

DOCKER FOR JAVASCRIPT DEVELOPERS

USE THE POWER OF DOCKER TO YOUR ADVANTAGE

by Peter Cosemans

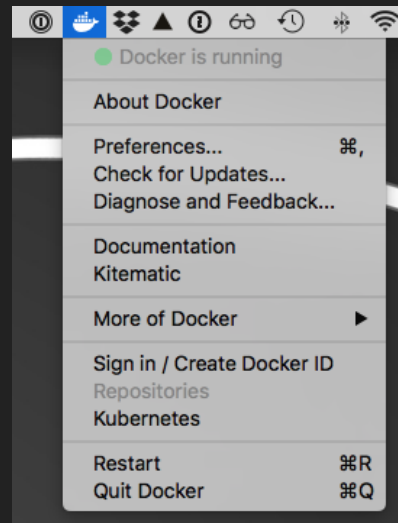
v1.0

Copyright (c) 2018 Euricom nv. Licensed under the [MIT license](#).

SETUP DOCKER FOR MACOS

Follow this link: <https://docs.docker.com/docker-for-mac/install>

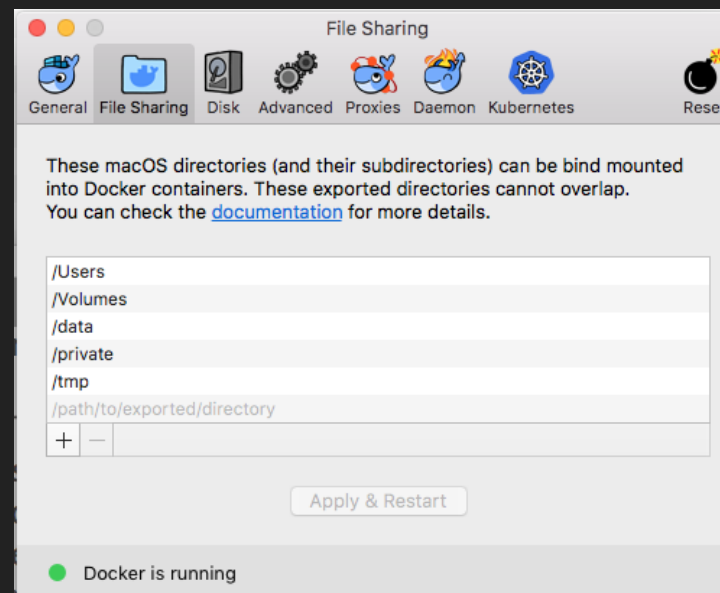
Docker is started automatically.



FILE SHARING

```
sudo mkdir -p /data/docker
sudo chown $USER /data/docker
```

Add File Sharing (for MacOS)



USE DOCKER TO RUN SERVICES

Extend your development toolbox

RUN MONGODB FROM DOCKER

Create container & run

```
# mongodb 3.6
docker run --publish 27017:27017 \
  --name mongodb \
  --volume /data/docker/mongo-3.6:/bitnami \
  bitnami/mongodb:3.6
```

Connect DB with shell

```
$ docker exec -it mongodb mongo
MongoDB shell version v3.6.2
connecting to: mongodb://127.0.0.1:27017/localhost
MongoDB server version: 3.6.6
>
```

Restart

```
docker start mongodb
```

DOCKERIZING A NODE.JS APP

You app in docker

A MINI APPLICATION

```
const http = require('http');
const fs = require('fs');

http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end(`<h1>Hello from NodeJS</h1>`);
}).listen(8080);
```

DOCKERIZING NODE.JS

Dockerfile

```
FROM node
RUN mkdir -p /app
COPY index.js /app
EXPOSE 8080
CMD [ "node", "/app/index" ]
```

build it

```
docker build -t node-app .
```

and run it

```
docker run -p 8081:8080 -d node-app
```

CREATE NODEJS APP WITH DEPENDENCIES

A more real live application with Express

```
{
  "name": "node-express",
  "version": "1.0.1",
  "scripts": {
    "start": "node ./src/server.js",
    "start:debug": "nodemon ./src/server.js",
    "lint": "eslint \"**/*.js\"",
    "docker:build": "docker build -t node-express .",
    "docker:run": "docker run -p 8081:8080 -d node-express"
  },
  "dependencies": {
    "express": "^4.16.3"
  },
  "devDependencies": {
    "eslint": "^4.19.1",
    "eslint-config-airbnb-base": "^12.1.0",
    "eslint-config-prettier": "^2.9.0",
    "eslint-plugin-import": "^2.12.0",
    "nodemon": "^1.17.5",
    "prettier": "^1.6.1"
  }
}
```


DOCKERIZE THE NODEJS APPLICATION

```
# Dockerfile
FROM node

# Create app directory
WORKDIR /app

# Install app dependencies
COPY package*.json /app/
ENV NPM_CONFIG_LOGLEVEL warn
RUN npm install --production --quiet

# Bundle app source
COPY . /app

# Start app
CMD [ "npm", "start" ]
EXPOSE 8080
```

DOCKERIZE THE APPLICATION

Only include files you really want with `.dockerignore`

```
# Ignore everything
**

# Allow files and directories
!package.json
!yarn.lock
!/src/**

# Ignore unnecessary files inside allowed directories
# This should go after the allowed directories
**/*~
**/*.log
**/.DS_Store
**/Thumbs.db
```

TIPS

LIMIT MEMORY

By default, any Docker Container may consume as much of the hardware such as CPU and RAM.
Better to limit usages.

```
$ docker run -p 8080:3000 -m "300M" --memory-swap "1G" demo
```

ENVIRONMENT VARIABLES

Run with `NODE_ENV` set to production.

```
$ docker run -p 8080:3000 -e "NODE_ENV=production" demo
```

This is the way you would pass in secrets and other runtime configurations to your application as well.

TAG DOCKER IMAGES WHEN BUILDING

In order to properly manage and maintain a deterministic build and an audit trail of a container, it is critical to create a good tagging strategy.

```
$ docker build -t appnamespace/app:0.0.1 .
```

MINIMIZE YOUR IMAGE SIZE

Using minimal node.js image, yarn and multi-stage builds

```
# Do the npm install or yarn install in the full image
FROM mhart/alpine-node:8
WORKDIR /app
COPY package.json yarn.lock ./
RUN yarn install --production

# And then copy over node_modules, etc from that stage to
# the smaller base image
FROM mhart/alpine-node:base-8
WORKDIR /app
COPY --from=0 /app .
COPY ./src /app/src

EXPOSE 8080
CMD [ "node", "src/server.js" ]
```

See <https://hub.docker.com/r/mhart/alpine-node/>

GRACEFULL SHUTDOWN

We can speak about the graceful shutdown of our application, when all of the resources it used and all of the traffic and/or data processing what it handled are closed and released properly.

LONG RUNNING REQUEST

A small simulation

```
app.get('/wait', (req, res) => {  
  const timeout = 5;  
  console.log(`received request, waiting ${timeout} seconds`);  
  setTimeout(() => {  
    res.send({  
      id: Date.now(),  
      message: 'Hello belated world'});  
  }, timeout * 1000)  
})
```

If you stop the nodeJS server (ctrl-C or kill) before the request is finished.

```
$ curl http://localhost:8080/wait  
curl: (52) Empty reply from server
```

GRACEFULL SHUTDOWN

React to sigint & sigterm to handle shutdown of the server

```
const shutdown = (signal) => {  
  console.log("shutdown by", signal);  
  httpServer.close((err) => {  
    console.log(`server stopped by ${signal}`);  
    process.exit(err ? 1 : 0);  
  });  
};  
  
process.on('SIGINT', () => shutdown('SIGINT')); // ctrl-c  
process.on('SIGTERM', () => shutdown('SIGTERM')); // kill
```

Limit Keep Alive

```
const httpServer = app.listen(8080, () => {  
  // limit keep alive to 6sec  
  httpServer.timeout = 6000;  
})
```

For production: <https://www.npmjs.com/package/@moebius/http-graceful-shutdown>

RUN IN DOCKER

Build and run

```
docker build -q -t grace . && docker run -p 1234:8080 --rm --name=grace grace
```

Stop container

```
docker stop grace
```

---> BAD: We don't see any signal handling <---

Lets look at the process tree.

```
$ docker exec -it grace /bin/sh  
> ps falx
```

```
# ps falx
```

F	UID	PID	PPID	PRI	NI	VSZ	RSS	WCHAN	STAT	TTY	TIME	COMMAND
4	0	33	0	20	0	4336	748	-	Ss	pts/0	0:00	/bin/sh
0	0	40	33	20	0	9088	820	-	R+	pts/0	0:00	_ ps falx
4	0	1	0	20	0	657520	38808	-	Ssl	?	0:00	npm
4	0	22	1	20	0	4340	704	-	S	?	0:00	sh -c node ./src/server.js
4	0	23	22	20	0	488468	36568	-	Sl	?	0:00	_ node ./src/server.js

GRACEFULL DOCKER SHUTDOWN

To shutdown gracefully

```
# Don't start with npm
# Always start node process directly
CMD [ "node", "src/server.js" ]
EXPOSE 8080
```

Stop with timeout

```
# stop container with 30 timeout before sending KILL
docker stop grace --time 30
```

Build, run & shutdown

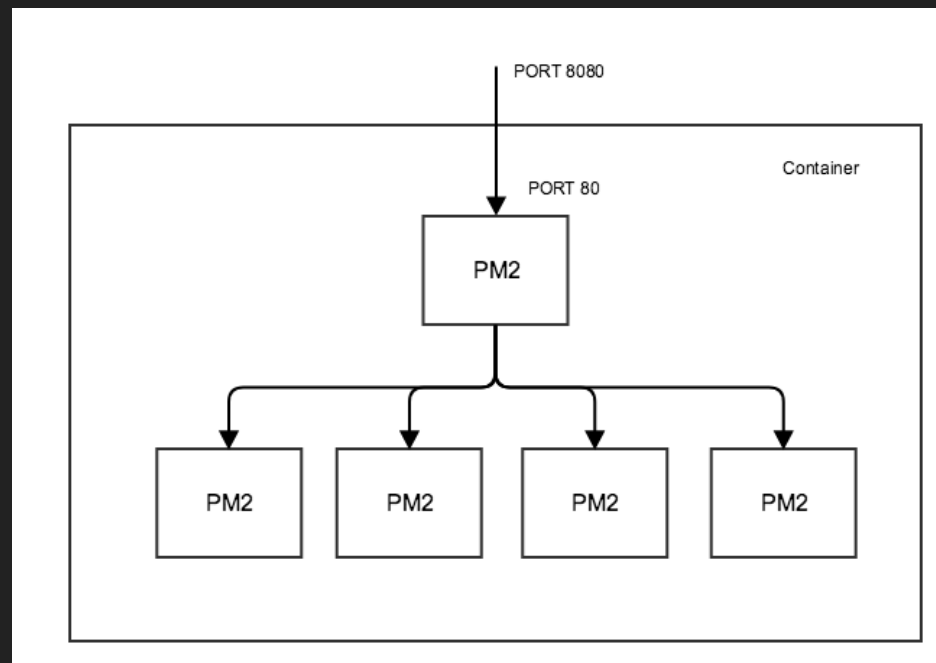
```
$ docker build -q -t grace . && docker run -p 1234:8080 --rm --name=grace grace
Shutdown by SIGTERM
server stopped.
```

CLUSTER NODE APPLICATIONS

Set up high availability

CLUSTERING WITH PM2

High available application



SETUP, CONFIG AND RUN PM2

Install

```
# install
npm install pm2 -g
```

Config

```
# pm2.config.js
module.exports = {
  apps : [{
    name      : 'API',
    script    : './src/server.js',
    instances: "auto",
    kill_timeout: 10000,
    instance_var: 'PM2_INSTANCE_ID',
    exec_mode: 'cluster',
  }],
};
```

Startup & monitor

```
# Start PM2 demon
pm2 start pm2.config.js

# Other commands
pm2 status
pm2 logs
```

RUNNING PM2 IN DOCKER

```
# Dockerfile
FROM keymetrics/pm2:latest-alpine

# Create app directory
WORKDIR /app

# Install app dependencies
COPY package*.json /app/
COPY ecosystem.config.js /app/
RUN npm install --production --quiet

# Bundle app source
COPY . /app/

# Start app
CMD [ "pm2-runtime", "start", "ecosystem.config.js" ]

EXPOSE 8080
```


USEFULL PM2 COMMANDS

Usefull commands

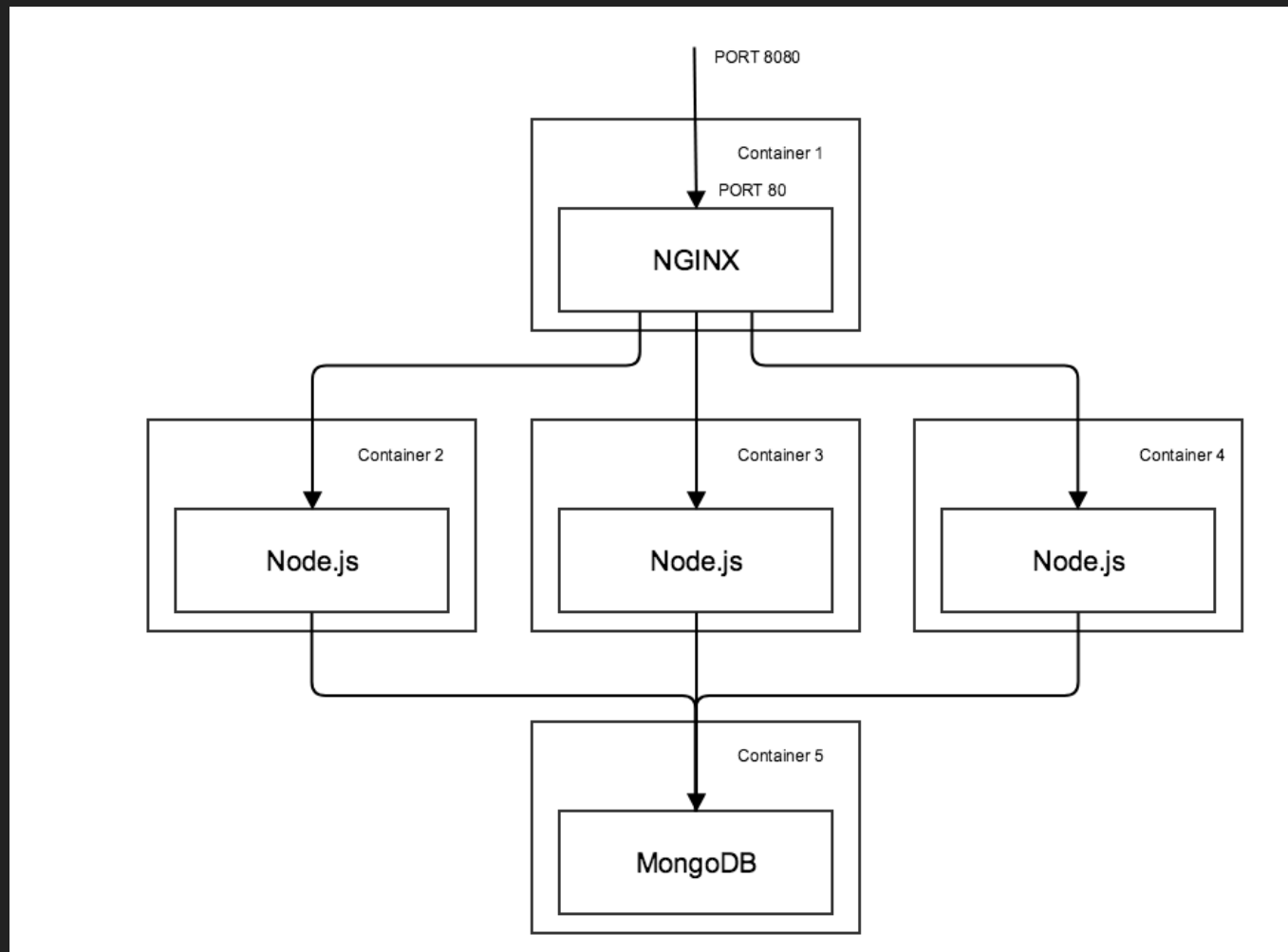
```
# Listing managed processes
$ docker exec -it <container-id> pm2 list

# Monitoring CPU/Usage of each process
$ docker exec -it <container-id> pm2 monit
```

LOAD BALANCING WITH NGINX

MULTIPLE DOCKER IMAGES

Let's configure an instance of NGINX to load balance requests between different docker instances.



DOCKER-COMPOSE

Compose is a tool for defining and running multi-container Docker applications.

```
version: '2'
services:
  nginx:
    build: ./nginx
    ports:
      - "8080:80"
    depends_on:
      - node1
      - node2
  node1:
    build: .
    depends_on:
      - mongo
    environment:
      MONGO_URL: mongodb://mongo/todoDemo
  node2:
    build: .
    depends_on:
      - mongo
    environment:
      MONGO_URL: mongodb://mongo/todoDemo
  mongo:
    image: mongo:3.2
    volumes:
      - ../mongo-data:/data/db
```

NGINX

Nginx is a high performance load balancer.

nginx.conf

```
server {  
    listen 80;  
  
    location / {  
        proxy_pass http://node-app;  
    }  
}  
  
upstream node-app {  
    server node1:3000 weight=10 max_fails=3 fail_timeout=30s;  
    server node2:3000 weight=10 max_fails=3 fail_timeout=30s;  
}
```

DOCKERIZE NGINX

Dockerfile

```
FROM nginx
RUN rm /etc/nginx/conf.d/default.conf
COPY nginx.conf /etc/nginx/conf.d/default.conf
```

COMPOSE: BUILD AND RUN

```
# build all docker images defined in the docker-compose file
$ docker-compose build

# startup docker cluster
$ docker-compose up
```

APPENDIX

Good to know

BEST PRACTICES FOR WRITING DOCKERFILES

- Use a .dockerignore file
- Use multi-stage builds
- Avoid installing unnecessary packages
- Each container should have only one concern (one process per container)
- Minimize the number of layers

USEFULL DOCKER COMMANDS

```
# Docker build
docker build -t node-app .

# List all running containers
docker ps

# List all running docker containers
docker ps

# Remove all dangling images, temp/cached containers
docker system prune

# Stop all containers
docker stop $(docker ps -a -q)

# Remove all containers
docker rm $(docker ps -a -q)

# Remove/delete all images
docker rmi -f $(docker images -q)

# Stop (and after 10sec kill) a docker container
docker kill <container-id>

# Run interactive
docker run -p 8082:8080 -d <image-name> -i -t /bin/bash

# Run interactive on running container
docker exec -it <container-id> /bin/bash
```

RESOURCES

- [Using Yarn with Docker](#)
- [Why we switched from docker to serverless](#)
- [Load Balancing Node.js Applications with NGINX and Docker](#)
- [Best practices for writing Dockerfiles](#)
- [Using PM2 with Docker](#)

RESOURCES

- Building Graceful Node Applications in Docker
- How To Prevent Your Node JS Process From Crashing
- How to write faster, leaner Dockerfiles for Node with Yarn and Alpine
- <https://medium.com/@gchudnov/trapping-signals-in-docker-containers-7a57fdda7d86>