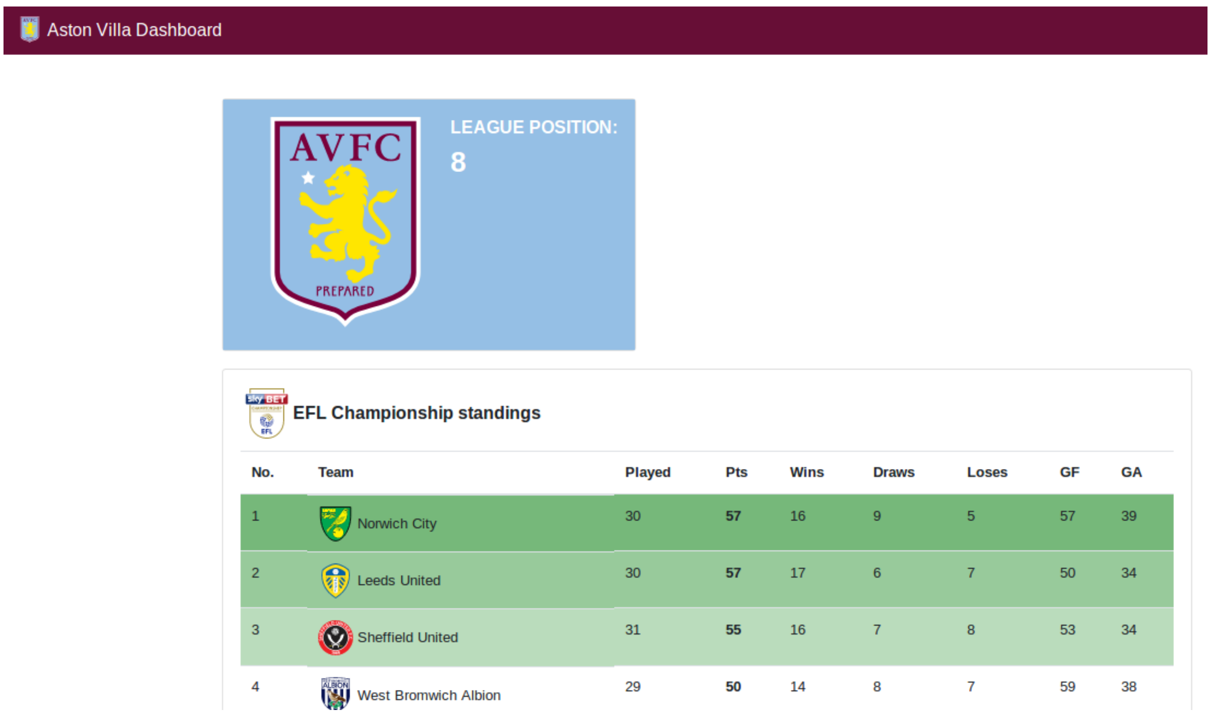
Build and run Angular application in a Docker container

In this tutorial, we will see you how to run your Angular application in a Docker container, then we'll introduce a multi-stage Docker build which will make the container smaller and your work more automated.

Without further introduction, let's get hands dirty and put an Angular app into the Docker container. For purpose of this tutorial we'll use a project — aston-villa-app. It's a simple dashboard with league standings of favourite football club — Aston Villa.



Aston Villa Dashboard

LEAGUE POSITION:
8

EFL Championship standings

No.	Team	Played	Pts	Wins	Draws	Loses	GF	GA
1	Norwich City	30	**57**	16	9	5	57	39
2	Leeds United	30	**57**	17	6	7	50	34
3	Sheffield United	31	**55**	16	7	8	53	34
4	West Bromwich Albion	29	**50**	14	8	7	59	38

Ok, so to get my application you'll need first to clone it, so open a terminal and type:

```
$ git clone https://github.com/wkrzywiec/aston-villa-app.git
```

Now, you've got it in your local folder and then enter it. Next make sure that you have installed Node.js and Angular CLI on your local PC. Full instruction can be found on the official Angular website.

Now if you installed both prerequisites you can compile an Angular app. Therefore open a terminal in the root folder of the app and type:

```
$ ng build --prod
```

This will result in creation of a new folder called **dist/aston-villa-app** in which all compiled files are put.

Then create a new file called **Dockerfile** that will be located in the project's root folder. It should have these following lines:

```
FROM nginx:1.17.1-alpine
COPY nginx.conf /etc/nginx/nginx.conf
COPY /dist/aston-villa-app /usr/share/nginx/html
```

This simple Dockerfile will tell Docker to do three things:
first to get a nginx Docker image from Docker Hub tagged with *1.17.1-alpine* (it's like a version number), then copy-paste the default nginx configuration, and finally copy-paste the compiled application (we done it in previous step) to the container.

My default nginx configuration file looks as follows (it's located in the same directory as Dockerfile):

```
events{}
http {
   include /etc/nginx/mime.types;
   server {
      listen 80;
      server_name localhost;
      root /usr/share/nginx/html;
      index index.html;
      location / {
         try_files $uri $uri/ /index.html;
      }
   }
}
```

I don't want to go that much into details what each line means here (if you would like there are two very nice links with more explanation at the end of this article).

In general, here we define the server on which application will be hosted, its port and default behaviour.
Finally, go back to the terminal and use this command:

```
$ docker build -t av-app-image .
```

If you check the list of locally available Docker images you should get similar output:

```
$ docker image ls
```

```
REPOSITORY    TAG           IMAGE ID
av-app-image  latest        a160a7494a19
nginx         1.17.1-alpine  ea1193fd3dde
```

To run the image you've just created use following command:

```
$ docker run --name av-app-container -d -p 8080:80 av-app-image
```

With it first you give a name to the container (`--name av-app-container`), then make sure that it will run in the background (`-d`), next you map container port to your local (`-p 8080:80`) and finally you pick a base Docker image to be that you've just created - `av-app-image`.

To check if new container is running in terminal type:

```
$ docker container ls
```

```
CONTAINER ID  IMAGE        STATUS        NAMES
2523d9f77cf6  av-app-image  Up 26 minutes  av-app-container
```

Or you can enter a web browser and go to http://localhost:8080/.
Alright! That was easy! Wasn't it? I hope it was for you, but you could see that it's a multi step process and as such, beside being time consuming, is also error-prone.

So how it can be done better? Maybe it could be better when we include the compile phase (`ng build --prod`) into the Docker build? That sounds promising, let's do it!
To achieve it I'd like to introduce something that's called Multi-stage Docker build.

It was introduced in Docker 17.05 and it's main goal was to create smaller containers without loosing the readability of a Dockerfile. With this approach we can divide building a Docker image into smaller phases (stages) where result of previous one (or part of it) can be used in another.

To put it into our context, we'll divide our Docker build into two stages:
compiling the source code into production ready output,
running compiled app in a Docker image.
Only compiled output from first stage will be moved to the second so small size of the container will be preserved.
Until this point we've done the second step, so let's focus on a first one.
For compiling the source code we'll go with different Docker image as a base, which is that containing Node.js. The part of Dockerfile that covers building stage is:

```
FROM node:12.7-alpine AS build
WORKDIR /usr/src/app
COPY package.json package-lock.json ./
RUN npm install
COPY . .
RUN npm run build
```

In this case there are more lines which are responsible for:

(FROM) getting *node* Docker image from registry and naming the compilation stage as build (so we will be able to refer to it in another stage),
(WORKDIR) setting default work directory,
(COPY) copying package.json & package-lock.json files from local root directory — this file contains all dependencies that our app requires,
(RUN) installing necessary libraries (based on a file copied in previous step),
(COPY) copying all remaining files with a source code,
(RUN) and finally compiling our app.

To make building our Docker image even more efficient we can add to the project's root additional file called .dockerignore . This one works similar to *.gitignore* and in it we can define what files and folders we want to Docker to ignore. In our case we don't want to copy any files from *node_modules* and *dist* folders, because they're not needed in compilation.

Therefore this file should look like as follows:

```
dist
node_modules
```

Alright so let's combine both Docker stages into one and as a result we'll get:

```
### STAGE 1: Build ###
FROM node:12.7-alpine AS build
WORKDIR /usr/src/app
COPY package.json package-lock.json ./
RUN npm install
COPY . .
RUN npm run build### STAGE 2: Run ###
FROM nginx:1.17.1-alpine
COPY nginx.conf /etc/nginx/nginx.conf
COPY --from=build /usr/src/app/dist/aston-villa-app /usr/share/nginx/html
```

The only adjustment that I made here is that I've added comments before each stage and also I've added --from=build flag to tell Docker that it needs to copy compiled files from *build* stage (also the source path have changed, because files are located in a different folder).

Going back to a terminal, first you need to create a Docker image:

```
$ docker build -t av-app-multistage-image .
```

And then run the app (on a different port):

```
$ docker run --name av-app-multistage-container -d -p 8888:80 av-app-multistage-image
```

And if you now enter http://localhost:8888/ you'll see that it's running!