[MEAN Stack app on Docker containers : micro services via docker-compose](http://www.bogotobogo.com/MEAN-Stack/MEAN-Stack-NodeJS-Angular-Docker-Compose.php)

Introduction

In this tutorial, we'll deploy MEAN application to two Docker containers using **docker-compose**. Our local machine will be hosting the two containers:

1. mongodb container
2. node/express/angular app

In another article ([MEAN Stack app on Docker containers : micro services](http://www.bogotobogo.com/MEAN-Stack/MEAN-Stack-NodeJS-Angular-Docker.php)), we've done it in a manual fashion.

In this tutorial, we are going to use **docker-compose** to define service dependency, link it to the container running our MEAN application, and mongodb.

Initially, just for demonstration purpose, we'll run our app semi-automatically:

1. **Run** mongodb container using **mongo** image from the Docker hub.
2. **Build** MEAN app container via **docker build** command which gets the instructions from **Dockerfile**.
3. **Run** MEAN app container via **docker run** command.

Then, in later section, we'll do the same using **docker-compose** command which enables us automate those steps (build the two containera and run the app).

Our source code for MEAN app is available from [Github](https://github.com/Einsteinish/akaML" \t "_blank).

Dockerfile for MEAN container

For our MEAN app container, we'll build the image using **Dockerfile**.

A Dockerfile is a set of instructions to build and create an image.

After each instruction the docker constructs a new layer. If a layer hasn't changed, it doesn't need to be rebuilt the next time the build runs, instead the cached layers are used. This layering system is the reason why Docker is so fast.

We'll get started from the official **node** image, and then install necessary packages. Tee **Dockerfile** looks like this:

# Tells the Docker which base image to start.

FROM node

# Adds files from the host file system into the Docker container.

ADD . /app

# Sets the current working directory for subsequent instructions

WORKDIR /app

RUN npm install

RUN npm install -g bower

RUN bower install --allow-root

RUN npm install -g nodemon

#expose a port to allow external access

EXPOSE 3000

# Start mean application

CMD ["nodemon", "server.js"]

Let's go over line by line of the file:

1. **FROM** directive sets the Base Image for subsequent instructions.
2. **ADD** copies all files in our local repo (where we cloned our mean source code) into the containers **app** folder.
3. **WORKDIR** sets the current working directory for subsequent instructions.
4. **RUN** command executes any commands in a new layer on top of the current image and then commits the results. The resulting image will then be used in the next steps.
5. **EXPOSE** will open up a port on our container, but not the host.
6. **CMD** is what will happen when we run our container using **docker run** from the command line. It takes arguments as an array.

Docker build - MEAN container

Now that our Docker file of MEAN is ready, let's build it:

$ docker build -t my-mean-app .

Here we specified the repository name (tag) for the image, and the dot('.') at the end of the command indicates that the location of **Dockerfile** is local.

Running mongodb container

MongoDB should be running before we run MEAN container:

$ docker run --name mymongodb -d mongo

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

5cc4e76a1a9e mongo "/entrypoint.sh mongo" 5 seconds ago Up 2 seconds 27017/tcp mymongodb

Running MEAN container

$ docker run --link mymongodb:db\_1 -p 80:3000 -d my-mean-app

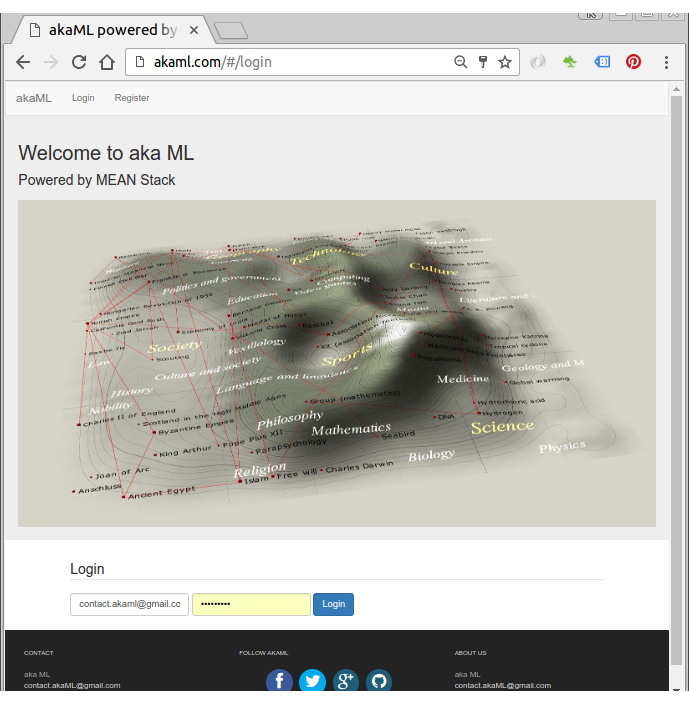
$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

0758bf8e89ae my-mean-app "nodemon server.js" 8 seconds ago Up 4 seconds 0.0.0.0:80->3000/tcp grave\_meitner

5cc4e76a1a9e mongo "/entrypoint.sh mongo" 9 minutes ago Up 9 minutes 27017/tcp mymongodb

Now our app is up an running:



docker-compose install

Let's install Docker Compose.

$ sudo apt-get -y install python-pip

$ sudo pip install docker-compose

To check whether our installation is working correctly, in a temporary folder, let's write a simple file, **docker-compose.yml**:

my-test:

image: hello-world

Then on a command line:

$ docker-compose up

Recreating helloworld\_my-test\_1

Attaching to helloworld\_my-test\_1

my-test\_1 |

my-test\_1 | Hello from Docker.

my-test\_1 | This message shows that your installation appears to be working correctly.

my-test\_1 |

...

helloworld\_my-test\_1 exited with code 0

The output also contains the explanation about what Docker is doing:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

docker-compose.yml

**docker-compose**'s yaml file (**docker-compose.yml**) for our MEAN app, and it looks like this:

mean:

build: .

links:

- db

ports:

- "80:3000"

db:

image: mongo

ports:

- "27017:27017"

The line **mean** defines a service. The command **build** tells docker-compose that we want the service to be built by running **docker build** in the path provided(".").

Next, we instruct docker-compose to link our service to the **db** service using the **links** key.

docker-compose up

**docker-compose up** command does the following (ref: <https://docs.docker.com/v1.5/compose/cli/>):

1. Builds, (re)creates, starts, and attaches to containers for a service.
2. Linked services will be started, unless they are already running.
3. By default, **docker-compose up** will aggregate the output of each container and, when it exits, all containers will be stopped. Running **docker-compose up -d**, will start the containers in the background and leave them running.
4. By default, if there are existing containers for a service, **docker-compose up** will stop and recreate them (preserving mounted volumes with **volumes-from**), so that changes in **docker-compose.yml** are picked up. If you do not want containers stopped and recreated, use **docker-compose up --no-recreate**. This will still start any stopped containers, if needed.

~/akaML$ docker-compose up

Starting akaml\_db\_1

Creating akaml\_mean\_1

Attaching to akaml\_db\_1, akaml\_mean\_1

...

mean\_1 | [nodemon] starting `node server.js`

...

mean\_1 | Express server listening on port : 3000

mean\_1 | Mongoose connection open to mongodb://172.17.0.2:27017/myApp

...

Our MEAN app is up and running now!

Let's check our container state with **docker-compose ps** command:

$ docker-compose ps

Name Command State Ports

-----------------------------------------------------------------------

akaml\_db\_1 /entrypoint.sh mongod Up 0.0.0.0:27017->27017/tcp

akaml\_mean\_1 nodemon server.js Up 0.0.0.0:80->3000/tcp

docker-compose stop

To stop all running Docker containers, issue the following command in the same directory as the docker-compose.yml file used to start the Docker group:

$ docker-compose stop

Stopping akaml\_mean\_1 ... done

Stopping akaml\_db\_1 ... done