DOCKER FOR JS DEVELOPERS

USE THE POWER OF DOCKER TO YOUR ADVANTAGE

by Peter Cosemans

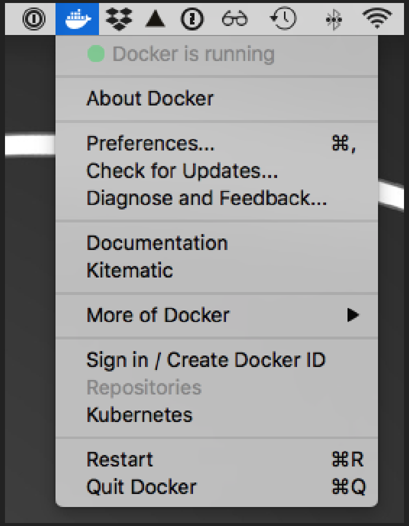
v1.0

SETUP DOCKER

FOR MACOS

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

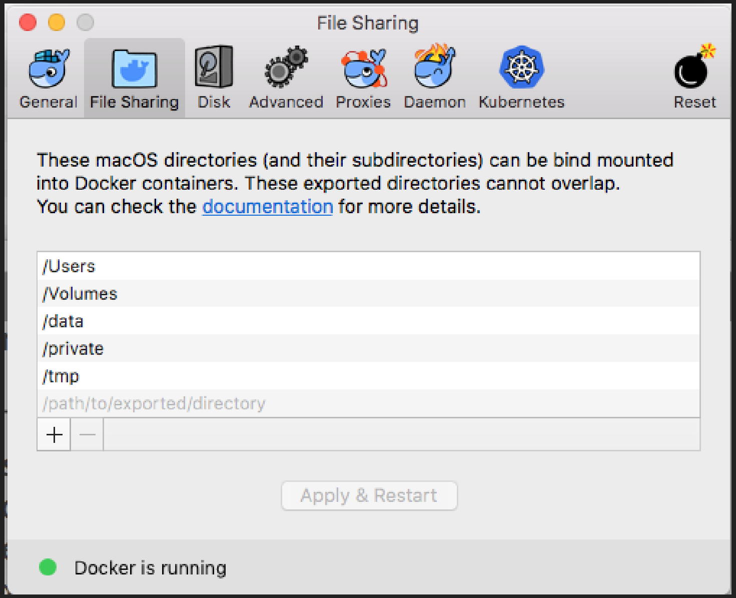
Docker is started automatically.



FILE SHARING

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

Add File Sharing (for MacOS)



These macOS directories (and their subdirectories) can be bind mounted into Docker containers. These exported directories cannot overlap.
You can check the documentation for more details.

/Users
/Volumes
/data
/private
/tmp
/path/to/exported/directory

Docker is running

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

```json
{
  "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
```

MINIMIZE YOUR IMAGE SIZE

Make it small

WHICH BASE IMAGE?

name	Linux	remark	size
node	Debian	latest (inc tools)	673MB
node:6	Debian	latest v6 (6.14.4)	659MB
node:slim	Debian	less tools	183MB
node:8-slim	Debian	less tools & v8	
node:alpine	Alpine	optimized for node	69 MB
mhart/alpine-node	Alpine	latest (npm & yarn)	68 MB
mhart/alpine-node:base	Alpine	latest	42 MB

```
$ docker run mhart/alpine-node:10 node --version
v10.11.0
```

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

MULTI-STAGE BUILDS

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"]
```

HEALTHCHECK

Monitor your docker image

HEALTHCHECK

Docker provide a native health check (> 1.12)

```
# Dockerfile
FROM node

...

# check every 30s to ensure this service returns HTTP 200
HEALTHCHECK --interval=30s CMD node healthcheck.js

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

Status

```
CONTAINER ID  IMAGE     COMMAND      STATUS
7f98cf0d23ae  health    "npm start"  Up 30 seconds (healthy)
```

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;
});
```

RUN IN DOCKER

Build and run

```
docker build -t node-express-shutdown .
docker run -p 8080:80 --rm --name=expressShutdown node-express-shutdown
```

Stop container

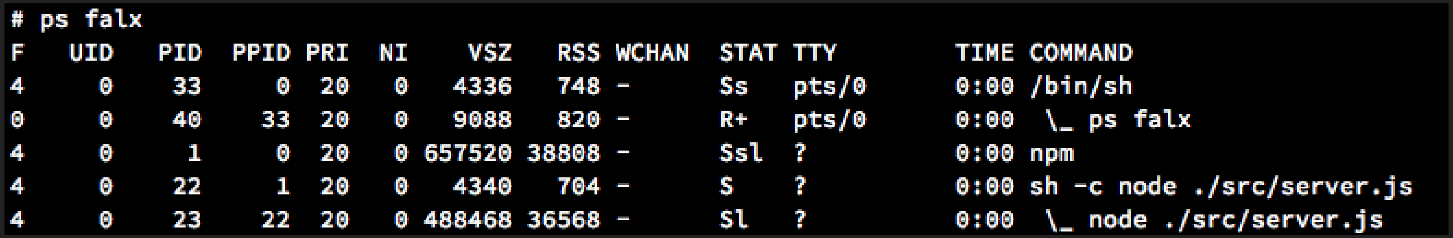
```
docker stop expressShutdown
```

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

SIGNAL PROCESSING IN DOCKER

Lets look at the process tree.

```
$ docker exec -it expressShutdown /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" ]
```

Stop with timeout

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

Build, run & shutdown

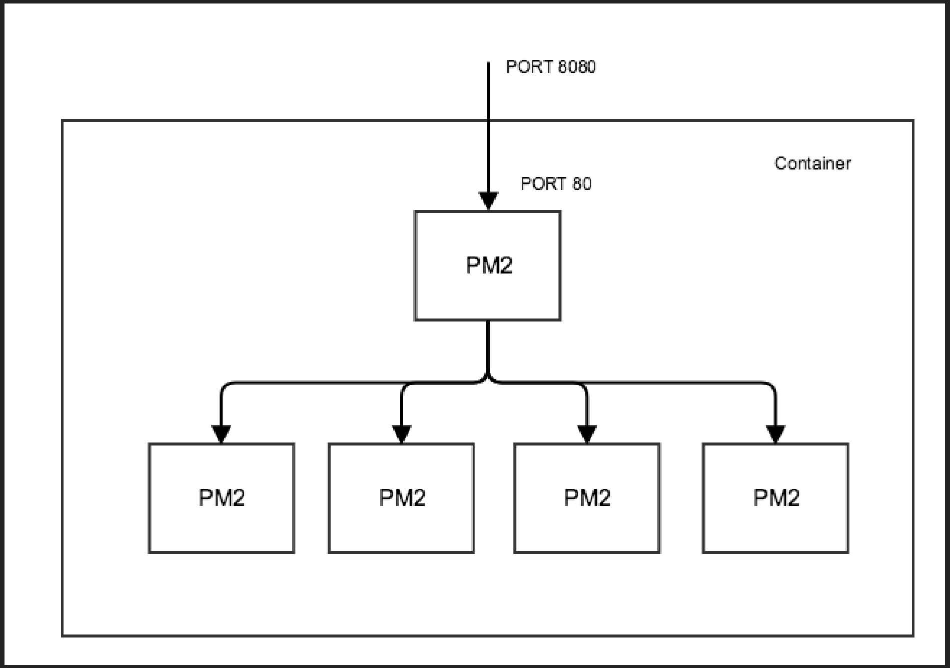
```
$ docker run -p 8080:80 --rm --name=expressShutdown node-express-shutdown
Shutdown by SIGTERM
  server stopped.
```

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

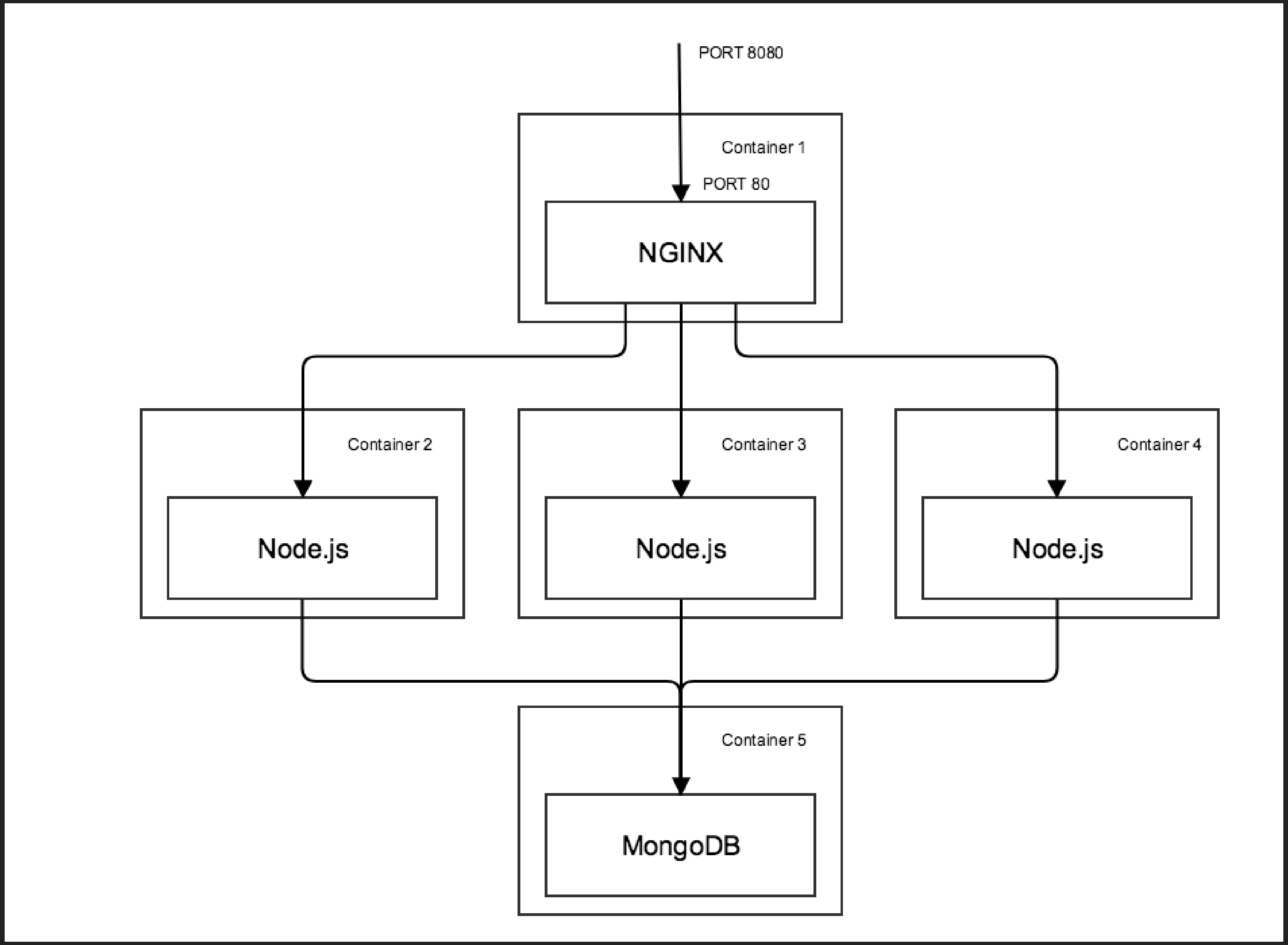
```
# Listing managed processes
$ docker exec -it <container-id> pm2 list

# 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.

```yaml
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

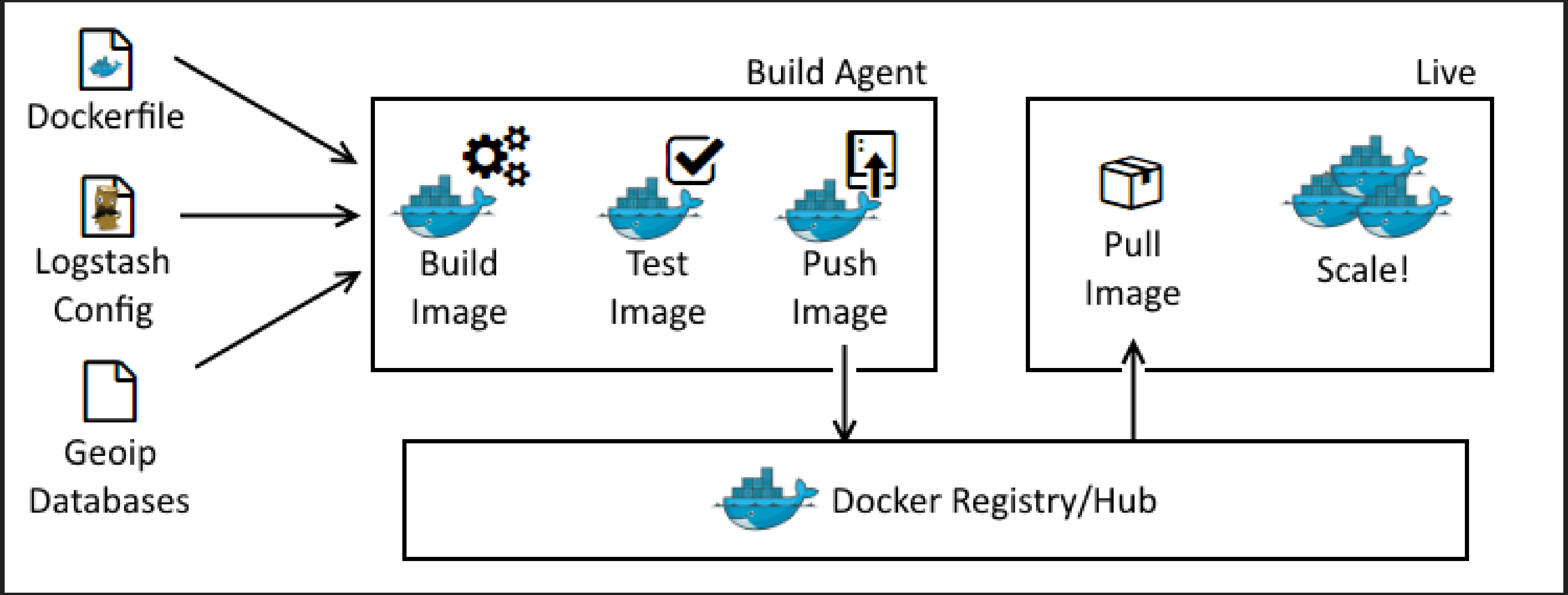
```
FROM nginx
RUN rm /etc/nginx/conf.d/default.conf
COPY nginx.conf /etc/nginx/conf.d/default.conf
```

COMPOSE: BUILD AND RUN

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

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

DEPLOY


Dockerfile
Logstash Config
Geoip Databases
Build Agent
Build Image
Test Image
Push Image
Live
Pull Image
Scale!
Docker Registry/Hub

PUSH IMAGE TO REGISTRY

```
# login to dockerhub: https://hub.docker.com/
docker login

# tag (label) image
docker tag my_image euricom/my_image

# push to repository
docker push euricom/my_image
```

CI & CD

- CircleCI: build & test image
- Azure DevOps (VSTS)
- GitLab Continuous Integration

CIRCLECI

Base Image	Service Image	Tools
Node	MongoDB	curl
JRuby	MySQL	git
Go	PostgresSQL	zip/tar
PHP	…	docker
…	…	jq
Custom	Custom	apt-get

build config

```
version: 2
jobs:
  build:
    docker:
      - image: circleci/node:10
      - image: mongo:3.4.4
```

SIMPLE DEPLOYMENT

- Now (Zeit)
- Azure Container Instances
- Heroku Docker
- AWS Fargate

NOW

```
# build and run container
now
```

DEMO: .../demos/nextjs-with-now

AZURE CONTAINER INSTANCES

```
# Create resource group
az group create --name timACI --location northeurope

# Create container
az container create --name simpleservice \
    --image magneticio/simpleservice:1.0.0 \
    --resource-group timACI --ip-address public --port 3000

# Start container
az container show --name simpleservice --resource-group timACI

# Delete container
az container delete --name simpleservice --resource-group timACI
```

SCALE, HIGH AVAILABLE AND ORCHESTRATE CONTAINERS

- Azure Container Service
- AWS Elastic Container Service (ECS)
- Google Container Engine

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 8080:80 -m "300M" --memory-swap "1G" demo
```

ENVIRONMENT VARIABLES

Run with `NODE_ENV` set to production.

```
$ docker run -p 8080:80 -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:1.1.0 .
```

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 images
docker images

# List all running containers
docker ps

# List all containers
docker ps -a
```

```
# 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 running container
docker stop <container-id or name>
docker stop -f <container-id  or name>
```

USEFULL DOCKER COMMANDS

```
# Run interactive
docker run -it <image-name>

# Run interactive with
docker run -it --entrypoint bash <image-name>

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

```
# Stop and remove all stuff (containers, images, cache, ...)
docker system prune
```

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
- Docker for local development