
**확장본#2 - 자바 개발자를 위한 컨테이너 이미지 빌드**

---

**조 훈(Hoon Jo)**

**심근우(Gnu Shim)**

**문성주(Martin Moon)**

# 자바 개발자를 위한 컨테이너 이미지 빌드

쿠버네티스 기반의 시스템을 운영하고 있다면 CI/CD 과정에서 반드시 컨테이너 이미지를 생성하는 과정을 거쳐야 합니다. 이 글에서는 컨테이너의 기본 개념과 자바 개발자를 위한 효과적인 컨테이너 이미지 생성 방법을 알아봅니다.

## 1. 컨테이너 이미지의 기본 개념

우리는 Docker 와 같은 도구를 이용해 컨테이너를 쉽게 구동합니다. 이 때 컨테이너는 이미지를 바탕으로 실행됩니다. 컨테이너의 바탕이 되는 이미지는 어떤 구조를 가지고 있을지를 살펴보기 위해 `docker save` 명령어를 통해 추출한 이미지 파일의 내부를 살펴봅시다.

**[그림 1] docker save 명령어로 저장한 이미지의 압축을 해제한 디렉터리**

```
[root@m-k8s nginx]# ls -al
total 20
drwxr-xr-x. 8 root root 4096 Jan 30 03:49 .
dr-xr-x---. 8 root root  242 Jan 30 04:00 ..
drwxr-xr-x. 2 root root   50 Jan 26 17:58 2061a5d5b8778799056bad17ea4a0ee809ad6b36adb257ee821d735495a62000
drwxr-xr-x. 2 root root   50 Jan 26 17:58 3ddaab3a4de739cd769591cdf10b41b1de0251367fc8c48a1550add140fcb2b5
drwxr-xr-x. 2 root root   50 Jan 26 17:58 7c29ffe9b70e8a6ca4e857a583e089c503d58013f0fd8d81eb8f8f05b6857ed2
drwxr-xr-x. 2 root root   50 Jan 26 17:58 8d9e9db36ad9232f39441734f778fb2529ef25f2d8961514905174f4f6ac93fb
drwxr-xr-x. 2 root root   50 Jan 26 17:58 b204bacdb8dfec3fbfbe23c8fb04d2195f5c4644c474d22b4acf90fa689ec2d7
-rw-r--r--. 1 root root 7657 Jan 26 17:58 c316d5a335a5cf324b0dc83b3da82d7608724769f6454f6d9a621f3ec2534a5a.json
drwxr-xr-x. 2 root root   50 Jan 26 17:58 cdc6084f4bc3608d302bb86c8ef7b1c26f196ed97b8b32d8ae04f42341542e52
-rw-r--r--. 1 root root  586 Jan  1  1970 manifest.json
-rw-r--r--. 1 root root   88 Jan  1  1970 repositories
```

`docker save` 명령어를 통해 이미지를 파일로 저장한 후 압축을 해제하면 해당 디렉터리는 명세 정보를 관리하는 manifest.json, 구동과 관련된 세부 사항이 담긴 config 관련 json 그리고 디렉터리로 구분된 각각의 레이어를 포함하고 있습니다.

먼저 명세 정보를 관리하는 **manifest.json** 을 살펴보도록 하겠습니다.

**[그림 2] 이미지의 명세 정보가 담긴 manifest.json 파일**

```
[root@m-k8s nginx]# cat manifest.json |jq .
[
  {
    "Config": "c316d5a335a5cf324b0dc83b3da82d7608724769f6454f6d9a621f3ec2534a5a.json",
    "RepoTags": [
      "nginx:latest"
    ],
    "Layers": [
      "8d9e9db36ad9232f39441734f778fb2529ef25f2d8961514905174f4f6ac93fb/layer.tar",
      "7c29ffe9b70e8a6ca4e857a583e089c503d58013f0fd8d81eb8f8f05b6857ed2/layer.tar",
      "3ddaab3a4de739cd769591cdf10b41b1de0251367fc8c48a1550add140fcb2b5/layer.tar",
      "cdc6084f4bc3608d302bb86c8ef7b1c26f196ed97b8b32d8ae04f42341542e52/layer.tar",
      "b204bacdb8dfec3fbfbe23c8fb04d2195f5c4644c474d22b4acf90fa689ec2d7/layer.tar",
      "2061a5d5b8778799056bad17ea4a0ee809ad6b36adb257ee821d735495a62000/layer.tar"
    ]
  }
]
```

명세 정보 파일에는 세부사항이 담긴 config 파일의 이름과 이미지의 태그 정보, 그리고 [그림 1]에서 보았던 각 디렉터리 하위에 있는 레이어 관련 파일의 정보가 담겨 있는

것을 볼 수 있습니다. 이미지는 이처럼 여러 레이어로 구성되어 있으며 이미지 빌드시 동일한 내용을 가진 레이어는 재사용될 수 있습니다.

다음으로 config 관련 json 파일을 살펴보겠습니다. 이 파일은 매우 길기 때문에 일부만 살펴보도록 합니다.

**[그림 3] 이미지의 세부 사항이 담긴 config 파일**

```
[root@m-k8s nginx]# cat c316d5a335a5cf324b0dc83b3da82d7608724769f6454f6d9a621f3ec2534a5a.json |jq .
{
  "architecture": "amd64",
  "config": {
    "Hostname": "",
    "Domainname": "",
    "User": "",
    "AttachStdin": false,
    "AttachStdout": false,
    "AttachStderr": false,
    "ExposedPorts": {
      "80/tcp": {}
    },
    "Tty": false,
    "OpenStdin": false,
    "StdinOnce": false,
    "Env": [
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
      "NGINX_VERSION=1.21.6",
      "NJS_VERSION=0.7.2",
      "PKG_RELEASE=1~bullseye"
    ],
    "Cmd": [
      "nginx",
      "-g",
      "daemon off;"
    ],
    "Image": "sha256:1c5303e11e7dbdd166101b7088de8e64576d70ab36e850d8c92a759ce32fe5a9",
    "Volumes": null,
    "WorkingDir": "",
    "Entrypoint": [
      "/docker-entrypoint.sh"
    ],
    "OnBuild": null,
    "Labels": {
      "maintainer": "NGINX Docker Maintainers <docker-maint@nginx.com>"
    },
    "StopSignal": "SIGQUIT"
  },
```

파일의 일부를 살펴보면 플랫폼(CPU 및 특정 OS), 노출할 포트, 환경 변수, 이미지를 컨테이너로 구동할 때 사용할 명령어, 레이블 등 이미지를 실행과 관련된 세부 사항이 이 파일에 담겨 있는 것을 알 수 있습니다.

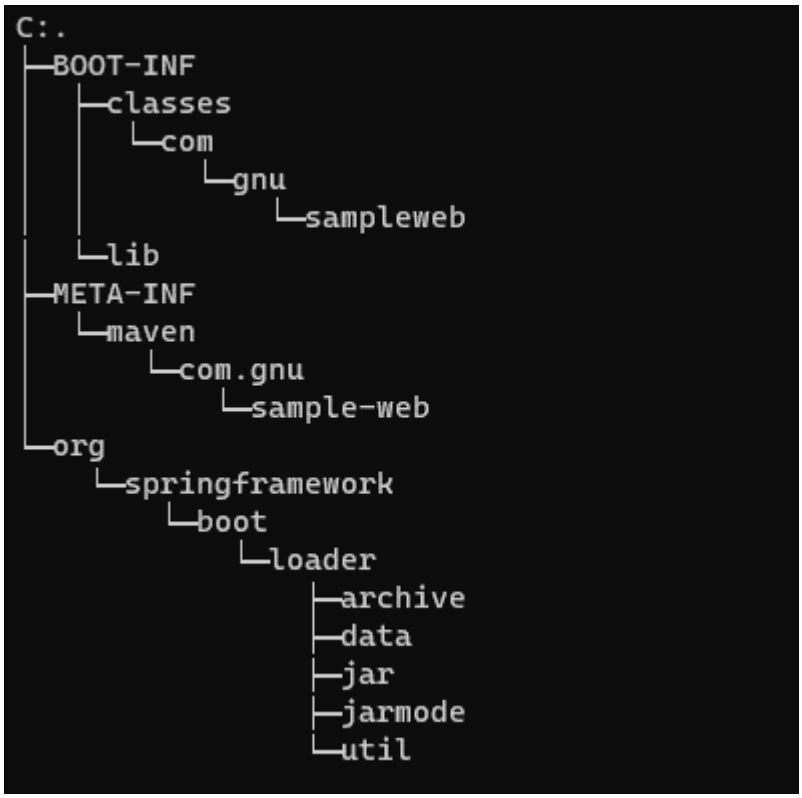
컨테이너를 구동하는데 필요한 추가적인 옵션이나 연결할 볼륨 등을 입력하고 `docker run` 을 실행하면 위와 같은 이미지 정보와 조합하여 컨테이너가 구동되는 것입니다.

## 2. 자바 개발자의 컨테이너 이미지 빌드

자바로 개발한 애플리케이션은 .jar 혹은 .war 의 확장자를 가지는 패키지 파일로 만들어집니다. 이를 컨테이너 기반으로 구동하기 위해 우리는 JDK 를 베이스 이미지로 가지는 Dockerfile 을 작성하고 자바 패키지 파일을 이미지 내부로 복사하도록 한 후 컨테이너의 구동을 위한 ENTRYPOINT 나 CMD 에 자바 애플리케이션을 구동하는

명령어를 삽입합니다. 이 방법은 매우 일반적인 방법이며 잘 동작합니다. 그러나 컨테이너 이미지의 관점에서는 약간의 비효율이 있습니다. 왜 이 방법이 비효율적인지 알아보기 전 널리 사용되는 메이븐(Maven)으로 빌드된 스프링부트 기반 자바 애플리케이션의 패키지 구조를 살펴봅시다.

**[그림 4] 스프링부트 기반 자바 애플리케이션의 패키지 구조**




이 패키지는 구동 관련 정보가 담긴 BOOT-INF 디렉터리, 명세 정보가 담긴 META-INF 디렉터리, 로더와 관련된 정보가 담긴 org 디렉터리로 구성되어 있습니다. 이 때 주목할 것은 구동과 관련된 BOOT-INF 디렉터리입니다. 사용자가 작성한 소스 코드는 BOOT-INF 하위의 classes 디렉터리에 담기게 되며, 외부에서 불러온 의존성 라이브러리들은 lib 디렉터리에 담기게 됩니다. 이 샘플 프로젝트를 기준으로 패키지의 전체 크기는 16MB 이고 그 중 의존성 라이브러리 크기는 12MB 입니다. 반면 직접 작성한 코드는 2KB 에 불과합니다. 사용자가 작성한 코드는 빈번하게 변경되지만 의존성 라이브러리는 프로젝트가 구성된 이후 변경되는 빈도가 낮습니다.

그럼에도 불구하고 이 프로젝트를 앞에서 설명드린 .jar 와 같은 패키지 파일을 생성한 후 컨테이너 이미지 내부로 복사하는 방식으로 컨테이너 이미지를 만들 경우 매번 16MB 의 용량을 차지하게 될 것입니다. 이는 이미지를 사용자의 로컬에 보관하거나 원격에 push 하거나 pull 을 받을 때 디스크 사용량 및 네트워크 전송량의 비효율을 발생시킵니다. 반면에 .jar 파일을 그대로 컨테이너 이미지 내부로 복사하지 않고

사용자가 작성한 코드와 의존성 라이브러리를 분리하여 컨테이너 이미지 내부로 복사하면 이것들이 각각 컨테이너 이미지 생성시 개별 레이어를 구성하게 됩니다.

이 경우 사용자가 작성한 소스 코드에 일부 변경이 있었다면 의존성 라이브러리와 관련된 부분은 재사용하고 사용자가 작성한 소스 코드만을 새로 이미지에 반영할 수 있게 됩니다.

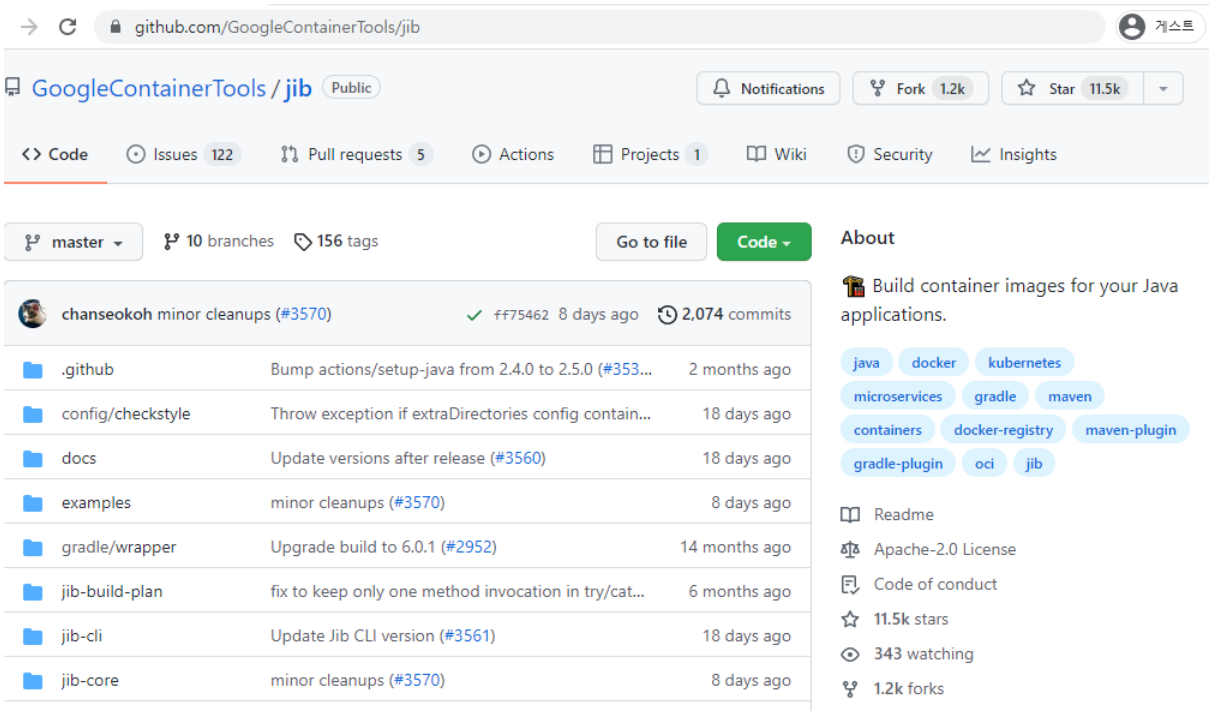
## 3. 좀 더 쉽게 컨테이너 이미지를 생성하는 방법

생성된 .jar 파일을 통해서 효율적으로 컨테이너 이미지를 만들 수 있습니다. 하지만 이를 수행하는 단계에서 패키지의 압축을 해제하고, 일일이 레이어가 될 디렉터리를 구분하여 복사한 후 옵션을 추가한 Dockerfile 을 작성해야 하는 번거로움이 있습니다. 또한 무엇보다도 도커를 구동하기 위해서는 리눅스 기반의 환경이 필요하기 때문에 윈도우 혹은 맥 기반의 개발 환경을 구축한 사용자들은 도커를 사용하기 위해 하이퍼바이저 기반의 별도 환경 구성해야 하는 번거로움이 있습니다.

이제 좀 더 쉽게 이를 수행할 수 있는 방법을 알아봅시다. 스프링부트 기반의 자바 애플리케이션 개발자가 사용할 수 있는 도구로 유명한 것은 크게 두 가지가 있습니다. 하나는 스프링부트에 통합된 Container Native BuildPacks(이하 Buildpacks)이며 또 다른 하나는 **Jib** 입니다. Buildpacks 에 대해서는 이 글에서 자세히 다루지 않고 소개를 하고 넘어가도록 하겠습니다. Buildpacks 은 쿠버네티스를 관리하고 있는 CNCF 에서 출시한 이미지 빌드 도구로 스프링부트와 통합하여 스프링부트 애플리케이션의 패키징 작업과 컨테이너 이미지 생성 과정을 통합할 수 있습니다. 앞서 말씀드린 이미지의 레이어 분리 작업 또한 자동으로 수행할 수 있는 장점이 있습니다. 다만 Buildpacks 은 Docker daemon 에 의존적이기 때문에 윈도우나 맥 기반의 개발 환경에서 별도의 도커를 설치하거나 원격에 설치된 Docker daemon 과 연결을 위한 설정이 필요하다는 단점이 있습니다. 반면에 윈도우나 맥에서도 별도의 Docker daemon 없이 컨테이너 이미지를 생성할 수 있는 도구가 Jib 입니다. 그럼 Jib 에 대해서 알아봅시다.

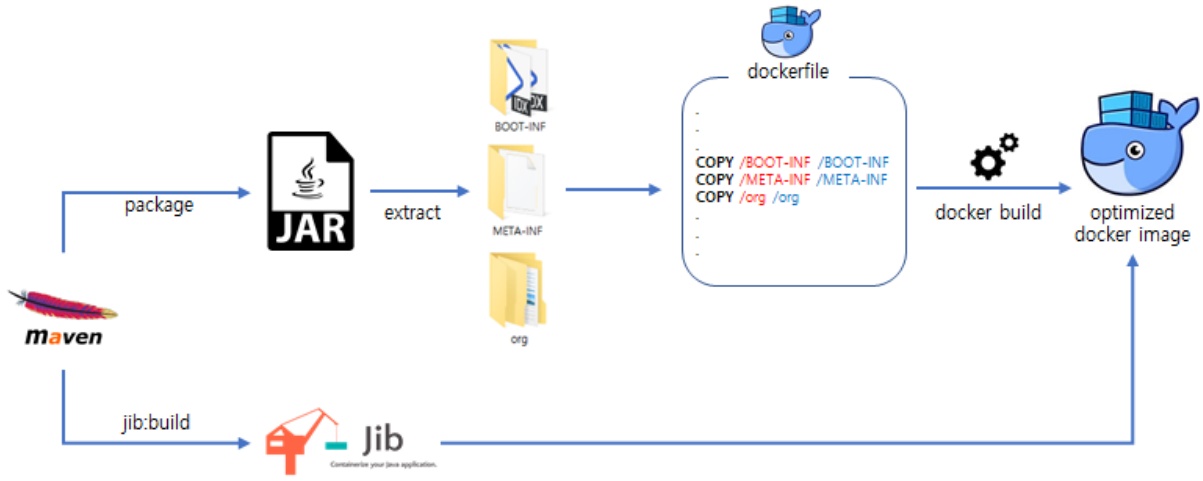
Jib 은 오픈소스 컨테이너 이미지 도구로 구글이 지원하고 있습니다.

**[그림 5] 구글에서 지원하는 jib 깃허브 페이지(https://github.com/GoogleContainerTools/jib)**




Jib 은 명령줄을 통해 사용할 수 있는 Jib CLI 혹은 자바 애플리케이션 빌드 도구인 메이븐 혹은 그레이들(Gradle)과 통합해서 사용할 수 있는 플러그인을 제공하고 있습니다. 주요한 장점으로 내세우는 것은 쉽고 빠른 레이어 재사용, Dockerfile 이나 Docker daemon 이 필요 없는 이미지 생성입니다.

**[그림 6] Jib 을 통해 간편히 도커 이미지를 만드는 과정**




Jib 은 어떻게 윈도우나 맥에서 Docker 에 의존하지 않고 이미지를 생성할 수 있는지를 생각 해 보면 1 절에서 살펴본 것과 같이 컨테이너 이미지는 명세 및 설정 정보와 사용할 레이어 파일을 묶어 놓은 것이므로 이 형식에 맞춘 파일과 디렉터리만 있다면

컨테이너 이미지를 생성하는데는 특정 OS 나 특정 도구가 필요한 것은 아님을 알 수 있습니다.

## 4. jib-maven-plugin 을 활용한 컨테이너 이미지 생성

지난 내용을 정리하자면 Jib 을 통해 컨테이너 이미지를 생성하는 방법은 크게 두 가지가 있습니다. 첫 번째는 Jib CLI 를 이용하는 방법입니다. 이 방법은 Jib 이 지원하는 자체 방식의 YAML 파일을 이용하여 컨테이너 이미지를 생성할 수 있도록 해 줍니다. 두 번째는 자바 애플리케이션 빌드 도구의 플러그인으로 Jib 플러그인을 삽입하는 것입니다. Jib 을 이렇게 사용할 경우 자바 애플리케이션을 빌드하는 작업 과정에서 컨테이너 이미지를 생성할 수 있어 매우 효율적입니다. Jib 은 메이븐 및 그레이들과 통합할 수 있는 플러그인을 지원합니다. 메이븐 기반의 자바 애플리케이션에 jib-maven-plugin(이하 플러그인)을 사용하여 자바 애플리케이션을 컨테이너 이미지로 만들어 보겠습니다.

메이븐 기반의 자바 애플리케이션의 빌드 관련 정보는 `pom.xml` 에 기록됩니다. 이 때 플러그인 정보는 <plugins></plugins> 섹션에 위치하고 있습니다. 기본적으로 별도의 플러그인 설정을 하지 않은 스프링부트 애플리케이션의 플러그인 정보는 아래와 같습니다.

**[그림 7] 스프링 부트 애플리케이션의 pom.xml**

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

이 부분의 아래에 `jib-maven-plugin` 과 관련된 내용을 삽입하여 [그림 6]과 같은 내용이 되도록 합니다.

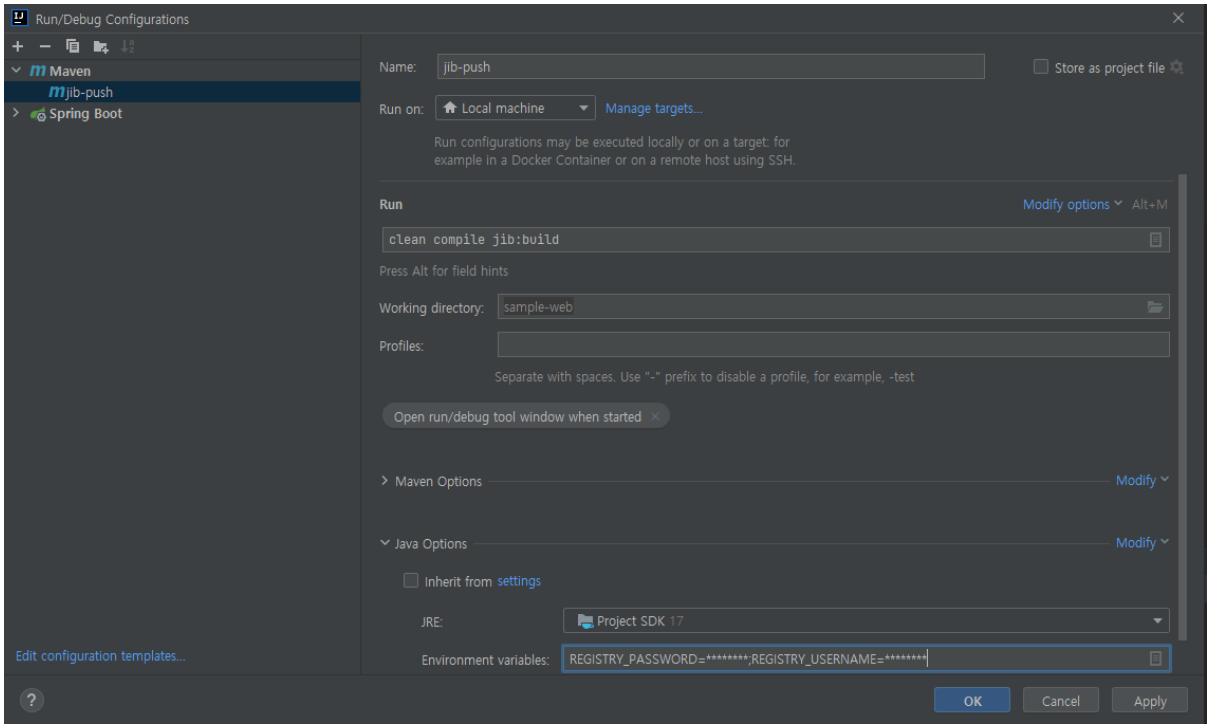
**[그림 8] jib-maven-plugin 삽입**

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
        <plugin>
            <groupId>com.google.cloud.tools</groupId>
            <artifactId>jib-maven-plugin</artifactId>
            <version>3.2.0</version>
            <configuration>
                <to>
                    <image>webfuel/jib-test-project</image>
                    <auth>
                        <username>${env.REGISTRY_USERNAME}</username>
                        <password>${env.REGISTRY_PASSWORD}</password>
                    </auth>
                </to>
            </configuration>
        </plugin>
    </plugins>
</build>
```

이 때 <to></to> 의 사이에 컨테이너 이미지가 push 될 리포지토리 정보를 입력하고 <auth></auth> 의 사이에 레지스트리에 접속할 계정 정보를 입력합니다. 위 플러그인은 빌드된 자바 애플리케이션을 컨테이너 이미지로 생성한 후 환경 변수 REGISTRY_USERNAME, REGISTRY_PASSWORD 를 사용자명과 비밀번호로 도커 허브(Docker Hub)에 접속하여 webfuel/jib-test-project 리포지토리에 이미지를 push 합니다.

이제 널리 사용하는 개발 도구인 **인텔리제이 통합 개발 환경**(IntelliJ IDEA)을 기준으로 자바 애플리케이션을 빌드하는 설정을 해 보겠습니다.

**[그림 9] 인텔리제이 통합 개발 환경의 메이븐 빌드 설정**




이 때 설정을 해 주셔야할 것은 이 실행 환경 구성의 이름, 메이븐 빌드를 실행하기 위한 명령어(Run)와 앞서 pom.xml 에서 환경 변수로 읽어 오도록 설정하였던 사용자명과 비밀번호입니다. pom.xml 은 Github 과 같은 소스 코드 저장소에 공유되는 파일이기 때문에 파일에 직접 계정 정보를 입력하는 것은 위험합니다.

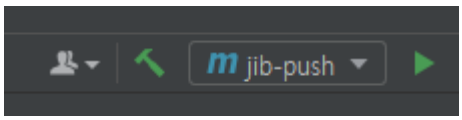
환경의 이름은 Name 항목에 jib-push 로 지정해 주었습니다.

Run 항목에서 입력한 빌드를 실행하기 위한 명령어의 의미는 과거 빌드된 결과물이 있다면 초기화를 시키고(clean) 애플리케이션을 컴파일(compile)한 후 jib-maven-plugin 을 통해서 컨테이너 이미지를 만들고 도커 허브에 push(jib:build)한다는 의미입니다. 이 때 앞서 설명드린 것과 같이 변경되지 않은 부분에 대해서는 레이어를 재사용하기 때문에 좀 더 효율적인 이미지 생성이 이뤄집니다.

또한 그림에서 Environment variables 에 ***로 처리된 사용자명과 비밀번호 부분은 사용하는 실제 값을 입력하셔야 합니다.

이 값을 저장한 후 자바 애플리케이션 빌드를 수행하고 결과를 확인해 보겠습니다.

**[그림 10] 빌드 실행 버튼**




화면 우측 상단의 목록에서 jib-push 를 선택한 후 실행을 누르면 빌드가 수행되면서 아래와 같은 결과를 확인할 수 있습니다.

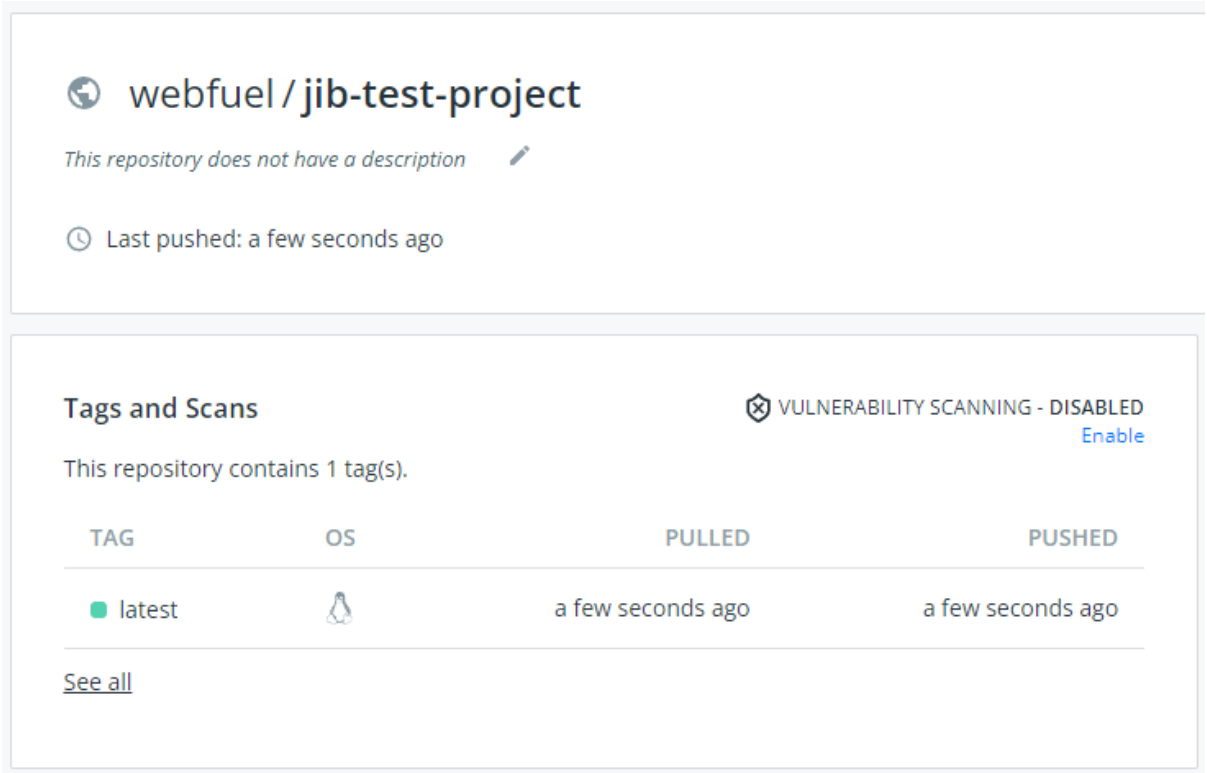
**[그림 11] 빌드 수행 및 컨테이너 이미지 생성**

```
[INFO] Executing tasks:
[INFO] [=======================       ] 75.0% complete
[INFO] > scheduling pushing manifests
[INFO] > pushing blob sha256:25bb960a2c264f210cd023750...
[INFO]
[INFO]
[INFO] Executing tasks:
[INFO] [========================      ] 83.3% complete
[INFO] > scheduling pushing manifests
[INFO]
[INFO]
[INFO] Executing tasks:
[INFO] [========================      ] 83.3% complete
[INFO] > launching manifest pushers
[INFO]
[INFO]
[INFO] Executing tasks:
[INFO] [========================      ] 83.3% complete
[INFO] > pushing manifest for latest
[INFO]
[INFO]
[INFO] Executing tasks:
[INFO] [============================  ] 91.7% complete
[INFO] > building images to registry
[INFO]
[INFO]
[INFO] Executing tasks:
[INFO] [============================  ] 91.7% complete
[INFO] > launching manifest list pushers
[INFO]
[INFO]
[INFO] Executing tasks:
[INFO] [==============================] 100.0% complete
```

Jib 가 자바 애플리케이션을 레이어로 만들고 명세 정보 및 이미지를 원격 저장소로 푸시하는 것을 확인할 수 있습니다.

작업이 완료되었으면 도커 허브에서 해당 리포지토리를 확인해 보도록 하겠습니다.

**[그림 12] 도커 허브 확인**




pom.xml 에서 입력했던 **webfuel/jib-test-project** 리포지토리에 이미지가 정상적으로 push 된 것을 확인할 수 있습니다. 이 과정에는 Dockerfile 작성이나 도커 설치가 필요하지 않습니다. Jib 을 사용하면 OS 환경이나 도커 설치에 구애받지 않고, 자바 애플리케이션에 최적화된 구조의 이미지를 개발 도구 안에서 쉽게 생성할 수 있음을 알아보았습니다.