Immutable infrastructure 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 is immutable infrastructure?
- What are its pros and cons?
- How can it be implemented with containers?
- Also: demos!

Immutable infrastructure

(a.k.a. immutable servers, phoenix servers, etc.)

Rule 1: never change what's on a server

- Don't install new packages
- Don't upgrade existing ones
- Don't remove or downgrade them
- (Even for security vulnerabilities!)
- Don't edit configuration files
- Don't update your app code
- (Even for small or urgent fixes!)

Rule 2: if tempted to change something...

- See *Rule 1*

(OK, we will see an exception later.)

How do we upgrade?

- Create new server from scratch
- Apply deployment process*
  (scripts, configuration management...)
- (Optional: test the new server)
- Replace old server with new server
- *Keep old server around,* just in case

* Configuration management helps, but is not mandatory here.

WHY?!?

Avoid *drift*

Avoid *drift*

Avoid *drift*

- Drift = differences between servers
  (when they are supposed to be identical)
- Caused by:
  - provisioning servers at different times
  - any manual operation
- Consequences:
  - seemingly random failures
  - same code, different behavior
  - gets worse with time

Coping with drift

- Careful replication of manual operations doesn't scale
  (and is error-prone)
- Automation seems simple at first,
  but has to deal with many edge cases
- Configuration management helps,
  but only deals with what you've defined

Automation fails

"Let's use `parallel-ssh`!" (Or your favorite tool)

- What if some servers...
  - are unreachable
  - become unreachable during the process
  - are being provisioned at the same time
- What if one of those services is (partially) down?
  - distro package repositories
  - code or artifact repositories

Config management fails

```
package { "openssl": ensure => "installed" }
```

Config management fails

```
package { "openssl": ensure => "installed" }
```

 A wild OpenSSL vulnerability appears!

Config management fails

```
package { "openssl": ensure => "installed" }
```

 A wild OpenSSL vulnerability appears!

```
package { "openssl": ensure => "1.0.1g" }
```

Config management fails

```
package { "openssl": ensure => "installed" }
```

 A wild OpenSSL vulnerability appears!

```
package { "openssl": ensure => "1.0.1g" }
```

 Something went wrong, abort, abort!

Config management fails

```
package { "openssl": ensure => "installed" }
```

 A wild OpenSSL vulnerability appears!

```
package { "openssl": ensure => "1.0.1g" }
```

 Something went wrong, abort, abort!

```
package { "openssl": ensure => "installed" }
```

Config management fails

```
package { "openssl": ensure => "installed" }
```

 A wild OpenSSL vulnerability appears!

```
package { "openssl": ensure => "1.0.1g" }
```

 Something went wrong, abort, abort!

```
package { "openssl": ensure => "installed" }
```

 We didn't roll back to whatever-we-had!

Config management fails

```
package { "openssl": ensure => "installed" }
```

 A wild OpenSSL vulnerability appears!

```
package { "openssl": ensure => "1.0.1g" }
```

 Something went wrong, abort, abort!

```
package { "openssl": ensure => "installed" }
```

 We didn't roll back to whatever-we-had!

```
package { "openssl": ensure => "1.0.1f" }
```

Config management fails

```
package { "openssl": ensure => "installed" }
```

 A wild OpenSSL vulnerability appears!

```
package { "openssl": ensure => "1.0.1g" }
```

 Something went wrong, abort, abort!

```
package { "openssl": ensure => "installed" }
```

 We didn't roll back to whatever-we-had!

```
package { "openssl": ensure => "1.0.1f" }
```

 This should do the trick. (Hopefully.)

More nightmares

```
package { "openssl": ensure => "1.0.1f" }
```

More nightmares

```
package { "openssl": ensure => "1.0.1f" }
```

😧 Package not found on the repos.

More nightmares

```
package { "openssl": ensure => "1.0.1f" }
```

- 😯 Package not found on the repos.
- 😨 "Well, actually" we want an older version.

More nightmares

```
package { "openssl": ensure => "1.0.1f" }
```

- 😧 Package not found on the repos.
- 😨 "Well, actually" we want an older version.
- 😰 Package even less likely to be found on the repos.

More nightmares

```
package { "openssl": ensure => "1.0.1f" }
```

😟 Package not found on the repos.

😨 "Well, actually" we want an older version.

😰 Package even less likely to be found on the repos.

😱 "Well, actually" we were using 0.9.8xxx.

When we requested 1.0.1g we upgraded the whole distro.

More nightmares

```
package { "openssl": ensure => "1.0.1f" }
```

😧 Package not found on the repos.

😨 "Well, actually" we want an older version.

😰 Package even less likely to be found on the repos.

😱 "Well, actually" we were using 0.9.8xxx.

When we requested 1.0.1g we upgraded the whole distro.

(╯°□°）╯︵ ┻━┻)

With immutable servers

- We still have the old server
- Just put it back into service
  (while we figure out the OpenSSL upgrade!)
- Also works for any kind of upgrade
  that needs to be rolled back

Alright, we have easy rollbacks.

But how does that help with *drift*?

"Trash your servers and burn your code"

(Chad Fowler)

- Reprovision your servers regularly
  (from scratch)
- Ensures that you're always using recent packages
- Any manual deviation gets fixed automatically

Improvement: golden image

- Create a server from scratch
- Apply deployment process
- Snapshot this server (create an image)
- (Optional: create a test server and validate it)
- Create multiple identical servers from the image

Avoids uncertainties in the deployment process: unreachable packages repositories etc.

Allows to keep (for cheap) past versions around.

Downsides

(and how to cope)

Problem: small changes are cumbersome

E.g. one line of CSS.

- Before: manual change, validate, replicate
  (a few minutes)
- After: manual change, validate, ...
  - create new golden image from scratch
    (one hour)
  - provision new servers from image
    (a few minutes)
  - switch old/new servers
  - decommission old servers after a while

Solution: automation

- All those operations have to happen
- But everything after the "validate" step should be automated
- The *clock time* will still be 1+ hour
- The *user time* will be a few minutes (just like before)

Note: intermediary golden images can help (provision from checkpoint instead of from scratch)

Problem: debugging is harder

E.g. troubleshoot network issues.

- Before:
  - install `tcpdump`
  - fiddle with `iptables`
  - accumulate logs and packet captures locally
- After:
  - install `tcpdu`-oops, the server was re-imaged
  - fiddle with `ipta`-oops, ...
  - logs and traces have to be shipped out

Solution 1: drift and self-destruct

- Tag a given machine to prevent its "re-imaging"
- Schedule it for self-destruct after e.g. 1 week
  (`shutdown +10000`)
- That machine is allowed to drift
  (you can install your tools on it,
  leave logs and traces locally...)
- If you need more time, reschedule the self-destruct

Solution 1: drift and self-destruct

- Tag a given machine to prevent its "re-imaging"
- Schedule it for self-destruct after e.g. 1 week
  (`shutdown +10000`)
- That machine is allowed to drift
  (you can install your tools on it,
  leave logs and traces locally...)
- If you need more time, reschedule the self-destruct

If you find yourself setting up a cron job to reschedule the self-destruct, you're doing it wrong!

Solution 2: bundle the tools

- Install `tcpdump` and friends in the golden image
- Enable traffic capture with feature switch
- (Alternate solution: statistical sampling)
- Automate shipping of logs and traces

It's more work in the beginning, but pays in the long run.

Problem: storing data

Databases and anything stateful!

- Before: just store it locally
- After: need to persist it somehow

Solution 1: not my problem

"Often you can pass the buck to a service which someone else maintains, like Amazon's RDS database service."

(Kief Morris)

- Easy!
- But what if:
  - there is no such service
  - I can't use it for `$REASONS`?

Solution 2: state = files

All you need is a mechanism to store files externally.

- NAS/SAN (on-prem)
- EBS, EFS (AWS)
- Ceph, Gluster... (anywhere)

But it's extra work, expensive, and/or slower.

Solution 3: ?

Solution 3: ?

SPOILER ALERT

Solution 3

Immutable containers

Let's review our process

- Create image:
  - from scratch
    (can take an hour or more)
  - from checkpoint
    (takes a few minutes, more complex)
- Deploy it N times
  (takes a few minutes)

How do we do that with containers?

Building container images

- We get the best of both worlds:
  - from scratch
    (clean rebuilds without side-effects)
  - incremental
    (fast rebuilds when changes are minor)
- Why and how?
  - container snapshots are *cheap*
    (seconds versus minutes)
  - simple DSL to break down the build into steps
    (each step = one command = one snapshot)

```
FROM debian:jessie
MAINTAINER Jessica Frazelle <jess@docker.com>

# Install dependencies
RUN apt-get update && apt-get install -y \
    build-essential \
    ... \
    --no-install-recommends

# Install node
RUN curl -sL https://deb.nodesource.com/setup | bash -
RUN apt-get install -y nodejs

# Clone atom
RUN git clone https://github.com/atom/atom /src
WORKDIR /src
RUN git fetch && git checkout \
    $(git describe --tags \
          `git rev-list --tags --max-count=1`)
RUN script/build && script/grunt install

# Autorun atom
CMD /usr/local/bin/atom --foreground
```

What happens during the first build?

```
FROM debian
RUN apt-get xxx
COPY . /src
RUN /src/build
```

- Create a container from `debian` base image
- Execute `apt-get xxx` in this container, take a snapshot
- Create a container from this snapshot
- Copy source files into `/src`, take a snapshot
- Create a container from this snapshot
- Execute `/src/build` in this container, take a snapshot

The final snapshot is our built image.

What happens during subsequent builds?

- Before executing each step:
  check if we already executed the same step before
  (and have a snapshot of its result)
  - if we do, use the snapshot and continue
  - otherwise, execute the step normally
    (and snapshot the result)
- As a result, we zoom through the build process,
  until we hit a step that has changed
- The end result is the same as a full clean build,
  but much faster

Demo

Running container images

- On physical or virtual machines
- Run multiple containers per machine
- Upgrading is faster
  (doesn't have to wait for IaaS VM to come up)
- Can reuse local data (Docker concept: "volumes")
- Solves the stateful service problem

Demo

Bonus

- Containers can share:
  - directories (e.g.: logs)
  - network stack (e.g.: traffic analysis)
  - ... and more!

Logging, backups, metrics collection, troubleshooting...
can be done from "sidekick" containers.

Demo

Other niceties

- Containers filesystem can be made read-only
  - enforces immutability
  - exception for data volumes (with `noexec`)
  - easier security audit
- Cheaper
  - consolidation
  - save a few ¢ or $ per server per deploy
    (great if your IAAS bills by the hour)

Conclusions

Immutable containers

- All the advantages of immutable servers (avoid drift, reliable rollbacks...)
- Build in seconds instead of minutes/hours
- Faster, simpler deployment
- Deal with stateful services
- Bonus: cheaper, safer, cleaner

Thanks!
Questions?

@jpetazzo
@docker