VMs repetibles y portátiles con Vagrant ~~y~~, Docker, Ansible ~~y~~, Terraform y *Lambdas*

(*versión ~~7~~ ~~8~~ 9*)

Carlos M. Martinez
@carlosm3011 -> 2015-2021,2022

Indice

Virtualización full vs Containers

Virtualización full:

Software que emula hardware, I/O y storage, haciéndole ‘creer’ a un S.O. (el ‘guest’) que está corriendo en un hardware físico (el ‘host’).

Ejemplos: KVM, VirtualBox, VMWare, AmazonAWS, GCE (Google Compute Engine).

Containers:

Kernels de S.O. que corren como procesos de usuario dentro de otro kernel similar.

Ejemplos: Virtuozzo, FreeBSD Jails, LXC Containers

Virtualización full vs Containers

Virtualización full:

Ventajas: se puede correr *casi* cualquier S.O. en *casi* cualquier host. Cosas como Windows Server 2012 sobre Linux/FreeBSD son posibles.

Desventajas: “Costoso/pesado” en términos de recursos para el host. Correr más de 5 o 6 VMs en una notebook se vuelve poco práctico.

Containers:

Ventajas: “Livianito” en términos de recursos para el host. Correr 50 o 100 contenedores en un PC relativamente modesto es posible.

Desventajas: Solo se pueden ejecutar derivados del S.O. host, es decir, otros kernels de Linux o FreeBSD

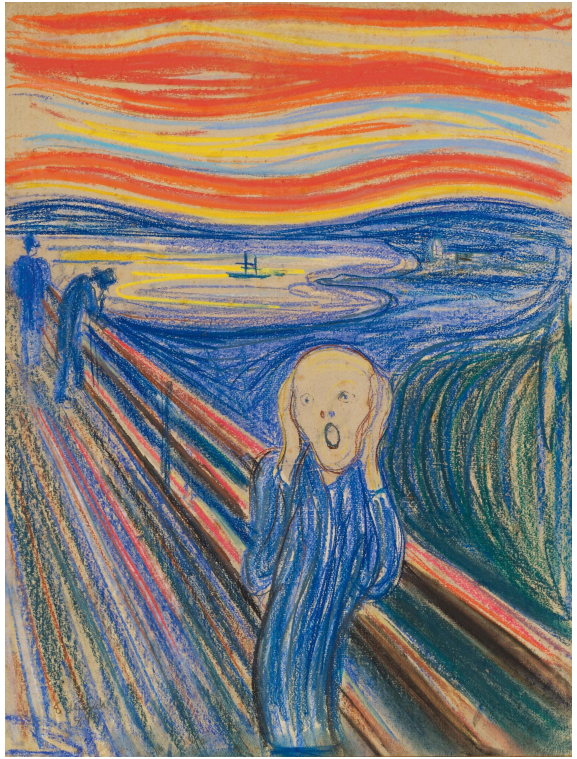
Ciclo de trabajo “tradicional” con VMs

¿Cómo trabajamos normalmente?

1. Instalamos VirtualBox
2. Nos bajamos el ISO del Ubuntu/Debian/CentOS
3. Armamos una VM
4. Instalamos el Linux
5. Instalamos un sillón de paquetes
6. Instalamos lo que precisamos (RPKI? Validador de RIPE? Quagga?)
7. Copiamos archivos de configuración, los ajustamos a mano
8. Tocamos la configuración de red hasta que queda funcionando

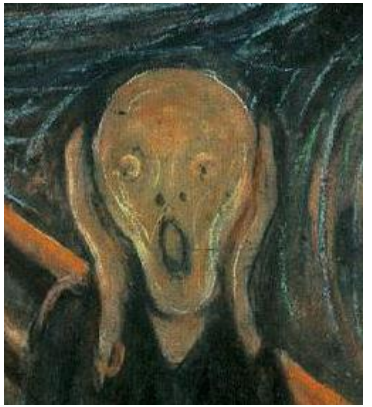
Ciclo de trabajo “tradicional” con VMs

¿Y cuando queremos armar el mismo sistema 6 meses después o actualizar un componente?



Ciclo de trabajo “tradicional” con VMs

¿Y cuando queremos armar el mismo sistema 6 meses después o actualizar un componente?



1. **Instalamos VirtualBox**
2. **Nos bajamos el ISO del Ubuntu/Debian/CentOS**
3. **Armamos una VM**
4. **Instalamos el Linux**
5. **Instalamos un sillón de paquetes**
6. **Instalamos lo que precisamos (RPKI? Validador de RIPE? Quagga?)**
7. **Copiamos archivos de configuración, los ajustamos a mano**
8. **Tocamos la configuración de red hasta que queda funcionando**

Vagrant/Docker al rescate

Vagrant [https://www.vagrantup.com/] es una capa de abstracción sobre proveedores de virtualización

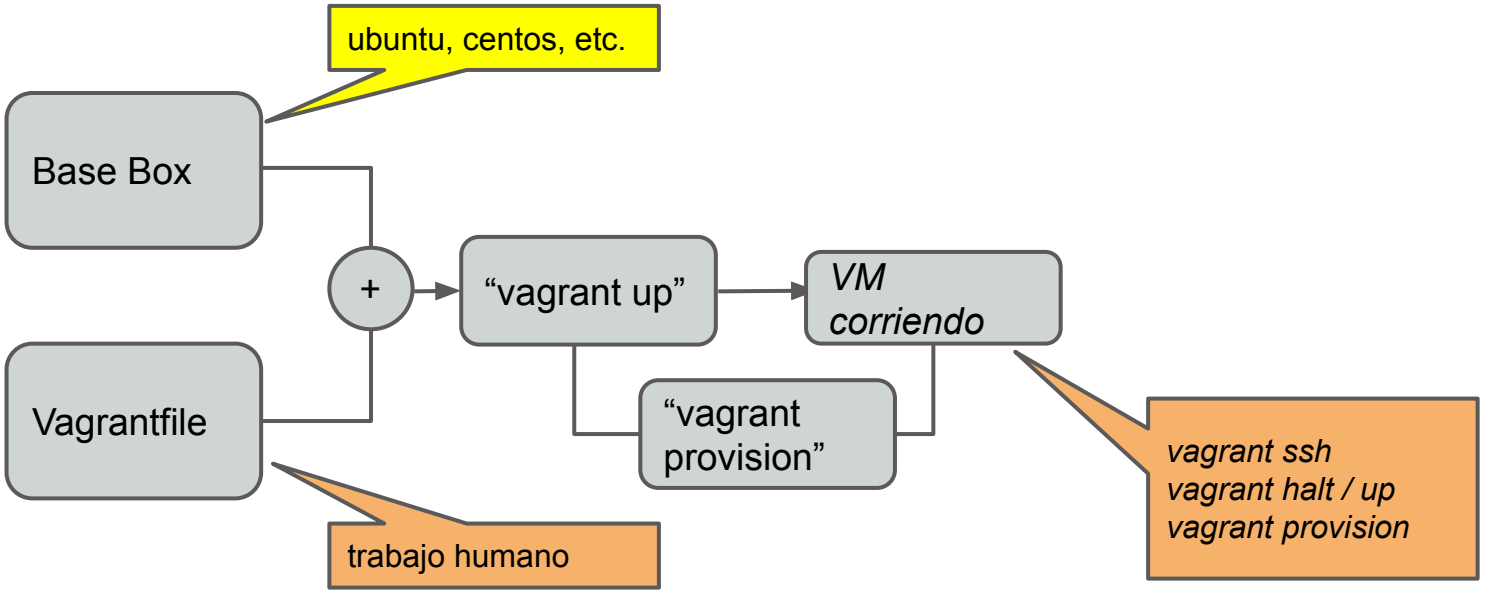
Vagrant nos permite automatizar la creación de ambientes virtualizados mediante ‘recetas’ o scripts llamados *Vagrantfiles*

Vagrant

Terminología:

- “**provider**”: Es el elemento que provee la función de virtualización y ejecución de las máquinas virtuales
  - Ejemplos: VirtualBox, VMWare, AWS, GCE
- “**provisioning”**: Acción de configurar una máquina virtual de acuerdo a una especificación (Vagrantfile)
- “**provisioners”:** Herramientas capaces de reflejar una especificación en una VM
  - Ejemplos: shell, Puppet, Chef, Ansible

Vagrant - Flujo de trabajo



Vagrant - Flujo de trabajo (ii)

vagrant up/halt

Arrancar / parar la VM. La primera vez que se corre implica tambien '*provision*'

vagrant provision

Aplicar las recetas contenidas en el Vagrantfile

vagrant destroy

Borrar completamente el estado acumulado para empezar nuevamente de cero

Vagrantfile

El **Vagrantfile** es un conjunto de recetas (‘provisioners’) que se aplican a la base box una vez que esta levantó.

Es la colección de *apt-get, yum y “configure && make && make install” que debemos hacer para dejar algo funcionando.*

Directorios compartidos

Las VMs Vagrant comparten por defecto el mismo directorio donde está el Vagrantfile con su VM, montado en “/vagrant”

Lo que se copie allí está visible tanto por la VM como por el host.

De esta manera podemos tener proyectos que trabajamos desde el host pero que ejecutamos o construimos en la VM:

- código fuente
- scripts
- archivos de configuración

¡Gracias!

Fin de la Parte 1

Virtualización full vs Containers

Full Virtualization:

A software layer emulates hardware, I/O and storage, thus leading a ‘guest OS’ that is running alone on a hardware platform.

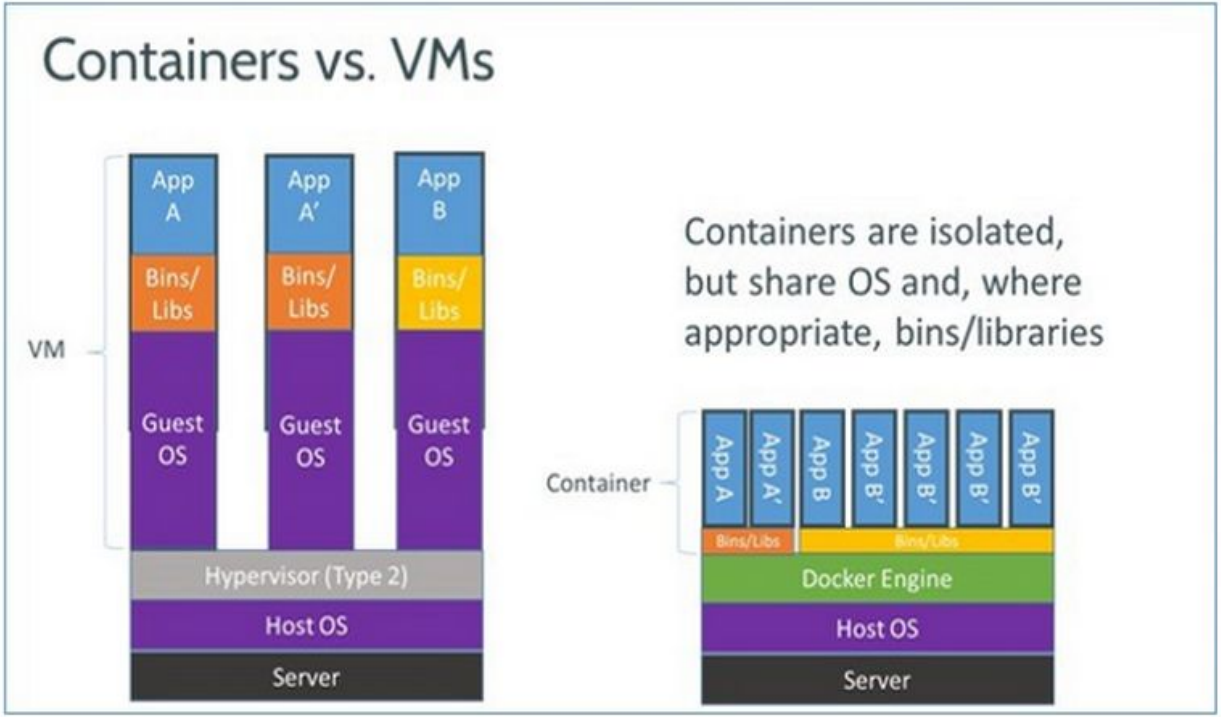
Some examples: KVM, VirtualBox, VMWare, AmazonAWS, GCE (Google Compute Engine).

Containers:

Isolated OS kernels, modified to run in user-mode, having access to a set of libraries and support files inside a similar kernel.

Some examples: Virtuozzo, FreeBSD Jails, LXC Containers

Docker


Containers vs. VMs
VM
App A
App A'
App B
Bins/ Libs
Bins/ Libs
Bins/ Libs
Guest OS
Guest OS
Guest OS
Hypervisor (Type 2)
Host OS
Server
Containers are isolated, but share OS and, where appropriate, bins/libraries
Container
App A
App A'
App B
App B'
App B'
App B'
App B'
Bins/Libs
Bins/Libs
Docker Engine
Host OS
Server

Docker

Images and Containers

- An **image** is a complete set of libraries and other support files that make up an operating system image
  - There are images for different ubuntu versions, centos, etc.
- A **container** is a particular instance of a command run inside an image
  - You can have a container for running **mysql**, or '**echo hello world**'
  - A container is launched as a process inside the host kernel, and it can either run as a daemon or as entry-exit normal process
  - Each time we run a command on a given image, a new container is created

Docker

Using docker:

- **docker** is the main tool used to manage containers and images

- many commands look and feel similar to the traditional ‘pstools’ (tools for managing processes)
  - docker ps
  - docker rm

- containers are identified by either a hash (created automatically) or a name (either automatically or manually assigned)

Docker - Managing Containers

Check created containers: docker ps [--all]

<<insert docker ps --all output>>

Starting a container: docker run

Stopping a container: docker stop <hash|name>

¡Gracias!

Fin de la Parte 2

Gestión de containers

Gestión de containers (o de procesos)

- Auto arranque (luego del *system boot*)
- Control (arranque / parada controlada)
- Auto arranque (en caso de fallas)
- Logging de estos eventos

Docker usa el *process manager* del sistema para esa función.

Gestión de containers

Process managers mas comunes:

- Upstart (Ubuntu)
- SystemD (Ubuntus mas recientes, debian)
- SupervisorD

Ejemplo de SystemD service file

ripeval.service

```
# cat /etc/systemd/system/ripeval.service
[Unit]
Description=RIPE-NCC RPKI Validator
After=network.target

[Service]
Type=forking
User=ripeval
WorkingDirectory=/home/ripeval
ExecStart=/home/ripeval/ripeval_latest/rpki-validator.sh start
ExecStop=/home/ripeval/ripeval_latest/rpki-validator.sh stop
Restart=on-abort

[Install]
WantedBy=multi-user.target
```

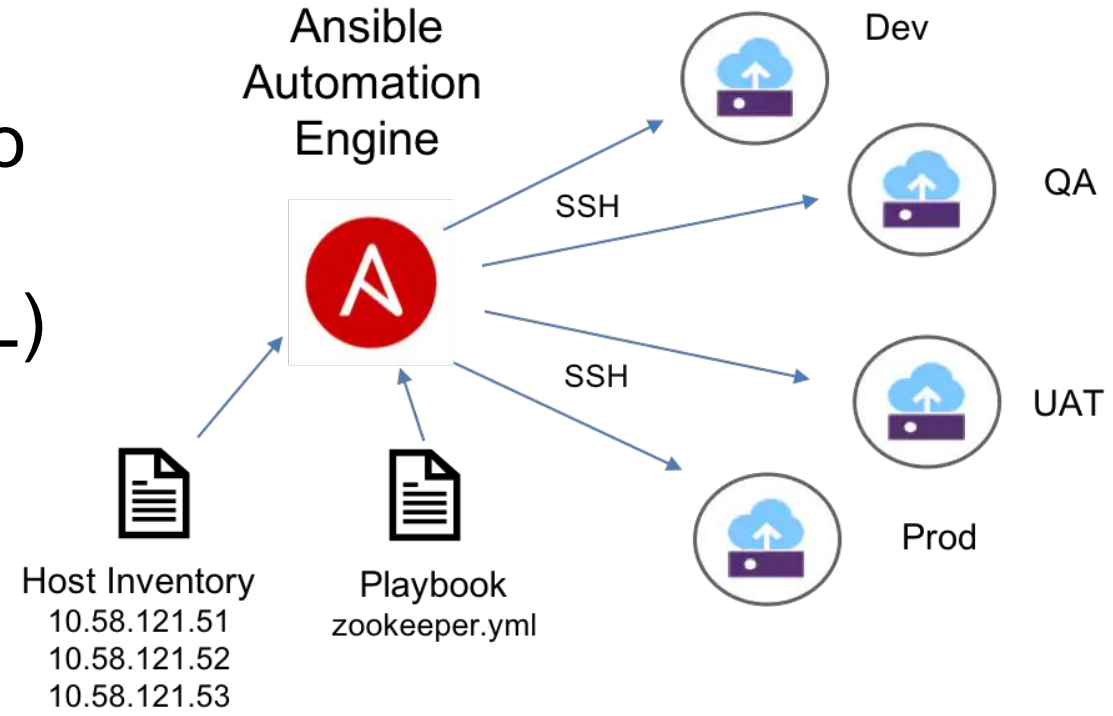
¡Gracias!

Fin de la Parte 3

Ansible y Terraform

Ansible

Configuración de ambientes usando un lenguaje declarativo (YAML)



Terraform

Instanciación de infraestructura expresada en un lenguaje declarativo (YAML)

- Creación de VMs y/o LXCs
- Configuración de red
- Configuración de registros DNS / reglas de firewall / entradas en load balancers

Vagrant pero para proveedores de cloud

Terraform

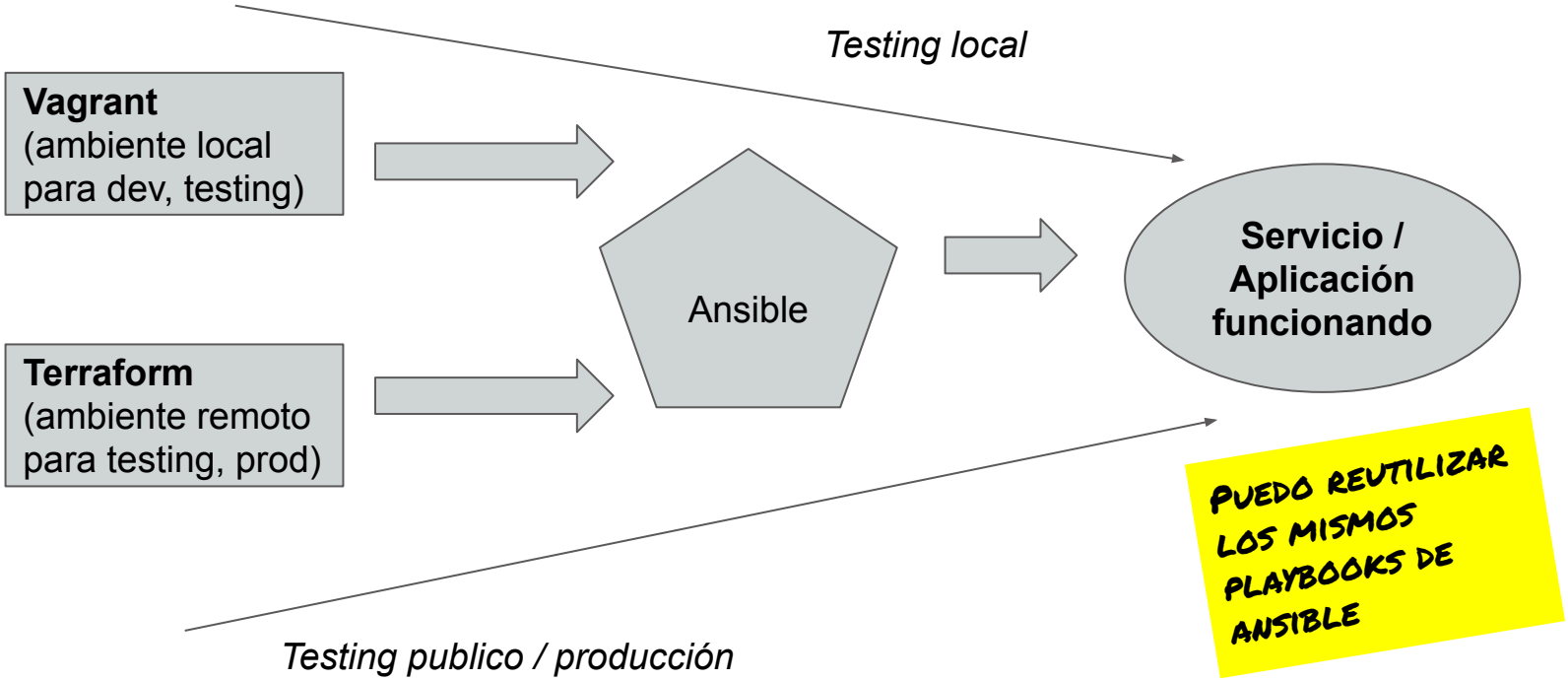
Providers:

- Proxmox
- Linode
  - Infra, DNS, Load balancing
- Amazon
- DNS dinámico

Provisioners:

- local-exec
  - Ejecución local de scripts
- Remote-exec
  - Ejecución remota de scripts
- file-copy
  - Copiar archivos locales a las instancias remotas

Terraform


Testing local
Vagrant
(ambiente local
para dev, testing)
Terraform
(ambiente remoto
para testing, prod)
Ansible
Servicio /
Aplicación
funcionando
PUEDO REUTILIZAR
LOS MISMOS
PLAYBOOKS DE
ANSIBLE
Testing publico / producción

Terraform

Un “*resource*” es un “*algo*” a crear/hacer:

- Creación y aprovisionamiento de VMs o CTs
- Entradas de DNS
- Copia de archivo de local al remoto
- Ejecución de un script remoto

El TF lleva cuenta del “estado” en el que está mi infraestructura

Terraform

Flujo básico:

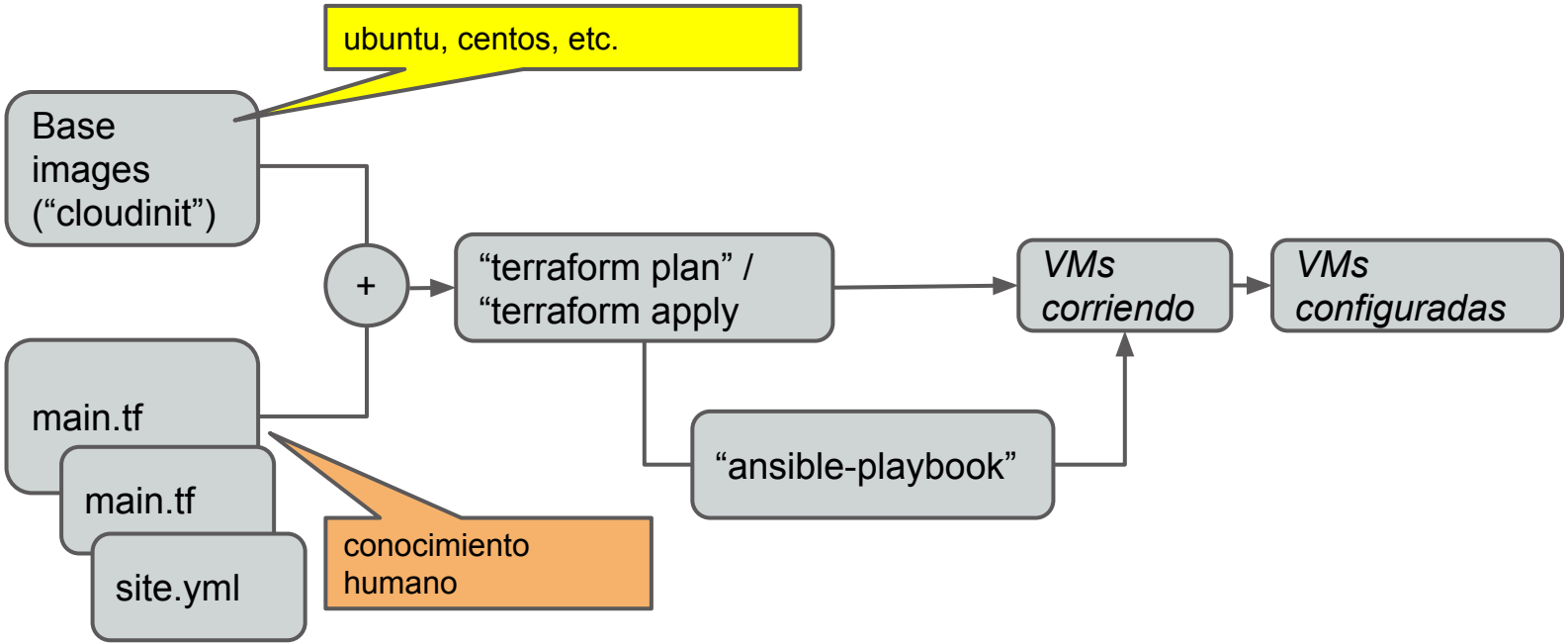
- Crear “main.tf” con declaraciones
- Correr “`terraform plan`”
- Correr “`terraform apply`”
- Correr “`terraform destroy`”

Terraform: Resources y Providers

Cada “provider” expone los “resources” que puede gestionar:

- Linode / DigitalOcean
  - Creación de VMs
  - Creación de registros DNS en zonas hosteadas en Linode
  - Creación y configuración de load balancers
  - … otros muchos ...

Terraform - Flujo de trabajo



Terraform

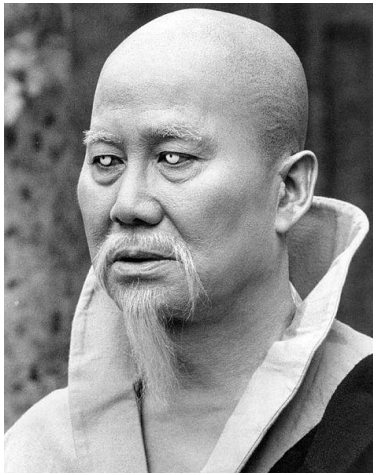
Ejemplo
(crear la vm)

```
provider “telmate/proxmox” “proxmox_vm” {

    clone “ubuntu2004-cloudinit”
    ssh_key SSADFASDFASDFE….
    disk {
        ...
    }
    cpu {
        …
    }
    ip 10.1.2.3

}
```

Terraform

<<Ejemplo: Kutt URL shortener>>

Lambdas



El camino de la virtualización

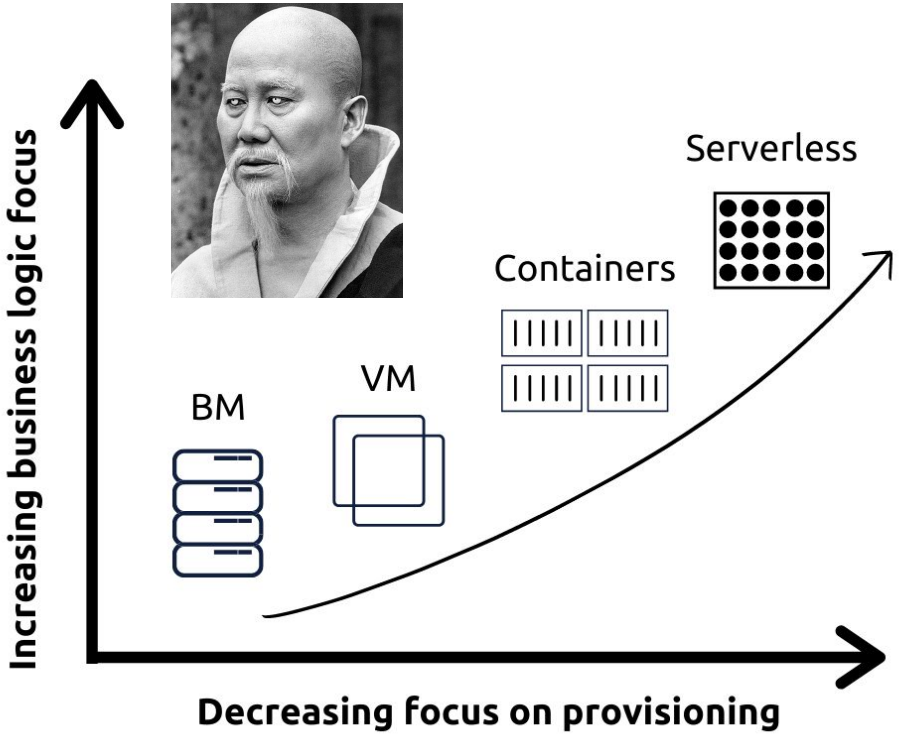
El camino de la miniaturización

- El server físico se partió en **máquinas virtuales**
- Las máquinas virtuales se partieron en **containers**
- Los containers se partieron** en **funciones**

El camino de la virtualización

¿Los containers se dividieron en funciones?

No exactamente, pero podemos pensarlo así.



Serverless

Los desarrolladores no se preocupan por la naturaleza de los recursos de cómputo.

Para ellos son *funciones*.

Cómo y dónde corren esas funciones deja de ser relevante.

Serverless (ii)

Para los *el equipo de infraestructura*, las *aplicaciones* que sus servidores soportan dejan de ser tan relevantes.

Son runtimes para las funciones. Todos los servers son más o menos iguales.

¿Que es una FaaS?

FaaS: Function as a Service

- Código que se ejecuta como respuesta a un evento
  - Evento: HTTP Request, un click en un link, la llegada de un paquete de red, el vencimiento de un timer
- Una FaaS no guarda estado y no tiene efectos secundarios** (o casi nunca). De ahí el nombre de *lambda*

¿Que es una FaaS? (ii)

FaaS: Function as a Service

- Una FaaS:
  - Tiene un tiempo de ejecución corto
  - No tiene demasiadas dependencias
- El *runtime:*
  - Se encarga de gestionar el inicio y el apagado
  - Soporta diferentes lenguajes para las funciones

Serverless - Ventajas

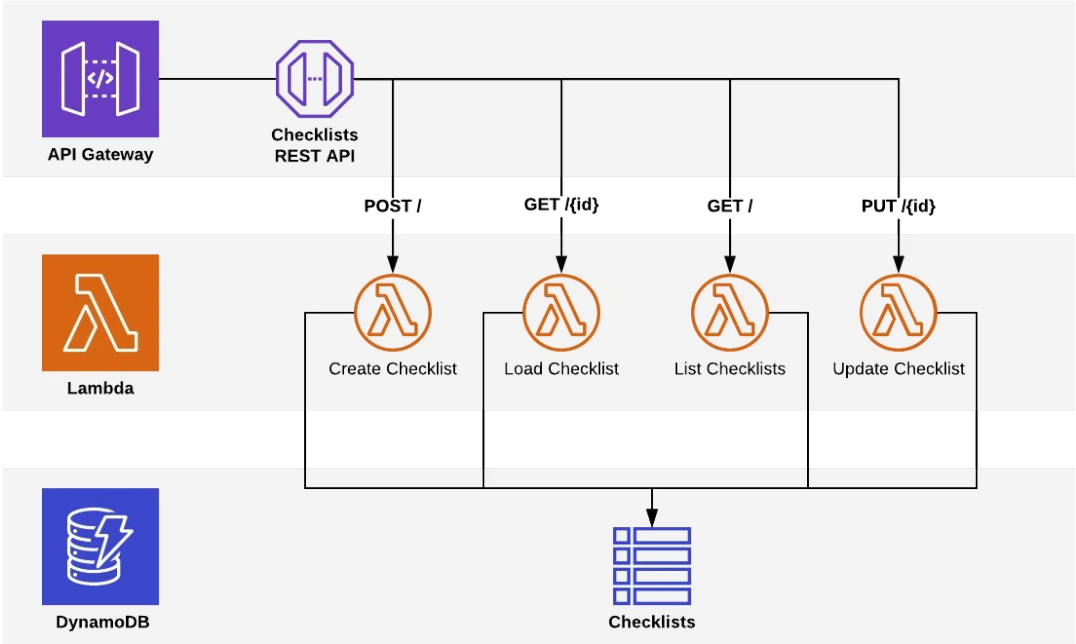
- Administración simplificada para los *equipos de infraestructura*
- Menos “carga mental” para los desarrolladores (?)
- Deploys mas ágiles

Serverless (ventajas)

- Escalabilidad granular

El runtime puede desde apagar todo a levantar cientos de instancias de cada función

Si cambio solo un método de una API, entonces solo tengo que hacer deploy de la función que la implementa



Serverless (DESventajas)

- Necesidad de monitorear contínuamente el estado de la infraestructura (si uno lo implementa onprem)
- Necesidad de monitorear continuamente *los costos* (si es en cloud)
- “Startup time” de las funciones
  - Cuando se van a dormir por inactividad, el primer evento puede sufrir una demora en su atención

Implementaciones

Cloud:

- Amazon Lambda
- Google:
  - Google Cloud Functions
  - Google Compute Engine

On premises:

- OpenFaas
- Apache OpenWhisk

OpenFaaS

“*Kubernetes para los que no quieren aprender kubernetes*”

OPENFAAS ABOUT BLOG EBOOKS CONSULTING DOCS TEAM SUPPORT

Simple, powerful functions from anywhere.

Whether you want to use Go, Java, Python, C#, Ruby - Express.js, Django, ASP.NET Core, even binaries like ffmpeg, ImageMagick, or anything else. We have you covered with the template store.

Node.js | Python | Go | Dockerfile | Bash | Java

```
$ faas-cli new --lang java11 java-fn
```

Handler.java

```
package com.openfaas.function;

import com.openfaas.model.IHandler;
import com.openfaas.model.IResponse;
import com.openfaas.model.IRequest;
import com.openfaas.model.Response;

public class Handler implements com.openfaas.model.IHandler {

    public IResponse Handle(IRequest req) {
        Response res = new Response();
        res.setBody("Hello, world!");

        return res;
    }
}
```

Apache OpenWhisk

[Link]

APACHE OpenWhisk

Documentation | Community | Downloads

Open Source Serverless Cloud Platform

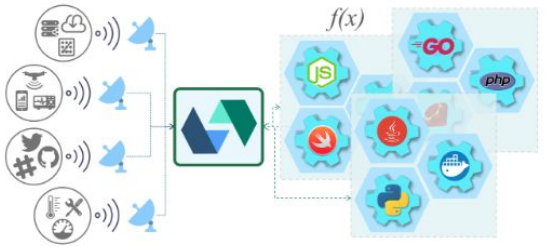
Executes functions in response to events at any scale

What is Apache OpenWhisk?

Apache OpenWhisk is an open source, distributed **Serverless** platform that executes functions (fx) in response to events at any scale. OpenWhisk manages the infrastructure, servers and scaling using Docker containers so you can focus on building amazing and efficient applications.

The OpenWhisk platform supports a programming model in which developers write functional logic (called **Actions**), in any supported programming language, that can be dynamically scheduled and run in response to associated events (via **Triggers**) from external sources (**Feeds**) or from HTTP requests. The project includes a REST API-based Command Line Interface (CLI) along with other tooling to support packaging, catalog services and many popular container deployment options.

Create Your Local Playground



Deploys anywhere

Since Apache OpenWhisk builds its components using containers it easily supports many deployment options both locally and within Cloud infrastructures. Options include many of today's popular Container frameworks such as **Kubernetes and OpenShift**, and **Compose**. In general, the community endorses deployment on Kubernetes using **Helm** charts since it provides many easy and convenient implementations for both Devlopers and Operators alike.

Serverless Demo

- Objetivo:
  - Desplegar tres funciones usando OpenFaaS (2 en python, una en XXX)
  - Runtime: faasd (OpenFaaS sobre containerd)
- Pasos:
  - Crear y compilar las funciones a sus containers
  - Subir los containers a un docker registry
  - Crear el ambiente de “producción” (Vagrant)
  - Deploy de las funciones
  - Testear las funciones

FIN (Por ahora)

¿Preguntas?