Creating an Image Including Files

In this lesson, you are going to learn how to include an HTML file in an image.

The image we just created in the previous lesson didn't need anything other than what the base image already contained, which is why our *Dockerfile* file was so simple. In a real-world scenario, however, I'm very likely to want files to be part of an image I create.

Suppose I have a file named *index.html* on my disk with the following contents:

index.html

```
<html>
  <body>
    <h1>Hello !</h1>
    <div>I'm hosted by a container.</div>
  </body>
</html>
```

I want to create an image that includes a web server that serves the above page over HTTP. NGINX is a good candidate. I could keep using the *debian* base image and add instructions to my *Dockerfile* file that install NGINX, but it's easier to base my work on images that are already configured and tested. The Docker Hub contains an NGINX image where NGINX has already been installed with a configuration that serves files found in the */usr/share/nginx/html* directory.

I create the following *Dockerfile* file in the same folder as the HTML file:

Dockerfile

```
FROM nginx:1.15
```

```
COPY index.html /usr/share/nginx/html
```

Apart from the *nginx* base image, you can see a *COPY* instruction. Its first parameter is the file to be copied from the build context and its second parameter is the destination directory inside the image.

The build context is basically the directory you provide to the *docker build* command. Its contents are available for *COPY* instructions to use, but only during the image build process. That means it's available only for the instructions in the *Dockerfile* file, and files from it won't be a part of the build image or containers that you'll spawn from that image unless you use the *COPY* instruction. This is why we have a *COPY* instruction; we want the *index.html* file to be part of the */usr/share/nginx/html* directory inside the image we create.

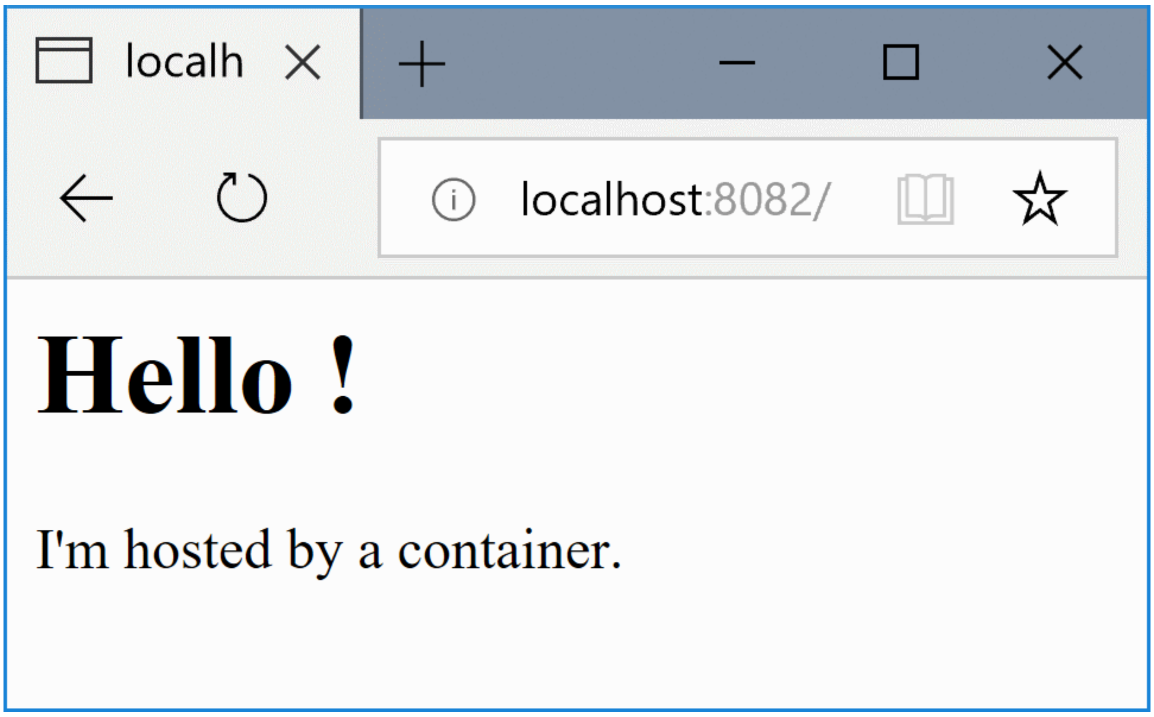
You may have noticed that this time the *Dockerfile* file contains no *CMD* instruction. Remember that the *CMD* instruction states which executable should be run when a container is created from my image, so it's weird that I don't include a *CMD* instruction that runs an NGINX server. The reason why I didn't include a *CMD* instruction is because the base *nginx:1.15* image already contains a *CMD* instruction to run the NGINX server. This is part of my image and I don't need to include my own *CMD* instruction as long as I don't want to run *another* executable on container startup.

Back to creating our HTTP server image, I open a command line inside the folder where my *index.html* and *Dockerfile* files are, and run the following commands:

Dockerfile

```
docker build -t webserver .
docker run --rm -it -p 8082:80 webserver
```

The above commands build a *webserver* from the *Dockerfile* file instructions, then start a container that listens to my machine's 8082 port and redirect the incoming connections to the container's 80 port. I can start a browser and point it to `http://localhost:8082`. This displays the HTML file contents in my point it to `http://localhost:8082`. This displays the HTML file contents in my browser since they are served over HTTP by the running container:



Hello !

I'm hosted by a container.

In the command-line I can see a log from NGINX that proves it received the HTTP request from my browser:

```
172.17.0.1 - - [.../2019:21:14:46 +0000] "GET / HTTP/1.1" 304 0 "-" "..."
"-"
```

When running my container, I added the *–rm* and *-it* switches simply for demo purposes. In reality, that server container would be long-running, so I'd run it without those switches. Here's why I used the switches:

- The *-it* switch allows me to stop the container using *Ctrl-C* from the command-line
- The *–rm* switch ensures that the container is deleted once it has stopped

---

Let's continue our discussion on image creation in the next lesson.