Node in Production

Module 3: Docker Containers

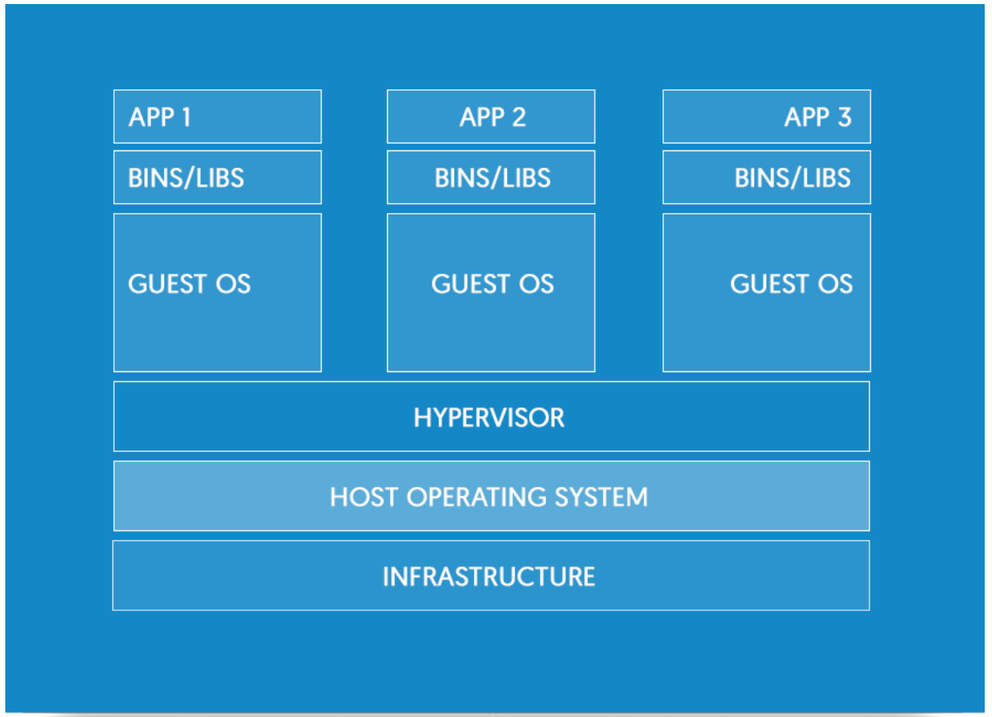
Azat Mardan @azat_co

Node University

Module 3: Docker Containers

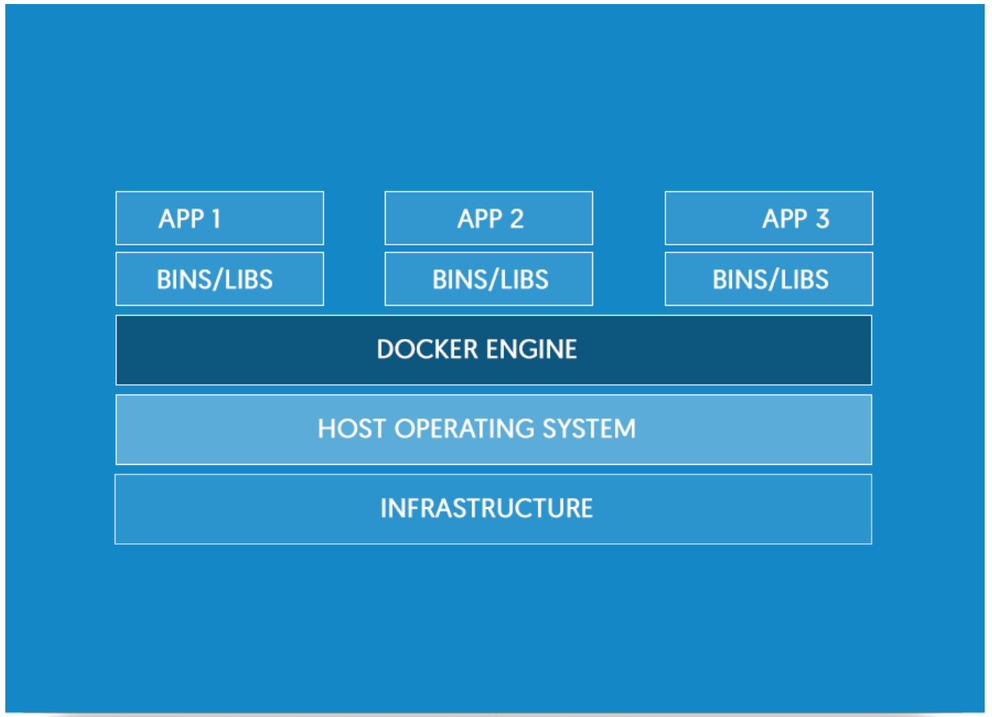
Containers vs. VMs

How VMs work



Source: docker.com

How containers work



Source: docker.com

Here are some of the benefits of Docker containers:

- Allow for rapid application development/deployment
- Allow consistent behavior from dev to production, i.e., portable across machines & environments - fewer bugs!
- Extendable (reuse, collaboration + community)
- Lightweight and fast (no guest OS)
- Easy to collaborate, use and maintain
- Can be versioned and components can be reused

Container and Docker Terminology

Docker Images

Images – The blueprints of our application which form the basis of containers. We will use the docker pull command to download the specified image.

Docker Containers

Containers – Created from Docker images and run the actual application. We create a container using docker run. A list of running containers can be seen using the docker ps command.

Docker Daemon

Docker Daemon – The background service running on the host that manages building, running and distributing Docker containers. The daemon is the process that runs in the operation system to which clients talk to. It is what makes Docker Engine work.

Docker Client

Docker Client – The command line tool that allows the user to interact with the daemon. There can be other forms of clients – such as Kitematic which provides a GUI.

Docker Hub

Docker Hub – A registry of Docker images. You can think of the registry as a directory of all available Docker images. If required, one can host their own Docker registries and can use them for pulling images.

Dockerfile

Dockerfile – A recipe from which you can create an image. Dockerfile has the base image, instructions to add or copy files, commands to execute, ports to expose and other information. Dockerfile is case sensitive.

Docker Compose

Docker Compose – A mechanism to orchestrate multiple containers needed for a service(s) from a single configuration file docker-compose.yml.

Host

Host – Your computer which hosts docker daemon or a remote machine which hosts docker deamon/engine.

Docker workflow

1. Code and create Dockerfile
2. Build an image, tag it, upload to registry (optional)
3. Run container from an image with env vars, volumes and networks
4. Re-build and re-run if needed
5. Stop container

Base Images

Plain OS Base Images:

- alpine
- ubuntu
- debian and jessie (debian v8)
- centos
- amazonlinux

Environment-Ready Base Images

- node v7.9 is based on buildpack-deps:jessie
- node:alpine v6 is based on alpine
- nginx v1.13 is based on debian:stretch-slim
- Apache httpd v2.2 is based on debian:jessie
- mongo v3 is based on debian:wheezy-slim
- postgres v9 is based on debian:jessie

Dockerfile

Dockerfile is a blueprint for the images which turn into containers.

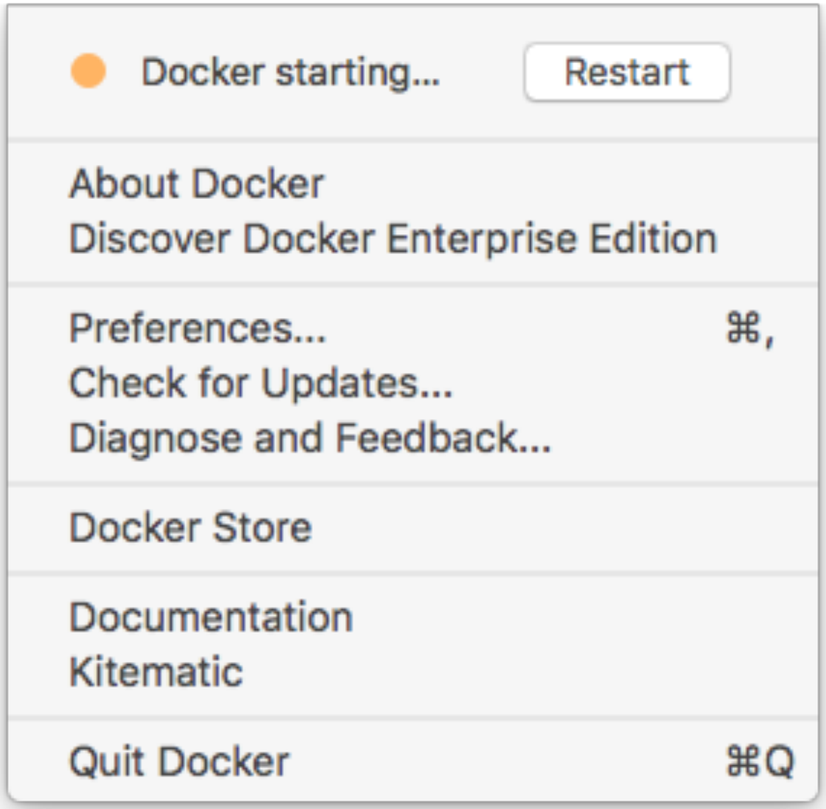
Main Dockerfile statements

- FROM
- RUN
- WORKDIR
- COPY
- EXPOSE
- CMD

Dockerfile Example

```
FROM node:6-alpine
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
COPY server.js /usr/src/app
EXPOSE 3000
CMD [ "node", "server.js" ]
```

Docker Engine macOS



Creating an Image

Create an image from the current project folder:

```
docker build .
```

Create an image with a name and a tag:

```
docker build -t {your-name}/{your-app-name}:{tag} .
```

Running Docker Container

`docker run`

Example of running Alpine and executing `echo` in it:

```
docker run alpine echo "hello from alpine"
```

For usage and all flags/options:

```
docker run --help
```

Some useful options to the `docker run` command are:

- `-d` will detach our terminal (bg/daemon).
- `-rm` will remove the container after running.
- `-it` attaches an interactive tty (teletype) in the container.
- `-p` will publish/expose ports for our container.
- `--name` a name for our container.
- `-v` mounts a volume to share between host and container.
- `-e` provides environment vars to the container.

Alpine Hello World

Run Alpine and execute `echo` in it (just one command):

```
docker run alpine echo "hello from alpine"
```

Result: `hello from alpine`

Ubuntu tty

Run Ubuntu and use `tty` in the container with the `it` flag:

```
docker run -it ubuntu
```

Poke around with `pwd` and `cd`.

Dockerfile Hello World example

Project folder structure:

```
/docker-node-hello
  Dockerfile
  server.js
```

Main app file

server.js:

```
const port = 3000
require('http')
  .createServer((req, res) => {
    console.log('url:', req.url)
    res.end('hello world')
  })
  .listen(port, (error)=>{
    console.log(`server is running on ${port}`)
  })
```

Hello World Dockerfile

Dockerfile:

```
FROM node:6-alpine
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
COPY server.js /usr/src/app
EXPOSE 3000
CMD [ "node", "server.js" ]
```

Build the image:

```
docker build .
```

```
docker build .
Sending build context to Docker daemon 3.072 kB
Step 1/6 : FROM node:6-alpine
6-alpine: Pulling from library/node
709515475419: Pull complete
cb714116d354: Pull complete
25b12811305e: Pull complete
Digest: sha256:823cf041070ca762608a3a5b4c8ee620df093b2e86c6def04d40d0e5acfe3467
Status: Downloaded newer image for node:6-alpine
 ---> 24d2680547f5
Step 2/6 : RUN mkdir -p /usr/src/app
 ---> Running in 8b52d4f59a57
 ---> 20f588f821c8
Removing intermediate container 8b52d4f59a57
Step 3/6 : WORKDIR /usr/src/app
 ---> 33998dd83654
Removing intermediate container cbf35b20a2a2
Step 4/6 : COPY server.js /usr/src/app
 ---> bc832e3172dd
Removing intermediate container 0529df33ca82
Step 5/6 : EXPOSE 3000
 ---> Running in e32c6f35d1b1
 ---> ead36d9c8800
Removing intermediate container e32c6f35d1b1
Step 6/6 : CMD node server.js
 ---> Running in a4b3a0eeeaef
 ---> a1888c63ecfb
Removing intermediate container a4b3a0eeeaef
Successfully built a1888c63ecfb
```

Verify and copy newly built image ID:

`docker images`

```
➜  docker-node-hello git:(master) ✗ docker images
REPOSITORY                                                      TAG                 IMAGE ID            CREATED             SIZE
<none>                                                          <none>              a1888c63ecfb        13 minutes ago      54.1 MB
node                                                            6-alpine            24d2680547f5        13 days ago         54.1 MB
<none>                                                          <none>              958bf18fb6d9        2 weeks ago         688 MB
<none>                                                          <none>              e90b7734f206        2 weeks ago         688 MB
<none>                                                          <none>              8cf558b70e97        2 weeks ago         688 MB
postgres                                                        latest              78e3985acac0        4 months ago        265 MB
node                                                            boron               00673888c33c        5 months ago        651 MB
161599702702.dkr.ecr.us-west-2.amazonaws.com/azat-container-registry   latest              73a541c344ed        5 months ago        654 MB
azat-container-registry                                         latest              73a541c344ed        5 months ago        654 MB
azat-co/node-test-7                                             latest              21edfe8594e9        5 months ago        646 MB
azat-co/node-test-6                                             latest              dc41cf87fe32        5 months ago        646 MB
azat-co/node-test-5                                             latest              dcb232fe7b38        5 months ago        647 MB
azat-co/node-test-4                                             latest              309671897f72        5 months ago        647 MB
azat-co/node-test-3                                             latest              1f4805fa19bf        5 months ago        647 MB
azat-co/node-test-2                                             latest              d342cfa9fd13        5 months ago        646 MB
azat-co/node-test                                               latest              24cc4a9b4708        5 months ago        646 MB
<none>                                                          <none>              e729f83d6f01        5 months ago        646 MB
node                                                            4-onbuild           ef38159cf51e        5 months ago        646 MB
azat-co/server                                                  latest              a18906bfed27        6 months ago        689 MB
node                                                            argon               fb5186e1595a        6 months ago        646 MB
mongo                                                           latest              092cc6fb995c        6 months ago        342 MB
alpine                                                          latest              baa5d63471ea        6 months ago        4.8 MB
ubuntu                                                          latest              f753707788c5        6 months ago        127 MB
hello-world                                                     latest              c54a2cc56cbb        10 months ago       1.85 kB
```

Run with mapping ports host:container:

```
docker run -p 80:3000 {image-id}
```

For example:

```
docker run -p 80:3000 a1888c63ecfb
```

Output (from container):

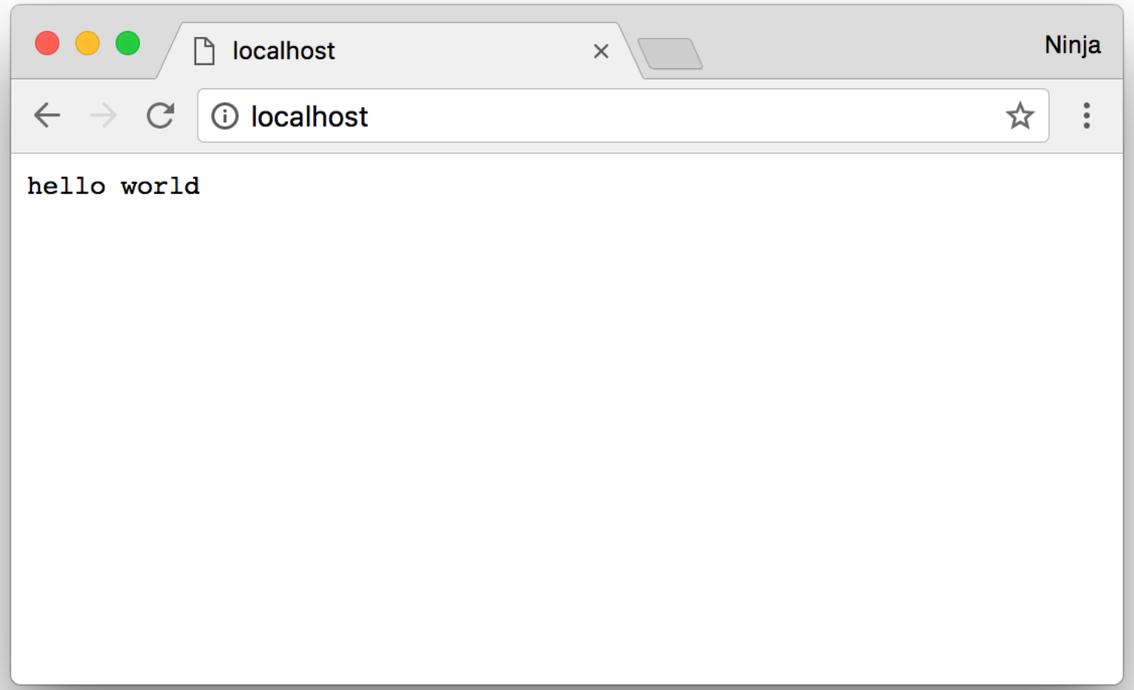
```
server is running on 3000
```

Verify "Hello world" in new terminal session.

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

or in the browser: http://localhost

Remember port 80 is the default port value for HTTP so no need to type it in the URL.



Also, verify with Docker Engine:

```
docker ps
```

Output example:

```
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                   NAMES
9e8fad455bff        a1888c63ecfb        "node server.js"    2 minutes ago       Up 2 minutes        0.0.0.0:80->3000/tcp    quizzical_ardinghelli
```

Stop:

```
docker stop {image-id}
```

For example:

```
docker stop 19b4df5a399c
```

Congrats, you've run your first Node web server in a container.

Working with containers

The ps will show you a list of all containers that we've run on the development machine (called host):

```
docker ps -a
```

Currently running containers:

```
docker ps
```

Remove

```
docker rm {container-id}
```

Remove all containers and images ⚠️

Delete all local and stopped containers

```
docker rm $(docker ps -a -q)
```

Delete all local images

```
docker rmi $(docker images -q)
```

❗ No way to restore them! ❗

Remove all containers and images ⚠️

One (newer) command to remove unused images, stopped containers, etc. ❗ No way to restore them! ❗

```
docker system prune -a

WARNING! This will remove:
    - all stopped containers
    - all volumes not used by at least one container
    - all networks not used by at least one container
    - all images without at least one container associated to them
```

? How do you connect to a database, make container flexible so it acts in different environments differently? ?

Use environment variables and/or build arguments

Variables, arguments and other parameters

- ARG in Dockerfile: built-time arguments which can be used in Dockerfile ($ and if/else) and supplied with `docker build --build-args` CLI
- ENV in Dockerfile: hard-coded in Dockerfile environment variables so they become baked into an image
- RUN with environment variables in Dockerfile, e.g., `RUN NODE_ENV=production npm i` - same as ENV in Dockerfile
- Environment variables with `docker run -e`: not-hard-coded, run-time and supplied at run

ARG vs. ENV

ARG and ENV are baked into images but ARG can be modified on built with `docker build --build-args`

Yet more Dockerfile Statements

- ADD
- LABEL
- VOLUME
- ENTRYPOINT (must have CMD or ENTRYPOINT)
- USER

COPY vs ADD

ADD can fetch from URLs and unzip[1]

[1] Docker/moby source code link

ENTRYPOINT vs CMD

CMD provides default commands. ENTRYPOINT command options will always run, but CMD can be overwritten by `docker run` [2]. ENTRYPOINT can be supplemented by CMD options:

```
ENTRYPOINT ["node"]
CMD [ "server.js" ]
```

[2] http://goinbigdata.com/docker-run-vs-cmd-vs-entrypoint

What to use

CMD is recommended for service-based images (like an API)[3], e.g.,

```
CMD [ "node", "server.js" ]
```

[3] https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/#cmd

Development with Docker

Two things in container: environment (not changing often) and app (changes often - every time you press save).

Source code change->Image Rebuild? 😞

Use volumes!

You can attach and detach volumes. Volumes allow sharing of data in the run-time between host and container.

- No need to rebuild an image each time you make a change
- Use the same image for development as for production
- Share data between containers

```
docker run -v $(pwd)/:/www/ {image-id}
docker run -v $(pwd)/app:/usr/src/app -it {name}
```

`$(pwd)` prints working directory, i.e., the absolute path to the location from which you run this command.

How to use the same image in production?

Use pm2 for dev and production

1. Use `docker run -v $(pwd)/api:/usr/src/api -e NODE_ENV=development {image-id}` to pass environment variable and attach volume
2. Install pm2 and run `npm start` in Dockerfile
3. Use if/else in `npm start` which will run `pm2-dev`
4. Any file changes on host will trigger restart in container

If/else in npm start script

This is just a shell if/else in the npm script start in package.json:

```
"start": "if [[ ${NODE_ENV} = production ]]; then pm2-docker start -i 0 server.js; else pm2-dev server.js; fi"
```

Recommended for Dockerizing Node

- Keep container single-concerned, ephemeral and stateless
- `FROM node:6-alpine`
- File restart
- `npm i --production`
- Environment variable

Recommended for Dockerizing Node

- Use volumes and the network
- Keep layers minimal (RUN create a layer)
- Use COPY and CMD
- Use if/else but sparingly

Production-level Dockerfile Example

```
FROM node:6-alpine
# 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.university"
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
```

Production-level Dockerfile Example (cont)

```
# User
#USER app

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
```

Production-level Dockerfile Example (cont)

```
# 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"]
```

Multiple containers with Docker networks

Each container should have only one concern

The best practice for containers is to have one app per container. Thus, often developers will need to connect two or more containers together. For example, Node API, Redis and MongoDB.

One Network allows sharing of connections between containers

Create a new network with Bridge driver:

```
docker network create --driver=bridge my-network
```

Verify it exists:

```
docker network inspect my-network
```

List of all networks:

```
docker network ls
```

When you launch a container into a network and use a name, other containers can access this container by name. For example, launch MongoDB from its official image at Docker Hub:

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

Verify that you have container in the network:

```
docker network inspect my-network
```

Copy the API image ID:

```
docker images
```

Run API container and use `mongod-banking-api-prod-container` as the hostname:

```
docker run --rm -t \
  --net=my-network --name banking-api \
  -e NODE_ENV=production \
  -e DB_URI="mongodb://mongod-banking-api-prod-container:27017/db-prod" \
  -p 80:3000 be327d49c00d
```

Lab 1: Dockerized Node

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

Detailed instructions and links are in labs/1-dockerized-node.md

Time to finish: 15 min 🍍