**3장 도커 이미지 만들기**

***3-1* 도커 허브에 공유된 이미지 사용하기**

**애플리케이션의 컨테이너 이미지를 내려받는 명령어**

docker image pull diamol/ch03-web-ping

**도커 이미지**: 물리적으로는 여러 개의 작은 파일로 구성

이미지를 내려받는 과정을 보면 여러 건의 파일을 동시에 내려받음 -> 각각의 파일을 **이미지 레이어**라고 함

도커가 이미지 레이어들을 조립해 컨테이너의 내부 파일 시스템을 만듦

모든 레이어를 내려받고 나면 전체 이미지를 사용할 수있게 됨

**내려받은 이미지로 컨테이너를 실행하고 실행된 애플리케이션의 기능을 확인하는 명령어**

docker container run -d --name web-ping diamol/ch03-web-ping

**-d**: --detach의 축약형. 컨테이너는 백그라운드에서 동작하게 함

**--name**: 컨테이너에 원하는 이름을 붙이고 이 이름으로 컨테이너를 지칭할 수 있음

**실행 중인 컨테이너를 삭제하고 환경 변수 TARGET의 값을 다른 값으로 지정한 새로운 컨**

**테이너를 실행하는 명령어**

docker rm -f web-ping

docker container run --env TARGET=google.com diamol/ch03-web-ping

-> 새로 실행한 컨테이너의 동작은 아까와는 다름

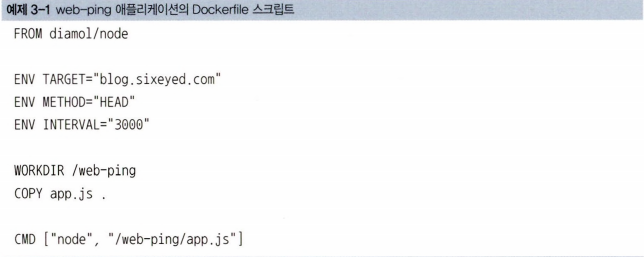
* --detach 플래그를 적용하지 않았으므로 애플리케이션의 출력 내용이 콘솔에 나타남
* HTTP 요청을 보내는 대상이 google.com으로 변경

-> 도커 이미지는 설정값의 기본값을 포함해 패키징되지만. 컨테이너를 실행할 때 이 설정값을 바꿀 수 있어야 함

***3-2* Dockerfile 작성하기**

**Dockerfile**

* 애플리케이션을 패키징하기 위한 간단한 스크립트로 일련의 인스트럭션으로 구성
* 인스트럭션을 실행한 결과 -> 도커 이미지가 만들어짐
* Dockerfile 문법은 배우기 쉬우며 어떠한 애플리케이션이라도 패키징 가능
* 자주 쓰이는 작업은 별도의 명령이 마련돼 있으며, 원하는 작업을 직접 작성할 수 있고. 표준 셸 문법도 사용 가능



web-ping 애플리케이션을 패키징하기 위한 전체 Dockerfile 스크립트

**FROM**: 모든 이미지는 다른 이미지로부터 출발. 이 이미지는 diamol/node 이미지를 시작

점으로 지정.

**ENV**: 환경 변수 값을 지정하기 위한 인스트럭션. key=”value” 형식

**WORKDIR**: 컨테이너 이미지 파일 시스템에 디렉터리를 만들고, 해당 디렉터리를 작업 디렉터리로 지정하는 인스트럭션

**COPY**: 로컬 파일 시스템의 파일 혹은 디렉터리를 컨테이너 이미지로 복사하는 인스트럭션. [원본경로] [복사경로] 형식으로 지정

**CMD**: 도커가 이미지로부터 컨테이너를 실행했을 때 실행할 명령을 지정하는 인스트럭션

***3-3* 컨테이너 이미지 빌드하기**

이미지를 빌드하려면 Dockerfile 스크립트 외에도 이미지의 이름, 패키징에 필요한 파일의 경로를 추가로 지정해 주어야 함

**docker image build 명령을 사용해 Dockerfile 스크립트로 이미지를 빌드하는 명령어**

docker image build --tag web-ping .

**--tag의 인잣값**: 이미지의 이름 (web-ping)

이어지는 인자: Dockerfile 및 이미지에 포함시킬 파일이 위치한 경로(.)

-> 마지막의 .은 ‘현재 작업 디렉터리’라는 뜻

build 명령에서 어떤 오류가 발생했다면 확인할 것

1. 도커 엔진이 정상적으로 동작 중인지 확인 -> 윈도나 맥 환경이라면 도커 데스크톱의 상태를 확인
2. 현재 작업 디렉터리가 정확한지 확인
3. build 명령을 정확하게 입력했는지 확인

**‘w’로 시작하는 태그명을 가진 이미지 목록을 확인하는 명령어**

docker image Is 'w\*'

**새로 빌드한 이미지로부터 컨테이너를 실행해 도커 웹 사이트에 5초마다 요청을 보내는 명령어**

docker container run -e TARGET=docker.com -e INTERVAL=5OO0 web-ping

-> 이 컨테이너는 포어그라운드(foreground)로 동작하므로 ctrl + c 를 눌러 중지

***3-4* 도커 이미지와 이미지 레이어 이해하기**

**이미지의 히스토리를 확인하는 명령어**

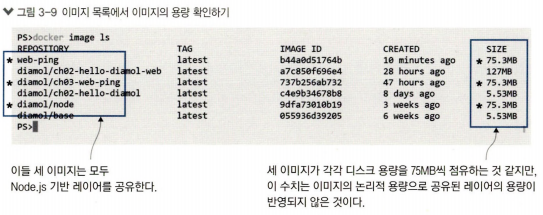
docker image history web-ping

Dockerfile 인스트럭션과 이미지 레이어는 1：1 관계

-> 이미지 레이어는 여러 이미지와 컨테이너에서 공유될 수 있음

ex) Node.js 애플리케이션이 실행되는 컨테이너를 여러 개 실행한다면 이들 컨테이너는 모두 Node.js 런타임이 들어 있는 이미지 레이어를 공유

docker image Is로 출력된 이미지 목록에서 각 이미지의 용랑을 확인 가능

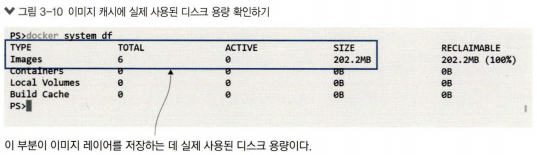


이미지 목록의 SIZE 항목에 나오는 수치는 이미지의 논리적 용량이지 해당 이미지가 실제로 차지하는 디스크 용량을 나타내는 것이 아님

-> 다른 이미지와 레이어를 공유하면 여기에 나온 수치보다 디스크 용량을 훨씬 덜 차지함

**이미지 저장에 실제 사용된 디스크 용량을 확인하는 명령어**

docker system df



-> 용량 총합이 363.96MB이였지만 이 명령을 입력해 나온 출력 결과를 보면 이미지 캐시의 실제 용량은 약 202.2MB를 차지하는 것으로 나옴. 163MB는 이미지끼리 레이어를 공유한 것으로, 약 45%의 디스크 공간이 절약됨

-> 절약되는 디스크 공간은 대개 런타임 등 같은 기반 레이어를 공유하는 애플리케이션의 숫자가 많을수록 더욱 늘어남

**공유되는 레이어는 수정할 수 없어야 함**

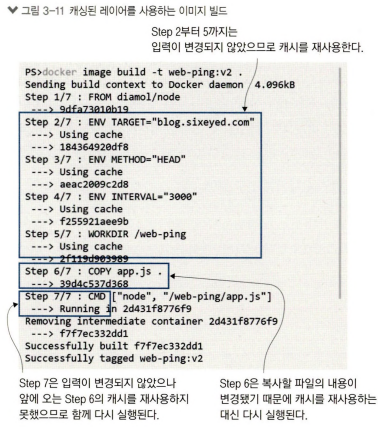
:만약 이미지의 레이어를 수정할 수 있다면 그 수정이 레이어를 공유하는 다른 이미지에도 영향을 미치기 때문

-> 이미지 레이어를 **읽기 전용**으로 만들어 두어 이런 문제를 방지

***3-5* 이미지 레이어 캐시를 이용한 Dockerfile 스크립트 최적화**

**파일 수정 후 새로운 버전의 도커 이미지를 빌드하는 명령어**

docker image build -t web-ping：v2 .



-> 인스트럭션의 결과가 이전 빌드와 같다면, 이전에 캐시된 레이어를 재사용. 이런 방법으로 똑같은 인스트럭션을 다시 실행하는 낭비를 줄일 수 있음

도커는 캐시에 일치하는 레이어가 있는지 확인하기 위해 해시값을 이용

* 해시는 입력값이 같은지 확인할 수 있는 일종의 디지털 지문
* 해시값은 Dockerfile 스크립트의 인스트럭션과 인스트럭션에 의해 복사되는 파일의 내용으로부터 계산되는데. 기존 이미지 레이어에 해시값이 일치하는 것이 없다면 **캐시 미스**가 발생하고 해당 인스트럭션이 실행
* 한번 인스트럭션이 실행되면 그다음에 오는 인스트럭션은 수정된 것이 없더라도 모두 실행 -> Dockerfile 스크립트의 인스트럭션은 잘 수정하지 않는 인스트럭션이 앞으로 오고 자주 수정되는 인스트럭션이 뒤에 오도록 배치해야 함 -> **스크립트의 최적화**