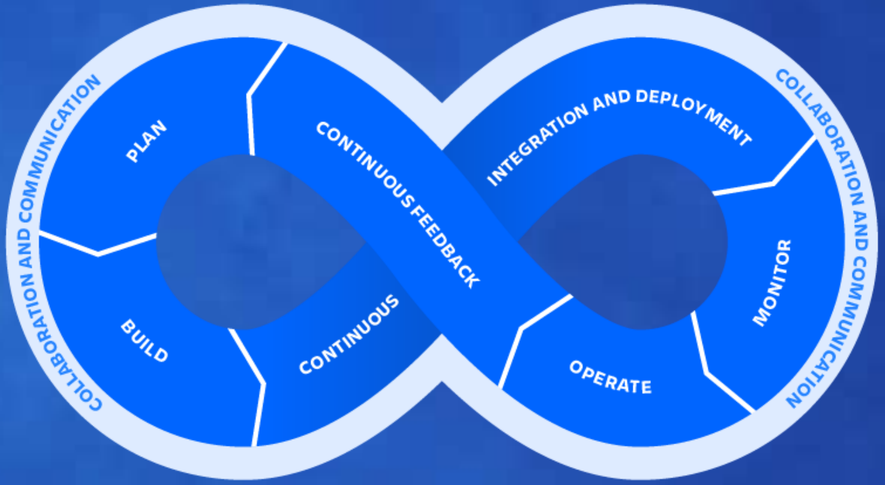

COLLABORATION AND COMMUNICATION
PLAN
CONTINUOUS FEEDBACK
INTEGRATION AND DEPLOYMENT
COLLABORATION AND COMMUNICATION
MONITOR
OPERATE
CONTINUOUS
BUILD

Session 13. Custom images

Ram N Sangwan

Agenda

- Design and code a Docker file to build a custom container image.
- Containerizing Apps using multiple images.
- Running a container from the custom image.

Create a Custom Image

- Install/verify required packages for building the program

```
# yum install -y gcc-c++ glibc-static libstdc++ compat-libstdc++ libstdc++-static
```

- Create a folder called tspscratch and create a file hello.cc inside it

```
# mkdir tspscratch
# cd tspscratch
```

Create a Program for the container

```
# vi hello.cc
#include <iostream>
using namespace std;
int main(){
cout << "Hello! Welcome to my Container - The SkillPedia \n ";
return 0;
}
```

Compile and Build the Image from scratch.

- Compile the program with

  # gcc -o hello -static hello.cc

- Test the Program.

  # ./hello

- Create the Dockerfile

  # vi Dockerfile

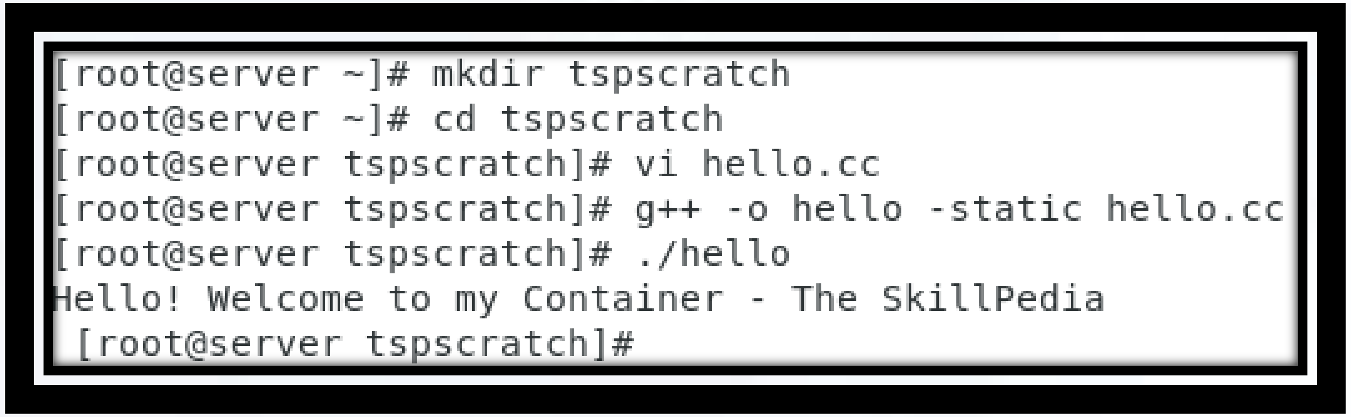
  FROM scratch

  ADD hello /

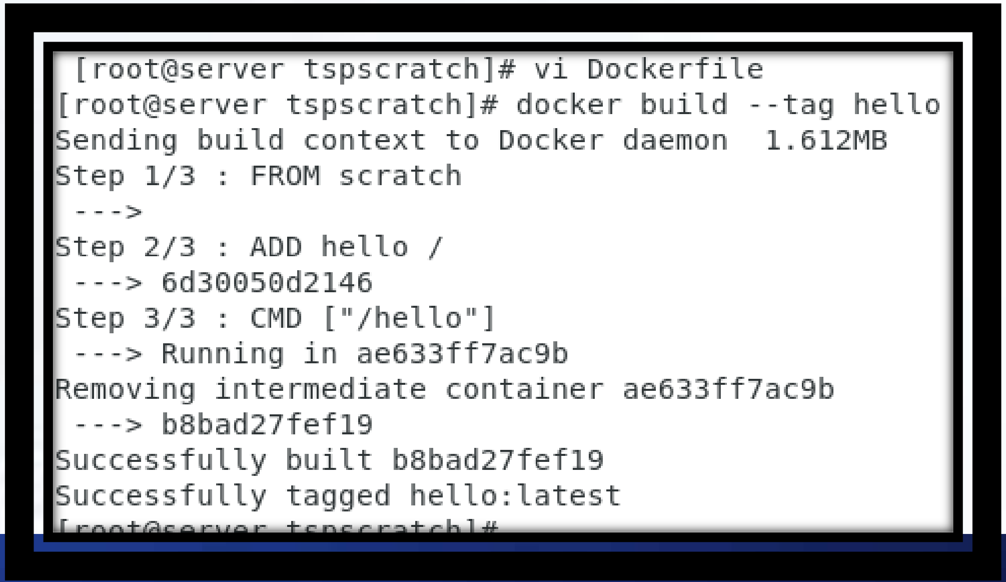
  CMD ["/hello"]

  Build Image from Dockerfile

  # docker build --tag hello .



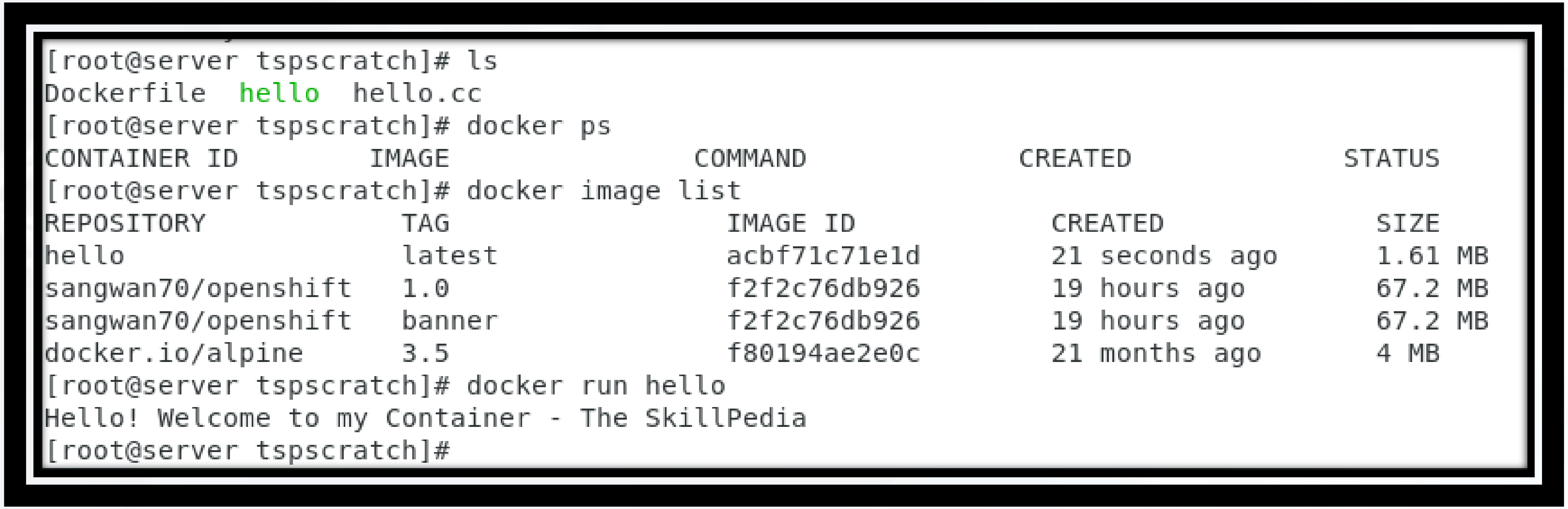
```
[root@server ~]# mkdir tspscratch
[root@server ~]# cd tspscratch
[root@server tspscratch]# vi hello.cc
[root@server tspscratch]# g++ -o hello -static hello.cc
[root@server tspscratch]# ./hello
Hello! Welcome to my Container - The SkillPedia
 [root@server tspscratch]#
```



```
 [root@server tspscratch]# vi Dockerfile
[root@server tspscratch]# docker build --tag hello
Sending build context to Docker daemon  1.612MB
Step 1/3 : FROM scratch
 --->
Step 2/3 : ADD hello /
 ---> 6d30050d2146
Step 3/3 : CMD ["/hello"]
 ---> Running in ae633ff7ac9b
Removing intermediate container ae633ff7ac9b
 ---> b8bad27fef19
Successfully built b8bad27fef19
Successfully tagged hello:latest
[root@server tspscratch]#
```

Run your Container

docker run hello



```
[root@server tspscratch]# ls
Dockerfile  hello  hello.cc
[root@server tspscratch]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
[root@server tspscratch]# docker image list
REPOSITORY            TAG                 IMAGE ID            CREATED             SIZE
hello                 latest              acbf71c71e1d        21 seconds ago      1.61 MB
sangwan70/openshift   1.0                 f2f2c76db926        19 hours ago        67.2 MB
sangwan70/openshift   banner              f2f2c76db926        19 hours ago        67.2 MB
docker.io/alpine      3.5                 f80194ae2e0c        21 months ago       4 MB
[root@server tspscratch]# docker run hello
Hello! Welcome to my Container - The SkillPedia
[root@server tspscratch]#
```

Container from Multiple Images - Multi-Stage Builds

What is It?

- Multi-stage builds are a method of organizing a Dockerfile to include multiple images.

How?

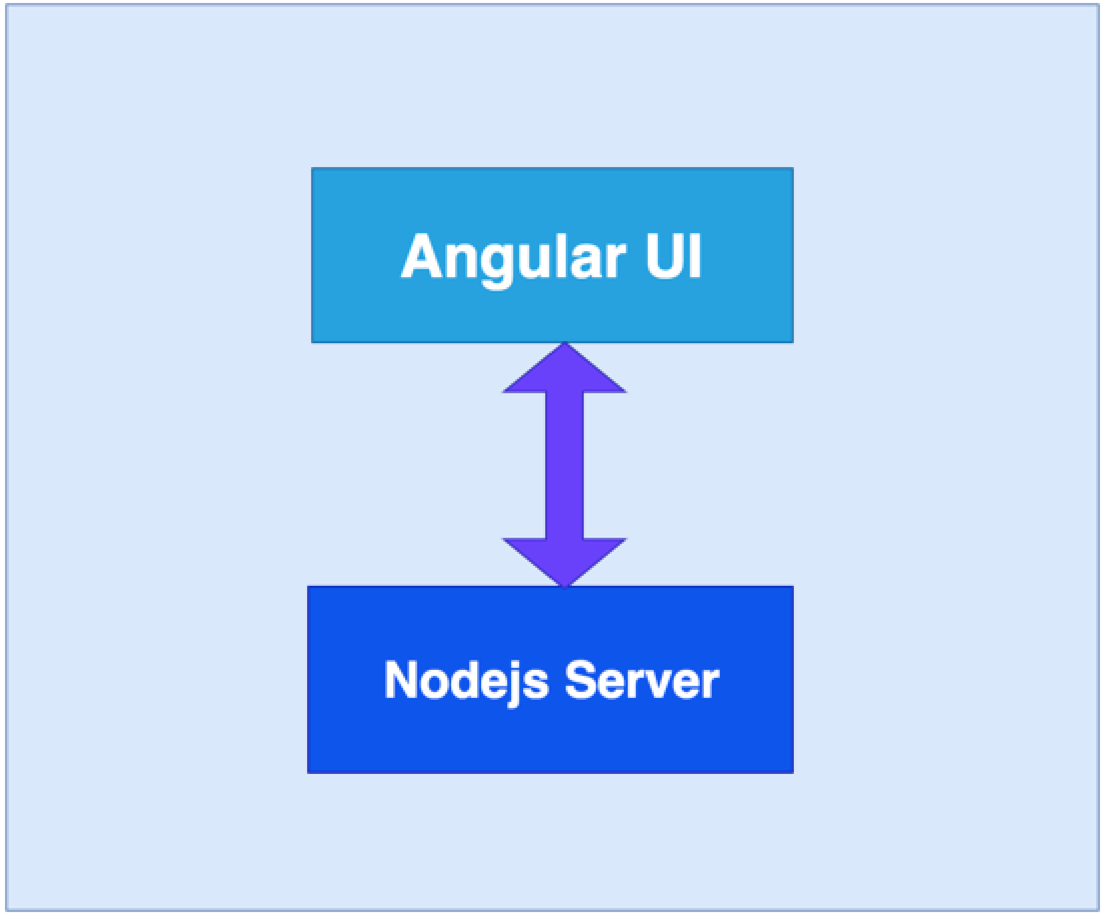
- By creating different sections of a Dockerfile, each referencing a different base image.
- This allows a multi-stage build to fulfill a function previously filled by using multiple docker files, copying files between containers, or running different pipelines.

Key Benefits

- Minimize the size of the final container.
- Improve run time performance
- Allow for better organization of Docker commands and files
- Provide a standardized method of running build actions.

Example Project

- It’s a simple web app with Angular and node app server.
- Look at the diagram to understand better.
- We have a UI built with Angular and running on the nodejs server.

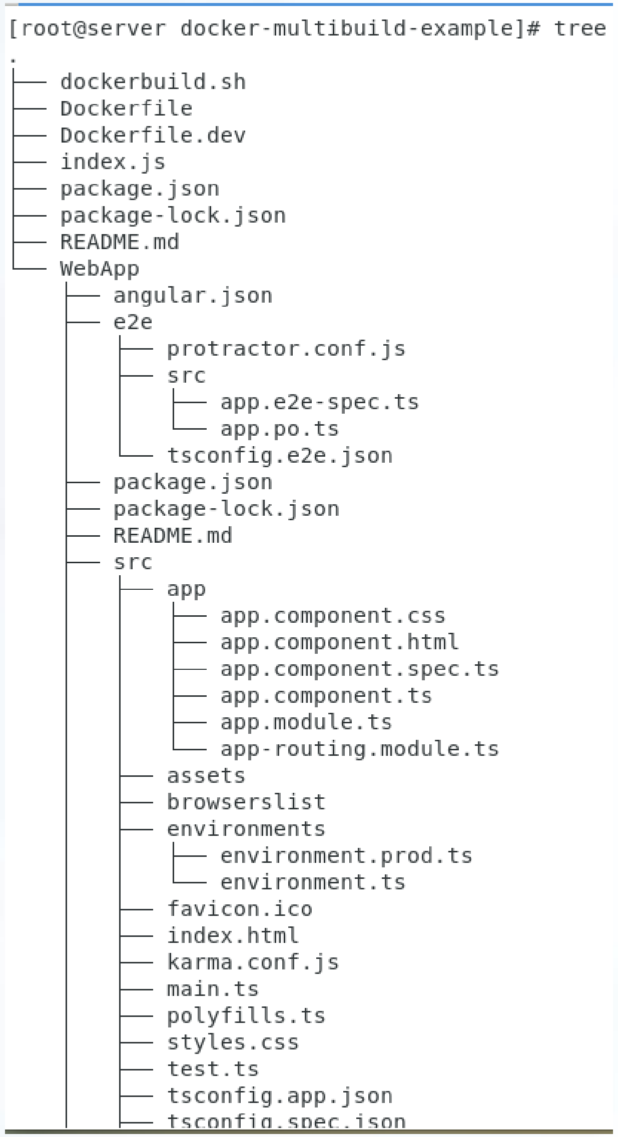


Setup the Environment

- Clone a project.

  # git clone https://github.com/Sangwan70/docker-multibuild-example.git

  # cd docker-multibuild-example/
- We are not going to build any functionality in the app just to keep it simple.
- We have a simple index.js for Nodejs server and serve Angular app on port 3070.
- Use tree command to check the Project Tree



```
[root@server docker-multibuild-example]# tree
.
├── dockerbuild.sh
├── Dockerfile
├── Dockerfile.dev
├── index.js
├── package.json
├── package-lock.json
├── README.md
└── WebApp
    ├── angular.json
    ├── e2e
    │   ├── protractor.conf.js
    │   ├── src
    │   │   ├── app.e2e-spec.ts
    │   │   └── app.po.ts
    │   └── tsconfig.e2e.json
    ├── package.json
    ├── package-lock.json
    ├── README.md
    ├── src
    │   ├── app
    │   │   ├── app.component.css
    │   │   ├── app.component.html
    │   │   ├── app.component.spec.ts
    │   │   ├── app.component.ts
    │   │   ├── app.module.ts
    │   │   └── app-routing.module.ts
    │   ├── assets
    │   ├── browserslist
    │   ├── environments
    │   │   ├── environment.prod.ts
    │   │   └── environment.ts
    │   ├── favicon.ico
    │   ├── index.html
    │   ├── karma.conf.js
    │   ├── main.ts
    │   ├── polyfills.ts
    │   ├── styles.css
    │   ├── test.ts
    │   ├── tsconfig.app.json
    │   ├── tsconfig.spec.json
```

Building Image Using Dockerfile

- Start from the base image *node:10*
- There are two package.json files: one is for nodejs server and another is for Angular UI.
- We need to copy these into Docker file system and install all the dependencies.
- We need this step first to build images faster in case there is a change in the source later.
- We don’t want to repeat installing dependencies every time we change any source files.
- Angular uses Angular/cli to build the app. So, install CLI and install all the dependencies.
- Run npm run build to build the Angular App and all the assets will be created under dist folder within WebApp folder.

Our Docker File

- Switch to the cloned directory and take the backup of existing Dockerfile

```
[root@server ~]# cd docker-multibuild-example/
[root@server docker-multibuild-example]# mv Dockerfile Dockerfile.multi
```

  - Now create a new Dockerfile to build a single image from our source.

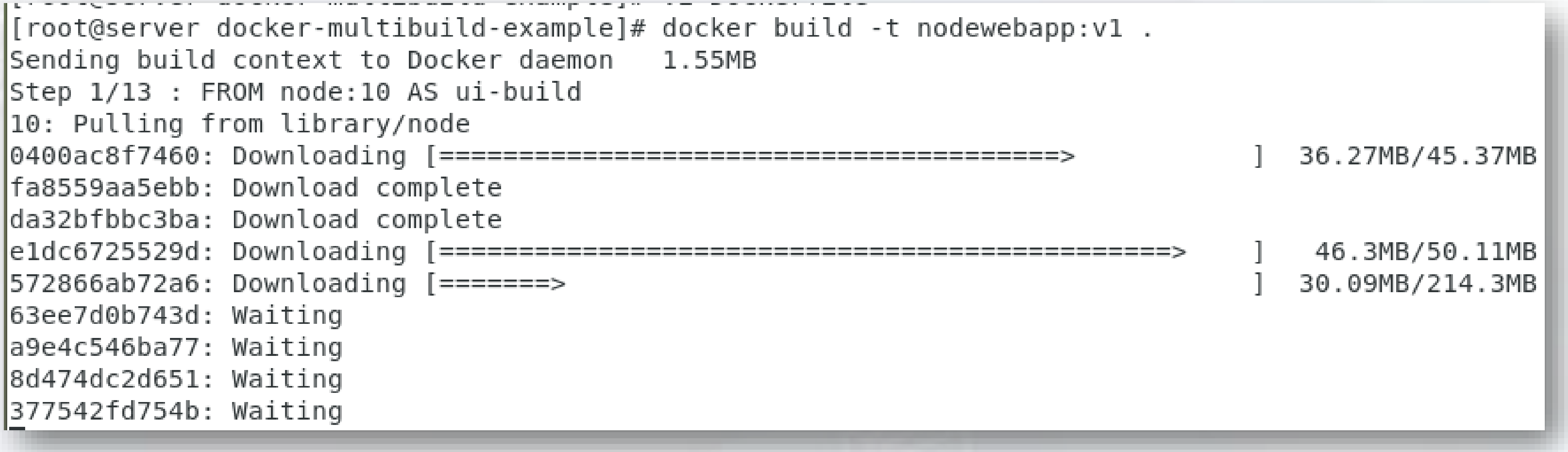
```
FROM node:10
WORKDIR /usr/src/app
COPY package*.json ./
COPY WebApp/package*.json ./WebApp/
# RUN npm install for node js dependencies
RUN npm install \
  && cd WebApp \
  && npm install @angular/cli \
  && npm install
```

```
# Bundle app source
COPY . .
RUN cd WebApp && npm run build
EXPOSE 3070
ENTRYPOINT ["node"]
CMD ["index.js"]
```

Run the Build Command

- Let’s build the image. I am giving it a tag nodewebapp:v1.
- It takes some time to build an image since we are installing two package.json dependencies and Angular/cli.
- Ignore the warnings generated

docker build -t nodewebapp:v1 .



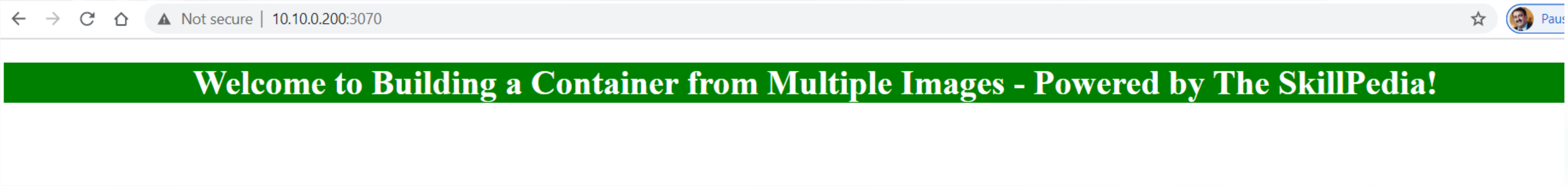
```
[root@server docker-multibuild-example]# docker build -t nodewebapp:v1 .
Sending build context to Docker daemon   1.55MB
Step 1/13 : FROM node:10 AS ui-build
10: Pulling from library/node
0400ac8f7460: Downloading [=========================================>          ]  36.27MB/45.37MB
fa8559aa5ebb: Download complete
da32bfbbc3ba: Download complete
e1dc6725529d: Downloading [==============================================>    ]   46.3MB/50.11MB
572866ab72a6: Downloading [========>                                          ]  30.09MB/214.3MB
63ee7d0b743d: Waiting
a9e4c546ba77: Waiting
8d474dc2d651: Waiting
377542fd754b: Waiting
```

Run the Image

- Let’s run this image as a container and see the result in the webpage.
- We are running a container with the interactive and detached mode and also exposing the port 3070 to the outside world.

```
# docker run -it -d -p 3070:3070 nodewebapp:v1
```

- Once you run the above command, we can see the result in the browser.



Welcome to Building a Container from Multiple Images - Powered by The SkillPedia!

Problems With Normal Build

- There are two main problems with this build
  1. size and
  2. larger surface area.

Size of the Image Build

- Let’s list the images that we have with this command docker images
- The size of our image is high, 1.22 GB.

```
[root@server docker-multibuild-example]# docker images
REPOSITORY      TAG      IMAGE ID       CREATED              SIZE
nodewebapp      v1       02dcd3b8ffcf   About a minute ago   1.22GB
```

larger Surface Area

- Another problem is the larger surface area which is prone to attacks.
- We included npm dependencies and the entire Angular CLI library in the image which are unnecessary in the final image.
- For images to be efficient, they have to be small in size and surface area.

Multi-stage Builds

- With multi-stage builds, we can use multiple FROM statements to build each phase.
- Every FROM statement starts with the new base and leave behind everything which you don’t need from the previous FROM statement.
- Before proceeding remove old builds and images.

```
docker ps – Take note of Container ID
docker stop <Container ID>
docker rm <Container ID>
docker rmi -f $(docker images -a -q)
```

New Dockerfile

- Here is the Dockerfile for the multi-stage build.

```
FROM node:10 AS ui-build
WORKDIR /usr/src/app
COPY WebApp/ ./WebApp/
RUN cd WebApp && npm install @angular/cli && npm install && npm run build

FROM node:10 AS server-build
WORKDIR /root/
COPY --from=ui-build /usr/src/app/WebApp/dist ./WebApp/dist
COPY package*.json ./
RUN npm install
COPY index.js .

EXPOSE 3070

ENTRYPOINT ["node"]
CMD ["index.js"]
```

Build and Run the Multi Container

- Build the new container with multiple images with

```
# docker build -f Dockerfile.multi -t nodewebapp:v2 .
```

- Now run the Container with a different port with following command

```
# docker run -it -d -p 3070:3070 nodewebapp:v2
```

Why to Use Multi-Stage Builds

- Allow you to separate build, test, and run time environments needing separate Dockerfiles.
- Minimize the actual size of the final Docker container, because the various layers are no longer stored in the final container.
- Allows you to ensure that there aren’t extra binaries in your deployed container, decreasing your attack vector.
- Ability to run steps/stage in parallel.
- Simplifies CI/CD pipeline and provides an easy way for developers to interact with the various expected gates on the way to a production deployment.

Thank You