Lab 1: Dockerized Node

Task

Build Node app images and run container in development and production modes

Time to finish: 15 min 🍍

Walk-through

If you would like to attempt the task, then skip the detailed walk-through and go for the task directly. However, if you need a little bit more hand holding or you would like to look up some of the commands or code or settings, then follow the walk-through.

1. Create Node project
2. Dockerize Node project
3. Use Docker networks for multi-container setup

1. Create/Copy Node project

Firstly, you need to have the application code itself before you can containerize anything. Of course, you can copy the existing code from code/banking-api but it's better for learning to create the project from scratch.

This is what we will do now. Create a new project folder somewhere on your local computer:

```
mkdir banking-api
cd banking-api
mkdir api
cd api
```

Create vanilla/default package.json and install required packages as regular dependencies with exact versions:

```
npm init -y
npm i express@4.15.2 errorhandler@1.5.0 express@4.15.2 globalog@1.0.0 monk@4.0.0 pm2@2
.4.6 -SE
```

Add the following npm scripts. First to test and the second to run the server using local pm2:

```
  "scripts": {
    "test": "sh ./test.sh",
    "start": "if [[ ${NODE_ENV} = production ]]; then ./node_modules/.bin/pm2-docker s
tart -i 0 server.js; else ./node_modules/.bin/pm2-dev server.js; fi"
  },
```

There are two CLI commands for pm2: `pm2-docker` for the container and `pm2-dev` for local development.

The full relative path `./node_modules/.bin` is recommended to make your command more robust. Local installation can be replaced with a global with `npm i -g pm2`. However, global installation is an extra step outside of package.json and `npm i` command and won't allow developers to use different versions of pm2 on one machine.

The source code for the Node+Express API (code/banking-api/api/server.js) is as follows:

```
require('globalog')
const http = require('http')
const express = require('express')
const errorhandler = require('errorhandler')
const app = express()
const monk = require('monk')

const db = monk(process.env.DB_URI, (err)=>{
  if (err) {
    error(err)
    process.exit(1)
  }
})

const accounts = db.get('accounts')

app.use(express.static('public'))
app.use(errorhandler())

app.get('/accounts', (req, res, next)=>{
  accounts.find({ }, (err, docs) =>{
    if (err) return next(err)
    return res.send(docs)
  })
})

app.get('/accounts/:accountId/transactions', (req, res)=>{
  accounts.findOne({_id: req.params.accountId}, (err, doc) =>{
    if (err) return next(err)
    return res.send(doc.transactions)
  })
})

http.createServer(app).listen(process.env.PORT, ()=>{
  log(`Listening on port ${process.env.PORT}`)
})
```

The key here is that we are using two environment variables: PORT and DB_URI. We would need to provide them in Dockerfile or in command-line so the app has them set during running.

Let's verify your application works without Docker by starting MondoGB and the app itself.

```
mongod
```

In a new terminal:

```
DB_URI=mongodb://localhost:27017/db-dev PORT=3000 npm start
```

Yet, in another terminal:

```
curl http://localhost:3000/accounts
```

The result will be `[]%` because it's an empty database and accounts collection. If you use MongoUI or mongo shell to insert a document to db-dev database and accounts collections, then you'll see that document in the response.

The app is working and now is the time to containerize it.

2. Creating Dockerfile

Go back to banking-api and create an empty Dockerfile which must be exactly `Dockerfile`- no extension and starts with capital letter D:

```
cd ..
touch Dockerfile
```

Then, write in banking-api/Dockerfile the base image which is Node v6 based on Alpine (lightweight Linux):

```
FROM node:6-alpine

# Some image metadata
LABEL version="1.0"
LABEL description="This is an example of a Node API server with connection to MongoDB.
 \
More details at https://github.com/azat-co/node-in-production and https://node.univers
ity"
#ARG mongodb_container_name
#ARG app_env

# Environment variables
# Add/change/overwrite with docker run --env key=value
# ENV NODE_ENV=$app_env
ENV PORT=3000
# ENV DB_URI="mongodb://${mongodb_container_name}:27017/db-${app_env}"
# agr->env->npm start->pm2-dev or pm2-docker
# User
#USER app
# Mount Volume in docker run command

# RUN npm i -g pm2@2.4.6

# Create api directory
RUN mkdir -p /usr/src/api
# From now one we are working in /usr/src/api
WORKDIR /usr/src/api

# Install api dependencies
COPY ./api/package.json .
# Run build if necessary with devDependencies then clean them up
RUN npm i --production

# Copy keys from a secret URL, e.g., S3 bucket or GitHub Gist
# Example adds an image from a remote URL
ADD "https://process.filestackapi.com/ADNupMnWyR7kCWRvm76Laz/resize=height:60/https://
www.filepicker.io/api/file/WYqKiG0xQQ65DBnss8nD" ./public/node-university-logo.png

# Copy API source code
COPY ./api/ .

EXPOSE 3000

# The following command will use NODE_ENV to run pm2-docker or pm2-dev
CMD ["npm", "start"]
```

Create app directory

```
# Create api directory
RUN mkdir -p /usr/src/api
# From now one we are working in /usr/src/api
WORKDIR /usr/src/api
```

Install app dependencies

```
# Install api dependencies
COPY ./api/package.json .
# Run build if necessary with devDependencies then clean them up
RUN npm i --production
```

Bundle app source

```
# Copy API source code
COPY ./api/ .
```

Open port and start server

```
EXPOSE 3000
CMD [ "npm", "start" ]
```

Build and verify

Build the image from the banking-api folder where you should have Dockerfile and api folder:

```
docker build .
```

Ah. Don’t forget to start the Docker Engine before building. Ideally, you would see 13 steps such as:

```
docker build .
Sending build context to Docker daemon 23.82 MB
Step 1/13 : FROM node:6-alpine
6-alpine: Pulling from library/node
79650cf9cc01: Pull complete
db515f170158: Pull complete
e4c29f5994c9: Pull complete
Digest: sha256:f57cdd2969122bcb9631e02e632123235008245df8ea26fe6dde02f11609ec57
Status: Downloaded newer image for node:6-alpine
 ---> db1550a2d1e5
Step 2/13 : LABEL version "1.0"
 ---> Running in 769ba6574e60
 ---> 63d5f68d2d01
Removing intermediate container 769ba6574e60
Step 3/13 : LABEL description "This is an example of a Node API server with connection
 to MongoDB. More details at https://github.com/azat-co/node-in-production and https:/
/node.university"
 ---> Running in f7dcb5dd35b6
 ---> 08f1211cbfe1

 ...

Step 13/13 : CMD npm start
 ---> Running in defd2b5776f0
 ---> 330df9053088
Removing intermediate container defd2b5776f0
Successfully built 330df9053088
```

Each step has a hash. Copy the last hash of the image, e.g., 330df9053088 in my case.

As an interim step, we can verify our image by using a host database. In other words, our app will be connecting to the host database from a container. This is good for development. In production, you'll be using managed database such as AWS RDS or Compose or mLabs or a database in a separate (second) container.

To connect to your local mongo instance (must be running) let's grab your host IP:

```
ifconfig | grep inet
```

Look for the one that says `inet`, e.g., `inet 10.0.1.7 netmask 0xffffff00 broadcast 10.0.1.255` means my IP is 10.0.1.7.

Put the IP in the environment variable in the run command for the Docker build of the app image. In other words, substitute the {host-ip} and {app-image-id} with your values:

```
docker run --rm -t --name banking-api -e NODE_ENV=development -e DB_URI="mongodb://{ho
st-ip}:27017/db-prod" -v $(pwd)/api:/usr/src/api -p 80:3000 {app-image-id}
```

```
docker run --rm -t --name banking-api -e NODE_ENV=development -e DB_URI="mongodb://10.
0.1.7:27017/db-prod" -v $(pwd)/api:/usr/src/api -p 80:3000 330df9053088
```

Each option is important. `-e` passes environment variables, `-p` maps host 80 to container 3000 (set in Dockerfile), `-v` mounts the local volume so you can change the files on the host and container app will pick up the changes automatically!

Go ahead and verify by using `curl localhost/accounts`. Then, modify your server.js without re-building or stopping the container. You can add some text, a route or mock data to the `/accounts`:

```
app.get('/accounts', (req, res, next)=>{
  accounts.find({ }, (err, docs) =>{
    if (err) return next(err)
    docs.push({a:1})
    return res.send(docs)
  })
})
```

Hit save in your editor on your host and boom. You'll see the changes in the response from the app container:

```
curl localhost/accounts
[{"a":1}]%
```

To stop the container, run

```
docker stop banking-api
```

Or get the container ID first with `docker ps` and then run `docker stop {container-id}`.

The bottom line is that our Dockerfile is production-ready but we can run the container in dev mode (NODE_ENV=development) with volumes and host database which allows us to avoid any modifications between images and/or Dockerfiles when we go from dev to prod.

3. Use Docker networks for multi-container setup

As mentioned, Dockerfile you created is ready to be deployed to cloud without modifications. However, if you want to run MongoDB or anyother service in a container (instead of a host or managed solution like mLab or Compose), then you can do it with Docker networks. The idea is to create a network and launch two (or more) containers inside of this network. Every container in a network can "talk" to each other just by name.

Creating network

Assuming you want to name your network `banking-api-network`, run this command:

```
docker network create --driver=bridge banking-api-network
```

Verify by getting `banking-api-network` details or list of all networks:

```
docker network inspect banking-api-network
docker network ls
```

You would see something like this:

```
docker network ls
NETWORK ID          NAME                   DRIVER              SCOPE
e9f653fffa25        banking-api-network    bridge              local
cd768d87acb1        bridge                 bridge              local
0cd7db8df819        host                   host                local
8f4db39bd202        none                   null                local
```

Next, launch vanilla mongo image in `banking-api-network` (or whatever name you used for your network). The name of the container mongod-banking-api-prod-container will become the host name to access it from our app:

```
docker run --rm -it --net=banking-api-network --name mongod-banking-api-prod-container
 mongo
```

Note: If you didn't have mongo image, Docker will download it for you. It'll happen just once, the first time.

Leave this mongo running. Open a new terminal.

Launch App into a Network

This is my command to launch my Node app in a production mode and connect to my mongo container which is in the same network (banking-api-network):

```
docker run --rm -t --net=banking-api-network --name banking-api -e NODE_ENV=production
 -e DB_URI="mongodb://mongod-banking-api-prod-container:27017/db-prod" -p 80:3000 330d
f9053088
```

The `330df9053088` must be replaced with your app image ID from the previous step when you did `docker build .`. If you forgot the app image ID, then run `docker images` and look up the ID.

This time, you'll see pm2 in a production clustered mode. I have 2 CPUs in my Docker engine settings, hence pm2-docker spawned two Node processes which both listen for incoming connections at 3000 (container, 80 on the host):

```
docker run --rm -t --net=banking-api-network --name banking-api -e NODE_ENV=production
 -e DB_URI="mongodb://mongod-banking-api-prod-container:27017/db-prod" -p 80:3000 330d
f9053088
npm info it worked if it ends with ok
npm info using npm@3.10.10
npm info using node@v6.10.3
npm info lifecycle banking-api@1.0.0~prestart: banking-api@1.0.0
npm info lifecycle banking-api@1.0.0~start: banking-api@1.0.0

> banking-api@1.0.0 start /usr/src/api
> if [[ ${NODE_ENV} = production ]]; then ./node_modules/.bin/pm2-docker start -i 0 se
rver.js; else ./node_modules/.bin/pm2-dev server.js; fi

[STREAMING] Now streaming realtime logs for [all] processes
0|server   | Listening on port 3000
1|server   | Listening on port 3000
```

The command is different than in the previous section but the image is the same. The command does NOT have volume and has different environment variables. There's no need to use a volume since we want to bake the code into an image for portability.

Again, open a new terminal (or use an existing tab) and run CURL:

```
curl http://localhost/accounts
```

If you see `[]%`, then all is good.

Inspecting your network with `docker network inspect banking-api-network` will show you have 2 running containers there:

```
  ...
        "Containers": {
            "02ff9bb083484a0fe2abb63ec79e0a78f9cac0d31440374f9bb2ee8995930414": {
                "Name": "mongod-banking-api-prod-container",
                "EndpointID": "0fa2612ebc14ed7af097f7287e013802e844005fe66a979dfe6cfb1
c08336080",
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            },
            "3836f4042c5d3b16a565b1f68eb5690e062e5472a09caf563bc9f11efd9ab167": {
                "Name": "banking-api",
                "EndpointID": "d6ae871a94553dab1fcd6660185be4029a28c80c893ef1450df8cad
20add583e",
                "MacAddress": "02:42:ac:12:00:03",
                "IPv4Address": "172.18.0.3/16",
                "IPv6Address": ""
            }
        },
    ...
```

Using the similar approach, you can launch other apps and services into the same network and they'll be able to talk with each other.

Note: The older `--link` flag/option is deprecated. Don't use it. See https://docs.docker.com/engine/userguide/networking/default_network/dockerlinks

Troubleshooting

- No response: Check that the port is mapped in the `docker run` command with `-p`. It's not enough to just have `EXPOSE` in Dockerfile. Developers need to have both.
- The server hasn't updated after my code change: Make sure you mount a volume with `-v`. You don't want to do it for production though.
- I cannot get my IP because your command is not working on my Windows, Linux, ChromOS, etc., see http://www.howtofindmyipaddress.com
- I can't understand networks. I need more info on networks. See https://blog.csainty.com/2016/07/connecting-docker-containers.html