Orchestrating Distributed Deployments with Docker and 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

- What are microservices?
- What are their challenges?
- How does Docker help?
- How can we do this today?
What are microservices?
Microservices: a style of software architecture

- Break big application down into many small services
- Example: e-commerce
  - web front-end
  - catalog of products
  - inventory/stock management
  - shipping calculator
  - payment processor
  - billing/invoicing
  - recommendation engine
  - user profiles
Why are we doing this? (1/2)

- Use different stacks for different services (use the right language/tool for the job)
- Replace or refactor individual services easily
- Less coordination required when deploying
  - deploy more often
  - less risk
  - more agility and speed
Why are we doing this? (2/2)

• Promotes many small teams instead of one big army
  ◦ smaller teams = less communication overhead
  ◦ cf Jeff Bezos' "two-pizza rule"

• Enables effective ownership of services
  ◦ a service always has an "owner" (=team)
  ◦ the owner is responsible (=on call)
  ◦ the owner is empowered (=can fix things)
What are the challenges associated with microservices?
Fast, efficient RPC calls

Docker *does not help* with that. See instead:

- ZeroRPC
- Cap'n Proto
- XMLRPC
- SOAP
- Dnode
- REST
- Queues (like AMQP), for long-running/async operations
Architecturing the application in small blocks

Docker *does not help* with that.

Some tips:

- Pretend that you're outsourcing parts of your stack
- Wrap database access; each object (or table) = one service
Efficient deployment system

- Instead of deploying one monolith once in a while, we deploy many services very frequently.

- The manual, tedious, uniquely tailored deployment system has to be replaced with something simple and reliable.

- It must be easy to add new components and deploy them.

Docker does help with that.
Network plumbing

- Instead of one monolith talking to a database, we now have 100s of services talking to each other.
- Those services will often be scaled independently.
- Instead of direct, deterministic library calls, we now have network calls across load balancers.
- Scaling, load balancing, monitoring must be easy to do.

Docker *does help* with that.
More challenges

See *Microservices: not a free lunch* by Benjamin Wootton.
How does Docker help us to implement microservices?
Building code, without Docker

- Deployment scripts are fragile
- Configuration management is hard (and not a silver bullet)
- Environments differ:
  - from dev to dev
  - from dev to prod
Building code, with Docker

- Write a `Dockerfile` for each component
- Builds are fast
  - each build step is saved
  - future builds will reuse saved steps
- Builds are reproducible
  - build environment can be well-defined
  - outside context is limited
root@dockerhost:~#
Shipping code, without Docker

- Push code or build artifacts to servers
- Distro packages (deb, rpm...) are great, but hard to build (too generic)
- Artifact libraries are great, but tailored to specific languages (too specific)
- When deployment fails, rollback isn't guaranteed to work
Shipping code, with Docker

- Container images can be pushed to *registries*
- Container engines can pull from those registries
- Docker Inc. provides a free registry (the Docker Hub)
- Works out of the box in 2 minutes (includes account creation and engine setup)
- Can also be deployed on prem
- Rollbacks are easy (see: immutable infrastructure with containers)
Network plumbing, without Docker

- Application code must be modified to handle:
  - service discovery
    (e.g. lookup endpoint addresses into Consul, Etcd, ZK)
  - fail over
    (same, but also needs to watch for changes)

- Development stack becomes either:
  - very complex
    (because of those extra components)
  - different from production
    (because the code path differs)
Network plumbing, with Docker

- Application code doesn't deal with plumbing
- Application code connects to services using DNS aliases
- DNS aliases are injected by Docker
- In dev, DNS aliases map directly to service containers
- In prod, DNS aliases map to special-purpose containers, implementing service discovery, load-balancing, failover
- Those containers are called *ambassadors*
How can we do this today?
Developing on a single node

Docker Compose

- Describe a stack of containers in a simple YAML file
- Start the stack with a single command
- Compose connects containers together with links
- Also provides simple scaling and log aggregation
Scaling with static resource scheduling

Docker Compose + Docker remote API

- Deploy stateful services separately
- Replace them with ambassadors in the stack
- Instantiate the stack multiple times
- Fits well with existing IAAS auto-scaling models
Scaling with dynamic resource scheduling

Docker Compose + Docker Swarm

- Swarm consolidates multiple Docker hosts into a single one
- Swarm "looks like" a Docker daemon, but it dispatches (schedules) your containers on multiple daemons
- Swarm talks the Docker API front and back
- Swarm is open source and written in Go (like Docker)
- Swarm was started by two of the original Docker authors (@aluzzardi and @vieux)
- Swarm is not stable yet (version 0.3 right now)
No demo
Thanks!
Questions?

@jpetazzo
@docker