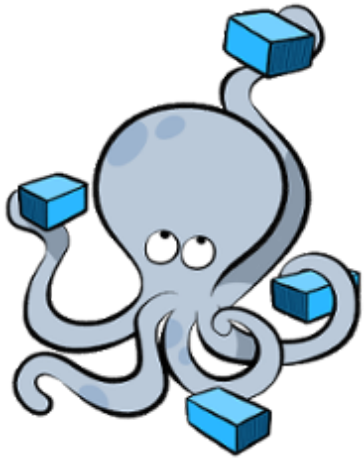
Docker compose with Node.js and MongoDB

Hagai Kahana Dec 3, 2017 · 5 min read

This is the first of a series of posts trying to demonstrate real (simple) examples of deploying multi distributed applications across multiple platform and environments.

The purpose of this post is to demonstrate a login application that persists the account data in MongoDB. We will want to demonstrate the use of docker and docker-compose to launch the application on a single host.



What is docker-compose?

Docker-compose is a tool for defining and running multi-container Docker applications. One common use case is to simulate a distributed application on your own development environment in an isolated manner, which allows you to iterate development faster on your local machine.

Another good example is for testing automation, as part of your CI pipeline having automated testing spin up your application stack locally and execute tests to check for correctness.

Objective

Demonstrate the ability to launch express Node.js application that persists information into MongoDB.

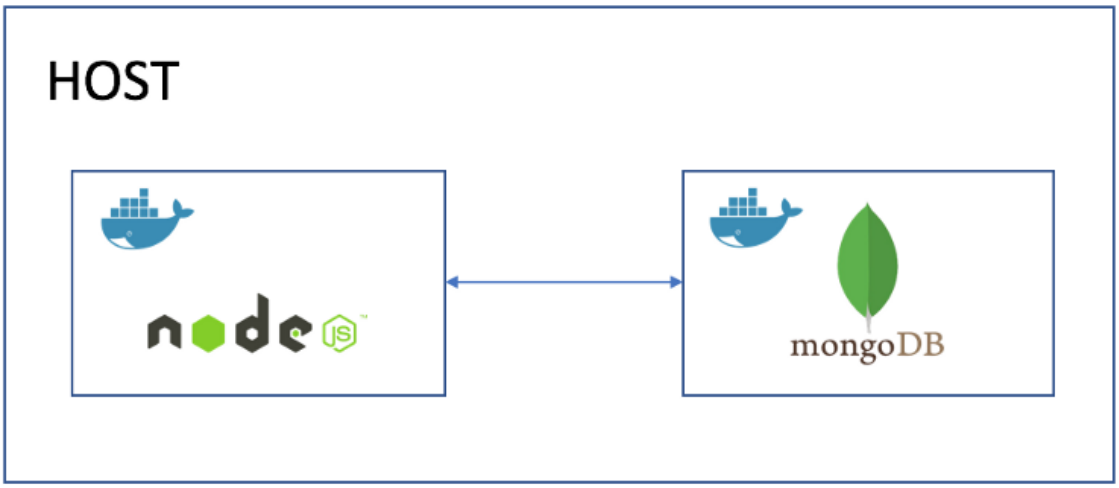
Prerequisites

- GIT
- Docker
- Docker-compose

Setup the Node.js application

At first, we need to obtain a simple application. In this post we will not build the application, but use the detailed guide provided by scotch.io as an example application. The guide creates an application called ‘easy-node-authentication’ and provides an example of a login form that utilizes several passport.js authentication capabilities.

We forked that code base and trimmed it down to only use a “local” passport login. This allows us to persist user information on MongoDB storage which can be a nice example of a 2-tier application. The express app and the MongoDB storage.



Use the following command to download the modified code:

```
git clone https://github.com/hagaik/easy-node-authentication.git
```

First, define the location of the database to be a local mongo. Go to file ‘config/database.js’ and modify it to contain the following configuration:

```
// config/database.js
module.exports = {
    'url' : 'mongodb://mongo:27017'
};
```

Build application docker image

To build a docker image create a Dockerfile in the root directory of the application code base:

```
FROM node:carbon

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
COPY package*.json ./
RUN npm install

# Copy app source code
COPY . .
```

```
#Expose port and start application
EXPOSE 8080
CMD [ "npm", "start" ]
```

Now, we can build our image:

```
$ docker build -t hagaik/auth-app .
```

At this point, we have a docker image with our application code. However, the application needs a running MongoDB to persist the user data and credentials.

Build Compose File:

That is where docker-compose comes in handy; When we want to be running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services.

Create a compose file called 'docker-compose.yml' :

```
version: "2"
services:
  web:
    build: .
    ports:
    - "8080:8080"
    depends_on:
    - mongo
  mongo:
    image: mongo
    ports:
    - "27017:27017"
```

This defines two dockers:

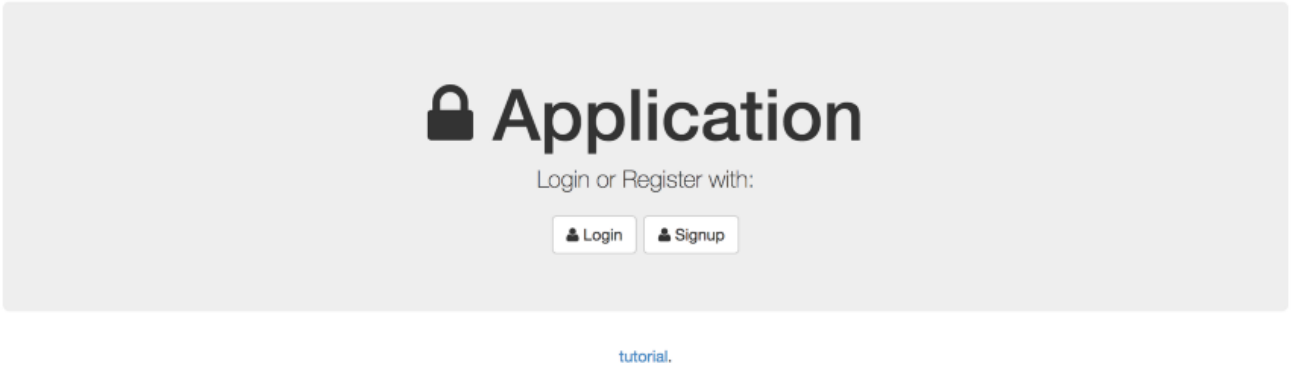
1. **web** which represents our application docker
2. **mongo** which represents the persistence layer docker

By default, "web" service can reach "mongo" service by using the service's name. That is why we configured the database URI to `mongodb://mongo:27017`

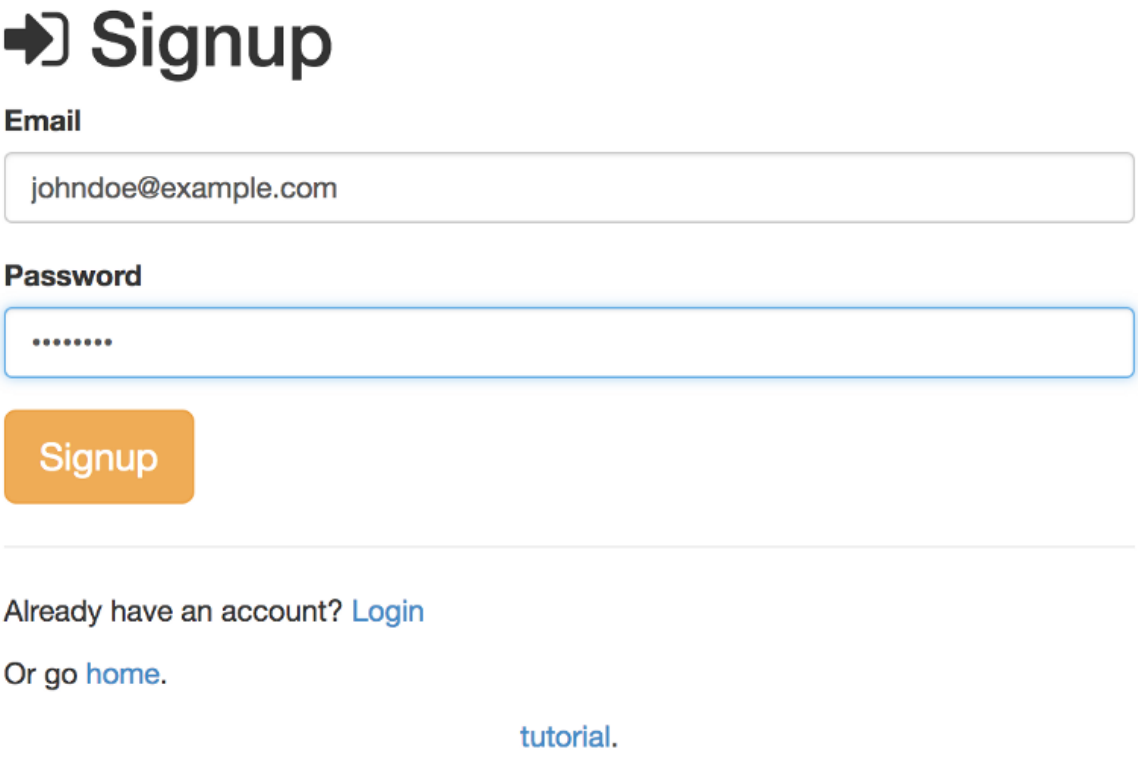
To run the two dockers using the compose file execute the command:

```
$ docker-compose up
```

Open a browser on http://localhost:8080/ to see our application:



From there we can Signup and create an account:



Let's validate that the user has been persistent in our MongoDB docker that is part of the composed work:

```
$ docker ps
CONTAINER ID        IMAGE                             COMMAND...
9a47337d30fe        easynodeauthentication_web   "npm start"...
c0f31757662b        mongo                             "docker..."...
$ docker exec -it c0f31757662b /bin/bash
$ mongo
MongoDB shell version v3.4.10
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.4.10
Welcome to the MongoDB shell.
> show dbs
admin     0.000GB
local     0.000GB
passport  0.000GB
> use passport
switched to db passport
> show collections
users
> db.users.find({})
{ "_id" : ObjectId("5a1f982615336e0010c38dd1"), "local" : {
"password" :
"$2a$08$rDBEIW7FBnH2LPk6E9ptFuv.J71/HrBZkQfdPjpiWDK/FUMHTto4y",
"email" : "johndoe@example.com" }, "__v" : 0 }
```

We use `docker ps` command to list all of the running dockers. From there, we can see the `mongo` docker id and login to it with `docker exec -i <docker-id/name> /bin/bash` .

Once we are logged to the docker, we run the `mongo` shell, list the databases, switch to `passport` database and list all of the users to find the new record of johndoe@example.com.

Persist data in a dedicated volume

In our current setup, we will lose our user data when we delete the mongo container or rebuild it. This is because MongoDB storage is part of the mongo container.

If we want to ensure that the data surpass the lifetime of the mongo container, we should use a named volume to store our mongo files.

The named volume will remain after the container is deleted and can be attached to other docker containers:

```
version: "2"
services:
```

```
  web:
    build: .
    ports:
    - "8080:8080"
    depends_on:
    - mongo
  mongo:
    image: mongo
    ports:
    - "27017:27017"
    volumes:
    - data-volume:/data/db
volumes:
  data-volume:
```

The only addition is the `volumes` section that defines a named volume call data-volume and in `mongo` service we added the volumes. That field maps the created data-volume into '/data/db' folder where the mongo storage file resides.

We can run to list the current running containers:

```
$ docker ps
```

To find the MongoDB container id and run the inspect command:

```
$ docker inspect -f '{{ .Mounts }}' <containerid>
[{volume easynodeauthentication_data-volume
/var/lib/docker/volumes/easynodeauthentication_data-volume/_data
/data/db local rw true}]
```

The first value `easynodeauthentication_data-volume` is the volume name. The seconds value is the path of the volume in the host file system (For Mac the Docker is running inside a VM. As such the path is relative to the VM, and not to your host Mac.). The third value is the mapping on the docker container file system, which match to our docker-compose YAML configuration.

Summary

That is it. We have a build a simple application that persists data in MongoDB. We were able to Dockerize that application and use docker-compose to lunch both the application and MongoDB in a single command. Lastly, we used volumes to ensure

the persisted data remains after the MongoDB container is destroyed. We can close both dockers with:

```
$docker-compose stop
```

and continue with our day.