

# Lean Containers

# Who am I?

- Jérôme Petazzoni ([@jpetazzo](#))
- French software engineer living in California
- Joined Docker (dotCloud) more than 4 years ago  
(I was at Docker *before it was cool!*)
- I have built and scaled the dotCloud PaaS
- I learned a few things about running containers  
(in production)

# Outline

- Brief intro about Docker and containers
- VMs and containers: technical differences
- VMs and containers: functional differences
- Lean containers
- Composing stacks of containers

# Brief intro about Docker and containers

Build, ship, and run  
any app, anywhere

# Take any Linux program, and put it in a container

- Web apps and services, workers  
(Go, Java, Node, PHP, Python, Ruby...)
- Data stores: SQL, NoSQL, big data  
(Cassandra, ElasticSearch, Hadoop, Mongo, MySQL, PostgreSQL, Redis...)
- Other server-y things  
(Consul, Etcd, Mesos, RabbitMQ, Zookeeper...)
- Command-line tools  
(AWS CLI, Ffmpeg...)
- Desktop apps  
(Chrome, LibreOffice, Skype, Steam...)

# What about non-Linux programs?

- Desktop apps with WINE  
(e.g.: Spotify client)
- Coming soon: Docker for Windows  
(run Windows apps on Windows machines)
- Coming soon: Docker for FreeBSD  
(port in progress)
- Coming eventually: Docker for OS X  
(technically possible; but is this useful?)

# Ship that container easily and efficiently

- Docker comes with an image distribution protocol
- Distribution server can be hosted by Docker Inc. (free for public images)
- Distribution protocol is public
- Open source reference implementation (used by Docker Inc. for the public registry)
- Container images are broken down into layers
- When updating and distributing an image, only ship relevant layers



# Run those containers anywhere

- Containers can run in VMs or in physical machines
- Docker is available on all modern Linux variants
- Many IAAS providers have server images with Docker
- On OS X and Windows dev machines: boot2docker
- There are distros dedicated to run Docker containers (Atomic, CoreOS, RancherOS, Snappy Core...)
- Other Docker implementations exist (e.g. Joyent Triton)

# VMs and containers: technical differences

# Containers are portable

- VMs can't easily be moved
  - nested hypervisors (VMs in VMs) exist, but still rare
  - VM images have to be converted and transferred (both are slow operations)
- The same container can run on any machine (physical or virtual)
- Containers use a stable interface
  - Intel 64 bits machine code
  - Linux system calls ABI

# Containers have low overhead

- Normal\* process(es) running on top of normal kernel
- No device emulation (no extra code path involved in I/O)
- Context switch between containers  
= context switch between processes
- Benchmarks show no difference at all  
between containers and bare metal  
(after adequate tuning and options have been selected)
- Containers have higher density

\* There are extra "labels" denoting membership to given namespaces and control groups. Similar to regular UID.

# VMs have stronger isolation

- Inter-VM communication must happen over the network  
(Some hypervisors have custom paths, but non-standard)
- VMs can run as non-privileged processes on the host  
(Breaking out of a VM will have ~zero security impact)
- Containers run on top of a single kernel  
(Kernel vulnerability can lead to full scale compromise)
- Containers can share files, sockets, FIFOs, memory areas...  
(They can communicate with standard UNIX mechanisms)

# Analogy: brick walls vs. room dividers

- Brick walls
  - sturdy
  - slow to build
  - messy to move
- Room dividers
  - fragile
  - deployed in seconds
  - moved easily

# Blurring lines

- Intel Clear Containers; Clever Cloud  
(stripped down VMs, boot super fast, tiny footprint)
- Joyent Triton  
(Solaris "branded zones," running Linux binaries securely, exposing the Docker API)
- Ongoing efforts to harden containers  
(GRSEC, SELinux, AppArmor)

# VMs and containers: functional differences



# Inside

- VMs typically contains\* everything they need  
(Backups, logging, periodic job execution, remote access...)
- Containers are the subject of an epic debate:
  - machine container  
(runs init, cron, ssh, syslog ... and the app)
  - application container  
(runs the app and nothing else;  
relies on external mechanisms)

\* No pun intended!

# Creation / deployment

- Containers are (typically) created from an image
- Updates = update the image, redeploy a new container
- "Immutable servers" pattern
- VMs can use the same pattern ("golden images"), but it's heavier to setup
- VMs often have a long lifecycle instead (provisioning→update→update→...→update→disposal)
  - easily leads to configuration drift (subtle differences that add up over time)
  - requires tight configuration management

# Development process (VMs)

- Hypothesis: app broken down in 10 components
- Production: 10+ VMs (each component in 1+ VM)
- Development: typically 1 VM for whole app
- Different components depending on environment (e.g.: logging, monitoring, service discovery...)
- Consequence: prod and dev deployments differ a lot

# Development process (containers)

- Hypothesis: app broken down in 10 components
- Production: 10+ containers (across any number of VMs)
- Development: 10 containers on 1 dev VM
- Re-use the same container images for prod and dev
- How do we provide container variants?

# Bloated containers

- Containers have all the software required for production
- In dev mode, only essential processes are started
- In prod mode, additional processes run as well
- Problems:
  - bigger containers
  - behavior can differ (because of extra processes)
  - extra processes duplicated between containers
  - hard to test those extra processes in isolation

# Lean containers

# Principle

- "Do one thing, do it well"
- One container for the component itself
- One container for logging
- One container for monitoring
- One container for backups
- One container for debugging (when needed)
- etc.

# Implementation (general principles)

- Containers can *share* almost anything, *selectively*
  - files  
(logs, data at rest, audit)
  - network stack  
(traffic routing and analysis, monitoring)
  - process space, memory  
(process tracing and debugging)



**Let's dive into the  
details**

# Logging (option 1: Docker logging drivers)

- Containers write to standard output
- Docker has different logging drivers:
  - writes to local JSON files by default
  - can send to syslog

Imperfect solution for now, but will be improved.  
Preferred in the long run.

# Logging (option 2: shared log directory)

- Containers write regular files to a directory
- That directory is shared with another container

```
docker run -d --name myapp1 -v /var/log myapp:v1.0
```

In development setup:

```
docker run --volumes-from myapp1 ubuntu \  
sh -c 'tail -F /var/log/*'
```

In production:

```
docker run -d --volumes-from myapp1 logcollector
```

# Logging takeaways

- Application can be "dumb" about logging
- Log collection and shipping happens in Docker, or in separate(s) container(s)
- Run custom log analyzer without changing app container (e.g. apachetop)
- Migrate logging system without changing app container

# "Yes, but..."

- "What about performance overhead?"
  - no performance overhead
  - both containers access files directly  
(just like processes running on the same machine)
- "What about synchronization issues?"
  - same as previous answer!

# Backups (file-based)

- Store mutable data on Docker volumes (same mechanism as for logs)
- Share volumes with special-purpose backup containers
- Put backup tools in the backup container (boto, rsync, s3cmd, unison...)

```
docker run --volumes-from mydb1 ubuntu \
    rsync -av /var/lib/ backup@remotehost:mydb1/
```

- The whole setup doesn't touch the app (or DB) container

# Backups (network-based)

- Run the backup job (pg\_dump, mysqldump, etc.) from a separate container
- Nothing complicated, but with VMs, this is overkill ("this VM does nothing at all; except a few minutes per day!")
- Advantages (vs. running in the same container):
  - nothing to install in the app (or DB) container
  - if the backup job runs amok, it remains contained (!)

# Network analysis

- Packet capture (tcpdump, ngrep, ntop, etc.)
- Low-level metrics (netstat, ss, etc.)
- Install required tools in a separate container image
- Run a container in the same *network namespace*

```
docker run -d --name web1 nginx
docker run -ti --net container:web1 tcpdump -pni eth0
docker run -ti --net container:web1 ubuntu ss -n --tcp
```



# Service discovery

- Docker can do *linking* and generic DNS injection
- Your code connects to e.g. `redis` (pretending that `redis` resolves to something)
- Docker adds a DNS alias\* so that `redis` resolves to the right container, or to some external service
- In dev, Docker Compose manages service dependencies
- In prod, you abstract service discovery from the container

\* Really, an entry in the container's `/etc/hosts`.

# Service discovery in practice

When service A needs to talk to service B...

1. Start container B on a Docker host

# Service discovery in practice

When service A needs to talk to service B...

1. Start container B on a Docker host
2. Retrieve host+port allocated for B

# Service discovery in practice

When service A needs to talk to service B...

1. Start container B on a Docker host
2. Retrieve host+port allocated for B
3. Start *ambassador* (relaying to this host+port)

# Service discovery in practice

When service A needs to talk to service B...

1. Start container B on a Docker host
2. Retrieve host+port allocated for B
3. Start *ambassador* (relaying to this host+port)
4. Start container A linked to ambassador

# Service discovery in practice

When service A needs to talk to service B...

1. Start container B on a Docker host
2. Retrieve host+port allocated for B
3. Start *ambassador* (relaying to this host+port)
4. Start container A linked to ambassador
5. Profit!

# General pattern

- Your code runs in the same container in dev and prod
- Add "sidekick\*" containers for additional tasks
- Developers don't have to be bothered about ops
- Ops can do their job without messing with devs' code

\* Kubernetes sometimes calls them "sidecars."

# Composing stacks of containers



# Docker Compose

# docker-compose.yml

```
rng:
  build: rng

hasher:
  build: hasher

webui:
  build: webui
  links:
    - redis
  ports:
    - "80:80"
  volumes:
    - "webui/files:/files/"

redis:
  image: redis

worker:
  build: worker
  links:
    - rng
    - hasher
    - redis
```

# Docker Compose

- Start whole stack with `docker-compose up`
- Start individual containers (and their dependencies) with `docker-compose up xyz`
- Takes care of container lifecycle (creation, update, data persistence, scaling up/down...)
- Doesn't automatically solve networking and discovery (yet)

# Docker Compose

- Start whole stack with `docker-compose up`
- Start individual containers (and their dependencies) with `docker-compose up xyz`
- Takes care of container lifecycle (creation, update, data persistence, scaling up/down...)
- Doesn't automatically solve networking and discovery (yet)
- ... However ...

# docker-compose.yml, reloaded

```
hasher:  
  build: hasher  
  
worker:  
  build: worker  
  links:  
    - rng  
    - hasherproxy:hasher  
    - redis  
  
hasherproxy:  
  image: jpetazzo/hamba  
  links:  
    - hasher  
  command: 80 hasher 80
```

(This was automatically generated by a tiny Python script.)

# Fair warning

- Docker networking is evolving quickly
- Docker 1.7 (to be released Real Soon Now) will support:
  - "networks" as first class objects
  - multiple networks
  - overlay driver allowing to span networks across multiple hosts
  - networking plugins from ecosystem partners

# Conclusions

# Conclusions

- Containers can share more context than VMs
- We can use this to decouple complexity (think "microservices" but for ops/devs separation)
- All tasks typically requiring VM access can be done in separate containers
- As a result, deployments are broken down in smaller, simpler pieces
- Complex stacks are expressed with simple YAML files
- Docker isn't a "silver bullet" to solve all problems, but it gives us tools that make our jobs easier



Thanks!  
Questions?

@jpetazzo  
@docker