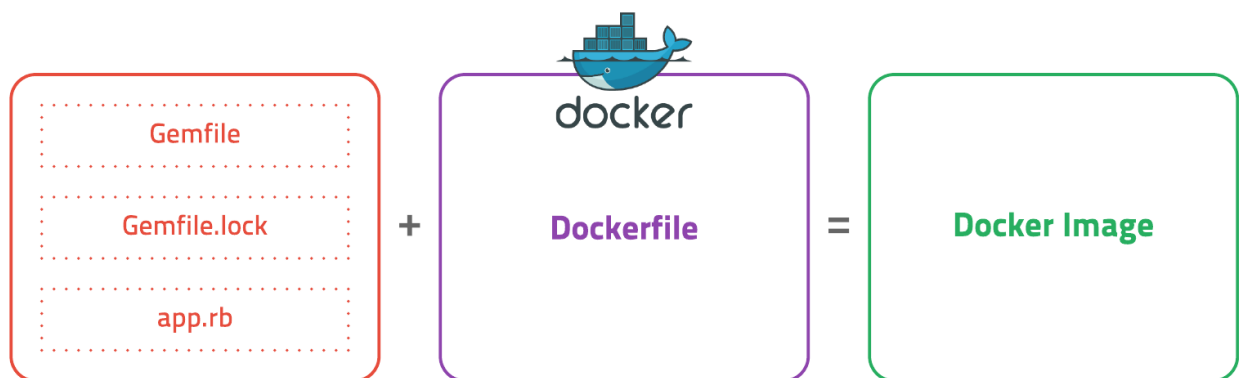
서버가 정상적으로 실행됐으면 웹 브라우저에서 테스트해봅니다. <http://localhost:4567>



Browser test

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

Ruby Application Dockerfile



Dockerfile

도커는 이미지를 만들기 위해 **Dockerfile** 이라는 이미지 빌드용 DSL(Domain Specific Language) 파일을 사용합니다. 단순 텍스트 파일로 일반적으로 소스와 함께 관리합니다.

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

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

이 과정을 그대로 쉘 스크립트로 옮겨봅니다.

```

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

```

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

```

1  # 1. ubuntu 설치 (패키지 업데이트 + 만든사람 표시)
2  FROM      ubuntu:16.04
3  MAINTAINER subicura@subicura.com
4  RUN      apt-get -y update
5
6  # 2. ruby 설치

```

```

7  RUN apt-get -y install ruby
8  RUN gem install bundler
9
10 # 3. 소스 복사
11 COPY . /usr/src/app
12
13 # 4. Gem 패키지 설치 (실행 디렉토리 설정)
14 WORKDIR /usr/src/app
15 RUN bundle install
16
17 # 5. Sinatra 서버 실행 (Listen 포트 정의)
18 EXPOSE 4567
19 CMD bundle exec ruby app.rb -o 0.0.0.0

```

Dockerfile hosted with ❤ by GitHub

[view raw](#)

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

빌드 파일을 작성했으니 이제 이미지를 만들어 봅니다.

Docker build

이미지를 빌드하는 명령어는 다음과 같습니다.

```
docker build [OPTIONS] PATH | URL | -
```

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

Dockerfile을 만든 디렉토리로 이동하여 다음 명령어를 입력합니다.

```
1  docker build -t app .
```

output:

```

1  Sending build context to Docker daemon 22.02 kB
2  Step 1/10 : FROM ubuntu:16.04
3  ----> f49eec89601e
4  Step 2/10 : MAINTAINER subicura@subicura.com
5  ----> Running in 06f20ac1017d
6  ----> fc41cd8ac52d
7  Removing intermediate container 06f20ac1017d
8  Step 3/10 : RUN apt-get -y update
9  ----> Running in c35738e75a51

```



```

10 Get:1 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
11 Get:2 http://archive.ubuntu.com/ubuntu xenial-updates InRelease [102 kB]
12 Get:3 http://archive.ubuntu.com/ubuntu xenial-security InRelease [102 kB]
13 Get:4 http://archive.ubuntu.com/ubuntu xenial/main Sources [1103 kB]
14 Get:5 http://archive.ubuntu.com/ubuntu xenial/restricted Sources [5179 B]
15
16 ... 생략 ...
17
18 Step 9/10 : EXPOSE 4567
19 ---> Running in 9c514a1f0c8e
20 ---> c5ce4376282e
21 Removing intermediate container 9c514a1f0c8e
22 Step 10/10 : CMD bundle exec ruby app.rb -o 0.0.0.0
23 ---> Running in 1f7a9ba8d63c
24 ---> 54d239c00f11
25 Removing intermediate container 1f7a9ba8d63c
26 Successfully built 54d239c00f11

```

```

~/Workspace/project/github.com/subicura/sinatra
> docker build -t app .
Sending build context to Docker daemon 22.02 kB
Step 1/10 : FROM ubuntu:16.04
---> f49eec89601e
Step 2/10 : MAINTAINER subicura@subicura.com
---> Running in 06f20ac1017d
---> fc41cd8ac52d
Removing intermediate container 06f20ac1017d
Step 3/10 : RUN apt-get -y update
---> Running in c35738e75a51
Get:1 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
Get:2 http://archive.ubuntu.com/ubuntu xenial-updates InRelease [102 kB]
Get:3 http://archive.ubuntu.com/ubuntu xenial-security InRelease [102 kB]
Get:4 http://archive.ubuntu.com/ubuntu xenial/main Sources [1103 kB]
Get:5 http://archive.ubuntu.com/ubuntu xenial/restricted Sources [5179 B]
Get:6 http://archive.ubuntu.com/ubuntu xenial/universe Sources [9802 kB]

```

▶ 00:00

docker-build

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

그럼 이미지가 잘 생성되었는지 확인해보겠습니다.

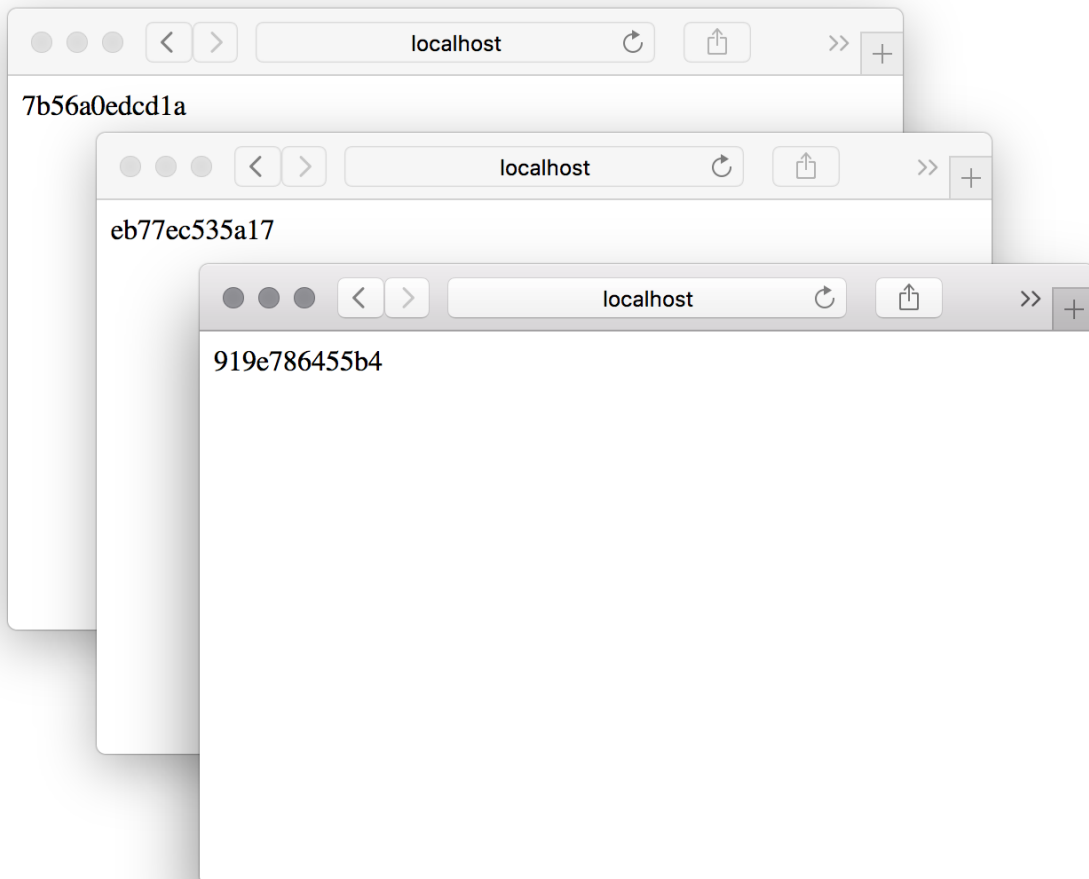
```
1 docker images
```

tput:

	REPOSITORY	TAG	IMAGE ID	CREATED	S
1	app	latest	54d239c00f11	4 minutes ago	2
2	ubuntu	16.04	f49eec89601e	2 weeks ago	1

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

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



Container test result

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

🐳 Dockerfile 기본 명령어

기지를 만드는 데 사용한 Dockerfile의 기본적인 명령어를 살펴보겠습니다.

FROM

```
FROM <image>:<tag>
FROM ubuntu:16.04
```

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

MAINTAINER

```
MAINTAINER <name>
MAINTAINER subicura@subicura.com
```

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

COPY

```
COPY <src>... <dest>
COPY . /usr/src/app
```

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

ADD

```
ADD <src>... <dest>
ADD . /usr/src/app
```

COPY 명령어와 매우 유사하나 몇가지 추가 기능이 있습니다. src 에 파일 대신 URL을 입력할 수 있고 src 에 압축 파일을 입력하는 경우 자동으로 압축을 해제하면서 복사됩니다.

RUN

```
RUN <command>
RUN ["executable", "param1", "param2"]
RUN bundle install
```

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

CMD

```
CMD ["executable","param1","param2"]  
CMD command param1 param2  
CMD bundle exec ruby app.rb
```

도커 컨테이너가 실행되었을 때 실행되는 명령어를 정의합니다. 빌드할 때는 실행되지 않으며 여러 개의 `CMD` 가 존재할 경우 가장 마지막 `CMD` 만 실행됩니다. 한꺼번에 여러 개의 프로그램을 실행하고 싶은 경우에는 `run.sh` 파일을 작성하여 데몬으로 실행하거나 [supervisord](#)나 [forego](#)와 같은 여러 개의 프로그램을 실행하는 프로그램을 사용합니다.

WORKDIR

```
WORKDIR /path/to/workdir
```

`RUN`, `CMD`, `ADD`, `COPY`등이 이루어질 기본 디렉토리를 설정합니다. 각 명령어의 현재 디렉토리는 한 줄 한 줄마다 초기화되기 때문에 `RUN cd /path` 를 하더라도 다음 명령어에선 다시 위치가 초기화 됩니다. 같은 디렉토리에서 계속 작업하기 위해서 `WORKDIR` 을 사용합니다.

EXPOSE

```
EXPOSE <port> [<port>...]  
EXPOSE 4567
```

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

VOLUME

```
VOLUME ["/data"]
```

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

ENV

```
ENV <key> <value>  
ENV <key>=<value> ...  
ENV DB_URL mysql
```

컨테이너에서 사용할 환경변수를 지정합니다. 컨테이너를 실행할 때 `-e` 옵션을 사용하면 기존 값을 오버라이딩 하게 됩니다.

~기까지 Dockerfile에서 가장 많이 사용하는 명령어에 대해 알아보았습니다. 모든 명령어가
 금하신 분은 [공식문서](#)를 참고하세요.

Build 분석

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

```

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

(1) Sending build context to Docker daemon 5.12 kB

빌드 명령어를 실행한 디렉토리의 파일들을 빌드컨텍스트^{build context}라고 하고 이 파일들을 도커 서버(daemon)로 전송합니다. 도커는 서버-클라이언트 구조이므로 도커 서버가 작업하려면 미리 파일을 전송해야 합니다.

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

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

(3) ---> f49eec89601e

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

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

Dockerfile의 두 번째 명령어인 MAINTAINER 명령어를 수행합니다.

(5) ---> Running in f4de0c750abb

명령어를 수행하기 위해 바로 이전에 생성된 f49eec89601e 이미지를 기반으로 f4de0c750abb 컨테이너를 임시로 생성하여 실행합니다.

```
'\ ' ---> 4a400609ff73
```

ㄱ 령어 수행 결과를 이미지로 저장합니다.

```
(7) Removing intermediate container f4de0c750abb
```

명령어를 수행하기 위해 임시로 만들었던 컨테이너를 제거합니다.

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

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

```
(9) Successfully built 20369cef9829
```

최종 성공한 이미지 ID를 출력합니다.

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

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

도커 이미지 리팩토링

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

Base Image

위에서 만든 Ruby 애플리케이션 이미지는 `ubuntu` 를 베이스로 만들었지만 사실 휘어어얼씬 간단한 `ruby` 베이스 이미지가 존재합니다. ~~따라 이야기하지 않아 죄송..~~ 기존에 `ruby`를 설치했던 명령어는 `ruby` 이미지를 사용하는 것으로 간단하게 생략할 수 있습니다.

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

```
9 FROM ruby:2.3
10 MAINTAINER subicura@subicura.com
```

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

Build Cache

조금전에 빌드한 이미지를 다시 빌드해볼까요?

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

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

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

기존 소스에서 소스파일이 수정되면 캐시가 깨지는 부분은 다음과 같습니다.

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

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

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

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

명령어 최적화

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

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

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

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

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

이쁘게

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

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

최종

❗ 종 결과는 다음과 같습니다.

```
1 FROM ruby:2.3
2 MAINTAINER subicura@subicura.com
3 COPY Gemfile* /usr/src/app/
4 WORKDIR /usr/src/app
5 RUN bundle install --no-rdoc --no-ri
6 COPY . /usr/src/app
7 EXPOSE 4567
8 CMD bundle exec ruby app.rb -o 0.0.0.0
```

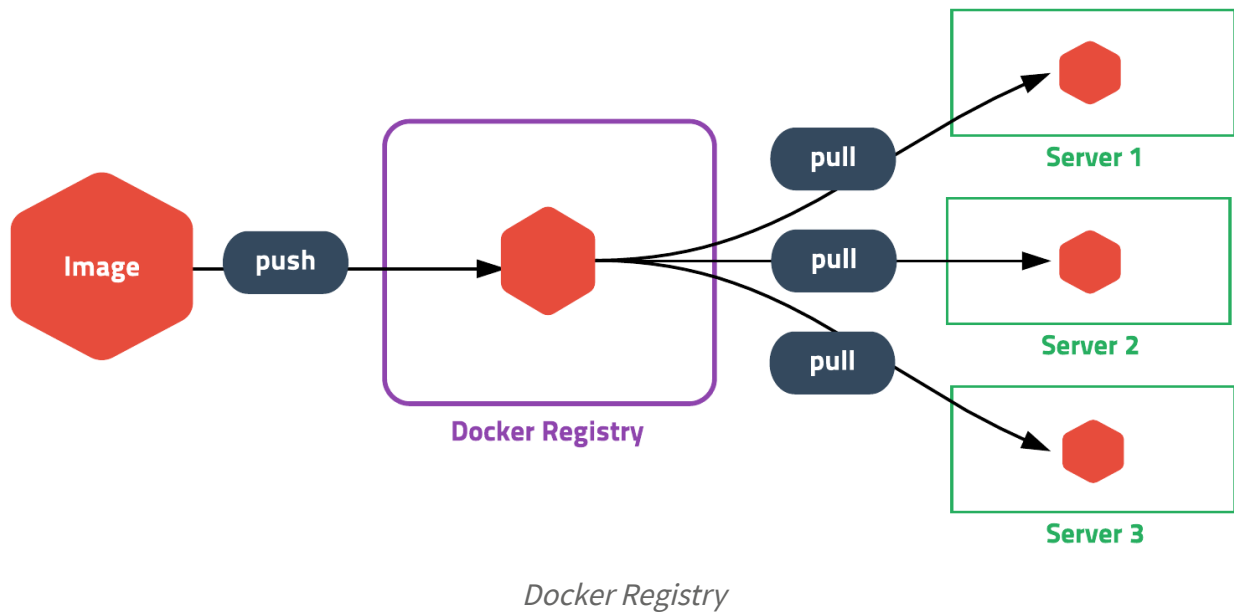
Dockerfile hosted with ❤ by GitHub

[view raw](#)

드디어 아까보다 훨씬 나은 이미지가 완성됐습니다. ~~만세~~

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

이미지 저장소



도커는 빌드한 이미지를 서버에 배포하기 위해 직접 파일을 복사하는 방법 대신 도커 레지스트리 Docker Registry라는 이미지 저장소를 사용합니다. 도커 명령어를 이용하여 이미지를 레지스트리에 푸시push하고 다른 서버에서 풀pull받아 사용하는 구조입니다. ~~git을 사용하는 느낌?~~

도커 레지스트리는 오픈소스로 무료로 설치할 수 있고 설치형이 싫다면 도커(Docker Inc.)에서 서비스 중인 도커 허브 Docker Hub를 사용할 수 있습니다.

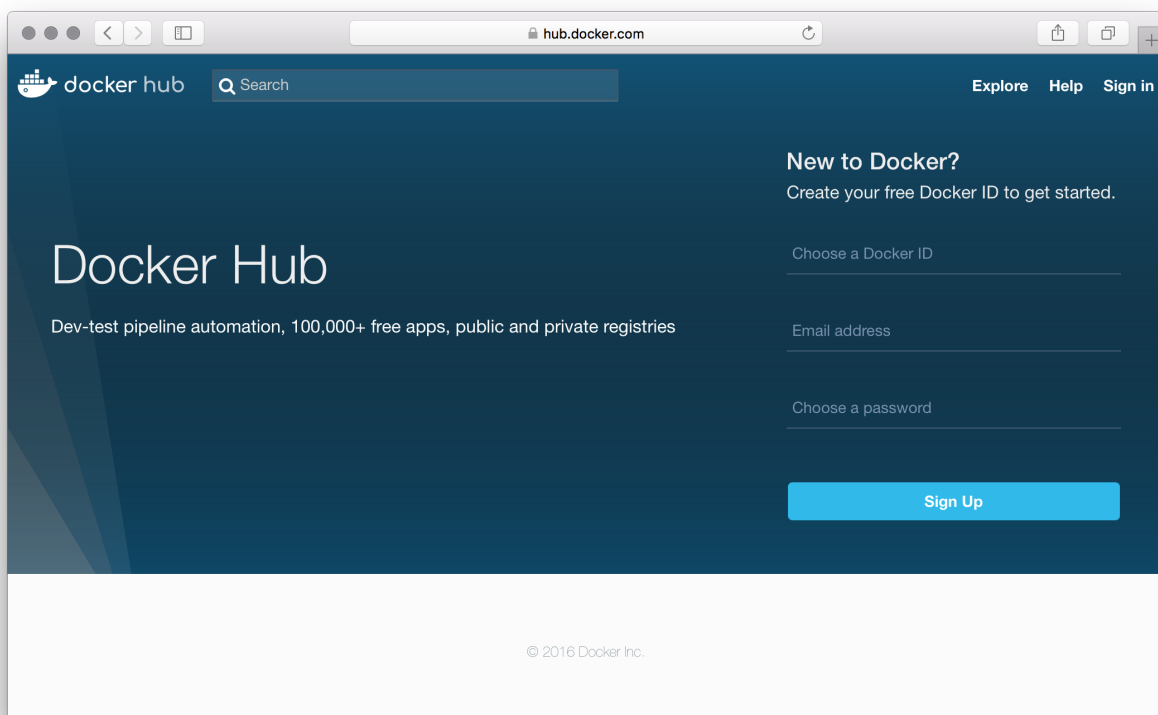
Docker Hub

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

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

회원가입을 하고 앞에서 만든 Ruby 웹 애플리케이션 이미지를 저장해보겠습니다.

회원가입



Docker Hub

도커 허브 사이트에 접속하면 쉽게 회원가입을 할 수 있습니다. 디자인이 참 마음에 들지 않는 데... 처음부터 지금까지 업데이트한 모습이 이 모양이라 앞으로도 크게 기대되지 않습니다. 다른 페이지는 다 이쁘데 ㅠㅠ

로그인

도커에서 도커 허브 계정을 사용하려면 로그인을 해야합니다.

```
1 docker login
```

output:

```
1 Login with your Docker ID to push and pull images from Docker Hub. If you don't h
2 Username: subicura
3 Password:
4 Login Succeeded
```

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

이미지 태그

도커 이미지 이름은 다음과 같은 형태로 구성됩니다.

```
[Registry URL]/[사용자 ID]/[이미지명]:[tag]
```

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

도커의 `tag` 명령어를 이용하여 기존에 만든 이미지에 추가로 이름을 지어줄 수 있습니다.

```
docker tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]
```

앞에서 만든 `app` 이미지에 계정정보와 버전 정보를 추가해보겠습니다.

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

`subicura` 라는 ID를 사용하고 이미지 이름을 `sinatra-app` 으로 변경했습니다. 첫 번째 버전 이므로 태그는 `1` 을 사용합니다. 이제 `push` 명령을 이용해 도커 허브에 이미지를 전송해 보 니다.

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

output:

```
1 The push refers to a repository [docker.io/subicura/sinatra-app]
2 2adeabae7edc: Pushed
3 8343e5bcf528: Pushed
4 af3b68c8b565: Pushed
```

```

5      40dd6783317f: Pushed
6      c6ae77e29c22: Pushed
7      5eb5bd4c5014: Mounted from library/ubuntu
8      d195a7a18c70: Mounted from library/ubuntu
9      af605e724c5a: Mounted from library/ubuntu
10     59f161c3069d: Mounted from library/ubuntu
11     4f03495a4d7d: Mounted from library/ubuntu
12     1: digest: sha256:af83aca920982c1fb17f08b4aa300439470349d58d63c921f67261054a0c946

```

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

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

Private Docker Registry

도커 이미지를 비공개로 저장하려면 [Docker Cloud](#)를 유료(\$7 for 5 repos/month)로 사용하거나 레지스트리 서버를 자체적으로 구축해야 합니다.

도커 레지스트리는 도커를 이용하여 쉽게 만들 수 있습니다. 도커 이미지를 저장할 서버를 도커 스스로 만들어서 도커 이미지를 관리하다니 뭔가 멋-_-합니다. ~~도커로 다하는 느낌~~

```

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

```

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

```

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

```

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

```

1      tree registry

```

output:

```

1      registry

```

```

2      └─ docker
3        └─ registry
4          └─ v2
5            └─ blobs
6              └─ sha256
7                └─ 05
8                  └─ 05e91de6d378244d3b4dcfbb978548e47b2c22d9918638444c0f9
9                    └─ data
10                 └─ 1b
11                   └─ 1bacd3c8ccb1f15609a10bd4a403831d0ec0b354438ddb644c95
12                     └─ data
13                 └─ 3e
14                   └─ 3e309f550af7bb4e3177e94e498938a24c20cf7404d90da4b24bd
15                     └─ data
16                 └─ 5b
17                   └─ 5b8ee2811f46dea888057c8a2eb1fcd43838bb2545588f7e108d4
18                     └─ data
19                 └─ 76
20                   └─ 761e0e61cc35ced6b08ce33b1db84a3e92a4843f908476d4cfc52
21                     └─ data
22                 └─ 86
23                   └─ 869d5d3f92f8bbdcaa1023ce719df7e337d34b2a13954cde7954c
24                     └─ data
25                 └─ 97
26                   └─ 976dd3af6adc358c852b97651cef1ceaade821a25f3bf566de3a8
27                     └─ data
28                 └─ a2
29                   └─ a2037218c9af7a6913308a60127a320272fd6d58405692a5be48c
30                     └─ data
31                 └─ e2
32                   └─ e2d7e96004fdd10e671372a8b5e861bae78bcf878e062d13f6c47
33                     └─ data
34                 └─ f4
35                   └─ f40e1388890a7ca2b5180e2fae69aa982926307cdfa98c8c6a2dd
36                     └─ data
37                 └─ f8
38                   └─ f8a4e25b40ce0ecc925cfb0b2ec7eec0a06948eda173e943c228c
39                     └─ data
40                 └─ fd
41                   └─ fdeb3bd5d1b49c094875dcacccb743b0bc913c7de80ba783f2f1b
42                     └─ data
43             └─ repositories
44               └─ subicura
45                 └─ app
46                   └─ _layers
47                     └─ sha256
48                       └─ 1bacd3c8ccb1f15609a10bd4a403831d0ec0b354438dd
49                         └─ link
50                       └─ 3e309f550af7bb4e3177e94e498938a24c20cf7404d90
51                         └─ link
52                       └─ 5b8ee2811f46dea888057c8a2eb1fcd43838bb2545588
53                         └─ link

```

```

54 |         | 761e0e61cc35ced6b08ce33b1db84a3e92a4843f90847
55 |         |   └─ link
56 |         | 869d5d3f92f8bbdcaa1023ce719df7e337d34b2a13954
57 |         |   └─ link
58 |         | 976dd3af6adc358c852b97651cef1ceaade821a25f3bf
59 |         |   └─ link
60 |         | a2037218c9af7a6913308a60127a320272fd6d5840569
61 |         |   └─ link
62 |         | e2d7e96004fdd10e671372a8b5e861bae78bcf878e062
63 |         |   └─ link
64 |         | f40e1388890a7ca2b5180e2fae69aa982926307cdfa98
65 |         |   └─ link
66 |         | f8a4e25b40ce0ecc925cfb0b2ec7eec0a06948eda173e
67 |         |   └─ link
68 |         └─ fdeb3bd5d1b49c094875dcacccb743b0bc913c7de80ba
69 |           └─ link
70 | └─ _manifests
71 |   └─ revisions
72 |     └─ sha256
73 |       └─ 05e91de6d378244d3b4dcfbb978548e47b2c22d99
74 |         └─ link
75 |   └─ tags
76 |     └─ latest
77 |       └─ current
78 |         └─ link
79 |       └─ index
80 |         └─ sha256
81 |           └─ 05e91de6d378244d3b4dcfbb978548e47
82 |             └─ link
83 | └─ _uploads
84 |
85 | 56 directories, 26 files

```

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

보안

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

관련 설정은 [문서](#)를 참고하시기 바랍니다.

포하기

드디어 도커 안내서의 마지막 주제, 서버관리의 꽃! 배포_{deploy}에 대해 알아보겠습니다.

컨테이너 배포 방식으로

컨테이너를 배포하는 방식은 기존에 애플리케이션을 배포하는 방식과 큰 차이가 있습니다.

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

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

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

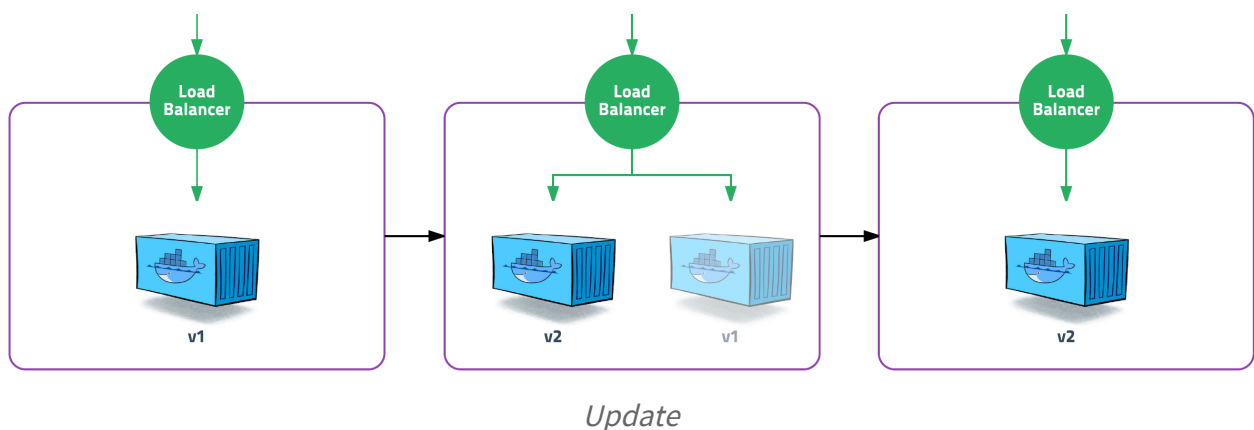
하지만, 컨테이너를 사용하면 어떤 언어, 어떤 프레임워크를 쓰든 상관없이 배포 방식이 동일해지고 과정 또한 굉장히 단순해집니다. 그냥 이미지를 다운받고 컨테이너를 실행하면 끝입니다.

음...

네, 그게 답니다.

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

컨테이너 업데이트



도커를 사용하면 업데이트하는 방식도 배포와 큰 차이가 없습니다.

*¹ 신 이미지를 기반으로 새 컨테이너를 만들고 이전 컨테이너를 중지(삭제)하면 됩니다. 최신 스를 어떻게 복사할지 서버 프로세스는 어떻게 재시작할지 고민할 필요가 없습니다. 그냥 통째로 바꿔버리는 겁니다.

단, 컨테이너를 중지하지 않고 graceful스무스하게 샤샤삭 교체하는 방법은 아쉽지만 존재하지 않습니다.

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

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

여기서는 개념만 소개하고 실제로 컨테이너를 업데이트하는 구체적인 방법은 [도커를 이용한 웹서비스 무중단 배포하기](#)~~막간 홍보글~~로 대신합니다.

배포에 대해 더 알아보기

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

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

자세한 내용은 초보를 위한 안내서의 범위를 벗어나므로 더 많은 내용이 궁금하신 분은 각자 공부하는 것으로... ~~화이팅!!~~

마무으리

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

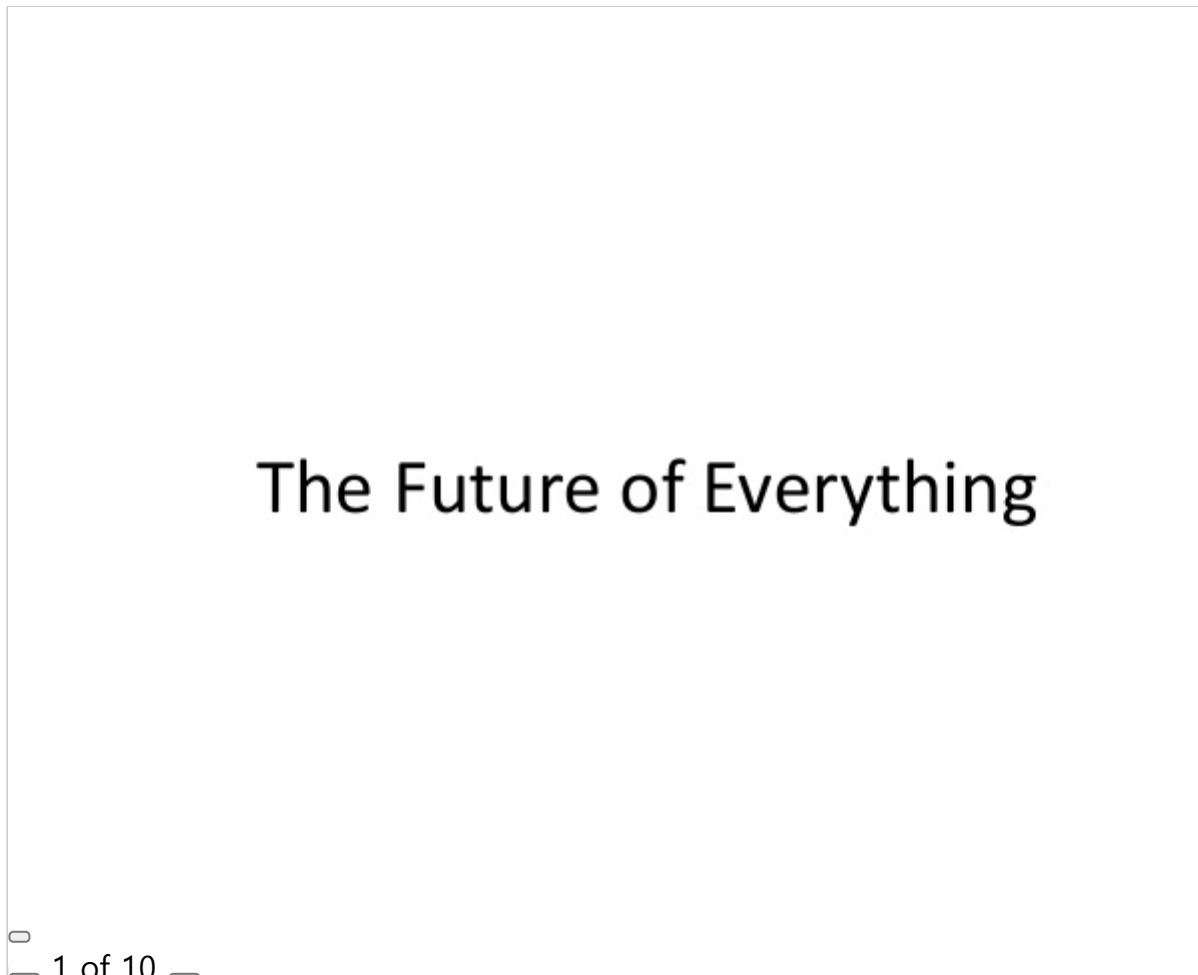
이러한 흐름 속에서 아무 문제 없이 서비스를 배포하고 운영하는 것은 사실 기적이라고 할 수 있습니다. 도커가 등장하여 복잡한 부분을 단순하게 하고 많은 문제를 해결해 주고 있지만 도

기'도 모든 문제를 해결해 주지는 않습니다. (대표적으로 데이터베이스처럼 stateful한 애플리케이션은 관리가 어렵습니다) 다만, 현재 시점에서 그 어떤 방법보다도 **좀 더 나은 방법**인 건 확실합니다.

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

도커에 대해 궁금한 점은 [오픈컨테이너 슬랙](#)으로 오시면 많은 도움을 받을 수 있습니다. 내용에 대한 다양한 피드백도 환영합니다. 읽어주셔서 감사합니다~

마지막으로 “Future of Everything(모든것의 미래)”라는 슬라이드를 소개하며 글을 마칩니다.



[The Future of Everything](#) from [Michael Ducy](#)

초보를 위한 도커 안내서 - 인프런

도커를 1도 모르는 입문자, 초보자분들을 위한 도커 안내서입니다. 복잡한 내용을 제외하고 도커가 왜 인기가 많고 어떻게 사용하는지 빠르게 익힐 수 있도록 집중하였습니다. 입문 서버

<https://www.inflearn.com/course/도커-입문>



docker

초보를 위한 도커안내서

[이제 도커안내서를 영상으로 만나보세요!](#)

[Docker](#) [DevOps](#) [Server](#) [Container](#)

좋아요 171개

공유하기




WRITTEN BY


SUPPORTED BY


subicura

Published 10 Feb 2017

Proudly published with Jekyll

 Feedly 구독하기

 RSS 구독하기

 Email 구독하기

READ THIS NEXT

Docker Swarm을 이용한 쉽고 빠른 분산 서버 관리

도커 스웜은 오케스트레이션 툴은 관리가 어렵고 사용하기 복잡하다는 편견을 완전히 바꿔놓았습니다. 구축 비용이 거의 들지...

YOU MIGHT ENJOY

초보를 위한 도커 안내서 - 설치하고 컨테이너 실행하기

초보를 위한 도커 안내서 2번째 글입니다. 도커의 기본적인 내용을 이야기 했던 첫번째 글에 이어 실제로...

Subicura's Blog © 2020

!글 30개

정렬 기준 인기순



댓글 달기...

**이현기**

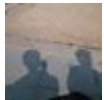
좋은 자료 감사합니다!

좋아요 · 답글 달기 · 1주

**김진**

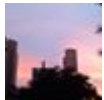
진심 많은 도움이 되었습니다. 감사합니다.

좋아요 · 답글 달기 · 10주

**최원표**

정말 많은 도움이 됐습니다. 감사합니다.

좋아요 · 답글 달기 · 12주

**최성욱**

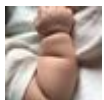
Docker 공부하면서 여러자료를 같이 봤는데 해당글을 결국 제일 많이 정독하게 되네요...!! 좋은 자료작성해주신것에 너무 감사합니다. 도움이 많이 되었습니다.

좋아요 · 답글 달기 · 14주

**예상오**

좋은 자료 감사합니다!!

좋아요 · 답글 달기 · 17주

**Wonil Seo**

감사합니다!!

좋아요 · 답글 달기 · 17주

**문재운**

좋은 자료 감사합니다.

좋아요 · 답글 달기 · 38주

**Haeyong Hwang**

박수!!!!!!

좋아요 · 답글 달기 · 40주

**이세현**

감사합니다~!

좋아요 · 답글 달기 · 42주

**방경민**

정말 좋은글 감사드립니다

좋아요 · 답글 달기 · 1년

댓글 10개 더 읽어들이기