IBM Cloud: Containers

Marek Sadowski

Developer Advocate IBM

V2 2020 08 20

IBM Developer

IBM

Agenda:

1. microservices in a cloud
2. containers to host microservices
3. Docker to build-ship-run containers
4. orchestration of many containers
5. best practices (layers, min size, etc)
6. the registry – pull/push images

Not this talk:

- *Kubernetes*
- *Helm*
- *Istio*
- *OKD vs Red Hat OpenShift vs Kubernetes*

Container story

About Microservice
APIs
Docker

How you build a microservice

IT IS A SIMPLE PROGRAM – RESPONDING TO CRUD OPERATIONS

WHAT is a microservice?

Why are there microservices?

A large system might consist of many services: HR, a basket checkout, payments, CRM operations, etc.

These services might be built out of microservices – change the prescription's content: add, remove, update, delete

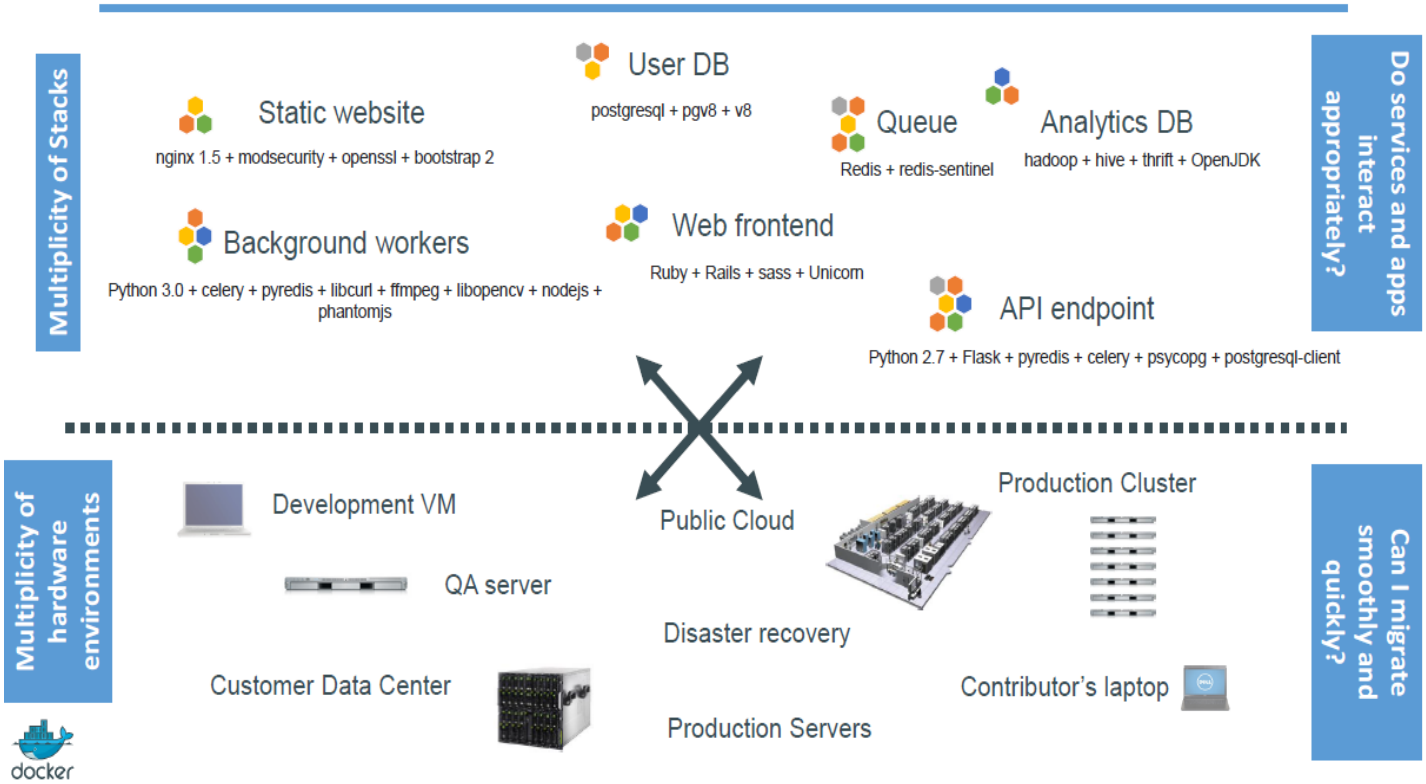
- These are microservices – that might be deployed as containers

Why Containers?

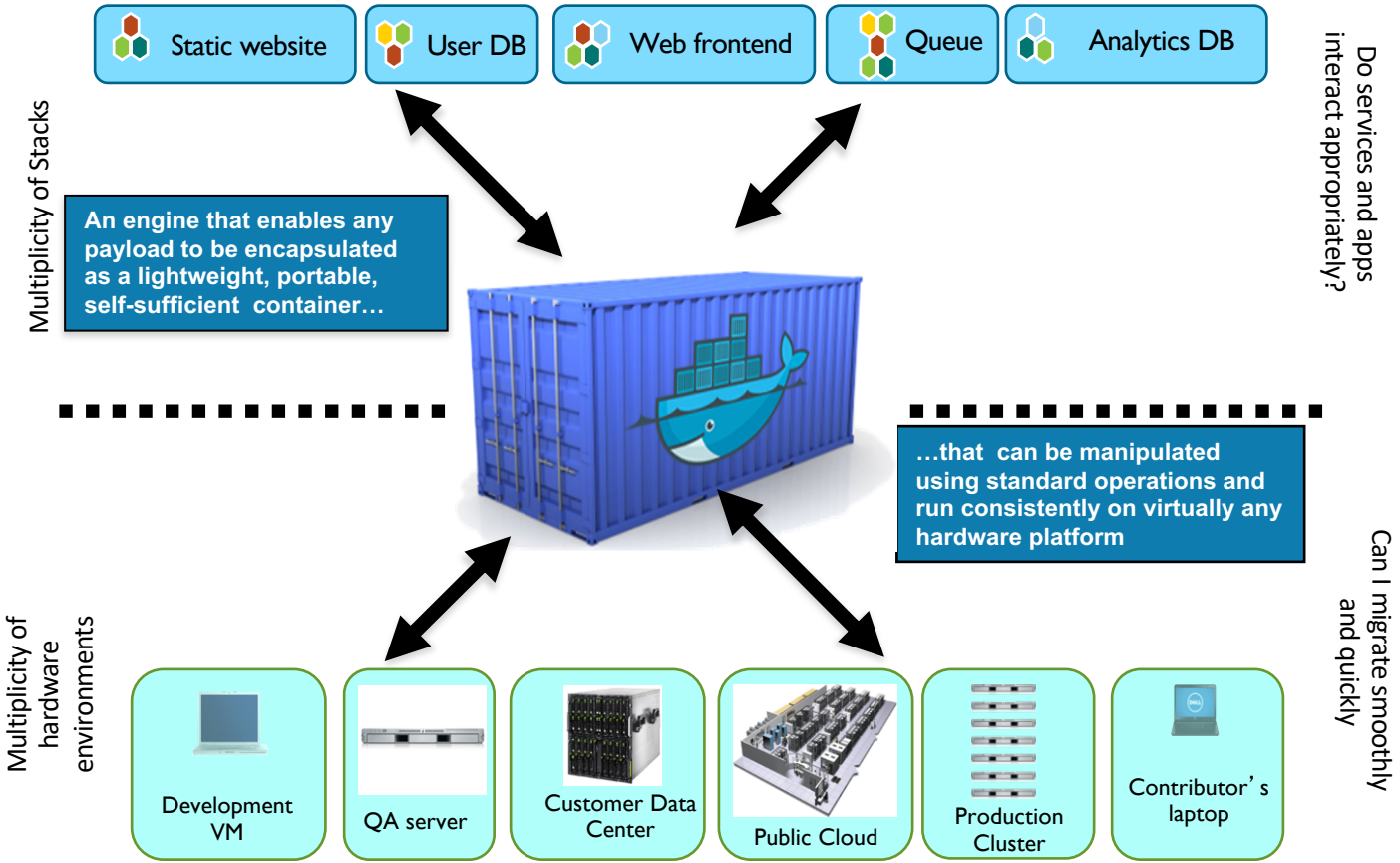
Everyone Loves Containers

- A standard way to package an application and all its dependencies so that it can be moved between environments and run without changes
- Containers work by isolating the differences between applications inside the container so that everything outside the container can be standardized

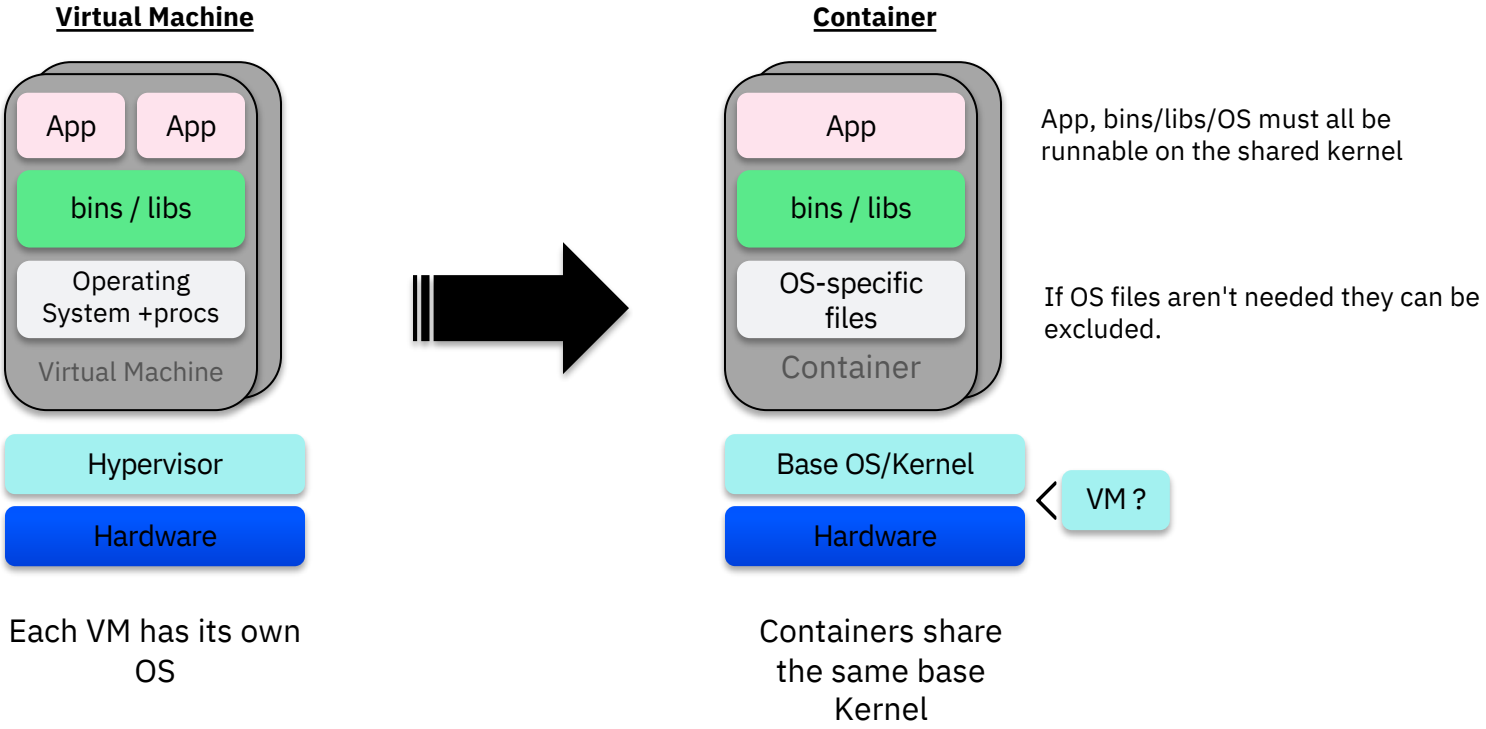
The Challenge



The Solution - A Shipping Container for Code

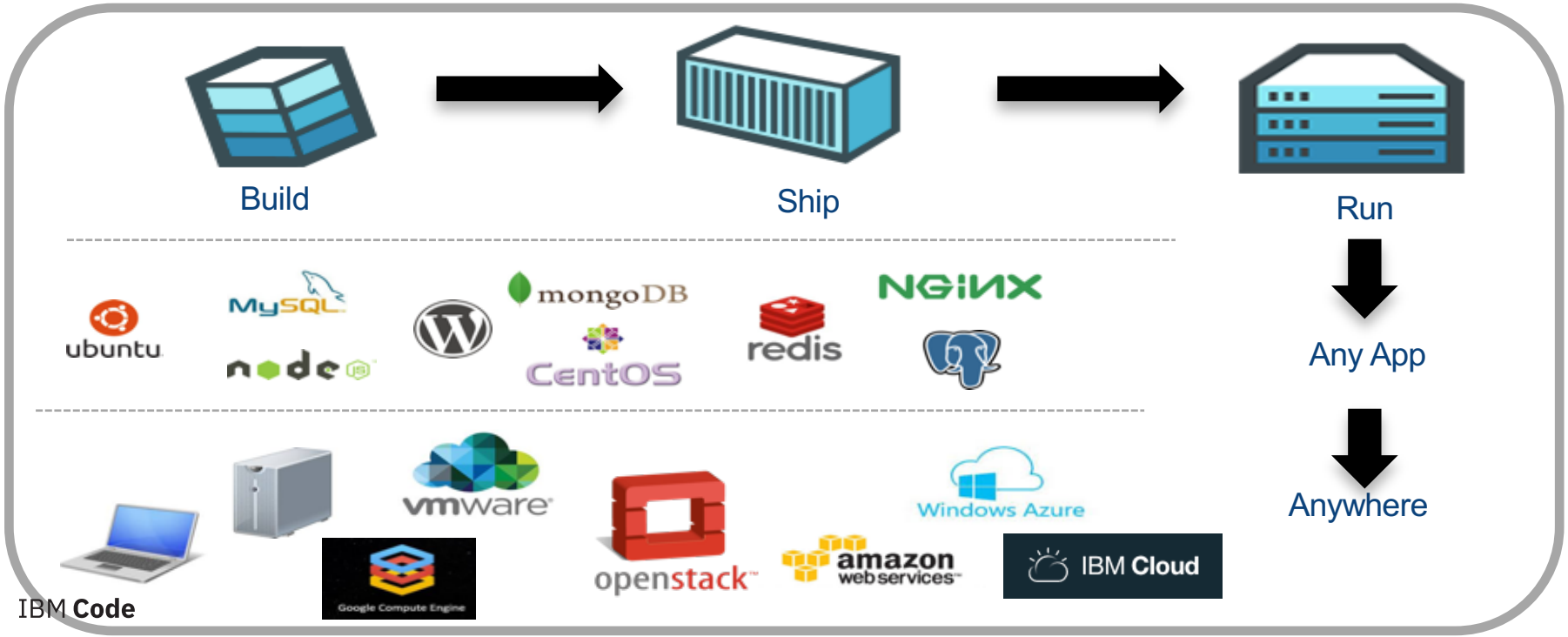


VM vs Container



Docker Mission

Docker is an **open platform** for building distributed applications for **developers** and **system administrators**



Docker Component Overview

Docker Engine

- Manages containers on a host
- Accepts requests from clients
  - REST API
- Maps container ports to host ports
  - E.g. 80 → 3582

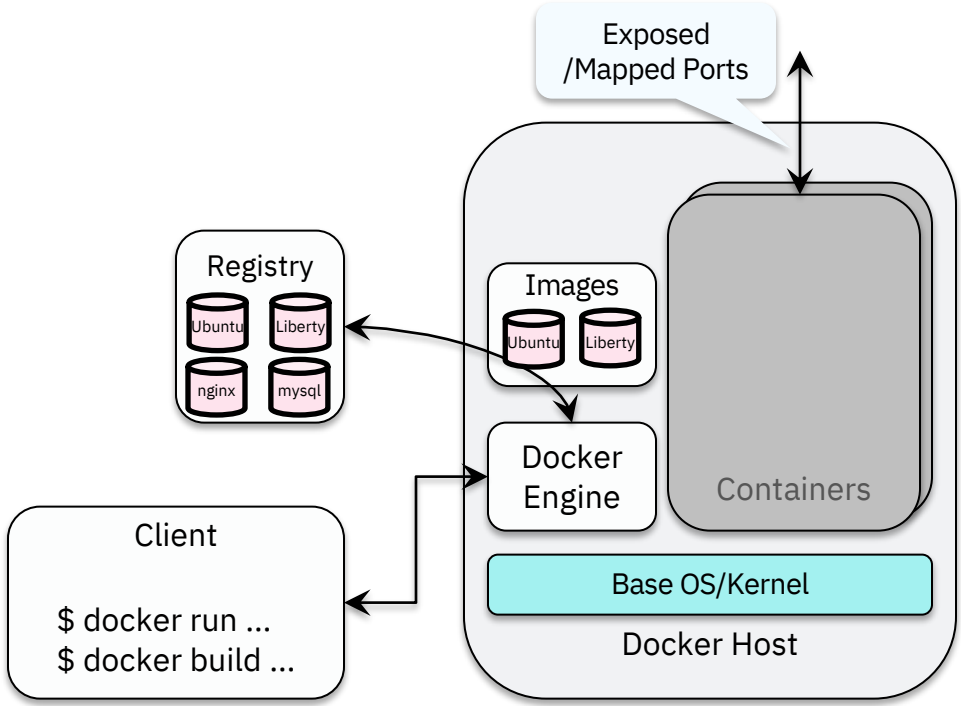
Images

Docker Client

- Drives engine
- Drives "builder" of Images

Docker Registry | Container Registry

- Image DB



Shared / Layered / Union Filesystems

Docker uses a copy-on-write (union) filesystem

New files(& edits) are only visible to current/above layers

Layers allow for reuse

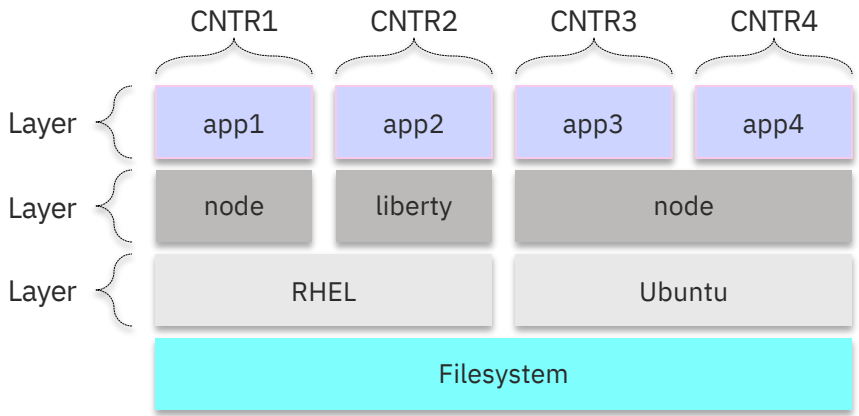
– More containers per host

– Faster start-up/download time

Images

– Tarball of layers

Think: Transparencies on projector



IBM Developer

Our First Container

```
$ docker run ubuntu echo Hello World
Hello World
```

What happened?

- Docker created a directory with a "ubuntu" filesystem (image)
- Docker created a new set of namespaces
- Ran a new process: `echo Hello World`
  - Using those namespaces to isolate it from other processes
  - Using that new directory as the "root" of the filesystem (`chroot`)
- That's it!
  - Notice as a user I never installed "ubuntu"
- Run it again - notice how quickly it ran

```
Mareks-MBP:~ mareksadowski$ docker run ubuntu echo Hello World
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
6b98dfc16071: Pull complete
4001a1209541: Pull complete
6319fc68c576: Pull complete
b24603670dc3: Pull complete
97f170c87c6f: Pull complete
Digest: sha256:5f4bdc3467537cbbe563e80db2c3ec95d548a9145d64453b06
Status: Downloaded newer image for ubuntu:latest
Hello World
Mareks-MBP:~ mareksadowski$ docker run ubuntu echo Hello World
Hello World
Mareks-MBP:~ mareksadowski$
```

A look under the covers

```
$ docker run ubuntu ps -ef
UID          PID    PPID  C
STIME TTY          TIME CMD
root           1       0  0
14:33 ?        00:00:00 ps -ef
```

Things to notice with these examples

- Each container only sees its own process(es)
- By default running as "root”,
- And running as PID 1
- The good security practice is to run services as the user other than root, and avoid the privileged mode
- Limit the resources used by container – to avoid noisy neighbor effect

Hands-on

LABs

Lab 1 - with Docker: http://ibm.biz/202008-containers-lab1

Lab 2 - with Watson microservice and Node.js app:
http://ibm.biz/202008-k8s-docker
Please first signup / sign in with this link (I get brownie points for using this link): http://ibm.biz/202008-containers-login

Lab 2 - Dockerfile

FROM node:10 ... builds our image on top of the Node.js 10 image.

WORKDIR /usr/src/app ... creates a working directory for our application to live in.

COPY package*.json ./ ... copies the package.json file to our working directory. RUN npm **install** ... install our dependencies. We just have two dependencies in this application: express and ibm-watson.

COPY . . ... copy the rest of our source code into the docker image

EXPOSE 8080 ... expose port 8080. We will still have to forward our local port to this docker container port later.

CMD ["node", "server.js"] ... starts the application by running node server.js.

Run the docker image:

```
docker run -p 8080:8080 -e lt_key=<api_key> -e lt_url="<api_url>" -d <docker-username>/node-container
```

LABs stretch goals:

- Step 1: first signup / sign in with this link (I get brownie points for using this link): http://ibm.biz/202008-containers-login
- Step 2: then get the access to Kubernetes cluster (based on all having, with less operations Kubernetes flavor based on Red Hat OpenShift [think about it alike Red Hat Enterprise flavored Linux]): http://ibm.biz/202008-containers-freecluster (the cluster is going to be available till end of Sunday 8/23) | a user name that you signed up in Step 1, and the key **oslab**
- then follow the steps in the Medium article here (use my friendly links - no paywall :-9 and give me some claps if you like them):

- **Lab 3** (a friendly link) : https://medium.com/zero-equals-false/running-a-microservice-on-kubernetes-straight-from-a-container-image-3eb8593bc94c?source=friends_link&sk=7cc4a9ab4cd8f88f3cae9395b2945165
- **Lab 4** (a friendly link) : https://medium.com/@blumareks/running-a-microservice-containers-on-kubernetes-straight-from-a-github-ff73047e877b?source=friends_link&sk=ba67153ca54c0c2f1acc3befbea1ceb3

Best practices (generic):

- Reuse images
- Maintain compatibility within tags
- Avoid multiple processes
- Use exec in wrapper scripts
- Clean temporary files
- Place instructions in the proper order
- Mark important ports
- Set environment variables
- Avoid default passwords
- Avoid sshd
- Use volumes for persistent data

Best OpenShift Container Platform-specific practices:

- Enable images for source-to-image (S2I)
- Support arbitrary user ids
- Use services for inter-image communication
- Provide common libraries
- Use environment variables for configuration
- Set image metadata
- Clustering
- Logging
- Liveness and readiness probes
- Templates
- Including metadata in images
- Defining image metadata
- Testing S2I images
- Understanding testing requirements
- Generating scripts and tools
- Testing locally
- Basic testing workflow
- Using OpenShift Container Platform for building the image