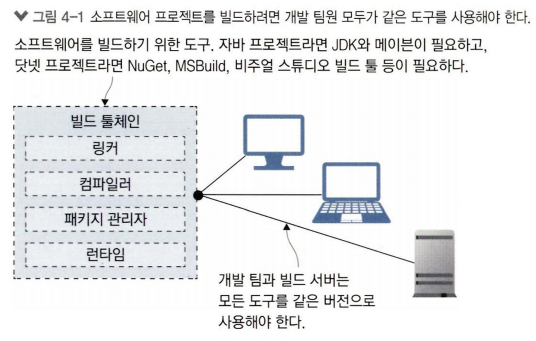
**4장 애플리케이션 소스 코드에서 도커 이미지까지**

***4-1* Dockerfile이 있는데 빌드 서버가필요할까?**

대부분의 프로그래밍 언어는 프로젝트를 빌드하기 위해 다양한 도구가 필요

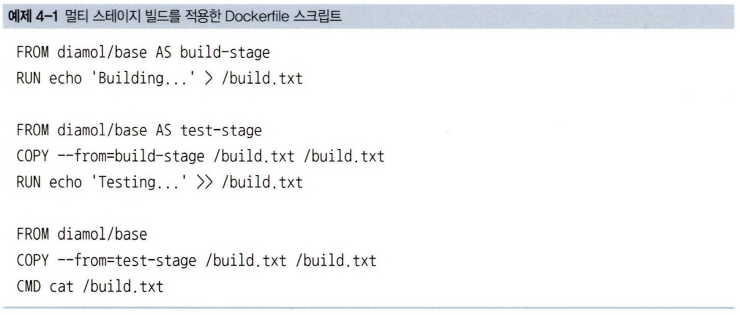


문제점:

1. 이 과정에는 유지 보수를 위한 큰 오버헤드가 발생
2. 신규 개발자는 이 도구를 다 설치해야 함
3. 빌드 서버와 버전이 달라지는 것만으로도 빌드 실패 가능성

해결법: 빌드 툴체인을 한 번에 패키징해서 공유할 수 있다면 편리 -> 도커를 사용하면 가능

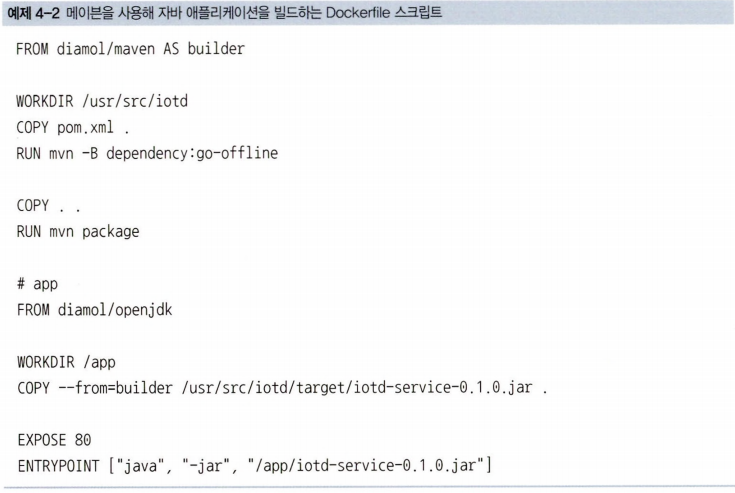
1. 개발에 필요한 모든 도구를 배포하는 Dockerfile 스크립트를 작성
2. 이를 이미지로 만듦
3. 애플리케이션 패키징을 위한 Dockerfile 스크립트에서 이 이미지를 사용해 소스 코드를 컴파일함으로써 애플리케이션을 패키징함



* 이 스크립트는 빌드가 여러 단계로 나뉘는 **멀티 스테이지 빌드**를 적용한 것
* 각 빌드 단계는 **FROM** 인스트럭션으로 시작
* 빌드 단계에 **AS** 파라미터를 이용해 이름을 붙일 수 있음
* 최종 산출물은 마지막 단계의 내용물을 담은 도커 이미지
* 각 빌드 단계는 독립적으로 실행되지만, 앞선 단계에서 만들어진 디렉터리나 파일을 복사할 수 있음 (COPY --from)
* **RUN** 인스트럭션은 빌드 중에 컨테이너 안에서 명령을 실행한 다음 그 결과를 이미지 레이어에 저장하는 기능 -> FROM 인스트럭션에서 지정한 이미지에서 실행할 수 있어야 함
* 각 빌드 단계는 서로 격리되어있고 단계별로 기반이미지도 다를 수 있으므로 사용할 수 있는 도구도 달라질 수 있음
* 어느 한 단계에서라도 명령이 실패하면 전체 빌드가 실패함

빌드 서버에 도커가 설치되면 빌드 도구를 도커 이미지를 통해 중앙 집중적으로 관리할 수 있으며 거의 모든 주요 애플리케이션 프레임워크는 이미 도커 허브를 통해 빌드 도구가 내장된 공식 이미지를 제공함

***4-2* 애플리케이션 빌드 실전 예제: 자바 소스 코드**



builder 단계

1. 이미지에 작업 디렉터리를 만든 후 pom.xml 파일을 여기에 복사함
2. RUN 인스트럭션에서 메이븐이 실행돼 필요한 의존 모듈을 내려받음
3. COPY . . 인스트럭션을 통해 나머지 소스 코드가 복사됨
4. mvn package 명령을 실행 -> 애플리케이션을 빌드하고 패키징하라는 의미. 입력은 자바 소스 코드이며, 출력은 JAR 포맷으로 패키징 된 자바 애플리케이션

애플리케이션 이미지 생성

1. 기반이미지 diamol/openjdk는 자바 11 런타임을 포함하지만 메이븐은 포함하지 않음
2. 이미지에 작업 디렉터리를 만든 후 builder 단계에서 만든 JAR 파일을 복사함
3. EXPOSE 인스트럭션을 통해 외부로 공개
4. ENTRYPOINT 인스트럭션은 CMD 인스트럭션과 같은 기능을 하는 인스트럭션. 해당 이미지로 컨테이너가 실행되면 도커가 이 인스트럭션에 정의된 명령어를 실행

**이미지를 빌드하는 명령어**

docker image build -t image-of-the-day .

**컨테이너 간 통신에 사용되는 도커 네트워크를 생성하는 명령어**

docker network create nat

**앞서 빌드한 이미지로부터 컨테이너를 실행하되. 80번 포트를 호스트 컴퓨터를 통해 공**

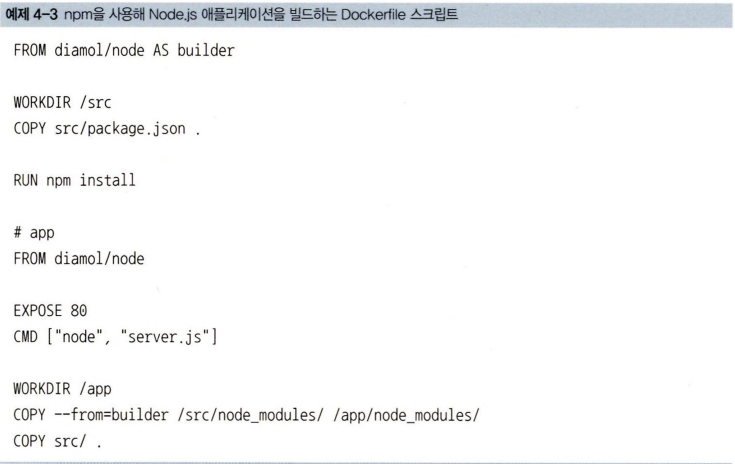
**개하고 nat 네트워크에 컨테이너를 접속하는 명령어**

docker container run --name iotd -d -p 800：80 --network nat image-of-the-day

***4-3* 애플리케이션 빌드 실전 예제: Node.js 소스 코드**

Node.js

* 자바와는 다른 스크립트 언어
* 이 빌드 패턴은 파이썬, PHP, 루비 등 여타 스크립트 언어에 그대로 적용 가능
* 애플리케이션은 자바스크립트로 구현
* 별도의 컴파일 절차가 필요 없음
* 멀티 스테이지 빌드가 필요하지 않은 것은 아님 -> 멀티 스테이지 빌드를 통해 의존 모듈 로딩을 최적화할 수 있음
* npm(node package manager)이라는 패키지 관리자를 사용해 의존 모듈을 관리



1. diamol/node를 기반 이미지로 사용
2. pakcage.json(builder 단계에서 애플리케이션의 의존 모듈이 정의됨) 파일을 복사
3. npm install 명령을 실행해 의존 모듈을 내려받음

-> 별도의 컴파일이 필요치 않으므로 빌드 과정은 이것이 전부

***4-4* 애플리케이션 빌드 실전 예제: Go 소스 코드**

Go

* 네이티브 바이너리로 컴파일되는 현대적인 크로스 플랫폼 언어

-> 원하는 어떤 플랫폼(윈도, 리눅스, amd64 아키텍처, ARM 아키텍처)이든 해당 플랫폼에서 동작하는 바이너리를 컴파일 할 수 있다는 의미

* 별도의 런타임이 필요하지 않음

-> 도커 이미지의 크기가 매우 작아짐



Dockerfile 스크립트의 각 빌드 단계는 서로 다른 기반 이미지를 사용

1. builder 단계의 기반 이미지는 Go 언어의 도구가 설치된 diamol/golang
2. Go 애플리케이션 빌드는 일반적으로 의존 모듈을 내려받는 단계 없이 곧장 빌드에 들어감
3. 애플리케이션 단계는 최소한의 운영체제 레이어만을 포함하는 이미지를 사용
4. 몇 가지 설정값을 환경 변수 형태로 설정하고 컴파일된 바이너리를 실행해 애플리케이션을 시작
5. 애플리케이션 단계는 builder 단계에서 빌드한 웹 서버 바이너리와 이 웹 서버가 제공할 HTML 파일을 복사하는 과정으로 마무리
6. 바이너리 파일이 chmod 명령을 통해 명시적으로 실행 권한을 부여받음

Go 컴파일러는 비교적 로그 양이 적으며 빌드에 실패한 경우에만 로그를 출력

**빌드에 사용된 Go 빌드 도구 이미지와 빌드된 Go 애플리케이션 이미지의 크기를 비교하는 명령어**

docker image Is -f reference=diamol/golang -f reference=image-gallery

예시에 따르면

Go 빌드 도구를 포함하는 이미지의 크기 : 800MB

실제 Go 애플리케이션 이미지의 크기 : 25MB

* 이 크기는 논리적 크기이므로 이미지 간에 많은 수의 레이어가 공유 됨
* 중요한 부분은 실제 이미지 크기가 아니라 최종 애플리케이션 이미지에 모든 것이 들어가지 않는다는 점
* Go 애플리케이션은 실행 시에 별도의 Go 빌드 도구가 필요 없음

-> 이미지의 크기를 최소한으로 유지하며 용량을 대폭 줄일 수 있는 것

-> 공격이 가능한 부분 자체를 줄일 수 있다는 점에서 큰 장점

***4-5* 멀티 스테이지 Dockerfile 스크립트 이해하기**

장점

1. 표준화

* 어떤 운영체제를 사용하든, 로컬 컴퓨터에 어떤 도구를 설치했는지와 상관없이 모든 빌드 과정은 도커 컨테이너 내부에서 이뤄짐
* 이들 컨테이너는 모든 도구를 정확한 버전으로 갖추고 있음

1. 성능 향상

* 멀티 스테이지 빌드의 각 단계는 자신만의 캐시를 따로 가짐
* 도커는 빌드 중에 각 인스트럭션에 해당하는 레이어 캐시를 찾음
* 처음에 Dockerfile 스크립트를 세심하게 최적화해서 작성한다면 이후로 캐시 재사용을 통해 90% 이상의 빌드 단계에서 시간을 절약 가능

1. 이미지를 작게 유지 가능

* 예시로 curl은 인터넷을 통해 필요한 파일을 내려받을 수 있는 중요한 도구이지만 파일 다운로드를 빌드 초기 단계에 모아 놓는다면 최종 이미지에는 curl을 포함시키지 않아도 됨
* 이미지 크기를 줄여서 애플리케이션의 시작 시간을 단축할 수 있으며 애플리케이션의 의존 모듈 자체를 줄여 취약점을 이용한 외부 공격의 가능성도 최대한 차단 가능