Dockerfile 작성하기

Dockerfile 형식

Docker는 Dockerfile의 인스트럭션(instruction)을 읽어 자동으로 이미지를 빌드할 수 있습니다.
Dockerfile은 사용자가 이미지를 만들기 위해 커맨드라인에서 호출할 수 있는 모든 명령이 포함된 텍스트 문서입니다.
Dockerfile의 형식은 다음과 같습니다.

```
# Comment
INSTRUCTION arguments
```

인스트럭션은 대소문자를 구분하지 않지만 관례적으로 대문자를 사용합니다.

Docker는 Dockerfile의 명령을 순서대로 실행합니다.
Dockerfile은 FROM 명령어로 시작해야 합니다. 파서 지시문(Parser directives), 코멘트(Comments), 글로벌 스코프 ARG 뒤에 올 수 있습니다.
FROM 명령어는 빌드할 상위 이미지를 지정합니다.

BuildKit은 해당 행이 유효한 파서 지시문이 아닌 한 #으로 시작하는 행을 주석으로 처리합니다.
이전 버전과의 호환성을 위해 주석(#) 및 명령어(예: RUN) 앞의 공백은 무시되지만 권장되지 않습니다. 이러한 경우 선행 공백은 유지되지 않으므로 다음 예는 동일합니다.
그러나 명령에 전달된 인수(arguments)의 공백은 무시되지 않습니다.

```
    # this is a comment-line
  RUN echo hello
RUN echo world
```

```
# this is a comment-line
RUN echo hello
RUN echo world
```

파서 지시문(Parser directives)

파서 지시문은 선택 사항이며 Dockerfile의 후속 행(lines)이 처리되는 방식에 영향을 미칩니다.
파서 지시문은 빌드에 레이어(layer)를 추가하지 않으며 빌드 단계로 표시되지 않습니다.
파서 지시문은 # directive=value 형식의 특별한 타입의 주석으로 작성됩니다.
하나의 지시문은 한 번만 사용할 수 있습니다.

주석(comment), 빈 행(empty line), 빌더 명령(instruction)이 처리되면 BuildKit은 더 이상 파서 지시문을 찾지 않습니다.
대신 파서 지시문으로 형식화된 모든 것을 주석으로 처리하고 파서 지시문이 될 수 있는지 확인하려고 하지 않습니다.
따라서 모든 파서 지시문은 Dockerfile의 맨 위에 있어야 합니다.

파서 지시문은 대소문자를 구분하지 않지만 관례에 따라 소문자입니다.
또한 파서 지시문 뒤에 빈 줄을 포함하는 것이 관례입니다.
파서 지시문에서는 줄 연속 문자(Line continuation characters : '\')가 지원되지 않습니다.

이러한 규칙으로 인해 다음 예제들은 모두 유효하지 않습니다.

▶ 줄 연속 문자(\)로 인해 유효하지 않음

```
# direc \
tive=value
```

▶ 하나의 지시문이 두 번 나타나므로 유효하지 않음

```
# directive=value1
# directive=value2

FROM ImageName
```

▶ 명령 뒤에 나타나므로 주석으로 처리됨

```
FROM ImageName
# directive=value
```

▶ 지시문이 주석 뒤에 있어서 주석으로 처리됩니다.

```
# About my dockerfile
# directive=value
FROM ImageName
```

▶ 알 수 없는 지시문은 인식되지 않으므로 주석으로 처리됩니다. 따라서 뒤에 있는 지시문도 주석 뒤에 나타나기 때문에 주석으로 처리됩니다.

```
# unknowndirective=value
# syntax=value
```

▶ 파서 지시문은 줄 바꿈이 아닌 공백(space, horizontal tab)이 허용됩니다. 따라서 다음 행은 모두 동일하게 취급됩니다.

```
#directive=value
# directive =value
#       directive= value
# directive = value
#         dIrEcTiVe=value
```

다음 파서 지시문이 지원됩니다.

- syntax
  syntax 지시문을 사용하여 빌드에 사용할 Dockerfile 구문 버전을 선언합니다. 지정하지 않으면 BuildKit은 Dockerfile 프런트엔드의 번들 버전을 사용합니다.
  syntax 버전을 선언하면 BuildKit 또는 Docker 엔진을 업그레이드하거나 사용자 지정 Dockerfile 구현을 사용하지 않고도 최신 Dockerfile 버전을 자동으로 사용할 수 있습니다.
  대부분의 사용자는 이 파서 지시문을 `docker/dockerfile:1`로 설정하려고 할 것입니다. 그러면 BuildKit이 빌드하기 전에 Dockerfile 구문의 최신 안정 버전을 가져옵니다.

  ```
  # syntax=docker/dockerfile:1
  ```

- escape
  escape 지시문은 Dockerfile에서 문자를 이스케이프하는 데 사용되는 문자를 설정합니다. 지정하지 않으면 기본 이스케이프 문자는 \입니다.
  이스케이프 문자는 줄에 특정 문자를 이스케이프하고 또 개행(newline)을 이스케이프 하는 데 사용됩니다. 이를 통해 Dockerfile의 명령이 여러 줄에 걸쳐 있을 수 있습니다.

  excape 지시문이 Dockerfile에 포함여부와 관계없이 줄 끝을 제외하고는 `RUN` 명령에서 이스케이프가 수행되지 않습니다.
  이스케이프 문자를 `(Grave accent)로 설정하는 것은 Windows에서 특히 유용합니다. Windows는 디렉터리 구분을 \로 하기 때문입니다.
  Windows에서 명확하지 않은 방식으로 실패하는 다음 예를 고려하십시오.

  두 번째 줄 끝에 두 번째 \는 첫 번째 \에서 이스케이프 대상이 아니라 개행(newline)에 대한 이스케이프로 해석됩니다.
  마찬가지로 세 번째 줄 끝에 있는 \는 실제로 명령어로 취급되었다고 가정할 때, 줄 연속 문자(Line continuation characters)로 처리됩니다.
  이 Dockerfile의 결과는 두 번째 줄과 세 번째 줄이 하나의 명령어로 간주된다는 것입니다.

  ```
  FROM microsoft/nanoserver
  COPY testfile.txt c:\\
  RUN dir c:\
  ```

  다음과 같이 이스케이프 문자를 `(Grave accent)로 하면 해결됩니다.

  ```
  # escape=`

  FROM microsoft/nanoserver
  COPY testfile.txt c:\
  RUN dir c:\
  ```

Environment replacement

환경 변수(ENV 문으로 선언됨)는 Dockerfile에서 해석할 변수로 특정 명령(instructions)에서 사용될 수도 있습니다.
환경 변수는 $variable_name 또는 ${variable_name}으로 Dockerfile에 표시됩니다.
중괄호 구문은 일반적으로 ${foo}_bar와 같이 공백이 없는 변수 이름 문제를 해결하는 데 사용됩니다.

${variable_name} 구문은 아래에 지정된 몇 가지 표준 bash 수정자(Modifiers)를 지원합니다.

- ${variable:-word} : 변수에 값이 설정된 경우 결과가 해당 값이 됨을 나타냅니다. 변수가 설정되지 않은 경우 word가 결과 값이 됩니다.
- ${variable:+word} : 변수에 값이 설정된 경우 word가 결과이고, 그렇지 않은 경우 빈 문자열(empty string)이 결과가 됨을 나타냅니다.
- ${variable#pattern} : 문자열의 시작 부분부터 검색하여 변수에서 가장 짧은 패턴 일치를 제거합니다.

```
str=foobarbaz echo ${str#f*b}     # arbaz
```

- ${variable##pattern} : 문자열의 시작 부분부터 검색하여 변수에서 가장 긴 패턴 일치를 제거합니다.

```
str=foobarbaz echo ${str##f*b}    # az
```

- ${variable%pattern} : 문자열 끝에서 거꾸로 검색하여 변수에서 가장 짧은 패턴 일치를 제거합니다.

```
string=foobarbaz echo ${string%b*}    # foobar
```

- ${variable%%pattern} : 문자열의 끝에서 거꾸로 검색하여 변수에서 가장 긴 패턴 일치를 제거합니다.

```
string=foobarbaz echo ${string%%b*}   # foo
```

- ${variable/pattern/replacement} : 변수에서 처음 나타나는 패턴을 대체 항목으로 바꿉니다.

```
string=foobarbaz echo ${string/ba/fo}  # fooforbaz
```

- ${variable//pattern/replacement} : 변수의 모든 패턴 발생을 대체 항목으로 바꿉니다.

```
string=foobarbaz echo ${string//ba/fo}  # fooforfoz
```

pattern은 Glob Pattern 입니다. 여기서 '?'는 모든 단일 문자, '*'는 임의 개수의 문자(0개 포함)와 일치합니다.
문자 '?', '*'와 일치시키려면 백슬래시로 이스케이프 하면 된다 \?, * 이렇게.

몇가지 사용되는 Glob Pattern	
*(asterisk)	matches any number of characters (including none).
?	matches any single character
**(double asterisk)	matches all files and zero or more directories and subdirectories. If followed by a `/` it matches only directories and subdirectories.
[abc]	matches one character in the brackets
[a-z,A-Z]	matches any uppercase or lowercase letter.
[!abc]	matches any character not in the brackets
{abc,123}	comma-delimited set of literals, matched 'abc' or '123'
\\	matches a single backslash
\?	matches the question mark.

환경 변수는 Dockerfile의 다음 명령(instruction) 목록에서 지원됩니다.

- `ADD`
- `COPY`
- `ENV`
- `EXPOSE`
- `FROM`
- `LABEL`
- `STOPSIGNAL`
- `USER`
- `VOLUME`
- `WORKDIR`

환경 변수를 RUN, CMD 및 ENTRYPOINT 명령어와 함께 사용할 수도 있지만, 이 경우 변수 대체(variable substitution)는 빌더가 아닌 command shell에서 처리됩니다.
exec 형식을 사용하는 명령(instruction)은 command shell을 자동으로 호출하지 않습니다.
환경 변수 대체(substitution)는 전체 명령어에서 각 변수에 대해 동일한 값을 사용합니다. 변수 값 변경은 후속 명령(subsequent instructions)에서만 적용됩니다.

```
ENV abc=hello
ENV abc=bye def=$abc
ENV ghi=$abc
```

위 예제에서 변수 def의 값은 hello가 되고, 변수 ghi의 값은 bye가 됩니다.

Shell form과 exec form

RUN, CMD, ENTRYPOINT 명령어는 모두 두 가지 가능한 형식이 있습니다.

```
INSTRUCTION ["executable","param1","param2"] (exec form)
INSTRUCTION command param1 param2 (shell form)
```

exec 형식을 사용하면 shell 문자열 문제를 방지하고 특정 명령 shell이나, 기타 실행 파일을 사용하여 명령을 호출할 수 있습니다.
이는 배열의 각 요소가 명령(command), 플래그(flag), 인수(argument)인 JSON 배열 문법을 사용합니다.

shell 형식은 좀 더 편안하고 사용 편의성, 유연성, 가독성을 강조합니다. shell 형식은 자동으로 command shell을 사용하는 반면, exec 형식은 그렇지 않습니다.

Exec form

exec 형식은 JSON 배열로 구문 분석됩니다. 즉, 단어 앞뒤에 작은따옴표(')가 아닌 큰따옴표(")를 사용해야 합니다. exec 형식은 실행 시 재정의할 수 있는 기본 인수를 설정하기 위해 CMD과 결합된 ENTRYPOINT 명령어를 지정하는 데 가장 적합합니다.

▶ Variable substitution
위에서 shell 형식은 자동으로 command shell을 사용하는 반면, exec 형식은 그렇지 않다고 말했습니다.
이는 변수 대체(substitution)와 같은 일반적인 shell 처리가 발생하지 않음을 의미합니다.
예를 들어 RUN ["echo", "$HOME"]은 $HOME에 대한 변수 대체를 처리하지 않습니다.
shell 처리를 원할 경우 shell 형식을 사용하거나 다음 예제와 같이 exec 형식으로 직접 shell을 실행하세요.
`RUN [ "sh", "-c", "echo $HOME" ]`
exec 형식을 사용하고 shell을 직접 실행하는 경우, shell 형식의 경우처럼 환경 변수 대체를 수행한느 것은 빌더(builder)가 아닌 shell입니다.

▶ Backslashes
exec 형식에서는 백슬래시(\)를 이스케이프 처리해야 합니다. 이는 특히 백슬래시가 경로 구분 기호인 Windows에서 주의해야 한다.

Shell form

exec 형식과 달리 shell 형식을 사용하는 명령어(instructions)는 항상 command shell을 사용합니다.
shell 형식은 JSON 배열 형식을 사용하지 않고 일반 문자열을 사용합니다.
shell 형식은 문자열을 사용하면 이스케이프 문자(기본적으로 백슬래시)를 사용하여 개행(newline)을 이스케이프하여 다음 행에 단일 명령(instruction)을 계속할 수 있습니다.
이렇게 하면 긴 명령을 여러 줄로 나룰 수 있기 때문에 더 긴 명령을 사용하기가 더 쉬워집니다.

```
RUN source $HOME/.bashrc && \
echo $HOME
```

명령을 분할하기 위해 shell 형식과 함께 heredocs를 사용할 수도 있습니다.

```
RUN <<EOF
source $HOME/.bashrc && \
echo $HOME
EOF
```

다른 shell 사용하기

SHELL 명령(instruction)을 사용하여 기본 shell을 변경할 수 있습니다.

```
SHELL ["/bin/bash", "-c"]
RUN echo hello
```

인스트럭션(instruction)

FROM

```
FROM [--platform=<platform>] <image> [AS <name>]
FROM [--platform=<platform>] <image>[:<tag>] [AS <name>]
FROM [--platform=<platform>] <image>[@<digest>] [AS <name>]
```

FROM 명령은 새 빌드 단계를 초기화하고 후속 명령어에 대한 기본 이미지를 설정합니다.
따라서 유효한 Dockerfile은 FROM 명령어(instruction)로 시작해야 합니다. image는 모든 유효한 image일 수 있습니다.

- `ARG`는 FROM 앞에 올 수 있는 유일한 명령(instruction)입니다.
- FROM은 단일 Dockerfile 내에서 여러 번 나타나 여러 이미지를 생성하거나 하나의 빌드 단계를 다른 빌드 단계의 종속성(dependency)으로 사용할 수 있습니다.
  각각의 새로운 FROM 명령 이전에 커밋(commit)에 의해 출력된 마지막 이미지 ID를 기록해 두십시오. 각 FROM 명령은 이전 명령에서 생성된 모든 상태를 지웁니다.
- 선택적으로 FROM 명령어에 AS 이름을 추가하여 새 빌드 단계에 이름을 지정할 수 있습니다.
  이름은 후속 FROM 및 COPY --from=<name> 명령에서 이 단계에서 빌드된 이미지를 참조하는 데 사용될 수 있습니다.
- tag 또는 digest 값은 선택 사항입니다. 둘 중 하나를 생략하면 빌더는 기본적으로 latest 태그를 가정합니다.
  빌더는 tag 값을 찾을 수 없으면 오류를 반환합니다.

멀티 플랫폼 이미지를 참조하는 경우 선택적으로 --platform 플래그를 사용하여 이미지의 플랫폼을 지정할 수 있습니다.

▶ Understand how ARG and FROM interact(ARG와 FROM이 상호 작용하는 방식 이해)
FROM 명령(instruction)은 첫 번째 FROM 이전에 발생하는 ARG 명령에 의해 선언된 변수를 지원합니다.

```
ARG  CODE_VERSION=latest
FROM base:${CODE_VERSION}
CMD  /code/run-app

FROM extras:${CODE_VERSION}
CMD  /code/run-extras
```

FROM 이전에 선언된 ARG는 빌드 단계 외부에 있으므로 FROM 이후의 어떤 명령어(instruction)에도 사용할 수 없습니다.
첫 번째 FROM 이전에 선언된 ARG의 기본값을 사용하려면 빌드 단계 내부에 값이 없는 ARG 명령어를 사용하세요.

```
ARG VERSION=latest
FROM busybox:$VERSION
ARG VERSION
RUN echo $VERSION > image_version
```

RUN

RUN 명령어는 모든 명령을 실행하여 현재 이미지 위에 새 레이어를 생성합니다. 추가된 레이어는 Dockerfile의 다음 단계에서 사용됩니다.
위에서 설명했듯 RUN, CMD, ENTRYPOINT 명령은 exec 형식과 shell 형식이 있습니다.
shell 형식은 가장 일반적으로 사용되며 개행(newline), 이스케이프, heredoc을 사용하여 더 긴 명령을 여러 줄로 더 쉽게 나눌 수 있습니다.

CMD

CMD 명령어는 이미지에서 컨테이너를 실행할 때 실행할 명령을 설정합니다.
마찬가지로 exec 형식과 shell 형식이 있습니다.

Dockerfile에는 CMD 명령이 하나만 있을 수 있습니다. CMD를 두 개 이상 나열하면 마지막 CMD만 적용됩니다.
CMD의 목적은 실행 중인 컨테이너에 기본값을 제공하는 것입니다.
이러한 기본값에는 실행 파일이 포함되거나 실행 파일이 생략될 수 있으며, 이 경우 ENTRYPOINT 명령도 지정해야 합니다.

컨테이너가 매번 동일한 실행 파일을 실행하도록 하려면 ENTRYPOINT를 CMD와 함께 사용하는 것을 고려해야 합니다.
사용자가 docker run에 인수를 지정하면 CMD에 지정된 기본값을 재정의하지만 여전히 기본 ENTRYPOINT를 사용합니다.
CMD를 사용하여 ENTRYPOINT 명령어에 대한 기본 인수를 제공하는 경우 CMD 및 ENTRYPOINT 명령어 모두 exec 형식에 지정해야 합니다.
RUN과 CMD를 혼동하지 마십시오.
RUN은 실제로 명령을 실행하고 결과를 커밋합니다. CMD는 빌드 시 아무것도 실행하지 않지만 이미지에 대해 의도된 명령을 지정합니다.

LABEL

```
LABEL <key>=<value> <key>=<value> <key>=<value> ...
```

LABEL 명령어는 이미지에 메타데이터를 추가합니다. LABEL은 키-값 쌍입니다.
LABEL value에 공백을 포함시키려면 명령줄 구문 분석에서와 마찬가지로 따옴표와 백슬래시를 사용합니다.
몇 가지 사용 예시

```
LABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

이미지는 여러 개의 라벨(label)을 가질 수 있습니다. 한 줄에 여러 개의 라벨을 지정할 수 있습니다.
다음은 단일 명령에서 여러 라벨을 지정하는 예시입니다.

```
LABEL multi.label1="value1" \
multi.label2="value2" \
other="value3"
```

작은 따옴표(single quotes)가 아닌 큰 따옴표(double quotes)를 사용해야 합니다.
특히 문자열 보간법(예: LABEL example="foo-$ENV_VAR")을 사용하는 경우 작은 따옴표는 변수 값을 풀지 않고 문자열을 있는 그대로 사용합니다.

기본 이미지 또는 부모 이미지(FROM 줄의 이미지)에 포함된 라벨은 이미지에 상속됩니다.
라벨이 이미 존재하지만 값이 다른 경우 가장 최근에 적용된 값이 이전에 설정된 값을 재정의합니다.

이미지의 라벨을 보려면 `docker image inspect` 명령을 사용하세요. `--format` 옵션을 사용하여 라벨만 표시할 수 있습니다.

```
$ docker image inspect --format='{{json .Config.Labels}}' myimage
```

MAINTAINER (deprecated)

EXPOSE

```
EXPOSE <port> [<port>/<protocol>...]
```

EXPOSE 명령은 컨테이너가 런타임 시 지정된 네트워크 포트에서 listen 한다는 것을 Docker에게 알립니다.
포트가 TCP 또는 UDP에서 listen 하는지 지정할 수 있으며, 프로토콜을 지정하지 않으면 기본값은 TCP입니다.

EXPOSE 명령은 실제로 포트를 게시하지 않습니다.
이는 이미지를 구축(build)하는 사람과 게시할 포트에 대한 컨테이너를 실행하는 사람 사이의 일종의 문서 역할을 합니다.
컨테이너를 실행할 때 포트를 게시하려면 docker run에서 -p(소문자) 플래그를 사용하여 하나 이상의 포트를 게시하고 매핑하거나

-P(대문자) 플래그를 사용하여 노출된 모든 포트를 게시하고 상위 포트에 매핑합니다.

TCP와 UDP 모두에 노출하려면 다음 두 줄을 포함합니다.

```
EXPOSE 80/tcp
EXPOSE 80/udp
```

이 경우 docker run과 함께 -P(대문자)를 사용하면 포트가 TCP에 대해 한 번, UDP에 대해 한 번 노출됩니다.
-P(대문자)는 호스트에서 임시 상위 호스트 포트(high-ordered host port)를 사용하므로 TCP와 UDP는 동일한 포트를 사용하지 않습니다.

EXPOSE 설정과 관계없이 -p(소문자) 플래그를 사용하여 런타임에 재정의할 수 있습니다.

```
$ docker run -p 80:80/tcp -p 80:80/udp ...
```

호스트 시스템에서 포트(port) 리다이렉션 설정하려면 -P 플래그 사용을 참고하세요.
`docker network` 명령은 네트워크에 연결된 컨테이너가 모든 포트를 통해 서로 통신할 수 있기 때문에 특정 포트를 노출(expose)하거나 게시할 필요 없이 컨테이너 간 통신을 위한 네트워크 생성을 지원합니다.

ENV

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

ENV 명령은 환경 변수를 설정합니다.
이 value값은 빌드 단계의 모든 후속 명령들(instructions)에 대한 환경에 있으며 설정할 값이 많은 경우 인라인으로 대체될 수도 있습니다.
value는 다른 환경 변수에 대해 해석되므로 이스케이프되지 않으면 따옴표 문자는 제거됩니다.
명령줄 구문 분석(command line parsing)과 마찬가지로 따옴표와 백슬레시를 사용하여 value 내에 공백을 포함할 수 있습니다.

```
ENV MY_NAME="John Doe"
ENV MY_DOG=Rex\ The\ Dog
ENV MY_CAT=fluffy
```

아래와 같이 한 번에 여러 <key>=<value>... 변수를 설정할 수 있습니다. 두 예제 결과는 같습니다.

```
ENV MY_NAME="John Doe" MY_DOG=Rex\ The\ Dog \
MY_CAT=fluffy
```

ENV로 설정한 환경 변수는 결과 이미지에서 컨테이너가 실행(run)될 때 유지됩니다.
`docker inspect`를 사용하여 값을 확인하고 `docker run --env <key>=<value>`를 사용하여 값을 변경할 수 있습니다.

스테이지(stage)는 부모(parent) 스테이지 또는 모든 조상(ancestor) 스테이지에서 ENV를 사용하여 설정된 모든 환경 변수를 상속합니다. ==> 멀티 스테이지 빌드(Multi-stage builds) 참고

환경 변수 지속성은 예상치 못한 부작용을 일으킬 수 있습니다.
예를 들어, ENV DEBIAN_FRONTEND=noninteractive 로 설정하면 apt-get의 동작이 변경되어 이미지 사용자에게 혼란을 줄 수 있습니다.

ENV 명령은 '='를 생략하는 대체 문구 ENV <key> <value>도 허용합니다.
ENV MY_VAR my-value
⛔ 권장하는 구문은 아님!!, 이 구문은 단일 ENV 명령에 여러 환경 변수를 설정할 수 없고 혼란스러움

ADD

로컬 또는 원격(remote)에 있는 파일과 디렉터리를 추가합니다.
경로에 공백이 포함되어 있으면 아래의 형식 중 두 번째 형식을 사용하면 된다.
ADD 명령은 <src>로부터 새 파일, 디렉터리, 원격 파일 URL을 복사하여 <dest>경로에 있는 이미지의 파일 시스템에 추가합니다.
여러 <src> 리소스를 지정할 수 있지만, 이 리소스가 파일 또는 디렉터리인 경우 해당 경로는 빌드 컨텍스트의 소스에 상대적인 것으로 해석됩니다.
각각의 <src>에는 와일드카드가 포함될 수 있으며 `Go의 filepath.Match` 규칙을 사용하여 수행됩니다.
<dest>는 source가 대상 컨테이너 내부에 복사될 절대 경로 또는 WORKDIR에 대한 상대 경로입니다.

```
ADD [--chown=<user>:<group>] [--chmod=<perms>] [--checksum=<checksum>] <src>... <dest>
ADD [--chown=<user>:<group>] [--chmod=<perms>] ["<src>",... "<dest>"]
```

--chmod는 Dockerfile 1.3부터 지원됩니다. 현재는 8진수 표기법만 지원됩니다.

```
ADD hom* /mydir/            # "hom"으로 시작하는 모든 파일을 추가
ADD hom?.txt /mydir/        # ?는 임의의 단일 문자와 일치합니다. 예를 들어 "home.txt"
ADD test.txt relativeDir/   # 상대 경로를 지정하고, <WORKDIR>/relativeDir/ 에 "test.txt"를 추가합니다.
ADD test.txt /absoluteDir/  # 절대 경로를 지정하고, /absoluteDir/ 에 "test.txt"를 추가합니다.
```

특수 문자(예: '[', ']')가 포함된 파일이나 디렉터리를 추가할 때 Golang 규칙에 따라 이스케이프해야 합니다.
예를 들어 이름이 "arr[0].txt"인 파일을 추가하려면 다음을 사용합니다.

```
ADD arr[[]0].txt /mydir/          # 앞에 있는 대괄호를 이스케이프 처리???
```

--chown 플래그로 추가된 컨텐츠의 특정 소유권을 요청하기 위해 사용자 이름, 그룹 이름, 또는 UID/GID 조합을 지정하지 않으면, 모든 파일과 디렉터리는 UID 및 GID 가 0으로 생성됩니다.
--chown 플래그의 형식은 사용자 이름(username)과 그룹 이름(groupname) 문자열 또는 정수 UID와 GID의 조합입니다.
그룹 이름 없이 사용자 이름을 제공하거나 또는 GID 없이 UID를 제공하면 GID를 UID와 같은 숫자를 사용합니다.
사용자 이름 또는 그룹 이름이 제공되면 컨테이너의 /etc/passwd 과 /etc/group 파일이 각각 사용자 이름, 그룹 이름에서 정수 UID, GID로 변환을 수행하는 데 사용됩니다.
다음 예제는 --chown 플래그의 유효한 정의를 보여줍니다.

```
ADD --chown=55:mygroup files* /somedir/
ADD --chown=bin files* /somedir/
ADD --chown=1 files* /somedir/
ADD --chown=10:11 files* /somedir/
ADD --chown=myuser:mygroup --chmod=655 files* /somedir/
```

컨테이너의 루트 파일 시스템에 /etc/passwd 와 /etc/group 파일이 없으면 --chown 플래그 사용시 ADD 작업에서 빌드가 실패합니다.

<src>가 원격(remote) 파일 URL인 경우 <dest> 의 권한(permission)은 정수값 600입니다.

STDIN(docker build - < somefile)을 통해 Dockerfile을 전달하여 빌드하는 경우 빌드 컨텍스트가 없으므로 Dockerfile에는 URL 기반 ADD 명령만 포함될 수 있습니다.
STDIN을 통해 압축된 아카이브(archive)를 전달할 수도 있습니다 (docker build - < archive.tar.gz), 아카이브 루트에 있는 Dockerfile과 아카이브의 나머지 부분이 빌드 컨텍스트로 사용됩니다.

<src>의 내용이 변경된 경우 처음 만난 ADD 명령은 Dockerfile의 모든 후속 명령에 대한 캐시를 무효화합니다.
여기에는 RUN 명령에 대한 캐시 무효화가 포함됩니다.

🔴 `ADD`는 다음 규칙을 따릅니다. 🔴

- <src>경로는 빌드 컨텍스트 내에 있어야 합니다. builder는 컨텍스트의 파일에만 엑세스할 수 있고, ../something 은 빌드 컨텍스트 루트의 상위 파일 또는 상위 디렉터리를 지정하기 때문에
  COPY ../something /something 을 사용할 수 없습니다.
- <src>가 디렉터리인 경우 파일 시스템 메타데이터를 포함하여 디렉터리의 전체 내용이 복사됩니다.
  ▶ 디렉터리 자체는 복사되지 않고 해당 내용만 복사됩니다.
- <src>가 URL이고 <dest>가 후행 슬래시(trailing slash)로 끝나면 URL에서 파일 이름이 유추되고 파일이 <dest>/<filename>에 다운로드됩니다.
  예를 들어 ADD http://example.com/foobar /는 /foobar 파일을 생성합니다.
  이 경우 적절한 파일 이름을 찾을 수 있도록 URL에는 중요한(nontrivial) 경로가 있어야 합니다.
- <src>가 인식되는 압축 형식(identity, gzip, bzip2, xz)의 로컬 tar 아카이브인 경우 디렉토리로 압축이 풀립니다.
  원격(remote) URL의 리소스는 압축 해제되지 않습니다. 디렉터리를 복사하거나 압축 해제할 때 tar -x 와 같은 동작을 합니다.
  결과는 다음의 결합입니다.
  i. destination 경로에 존재하는 모든 것
  ii. 파일별로 "2"를 선호하여 충돌이 해결된 소스 트리의 내용

  파일이 인식되는 압축 형식(identity, gzip, bzip2, xz)으로 식별되는지 여부는 파일 이름이 아닌 파일 내용에 따라 결정됩니다.
  예를 들어 빈 파일이 .tar.gz로 끝나는 경우 압축 파일로 인식되지 않고, 압축 해제 오류 메시지를 생성하지 않으며, 파일이 destination에 복사됩니다.

- <src>가 다른 종류의 파일인 경우 메타데이터와 함께 개별적으로 복사됩니다.
  이 경우, <dest>가 후행 슬래시(trailing slash)로 끝나면 디렉터리로 간주되어 <src>의 내용이 <dest>/base(<src>)에 기록됩니다.
- 직접 또는 와일드카드 사용으로 인해 여러 <src> 리소스가 지정된 경우 <dest>는 디렉터리여야 하며 슬래시(/)로 끝나야 합니다.
- <dest>가 슬래시(/)로 끝나지 않으면 일반 파일로 간주되어 <src>의 내용이 <dest>에 기록됩니다.
- <dest>가 존재하지 않으면 해당 경로에 누락된 모든 디렉터리와 함께 생성됩니다.
  ===> 마치 리눅스 mkdir 명령의 -p 옵션 처럼??

COPY

```
COPY [--chown=<user>:<group>] [--chmod=<perms>] <src>... <dest>
COPY [--chown=<user>:<group>] [--chmod=<perms>] ["<src>",... "<dest>"]
```

위 두가지 형식 중 두 번째 형식은 공백이 포함된 경로에 필요합니다.
COPY 명령은 <src>에서 새로운 파일이나 디렉터리를 복사하여 <dest> 경로에 있는 컨테이너의 파일 시스템에 추가합니다.

여러 <src> 리소스를 지정할 수 있지만 파일 및 디렉터리의 경로는 빌드 컨텍스트의 source에 상대경로로 해석됩니다.

각 <src>에는 와일드카드가 포함될 수 있으며 `Go의 filepath.Match` 규칙을 사용하여 수행됩니다.

```
COPY hom* /mydir/                 # "hom"으로 시작하는 모든 파일을 추가
COPY hom?.txt /mydir/             # ?는 임의의 단일 문자와 일치합니다. 예를 들어 "home.txt"

# <dest>는 절대 경로 또는 source가 destination 컨테이너 내부에 복사될 WORKDIR에 대한 상대 경로입니다.
COPY test.txt relativeDir/        # 상대 경로를 지정하고, <WORKDIR>/relativeDir/ 에 "test.txt"를 추가합니다.
COPY test.txt /absoluteDir/       # 절대 경로를 지정하고, /absoluteDir/ 에 "test.txt"를 추가합니다.
```

특수 문자(예: '[', ']')가 포함된 파일이나 디렉터리를 복사할 때 Golang 규칙에 따라 이스케이프해야 합니다.
예를 들어 이름이 "arr[0].txt"인 파일을 복사하려면 다음을 사용합니다.

```
COPY arr[[]0].txt /mydir/        # 앞에 있는 대괄호를 이스케이프 처리???
```

--chown 플래그로 복사된 컨텐츠의 특정 소유권을 요청하기 위해 사용자 이름, 그룹 이름, 또는 UID/GID 조합을 지정하지 않으면, 모든 파일과 디렉터리는 UID 및 GID 가 0으로 생성됩니다.
--chown 플래그의 형식은 사용자 이름(username)과 그룹 이름(groupname) 문자열 또는 정수 UID와 GID의 조합입니다.
그룹 이름 없이 사용자 이름을 제공하거나 또는 GID 없이 UID를 제공하면 GID를 UID와 같은 숫자를 사용합니다.
사용자 이름 또는 그룹 이름이 제공되면 컨테이너의 /etc/passwd 과 /etc/group 파일이 각각 사용자 이름, 그룹 이름에서 정수 UID, GID로 변환을 수행한는 데 사용됩니다.
다음 예제는 --chown 플래그의 유효한 정의를 보여줍니다.

```
COPY --chown=55:mygroup files* /somedir/
COPY --chown=bin files* /somedir/
COPY --chown=1 files* /somedir/
COPY --chown=10:11 files* /somedir/
COPY --chown=myuser:mygroup --chmod=644 files* /somedir/
```

컨테이너의 루트 파일 시스템에 /etc/passwd 와 /etc/group 파일이 없으면 --chown 플래그 사용시 COPY 작업에서 빌드가 실패합니다.

STDIN(docker build - < somefile)을 통해 Dockerfile을 전달하여 빌드하는 경우 빌드 컨텍스트가 없으므로 COPY를 사용할 수 없습니다.

선택적으로 COPY 명령어는 source 위치를 사용자가 보낸 빌드 컨텍스트 대신 사용할 이전 빌드 스테이지(FROM ... AS <name>으로 생성됨)로 설정하는 데 사용할 수 있는 --from=<name> 플래그를 허용합니다.
지정된 이름의 빌드 스테이지(stage)를 찾을 수 없는 경우 동일한 이름의 이미지를 대신 사용하려고 시도합니다.

🔴 `COPY`는 다음 규칙을 따릅니다. 🔴

- <src>경로는 빌드 컨텍스트 내에 있어야 합니다. builder는 컨텍스트의 파일에만 엑세스할 수 있고, ../something 은 빌드 컨텍스트 루트의 상위 파일 또는 상위 디렉터리를 지정하기 때문에
  COPY ../something /something 을 사용할 수 없습니다.
- <src>가 디렉터리인 경우 파일 시스템 메타데이터를 포함하여 디렉터리의 전체 내용이 복사됩니다.
  ▶ 디렉터리 자체는 복사되지 않고 해당 내용만 복사됩니다.
- <src>가 다른 종류의 파일인 경우 메타데이터와 함께 개별적으로 복사됩니다.
  이 경우, <dest>가 후행 슬래시(trailing slash)로 끝나면 디렉터리로 간주되어 <src>의 내용이 <dest>/base(<src>)에 기록됩니다.
- 직접 또는 와일드카드 사용으로 인해 여러 <src> 리소스가 지정된 경우 <dest>는 디렉터리여야 하며 슬래시(/)로 끝나야 합니다.
- <dest>가 슬래시(/)로 끝나지 않으면 일반 파일로 간주되어 <src>의 내용이 <dest>에 기록됩니다.
- <dest>가 존재하지 않으면 해당 경로에 누락된 모든 디렉터리와 함께 생성됩니다.
  ===> 마치 리눅스 mkdir 명령의 -p 옵션 처럼??

처음 만나는 COPY 명령은 <src>의 내용이 변경된 경우 Dockerfile의 모든 후속 명령에 대한 캐시를 무효화합니다.
여기에는 RUN 명령어에 대한 캐시 무효화가 포함됩니다.

ENTRYPOINT

ENTRYPOINT를 사용하면 실행 파일로 실행될 컨테이너를 구성할 수 있습니다.
위에서 설명했듯 exec form과 shell form이 있습니다.

```
ENTRYPOINT ["executable", "param1", "param2"] (exec form)
ENTRYPOINT command param1 param2 (shell form)
```

docker run <image>에 대한 명령줄 인수는 exec 형식으로 ENTRYPOINT의 모든 요소 뒤에 추가되며 CMD를 사용하여 지정된 모든 요소를 재정의합니다.
이를 통해 인수(arguments)가 entry point에 전달될 수 있습니다. 즉, docker run <image> -d 는 -d 인수를 entry point에 전달합니다.
🟢 docker run --entrypoint 플래그를 사용하여 ENTRYPOINT 명령을 재정의할 수 있습니다.

🟢 ENTRYPOINT의 shell 형식은 CMD 명령줄 인수가 사용되는 것을 방지합니다.
또한 시그널을 전달하지 않는 /bin/sh -c의 하위 명령으로 ENTRYPOINT를 시작합니다. 이는 실행 파일이 컨테이너의 PID 1이 아니며 Unix 시그널을 수신하지 않음을 의미합니다.
이 경우 실행 파일은 docker stop <container>에서 SIGTERM을 수신하지 않습니다.

Dockerfile의 마지막 ENTRYPOINT 명령만 적용됩니다.

다음 Dockerfile은 ENTRYPOINT를 사용하여 Apache를 포그라운드(foreground)에서 실행하는 방법을 보여줍니다.

```
FROM debian:stable
RUN apt-get update && apt-get install -y --force-yes apache2
EXPOSE 80 443
VOLUME ["/var/www", "/var/log/apache2", "/etc/apache2"]
ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

실행 파일의 시작 스크립트를 작성해야 하는 경우 exec 및 gosu 명령을 사용하여 최종 실행 파일이 Unix 시그널을 수신하도록 할 수 있습니다.

```
#!/usr/bin/env bash
set -e
```

```
if [ "$1" = 'postgres' ]; then
    chown -R postgres "$PGDATA"

    if [ -z "$(ls -A "$PGDATA")" ]; then
        gosu postgres initdb
    fi

    exec gosu postgres "$@"
fi

exec "$@"
```

- ENTRYPOINT의 shell 형식은 `/bin/sh -c` 에서 실행됩니다.
- shell 형식은 shell 환경 변수를 대체하고, CMD 또는 docker run 명령의 인수를 무시합니다.
- `docker stop`이 오랫동안 실행 중인 ENTRYPOINT 실행 파일에 올바르게 시그널을 보내도록 하려면 `exec`로 시작해야 합니다.

```
FROM ubuntu
ENTRYPOINT exec top -b
```

CMD와 ENTRYPOINT가 상호 작용하는 방식 이해

`CMD`, `ENTRYPOINT` 명령어는 컨테이너를 실행할 때 실행되는 명령(command)을 정의합니다.
CMD, ENTRYPOINT 두 명령의 협력(cooperation)을 설명하는 규칙이 조금은 있습니다.

- Dockerfile은 CMD 또는 ENTRYPOINT 명령 중 하나 이상을 지정해야 합니다.
- 컨테이너를 실행 파일로 사용할 때 ENTRYPOINT를 정의해야 합니다.
- CMD는 ENTRYPOINT 명령에 대한 기본 인수를 정의하거나 컨테이너에서 임시(ad-hoc) 명령(command)을 실행하는 방법으로 사용해야 합니다.
- 컨테이너를 대체 인수와 함께 실행할 때 CMD가 재정의됩니다.
- 아래의 표를 보면 JSON 배열 형식으로 입력하지 않으면 CMD와 ENTRYPOINT에 명시된 명령어의 앞에 `/bin/sh -c`가 추가됩니다.

	ENTRYPOINT 없음	ENTRYPOINT exec_entry p1_entry	ENTRYPOINT ["exec_entry", "p1_entry"]
CMD 없음	ERROR 발생	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD ["exec_cmd", "p1_cmd"]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry exec_cmd p1_cmd
CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

CMD가 기본 이미지(base image)에서 정의된 경우 ENTRYPOINT를 설정하면 CMD가 빈 값으로 재설정됩니다.
이 시나리오에서는 CMD가 값을 가지려면 현재 이미지에 정의되어야 합니다.

VOLUME

USER

WORKDIR