Microservices

June 13, 2017

1 Microservices with Docker.

1.1 Mateusz Urbanczyk

1.2 Agenda

- Why Microservies are great
- Why Docker is awesome
- When you should NOT use them

2 Overview

3 Implementation example

4 Why would you want to use Microservices?

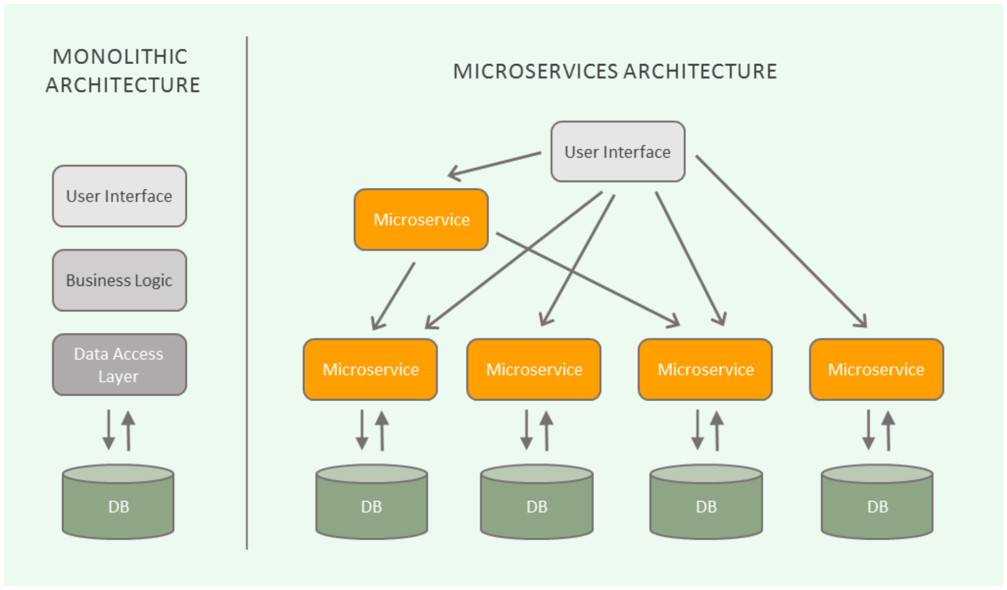
- Partial deployment
- Scaling independently when needed
- Getting rid of stale parts easily
- Constraints on more clean architecture
- Decoupled components
- Technology flexibility
- Multiple points of failure
- In general: **Maintainability**

5 How to handle implementation of Microservices?

6 Docker vs Virtual Machine

6.1 Container vs Image

Image



microservices-architecture

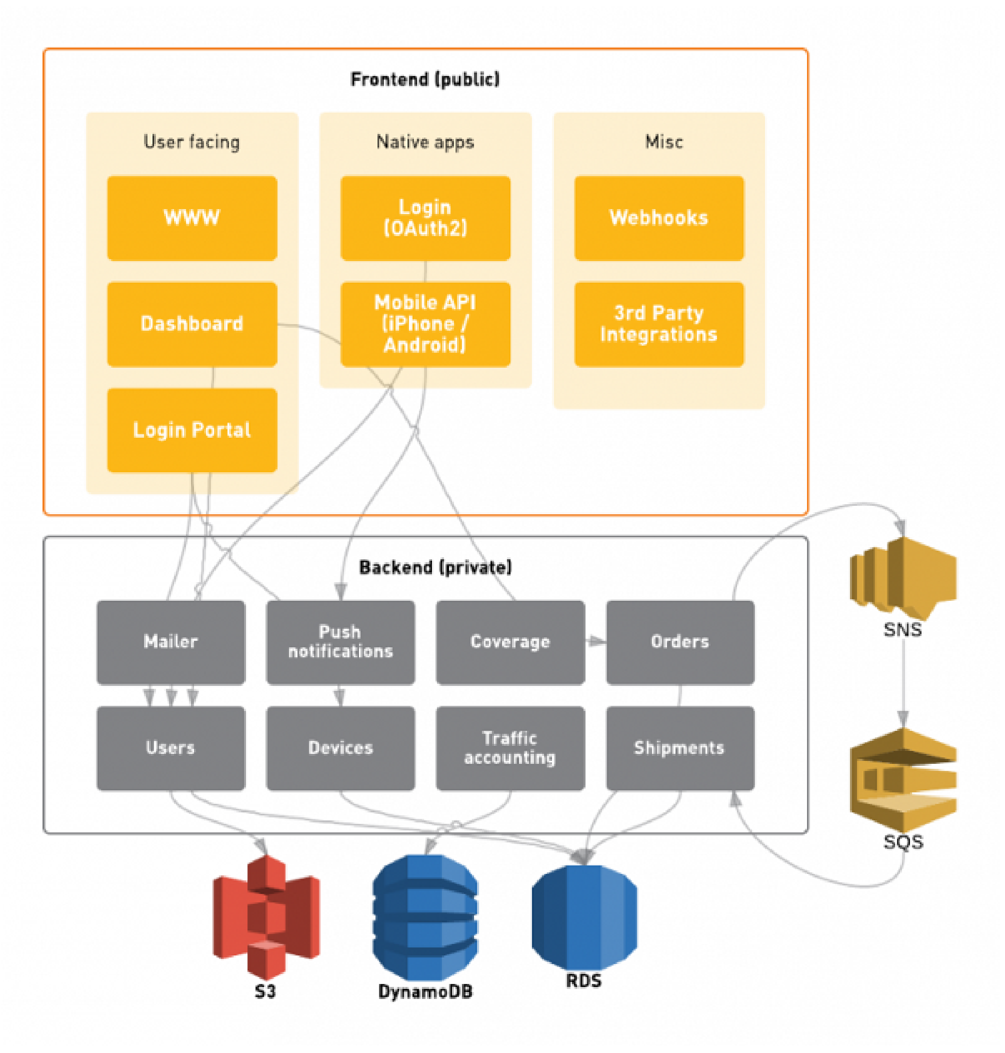
- Stateless, immutable file
- snapshot of a container
- composed of layers of other images

Container

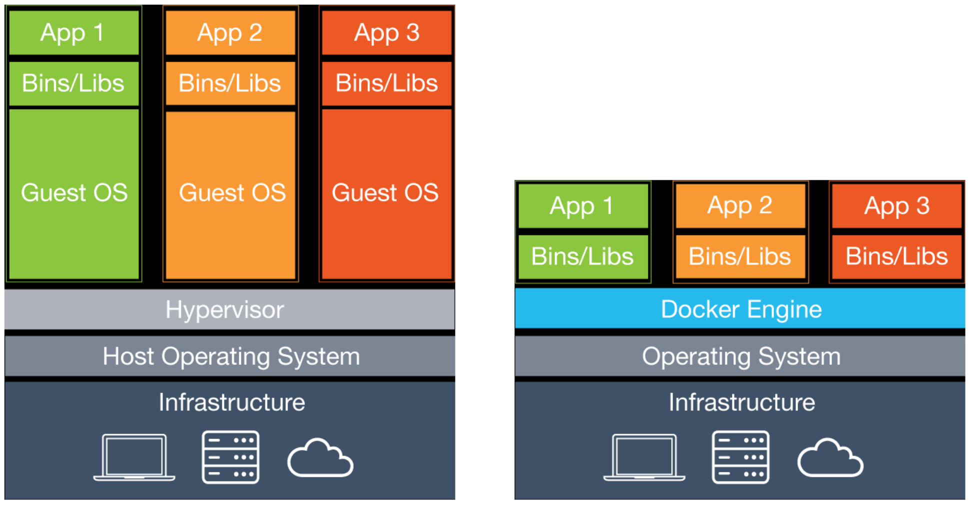
- An active instance of an image
- Encapsulation of environment in which we run application.

6.1.1 Docker (initial release: 2013)

- based on LCX Containers (initial release: 2008)
- Isolated environment
- Easier deployment
- Composability
- Lightweight and portable
- Resources constraint (e.g RAM, CPU usage)
- Blueprint of your system (Docker Image)
- Shared kernel, libraries across conainers
- Communicate with well-defined network
- Docker hub register with tons of (production) ready images (RedHat OS, nginx, Postgres, etc.)



microservices-example



vm-vs-docker-container

6.1.2 Real-life scenario:

Containers:

- nginx
- Postgres
- Mongo "database"
- RabbitMQ
- Two redis instances
- Three dockerized applications with business logic

7 Container Orchestration

8 Docker compose

- Compose containers and manage them in one file

  Not too good for complex production systems

9 Kubernetes

9.1 Kubernetes

- Declarative in nature
- Helping in container deployment
- Making sure all of the containers are running
- Rolling upgrades
- Load balancing

9.2 Docker - What can go wrong?

And why you should not follow every hype on software

- Space issues - ~1GB for each image, multiple layers
- Entirely abstracting the host networking.
  - Big mess of port redirection
  - DNS tricks and virtual networks.

Issues with kernel

- **Linux 3.x:** Unstable storage drivers
  - The only usable, widely supported storage driver is AUFS
  - Unstable.
  - Suffers from critical bugs provoking kernel panics and data corruption. In some companies this even happened every day.

- **Linux 4.x:** The kernel officially dropped AUFS support. **AUFS is entirely gone**

  Docker staff wrote a new filesystem, *Overlay*. (Not backward compatible...).

  Is it even possible to write a filesystem in 1 year?

  ext4 was developed for **5 years** before first stable release (maintaining backwards compatibility)

Overlay development was abandoned within 1 year of its initial release, because of various failures and problems with it.

Then comes Overlay2. Works only from Linux kernel 4.0 and docker 1.12.

No backports, just moving fast and breaking things...

9.2.1 What about Operation Systems compatibility?

- Windows? Forget about, docker technology relies strictly on linux kernel
- CentOS/RHEL: Russian roulette - RedHat team struggle hard to get docker work all of the time
- Debian: Jumping off a plane naked - AUFS driver issues
- Ubuntu: Not sure

9.2.2 Then which OS should i use?

The reasonable answer is to wait until RHEL 8 and Debian 10.

Alternatives:

- CoreOS - Container Linux

  Operating system that can only run containers and is exclusively intended to run containers.

- Google Container Engine / Amazon Machine Images

  Probably the only way to have stable docker - delegate management to someone else

9.3 Conclusions?

Docker is not mature yet for **critical** infrastructures.

It is still experimental. Wait for it.

9.3.1 What about companies?

- CoreOS is working on their own solution, Rocket
- Google is also replacing Docker + they have their own orchestration

10 When you should use Microservices

- Your domain model will not likely change soon and you have broad knowledge about business.
- You have growing development team
- You REALLY want indepentend, scalable components

11 Why you should NOT use Microservices

- More complex than Monolith
- Tough integration testing
- Data scattered across multiple databases
- Performance - time consuming I/O between components
- Hard to accord to a new/modified domain model
  - and to redefining your architecture
- **Rule of thumb: If you hesitate, DON'T USE THEM.** Stay in the Monolithic application

11.1 Must-have strategies while using Microservices with Docker

- Deployment automation (Ansible, Chef, Puppet, etc.)
- Continuous integration (Jenkins, Travis)
- Continuous Deployment / Delivery

12 Refrences:

12.0.1 Read/Watch - Worthy

https://thehftguy.com/2016/11/01/docker-in-production-an-history-of-failure/ https://thehftguy.com/2017/02/23/docker-in-production-an-update/ https://www.upcloud.com/blog/docker-swarm-vs-kubernetes/ https://www.youtube.com/watch?v=2yko4TbC8cI

12.0.2 Docs:

https://coreos.com/blog/rocket.html https://docs.aws.amazon.com/autoscaling/latest/userguide/AutoScalin
https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html
https://cloud.google.com/container-engine/ https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/