Docker Multi-stage Image Build

In this workshop, you will learn how to build the smallest possible docker image using just a single Dockerfile that will describe the compilation and then the packaging images.

The resulting image will be more than x1000 times smaller than the one we start with.

Let's get to work!

1. Create a file called app.go with a simple program in Go language:

```
mkdir /project

cd /project

cat <<EOF > app.go
package main

import "fmt"

func main() {
  fmt.Println("Hello world!")
}
EOF
```

2. Inspect the file content and size

```
ls -altr app.go

cat app.go
```

3. Create a Dockerfile with build and packaging instructions:

```
cat <<EOF > Dockerfile.big
FROM golang:1.10
COPY app.go .
RUN go build -o /app app.go
CMD ["/app"]
EOF
```

   The golang:1.10 image is based on Debian 9 (Stretch) as the OS and then the Go compiler is installed as well.

4. Build a docker image, check its size and run the container

```
docker build -t go-big -f Dockerfile.big .

docker image ls go-big

docker run -ti --rm go-big
```

Notice how the go-big image takes a whopping 796mb!
That Debian OS and Go compiler sure take a lot of space.

5. Use a different base image that is using Alpine Linux instead of Debian

```
cat <<EOF > Dockerfile.alpine
FROM golang:1.10-alpine
COPY app.go .
RUN go build -o /app app.go
CMD [ "/app" ]
EOF

docker build -t go-alpine -f Dockerfile.alpine .

docker image ls go-alpine

docker run -ti --rm go-alpine
```

Notice how the go-alpine image takes only 396mb, that is quite an improvement.
This is because Alpine Linux takes only 4.5mb, most of the space is still occupied by the Go compiler.

6. What is taking so much space? It certainly not the little Hello World application.
Create an image that is not including the compiler in the resulting image.

```
cat <<EOF > Dockerfile.multi-stage
FROM golang:1.10-alpine AS build-stage
COPY app.go .
RUN go build -o /app app.go

FROM alpine
COPY --from=build-stage /app /
CMD [ "/app" ]
EOF

docker build -t go-multi-stage -f Dockerfile.multi-stage .

docker image ls go-multi-stage

docker run -ti --rm go-multi-stage
```

Amazing result! The image is just 6.42mb and works just like before by printing Hello World.
The Dockerfile mentions FROM twice, this is called a multi-stage build. Only the last stage is stored as the resulting image.

7. Take it to the extreme by using scratch

```
cat <<EOF > Dockerfile.scratch
FROM golang:1.10-alpine AS build-stage
COPY app.go .
RUN go build -o /app app.go
```

```
FROM scratch
COPY --from=build-stage /app /
CMD [ "/app" ]
EOF

docker build -t go-scratch -f Dockerfile.scratch .

docker image ls go-scratch

docker run -ti --rm go-scratch
```

The application binary is the only file in the resulting image, there are no other binaries. Which makes the image take **2mb** of space.

scratch is the zero-files docker image, it is a reserved word for "empty".

No more docker optimizations left, but can the application itself be made even smaller?

8. Why does a Hello World in Go take 2mb? Partly it has to do with debug symbols left in the resulting binary. Remove the symbols and try making the application itself even smaller.

```
cat <<EOF > Dockerfile.stripped
FROM golang:1.10-alpine AS build-stage
RUN apk update && apk add binutils
COPY app.go .
RUN go build -o /app app.go
RUN strip /app

FROM scratch
COPY --from=build-stage /app /
CMD [ "/app" ]
EOF

docker build -t go-stripped -f Dockerfile.stripped .

docker image ls go-stripped

docker run -ti --rm go-stripped
```

The build process is a bit longer due to the installation of binutils and using strip on the resulting binary. But the result of all that hard work is an image that takes just **1.2mb**!

9. We already made the resulting binary smaller by stripping it of symbols.
But how about we change the software itself to take even less space?

```
cat <<EOF > app-smaller.go
package main

func main() {
  println("Hello world!")
```

```
}
EOF

cat <<EOF > Dockerfile.smaller
FROM golang:1.10-alpine AS build-stage
RUN apk update && apk add binutils
COPY app-smaller.go .
RUN go build -o /app app-smaller.go
RUN strip /app

FROM scratch
COPY --from=build-stage /app /
CMD [ "/app" ]
EOF

docker build -t go-smaller -f Dockerfile.smaller .

docker image ls go-smaller

docker run -ti --rm go-smaller
```

It still prints a beautiful "Hello World!" but the image only takes **682kb**.

Remember the first image we built?

796mb was *1167* times bigger than **682kb**.

Quite a difference.

Can you apply these principles to your own applications?