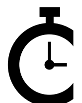
Casos de uso PaaS y de automatización completa

PID_00266616

Joan Caparrós Ramírez
Lorenzo Cubero Luque
Jordi Guijarro Olivares

Tiempo mínimo de dedicación recomendado: 5 horas



Universitat Oberta de Catalunya

Joan Caparrós Ramírez

Ingeniero superior en Informática por la UAB, máster en Seguridad de las TIC por la UOC y máster en Diseño y programación de aplicaciones móviles por La Salle. Desarrollador Fullstack web & mobile especializado en entornos de alto rendimiento y seguridad. Actualmente desarrolla funciones de técnico líder de proyectos en el Área de Cálculo y Aplicaciones en el Consorci de Serveis Universitaris de Catalunya (CSUC).

https://www.linkedin.com/in/joan-caparros

Lorenzo Cubero Luque

Ingeniero superior en Informática por la UPC y máster en Gestión de las tecnologías de la información por La Salle. Ha desarrollado proyectos de implantación de metodologías ágiles en equipos DevOps. Actualmente lidera el equipo responsable de los servicios TI de una multinacional suiza dedicada al marketing digital, Netcentric AG.

@lj_cubero

https://www.linkedin.com/in/lorenzocubero

Jordi Guijarro Olivares

Ingeniero en Informática por la UOC y máster en Gestión de tecnologías de la información por la URL. Coordinador de operaciones y ciberseguridad tecnológica en el CSUC, donde lidera las actividades técnicas en los ámbitos de servicios *cloud* y ciberseguridad del consorcio. También coordina el equipo de ciberseguridad CSUC-CSIRT de la red académica y de investigación catalana (Anella Científica), y colabora en proyectos a nivel europeo en grupos de trabajo como SIG-CISS y la TF-CSIRT de la red académica y de investigación europea GéANT.

@jordiguijarro

https://es.linkedin.com/in/jordiguijarro

La revisión de este recurso de aprendizaje UOC ha sido coordinada por el profesor: Josep Jorba Esteve (2019)

Segunda edición: septiembre 2019
© Joan Caparrós Ramírez, Lorenzo Cubero Luque, Jordi Guijarro Olivares
Todos los derechos reservados
© de esta edición, FUOC, 2019
Av. Tibidabo, 39-43, 08035 Barcelona
Realización editorial: FUOC

Ninguna parte de esta publicación, incluido el diseño general y la cubierta, puede ser copiada, reproducida, almacenada o transmitida de ninguna forma, ni por ningún medio, sea éste eléctrico, químico, mecánico, óptico, grabación, fotocopia, o cualquier otro, sin la previa autorización escrita de los titulares del copyright.

Índice

Objetivos.......... 5

1. Caso de uso 1: Docker Machine y OpenNebula.......... 7
1.1. Introducción a la Docker-Machine 7
1.2. Instalar docker-machine 8
1.3. Instalar el *driver* para OpenNebula de docker-machine 8
1.3.1. Preparación del sistema 8
1.3.2. Instalación del *driver*.......... 9
1.4. Instalar docker-engine 9
1.5. Cómo utilizar el *driver* de OpenNebula con docker-machine 10
1.5.1. Crear nuestro Docker Engine 10
1.5.2. Enlazar la *shell* local con la de la máquina virtual 10
1.6. Ejemplo de uso 11

2. Caso de uso 2: Desplegar un clúster de Kubernetes en AWS con RKE.......... 13
2.1. Introducción a Kubernetes 13
2.2. Crear la infraestructura desde la consola de AWS 14
2.3. Automatización: infraestructura como código 24
2.3.1. Terraform 24
2.3.2. *Workflow* de Terraform 25
2.4. Desplegar Kubernetes con RKE 30
2.4.1. Instalación de RKE 31
2.5. Administrar nuestro clúster de Kubernetes 35
2.5.1. Administrar Kubernetes desde el Dashboard 37

3. Herramientas de automatización completa y *testing*: Jenkins.......... 42
3.1. Servlet Demo 43
3.2. Ejecución de test del proyecto 44
3.3. Construcción del proyecto 46
3.3.1. Despliegue del proyecto 46
3.3.2. Página inicial 46
3.3.3. Página resultado 46
3.4. Dockerización del proyecto 47
3.4.1. Construcción de la imagen docker 47
3.4.2. Fichero Dockerfile 50
3.4.3. Ejecución de la aplicación mediante contenedor Docker 51
3.4.4. Detención de la aplicación mediante contenedor Docker 52

3.5. Jenkins 52
3.5.1. Instalación mediante imagen dockerizada 52
3.5.2. Instalación mediante fichero WAR oficial 53
3.5.3. Configuración de Jenkins 53
3.6. Registry Docker Hub 57
3.6.1. Registro mediante imagen dockerizada 58
3.6.2. Contribuir con nuestra imagen a la comunidad 60
3.7. Proceso de automatización completo (*pipeline*) 61
3.7.1. Compilar el código cada vez que este sufra un cambio 63
3.7.2. Construcción y publicación de la imagen 65
3.7.3. Notificación de errores durante el proceso de automatización 67

Objetivos

Introducir al estudiante varios casos de uso 100 % prácticos sobre escenarios reales con aplicación directa de los conocimientos cubiertos en el módulo didáctico «Infraestructura DevOps».

1. Caso de uso 1: Docker Machine y OpenNebula

Contenidos:

- Instalar docker-machine
- Instalar el *driver* para OpenNebula de docker-machine
  - Preparación del sistema
  - Instalación del *driver*

- Instalar docker-engine
- Cómo utilizar el *driver* de OpenNebula con docker-machine:
  - Crear nuestro Docker Engine
  - Enlazar la *shell* local con la de la máquina virtual

- Ejemplo de uso

Para poder crear máquinas Docker Engine en la OpenNebula podemos usar el *driver* docker-machine; una vez instalado, podremos crear e instanciar nuestros contenedores de Docker.

Requerimientos:

- **Docker Engine**: Nos permite crear los contenedores, gestionarlos e instanciarlos.

- **Docker Machine**: Nos permite crear y gestionar las máquinas virtuales.

- ***Driver* Docker Machine OpenNebula**: Nos permite utilizar el docker-machine en el *cloud* del OpenNebula.

1.1. Introducción a la Docker-Machine

Con docker-machine podemos aprovisionar y administrar múltiples Dockers remotos, además de aprovisionar clústeres Swarm en entornos Windows, Mac y Linux.

Es una herramienta que nos permite instalar el demonio Docker en *hosts* virtuales y administrar dichos *hosts* con el comando docker-machine. Además, podemos hacerlo en distintos proveedores (VirtualBox, OpenStack, OpenNebula, VmWare, etc.).

Usando el comando docker-machine podemos iniciar, inspeccionar, parar y reiniciar los *hosts* administrados, actualizar el cliente y demonio Docker, y configurar un cliente Docker para que interactúe con el *host*. Con el cliente podemos correr dicho comando directamente en el *host*.

1.2. Instalar docker-machine

Como usuario **privilegiado**, nos descargamos el binario y ejecutamos:

```
curl -L https://github.com/docker/machine/releases/download/v0.6.0/docker-machine-`uname -s`-
`uname -m` > /usr/local/bin/docker-machine && \
chmod +x /usr/local/bin/docker-machine
```

Una vez instalado, para comprobar que podéis ejecutar docker-machine, ejecutad desde vuestro usuario:

```
docker-machine version
```

Si os dijera que no encuentra el binario, ejecutad:

```
export PATH=$PATH:/usr/local/bin
```

Enlace de interés

Podéis encontrar más información en: <https://docs.docker.com/machine/install-machine/>.

1.3. Instalar el *driver* para OpenNebula de docker-machine

Antes de instalar el *driver*, tenemos que preparar el equipo para la instalación.

1.3.1. Preparación del sistema

1) Paquete GO

```
wget https://storage.googleapis.com/golang/go1.6.linux-amd64.tar.gz
tar -C /usr/local -xvzf go1.6.linux-amd64.tar.gz
export GOROOT=/usr/local/go
export PATH=$PATH:$GOROOT/bin
```

Creamos el directorio work en nuestro espacio de trabajo, por ejemplo, en el $HOME:

```
mkdir $HOME/work
```

Indicamos las variables de entorno necesarias:

```
export GOPATH=$HOME/work
```

```
export PATH=$PATH:$GOPATH/bin
```

2) Paquete GIT, BZR

```
sudo apt-get install git bzr
```

3) Paquete GODEP

```
go get github.com/tools/godep
```

1.3.2. Instalación del *driver*

Le damos la siguiente orden:

```
go get github.com/opennebula/docker-machine-opennebula
cd $GOPATH/src/github.com/opennebula/docker-machine-opennebula
make build
make install
```

1) ONE_AUTH y ONE_XMLRPC

Nos creamos un fichero donde indicaremos nuestro usuario y contraseña del OpenNebula:

```
mkdir -p $HOME/.one
echo usuario:password > $HOME/.one/one_auth
```

Creamos las variables de entorno necesarias:

```
export ONE_AUTH=$HOME/.one/one_auth
export ONE_XMLRPC=http://iaas.csuc.cat:2633/RPC2
```

Enlace de interés

Podéis encontrar más información en: <https://github.com/opennebula/docker-machine-opennebula>.

1.4. Instalar docker-engine

Para instalar el docker-engine, se recomienda visitar la página siguiente: <https://docs.docker.com/engine/installation/linux/>.

1.5. Cómo utilizar el *driver* de OpenNebula con docker-machine

1.5.1. Crear nuestro Docker Engine

Existe una imagen en la OpenNebula de un sistema operativo donde ya está instalado el docker y donde podemos lanzar contenedores; se llama boot2docker. También existe una versión de Ubuntu.

Si no está disponible, desde el *marketplace* del OpenNebula la descargamos. Una vez tenemos descargada la imagen, creamos la máquina virtual:

```
docker-machine create --driver opennebula --opennebula-image-id [num_imagen_boot2docker]
--opennebula-network-id [num_red] --opennebula-b2d-size [medida_en_MB] [nombre_máquina_virtual]
```

Más opciones:

- --opennebula-cpu: CPU value for the VM.
- --opennebula-vcpu: VCPUs for the VM.
- --opennebula-memory: Size of memory for VM in MB.
- --opennebula-template-id: Template ID to use.
- --opennebula-template-name: Template to use.
- --opennebula-network-id: Network ID to connect the machine to.
- --opennebula-network-name: Network to connect the machine to.
- --opennebula-network-owner: User ID of the Network to connect the machine to.
- --opennebula-image-id: Image ID to use as the OS.
- --opennebula-image-name: Image to use as the OS.
- --opennebula-image-owner: Owner of the image to use as the OS.
- --opennebula-dev-prefixe: Dev prefixe to use for the images: 'vd', 'sd', 'hd', etc.
- --opennebula-disk-resize: Size of disk for VM in MB.
- --opennebula-b2d-size: Size of the Volatile disk in MB (only for b2d).
- --opennebula-ssh-user: Set the name of the SSH user.
- --opennebula-disable-vnc: VNC is enabled by default. Disable it with this flag.

1.5.2. Enlazar la *shell* local con la de la máquina virtual

Ahora conectaremos nuestra *shell* local a la de la máquina virtual para que cada vez que ejecutemos docker, este pueda dar órdenes al docker instalado en la máquina virtual.

Obtenemos las variables de entorno en la máquina virtual:

```
docker-machine env [nombre_máquina_virtual]
```

Enlazamos la *shell*:

```
eval "$(docker-machine env [nombre_máquina_virtual])"
```

Ahora ya podemos utilizar docker normalmente.

Variables de entorno

Para mantener las nuevas variables de entorno de manera persistente, añadidlas a vuestro .profile ($HOME/.profile).

1.6. Ejemplo de uso

Actualmente disponéis de dos imágenes, ya precreadas con el Docker Engine instalado, que podéis usar: Boot2docker y Docker-Machine-Ubuntu.

Crear un contenedor con nginx en una máquina Docker con el *driver* de Docker Machine en el OpenNebula:

Crear una máquina en el OpenNebula con el Docker Engine y un disco volátil de 10 GB con el comando:

```
docker-machine create --driver opennebula --opennebula-network-id $NETWORK_ID --opennebula-image
-name boot2docker --opennebula-image-owner oneadmin --opennebula-b2d-size 10240 mydockerengine
```

Donde $NETWORK_ID será el ID de la red donde crearemos la máquina con Docker Engine.

Figura 1. Detalle del *dashboard* de OpenNebula

Fuente: Jordi Guijarro.

Podemos comprobar que todo ha funcionado bien con el comando:

```
docker-machine ls
```

Los siguientes comandos nos muestran las variables de entorno necesarias para acceder a la *shell*:

```
docker-machine env mydockerengine
eval $(docker-machine env mydockerengine)
```

Ahora podemos comenzar a usar docker sobre la máquina Docker Engine que acabamos de crear:

```
docker pull nginx
docker run --name mynginx -d -p 80:80 nginx
```

Finalmente, comprobamos que podemos acceder al servidor web que acabamos de crear.

2. Caso de uso 2: Desplegar un clúster de Kubernetes en AWS con RKE

2.1. Introducción a Kubernetes

Kubernetes (K8S), según la wikipedia, se define como un sistema *open source* para la automatización de despliegues, el escalado y la gestión de aplicaciones en contenedores, fue originalmente diseñado por Google y donado a la Cloud Native Computing Foundation (parte de la Linux Foundation), y está escrito en Golang. Se puede desplegar en múltiples entornos *cloud* o en bare-metal y soporta múltiples *runtimes* de contenedores.

Algunas de sus principales características son:

- **Escalado y autoescalado**: en función del uso de CPU permite el escalado vertical de las aplicaciones de manera automática o manual.

- **Descubrimiento de servicios y balanceo de carga**: no es necesario utilizar un mecanismo externo para el descubrimiento de servicios, ya que Kubernetes asigna a los contenedores sus propias direcciones IP y un nombre DNS único para un conjunto de contenedores y puede balancear la carga sobre ellos.

- **Despliegues y *rollbacks* automáticos**: cuando hay que actualizar una aplicación o cambiar su configuración, Kubernetes despliega los cambios de forma progresiva mientras monitoriza su salud para asegurar que no mata todas las instancias a la vez, y en caso de fallo, hace un *rollback* automático.

- **Planificación**: se encarga de decidir en qué nodo se ejecutará cada contenedor de acuerdo a los recursos que requiera y a otras restricciones.

- **Orquestación del almacenamiento**: puede montar automáticamente el sistema de almacenamiento necesario, ya sea almacenamiento local, almacenamiento en un proveedor de *cloud* público (como GCP o AWS), o incluso un sistema de almacenamiento de red como NFS, SCSI, Gluster, Ceph, Cinder o Flocker.

A continuación se muestra como desplegar un nuevo clúster de Kubernetes en una infraestructura de nube pública como AWS.

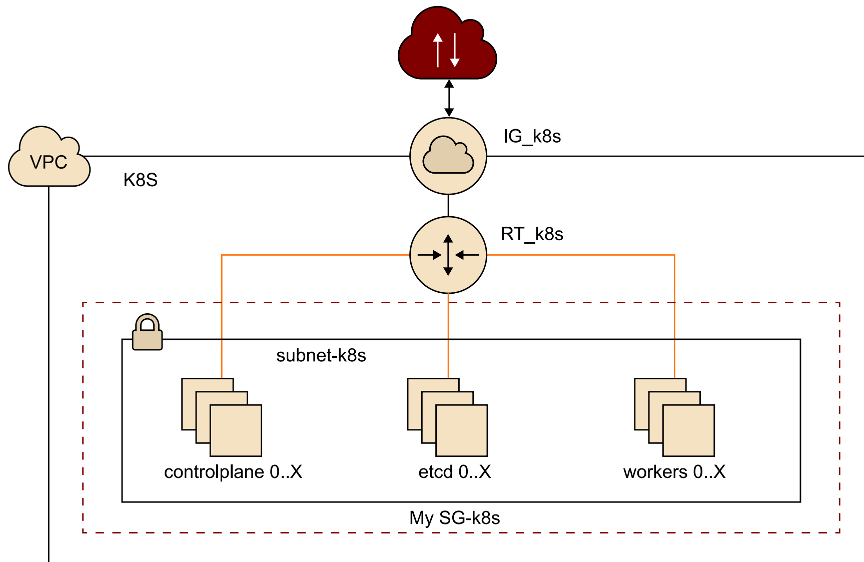
Enlace de interés

Para ampliar la información sobre Kubernetes y sus características, podéis consultar el siguiente enlace: <https://www.paradigmadigital.com/techbiz/por-que-todos-apuestan-por-kubernetes/>

2.2. Crear la infraestructura desde la consola de AWS

El primer paso será crear toda la infraestructura necesaria en AWS. En la siguiente imagen se muestra un esquema completo de la arquitectura que se desplegará:

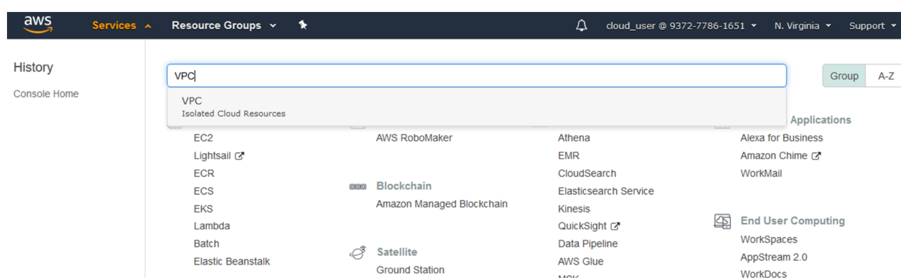
Figura 2



El primer elemento que necesitamos crear es nuestro Virtual Private Cloud (VPC) en terminología de AWS, que será nuestro «centro de datos» virtual; como necesitaremos acceder por SSH a las máquinas para configurar Kubernetes también será necesario configurar una subred, una puerta de enlace a Internet y una tabla de rutas en nuestro VPC.

Podemos utilizar la barra de búsqueda de AWS para buscar los recursos que deberemos ir creando:

Figura 3



Para crear un nuevo VPC, tan solo necesitamos proporcionar el rango de IPs (CIDR) que utilizaremos para las instancias, por ejemplo: 172.35.0.0/24. Para identificar más fácilmente nuestro VPC le hemos añadido el *tag* **k8s**.

Figura 4

VPCs > Create VPC

Create VPC

A VPC is an isolated portion of the AWS cloud populated by AWS objects, such as Amazon EC2 instances. You must s
Inter-Domain Routing (CIDR) block; for example, 10.0.0.0/16. You cannot specify an IPv4 CIDR block larger than /16.

Name tag: k8s

IPv4 CIDR block*: 172.35.0.0/24

IPv6 CIDR block: No IPv6 CIDR Block / Amazon provided IPv6 CIDR block

Tenancy: Default

* Required

Una vez completada la creación del VPC, ya podemos continuar con la creación de la subnet para nuestras instancias de EC2:

Figura 5

VPC Dashboard
Filter by VPC:
Select a VPC
Virtual Private Cloud
Your VPCs

Create subnet | Actions

Filter by tags and attributes or search by keyword

Name	Subnet ID	State	VPC	IPv4 CIDR
	subnet-01c4c8b822f59068c	available	vpc-01bfe5ad89270e1fa |...	10.0.0.0/24
	subnet-0b9ca24334c0962b1	available	vpc-01bfe5ad89270e1fa |...	10.0.1.0/24

En nuestro caso, podemos utilizar todo el rango asignado al VPC; nuevamente hemos añadido el *tag* **subnet-k8s**, para facilitar la identificación de la subnet.

Figura 6

Subnets > Create subnet

Create subnet

Specify your subnet's IP address block in CIDR format; for example, 10.0.0.0/24. IPv4 block sizes must be be
must be a /64 CIDR block.

Name tag: subnet-k8s

VPC*: vpc-0268b4f099e8f3cc4

VPC CIDRs:

CIDR	Status
172.35.0.0/24	associated

Availability Zone: No preference

IPv4 CIDR block*: 172.35.0.0/24

Modificaremos nuestra subnet, para que también autoasigne una IP pública a cada instancia.

Figura 7

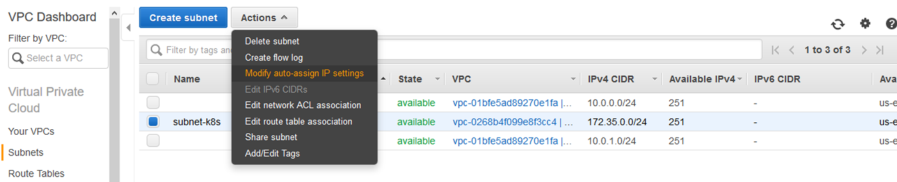


Figura 8

Subnets > Modify auto-assign IP settings

Modify auto-assign IP settings

Enable the auto-assign IP address setting to automatically request a public IPv4 or IPv6 address for a
time.

Subnet ID subnet-08ffef600976e90f3

Auto-assign IPv4 ☑ Enable auto-assign public IPv4 address

Así podremos acceder por SSH a nuestras instancias y configurar Kubernetes con RKE.

El siguiente paso será asignar una puerta de enlace a Internet (Internet Gateway) a nuestro VPC, así nuestras instancias de EC2 podrán tener salida a Internet para, por ejemplo, instalar nuevos paquetes o actualizarse.

Figura 9

Para la creación de un Internet Gateway simplemente tenemos que proporcionarle un *tag*, por ejemplo, **IG-k8s**.

Figura 10

Internet gateways > Create internet gateway

Create internet gateway

An internet gateway is a virtual router that connects a VPC to the internet. To create a new internet gateway specify the name for the gateway below.

Name tag IG-k8s

Una vez creado, tenemos que asociar este Internet Gateway a nuestro VPC, a través de la opción *Attach to VPC*.

Figura 11

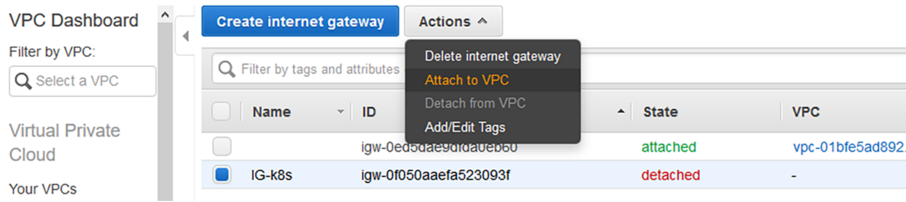


Figura 12

Internet gateways > Attach to VPC

Attach to VPC

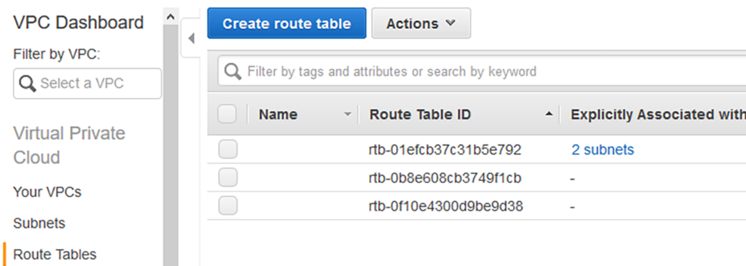
Attach an internet gateway to a VPC to enable communication with the internet. Specify the VPC you would like to attach below.

VPC* vpc-0268b4f099e8f3cc4

AWS Command Line Interface command

El siguiente paso será crear una tabla de rutas o *Route Table* en terminología de AWS para nuestro VPC.

Figura 13



Como hemos hecho anteriormente con otros recursos, le asignamos un *tag* para facilitar su identificación, en este caso, **RT-k8s**:

Figura 14

Route Tables > Create route table

Create route table

A route table specifies how packets are forwarded between the subnets within your VPC, the internet, and your VPN connection.

Name tag RT-k8s

VPC* vpc-0268b4f099e8f3cc4

Y asociaremos nuestra subnet a la Route Table que acabamos de crear:

Figura 15

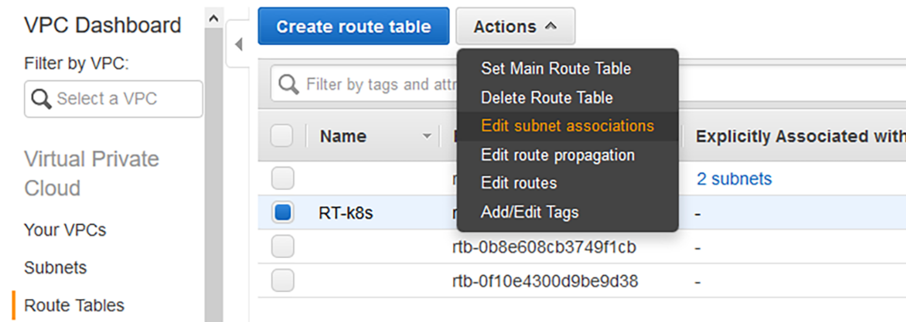
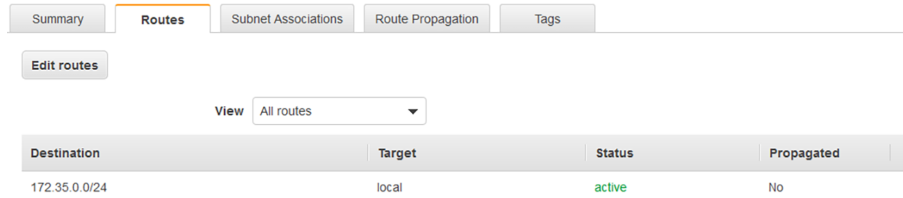


Figura 16

Una vez hayamos asociado la subnet a nuestra Route Table, deberemos añadir una ruta por defecto para la salida Internet a través del Internet Gateway que hemos creado previamente. Para ello hacemos clic en *Edit Routes*:

Figura 17



Y añadimos la siguiente ruta para el destino 0.0.0.0/0:

Figura 18

Nuestra tabla de rutas debería quedar de la siguiente manera:

Figura 19

Summary | Routes | Subnet Associations | Route Propagation | Tags

Edit routes

View: All routes

Destination	Target	Status	Propagated
172.35.0.0/24	local	active	No
0.0.0.0/0	igw-0f050aaefa523093f	active	No

También debemos revisar que nuestro VPC tenga habilitada la resolución DNS:

Figura 20

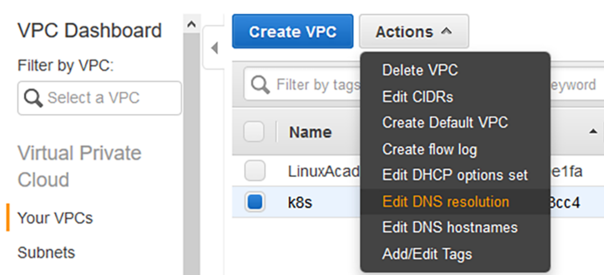


Figura 21

VPCs > Edit DNS resolution

Edit DNS resolution

VPC ID vpc-0268b4f099e8f3cc4

DNS resolution ☑ enable

Figura 22

VPCs > Edit DNS hostnames

Edit DNS hostnames

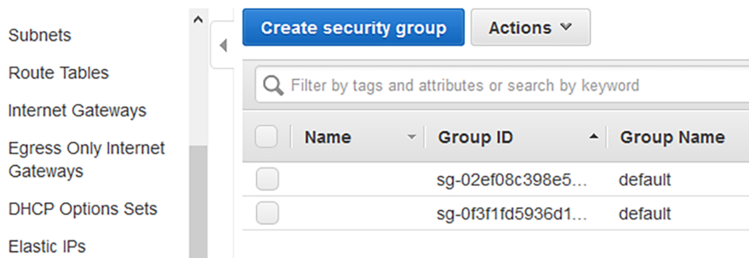
VPC ID vpc-0893579b4792183f1

DNS hostnames ☑ enable

Para finalizar con la parte relacionada con nuestro VPC, crearemos un nuevo Security Group con las siguientes reglas:

- Habilitaremos el SSH para acceder desde cualquier destino a nuestras instancias de EC2.
- Habilitaremos el puerto 6443 para la API de Kubernetes desde cualquier destino.
- Permitiremos todo el tráfico entre nuestras instancias de EC2.

Figura 23



Simplemente debemos asignarle un nombre, una descripción y asociarlo a nuestro VPC:

Figura 24

Security Groups > Create security group

Create security group

A security group acts as a virtual firewall for your instance to control inbound and outbound traffic. To create a new security group fill in the fields below.

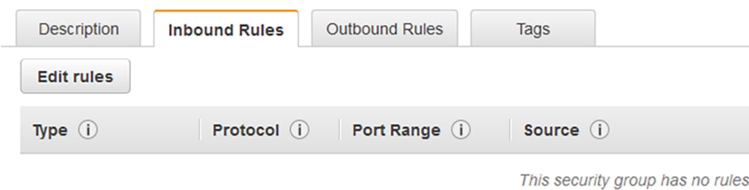
Security group name* MySG-k8s

Description* MySG-k8s

VPC vpc-0268b4f099e8f3cc4

Una vez creado el Security Group, debemos añadir las reglas anteriormente mencionadas mediante el botón *Edit Rules*:

Figura 25



La siguiente imagen muestra cómo debería quedar el Security Group una vez añadidas las reglas:

Figura 26

Security Groups > Edit inbound rules

Edit inbound rules

Inbound rules control the incoming traffic that's allowed to reach the instance.

Type	Protocol	Port Range	Source	
All traffic	All	All	Custom	sg-0c0ec8ccbab03c281
SSH	TCP	22	Custom	0.0.0.0/0
SSH	TCP	22	Custom	::/0
Custom TCP Rule	TCP	6443	Custom	0.0.0.0/0

Add Rule

Bien, ahora ya podemos crear las instancias EC2. En nuestro caso, crearemos tres instancias, cada una de ellas con un rol diferente. Una vez que estemos en el panel de EC2, simplemente hacemos clic sobre el botón *Launch Instance* que nos mostrará un **Wizard** para ir definiendo, paso a paso, los parámetros de nuestra instancia:

Figura 27

Create Instance

To start using Amazon EC2 you will want to launch a virtual server, known as an Amazon EC2 instance.

Launch Instance

Seleccionaremos la AMI de **Ubuntu Server 16.04 LTS**:

Figura 28

Step 1: Choose an Amazon Machine Image (AMI) Cancel and Exit

Ubuntu Server 16.04 LTS (HVM), SSD Volume Type - ami-0f9cf087c1f27d9b1 (64-bit x86) / ami-036ede09922dadc9b (64-bit Arm)

Free tier eligible

Ubuntu Server 16.04 LTS (HVM),EBS General Purpose (SSD) Volume Type. Support available from Canonical (http://www.ubuntu.com/cloud/services).

Root device type: ebs Virtualization type: hvm ENA Enabled: Yes

Select

64-bit (x86)
64-bit (Arm)

Para este laboratorio, podemos elegir como tipo de instancia: **t2.micro**

Figura 29

	Family	Type	vCPUs	Memory (GiB)	Instance Storage (GB)
	General purpose	t2.nano	1	0.5	EBS only
■	General purpose	t2.micro Free tier eligible	1	1	EBS only
	General purpose	t2.small	1	2	EBS only
	General purpose	t2.medium	2	4	EBS only
	General purpose	t2.large	2	8	EBS only

Cancel Pr

En el apartado de configuración de la instancia seleccionaremos que queremos crear **3 instancias** en el VPC y la subnet creada anteriormente.

Figura 30

Number of instances: 3 Launch into Auto Scaling Group

You may want to consider launching these instances into an Auto Sc future. Learn how Auto Scaling can help your application stay healt

Purchasing option: Request Spot instances

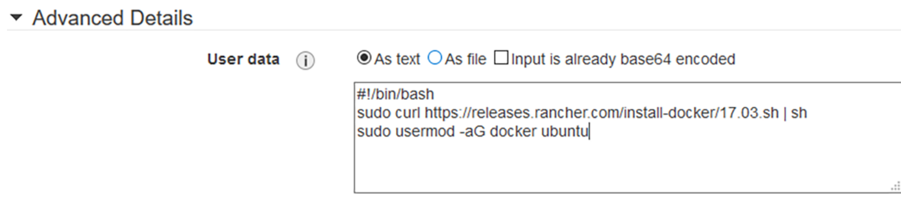
Network: vpc-0268b4f099e8f3cc4 | k8s Create n
No default VPC found. Create a new default VPC.

Subnet: subnet-08ffef600976e90f3 | subnet-k8s | us-east-1d Create n

Desplegaremos el apartado *Advanced Details*, y en el campo **User Data** pondremos el siguiente código como *script* de inicio:

```
#!/bin/bash
sudo curl https://releases.rancher.com/install-docker/17.03.sh | sh
sudo usermod -aG docker ubuntu
```

Figura 31



Este *script* se ejecutará cuando se inicie la instancia, de esta forma conseguiremos instalar ***docker*** en nuestra instancia. La instalación de ***docker*** es uno de requisitos previos que debe cumplir la máquina para poder ejecutar correctamente RKE e instalar Kubernetes.

Por último, asignamos el Security Group que ya habíamos creado previamente para las instancias.

Figura 32

Step 6: Configure Security Group

A security group is a set of firewall rules that control the traffic for your instance. On this page, you can add rules to allow specific traffic to reach your instance. For example, if you want to set up a web server and allow Internet traffic to reach your instance, add rules that allow unrestricted access to the HTTP and HTTPS ports. You can create a new security group or select from an existing one below. Learn more about Amazon EC2 security groups.

Assign a security group: Create a **new** security group

Select an **existing** security group

	Security Group ID	Name	Description	Actions
☐	sg-02ef08c398e544421	default	default VPC security group	Copy to new
■	sg-0c0ec8ccbab03c281	MySG-k8s	MySG-k8s	Copy to new

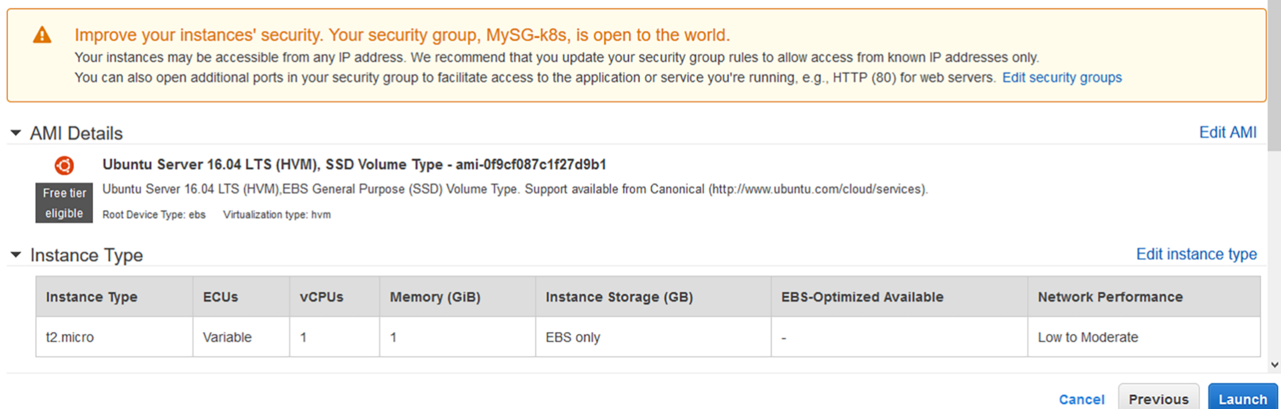
Inbound rules for sg-0c0ec8ccbab03c281 (Selected security groups: sg-0c0ec8ccbab03c281)

Type	Protocol	Port Range	Source	Description
All traffic	All	All	sg-0c0ec8ccbab03c281 (MySG-k8s)	
SSH	TCP	22	0.0.0.0/0	

Cancel Previous Review and Launch

Ahora simplemente debemos revisar que toda la configuración sea correcta y lanzar las instancias a través del botón *Launch*.

Figura 33



Para acceder posteriormente a las máquinas es necesario crear un par de claves SSH o usar una ya existente. Si creamos una nueva, es importante acordarse de descargar las claves antes de la lanzar la instancia.

Figura 34

Select an existing key pair or create a new key pair ✕

A key pair consists of a **public key** that AWS stores, and a **private key file** that you store. Together, they allow you to connect to your instance securely. For Windows AMIs, the private key file is required to obtain the password used to log into your instance. For Linux AMIs, the private key file allows you to securely SSH into your instance.

Note: The selected key pair will be added to the set of keys authorized for this instance. Learn more about removing existing key pairs from a public AMI .

Create a new key pair

Key pair name

k8s

Download Key Pair

You have to download the **private key file** (*.pem file) before you can continue. **Store it in a secure and accessible location.** You will not be able to download the file again after it's created.

Cancel Launch Instances

Si todo ha funcionado correctamente, en unos minutos nuestras tres instancias pasarán a estar en estado Running.

Figura 35

2.3. Automatización: infraestructura como código

Crear nuestra infraestructura a través de la consola de AWS puede ser interesante para familiarizarnos con los conceptos de AWS, pero en la práctica debemos buscar un modo para automatizar todo el proceso y ser mucho más ágiles creando la infraestructura, por ejemplo a través de la infraestructura como código o (IaC), que es una de las prácticas esenciales promovidas por la cultura DevOps.

El término «infraestructura como código» se refiere a la práctica de utilizar programación para configurar la infraestructura. Esta práctica trata la configuración de la infraestructura como si de software de programación se tratase.

La infraestructura como código permite a las instancias gestionarse de manera programada. Esto hace que la infraestructura sea «elástica», es decir, escalable y replicable. Una de las posibles herramientas de IaC que podemos utilizar para desplegar infraestructura es: Terraform.

2.3.1. Terraform

Terraform es una herramienta declarativa de aprovisionamiento que se basa en el paradigma infraestructura como código. Terraform es una herramienta de composición multipropósito: compone múltiples niveles (SaaS / PaaS / IaaS).

Instalación de Terraform

Para instalar Terraform, basta con descargar el paquete apropiado para tu sistema:

```
$ wget https://releases.hashicorp.com/terraform/0.11.11/terraform_0.11.11_linux_amd64.zip
$ unzip terraform_0.11.11_linux_amd64.zip
```

```
$ mv terraform /usr/local/bin
```

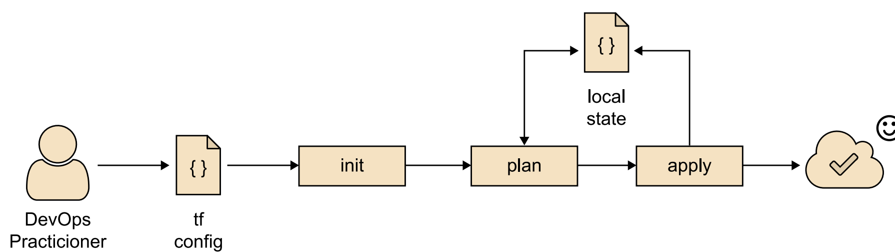
Podemos verificar que la instalación funciona, consultando la versión de Terraform:

```
$ terraform --version
```

2.3.2. *Workflow* de Terraform

La siguiente imagen muestra cuál sería la secuencia de etapas que seguiría un despliegue con Terraform:

Figura 36



Mediante ficheros de configuración que generamos en la sintaxis propuesta por Terraform (muy cercana al lenguaje natural), modelamos las necesidades de nuestra aplicación.

Normalmente usaremos un fichero para definir nuestra arquitectura, una serie de variables para poder parametrizar nuestro despliegue y algunas variables de salida para obtener los datos más importantes nuestra infraestructura una vez desplegada (por ejemplo, la IP de nuestras instancias de EC2).

En primer lugar, el comando terraform init revisará la sintaxis de nuestra configuración e instalará los *plugins* de los «*provider*» necesarios para el despliegue:

Figura 37

```
[root@node templates]# ls
init.tpl
[root@node templates]# cd ..
[root@node terraform-aws-ks8]# ls
main.tf  templates  variables.tf
[root@node terraform-aws-ks8]# terraform init

Initializing provider plugins...
- Checking for available provider plugins on https://releases.hashicorp.com...
- Downloading plugin for provider "aws" (1.58.0)...
- Downloading plugin for provider "template" (2.0.0)...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.aws: version = "~> 1.58"
* provider.template: version = "~> 2.0"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
[root@node terraform-aws-ks8]#
```

Una vez codificada nuestra infraestructura podemos hacer uso del comando terraform apply y comenzará el despliegue de nuestra infraestructura.

Figura 38

```
      cidr_blocks.0:                         "0.0.0.0/0"
      from_port:                             "22"
      protocol:                              "tcp"
      security_group_id:                     "${aws_security_group.kubernetes.id}"
      self:                                  "false"
      source_security_group_id:              <computed>
      to_port:                               "22"
      type:                                  "ingress"

  + aws_subnet.kubernetes
      id:                                    <computed>
      arn:                                   <computed>
      assign_ipv6_address_on_creation:       "false"
      availability_zone:                     <computed>
      availability_zone_id:                  <computed>
      cidr_block:                            "172.35.0.0/24"
      ipv6_cidr_block:                       <computed>
      ipv6_cidr_block_association_id:        <computed>
      map_public_ip_on_launch:               "true"
      owner_id:                              <computed>
      vpc_id:                                "${aws_vpc.vpc.id}"

  + aws_vpc.vpc
      id:                                    <computed>
      arn:                                   <computed>
      assign_generated_ipv6_cidr_block:      "false"
      cidr_block:                            "172.35.0.0/24"
      default_network_acl_id:                <computed>
      default_route_table_id:                <computed>
      default_security_group_id:             <computed>
      dhcp_options_id:                       <computed>
      enable_classiclink:                    <computed>
      enable_classiclink_dns_support:        <computed>
      enable_dns_hostnames:                  "true"
      enable_dns_support:                    "true"
      instance_tenancy:                      "default"
      ipv6_association_id:                   <computed>
      ipv6_cidr_block:                       <computed>
      main_route_table_id:                   <computed>
      owner_id:                              <computed>


Plan: 13 to add, 0 to change, 0 to destroy.
```

A continuación vamos a mostrar el mismo ejemplo creado anteriormente en el proveedor AWS por medio de su consola web, ahora utilizando Terraform; para ello generamos un fichero llamado main.tf:

```
provider "aws" {
region   = "${var.aws_region}"
access_key = "${var.aws_access_key}"
secret_key = "${var.aws_secret_key}"
}

# Create a VPC
resource "aws_vpc" "vpc" {
cidr_block = "${var.aws_vpc_cidr}"
enable_dns_support = true
enable_dns_hostnames = true
}

# Create a subnet with auto-assign public ip addresses
resource "aws_subnet" "kubernetes" {
vpc_id = "${aws_vpc.vpc.id}"
cidr_block = "${var.aws_subnet_cidr}"
map_public_ip_on_launch = true
}

# Create an Internet Gateway
resource "aws_internet_gateway" "gw" {
vpc_id = "${aws_vpc.vpc.id}"
}

# Create a Route Table on the VPC and add a route to the Internet
resource "aws_route_table" "kubernetes" {
vpc_id = "${aws_vpc.vpc.id}"
route {
cidr_block = "0.0.0.0/0"
gateway_id = "${aws_internet_gateway.gw.id}"
}
}

# Associate the route table with the Subnet
resource "aws_route_table_association" "kubernetes" {
subnet_id = "${aws_subnet.kubernetes.id}"
route_table_id = "${aws_route_table.kubernetes.id}"
}

# Create Security Group
resource "aws_security_group" "kubernetes" {
vpc_id = "${aws_vpc.vpc.id}"
```

```
name = "MySG-k8s"
}

# Allow SSH connections from ALL
resource "aws_security_group_rule" "allow_ssh_from_all" {
type    = "ingress"
protocol  = "tcp"
from_port  = 22
to_port   = 22
cidr_blocks = ["0.0.0.0/0"]
security_group_id = "${aws_security_group.kubernetes.id}"
}

# Allow API connections from ALL
resource "aws_security_group_rule" "allow_api_from_all" {
type    = "ingress"
protocol  = "tcp"
from_port  = 6443
to_port   = 6443
cidr_blocks = ["0.0.0.0/0"]
security_group_id = "${aws_security_group.kubernetes.id}"
}

# Allow the security group members to talk with each other without restrictions
resource "aws_security_group_rule" "allow_cluster_crosstalk" {
type = "ingress"
from_port = 0
to_port = 0
protocol = "-1"
source_security_group_id = "${aws_security_group.kubernetes.id}"
security_group_id = "${aws_security_group.kubernetes.id}"
}

# Create a Keypair
resource "aws_key_pair" "kubernetes" {
key_name = "${var.aws_keypair_name}"
public_key = "${var.aws_keypair_public_key}"
}

# Template for initial configuration script
data "template_file" "init" {
template = "${file("templates/init.tpl")}"
}

# Create the Kubernetes master node
resource "aws_instance" "etcd" {
count = 1
```

```
ami = "${var.aws_ami_id}"
instance_type = "t2.micro"

subnet_id = "${aws_subnet.kubernetes.id}"
user_data = "${data.template_file.init.rendered}"

vpc_security_group_ids = ["${aws_security_group.kubernetes.id}"]
key_name = "${aws_key_pair.kubernetes.id}"
}

# Create the Kubernetes controlplane node
resource "aws_instance" "controlplane" {
count = 1
ami = "${var.aws_ami_id}"
instance_type = "t2.micro"

subnet_id = "${aws_subnet.kubernetes.id}"
user_data = "${data.template_file.init.rendered}"

vpc_security_group_ids = ["${aws_security_group.kubernetes.id}"]
key_name = "${aws_key_pair.kubernetes.id}"
}

# Create the Kubernetes worker node
resource "aws_instance" "worker" {
count = 1
ami = "${var.aws_ami_id}"
instance_type = "t2.micro"

subnet_id = "${aws_subnet.kubernetes.id}"
user_data = "${data.template_file.init.rendered}"

vpc_security_group_ids = ["${aws_security_group.kubernetes.id}"]
key_name = "${aws_key_pair.kubernetes.id}"
}

output "vpc_id" {
value = "${aws_vpc.vpc.id}"
}

output "Node etcd public dns" {
value = "${aws_instance.etcd.public_dns}"
}

output "Node etcd private dns" {
value = "${aws_instance.etcd.private_dns}"
}
```

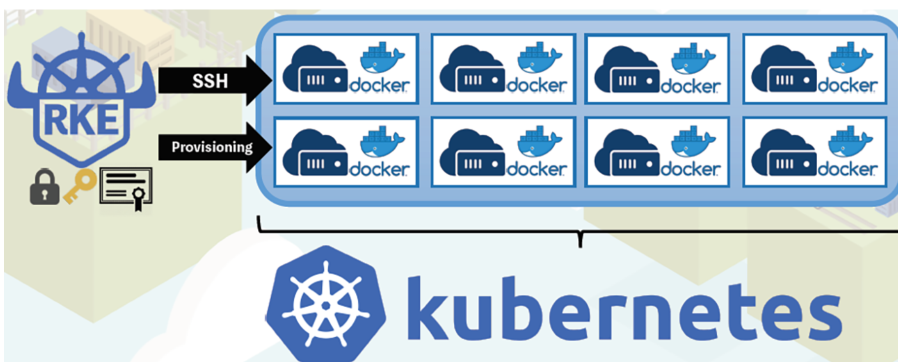
```
output "Node controlplane public dns" {
value = "${aws_instance.controlplane.ip}"
}

output "Node controlplane private dns" {
value = "${aws_instance.controlplane.private_dns}"
}

output "Node worker public dns" {
value = "${aws_instance.worker.public_dns}"
}

output "Node worker private dns" {
value = "${aws_instance.worker.private_dns}"
}
```

Ejecutamos el comando terraform apply y observaremos como Terraform se encarga de generar los recursos definidos en nuestro fichero hasta llegar al estado objetivo: tras la ejecución del comando hemos obtenido un fichero llamado terraform.tfstate. Dicho fichero almacena el estado actual de la infraestructura y es consultado por Terraform para obtener la información antes de aplicar nuevos cambios sobre la misma.

2.4. Desplegar Kubernetes con RKE

Una vez creada toda la infraestructura sobre AWS, ya podemos comenzar a trabajar sobre ella y empezar el despliegue de Kubernetes. Como la instalación de Kubernetes puede llegar a ser un proceso relativamente complejo, nos ayudaremos de un instalador que nos facilitará y automatizará todo el proceso de instalación desplegando los servicios de Kubernetes de forma «dockerizada» en diferentes *containers*.

Figura 39

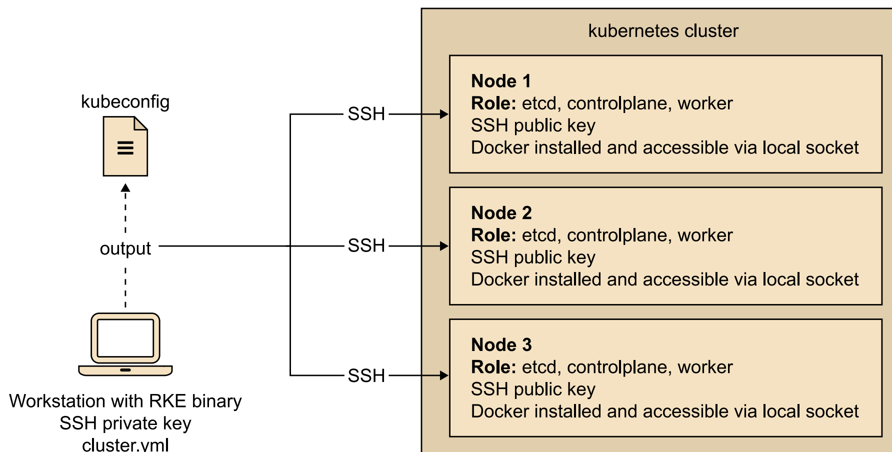


RKE és el acrónimo de Rancher Kubernetes Engine y es la herramienta de línea de comandos de Rancher para crear, administrar y actualizar clústeres de Kubernetes. Desde un archivo de texto, en formato YAML, con la configuración

del clúster (denominado cluster.yml de forma predeterminada), puede ejecutar el comando rke up y RKE creará un clúster Kubernetes de acuerdo con los detalles que especificó en el archivo de configuración.

RKE está escrito en Golang y utilizará las librerías predeterminadas para comunicarse con los nodos. Eso significa que el binario RKE es lo único que necesita instalar localmente para crear un clúster.

Figura 40



Para que RKE pueda aprovisionar Kubernetes, cada *host* debe tener lo siguiente:

- Servicio SSH accesible.
- Un par de claves SSH.
- Docker instalado y accesible para el usuario de SSH en un *socket* local (/var/run/docker.sock por defecto).

RKE se conectará a los *hosts* del clúster utilizando una librería local de SSH, se autentificará con el usuario configurado y la clave privada de SSH, y se conectará a la instancia de Docker del *host* a través del archivo de *socket*. A continuación, utilizará el *socket* de Docker para iniciar los contenedores necesarios para configurar el clúster.

Para ejecutar la herramienta RKE en sí, se necesita un sistema operativo compatible (Linux, MacOS o Windows) y tener acceso a la red a cada uno de los *hosts* que desea aprovisionar para su clúster.

2.4.1. Instalación de RKE

La instalación consiste únicamente en la descarga de la última versión de RKE en función de nuestro sistema operativo, en nuestro equipo de trabajo:

<https://github.com/rancher/rke/releases>

```
$ wget https://github.com/rancher/rke/releases/download/v0.1.5/rke
$ mv rke_linux-amd64 rke
```

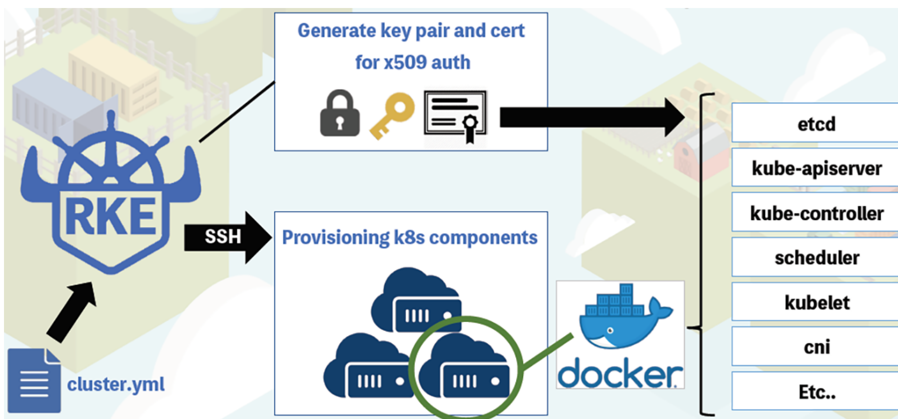
```
$ chmod +x rke
```

Copiaremos el binario de RKE a una carpeta de nuestro $PATH, por ejemplo, /usr/local/bin y renombraremos el binario a *rke*.

Confirmaremos que la instalación de RKE funciona correctamente ejecutando el siguiente comando:

```
$ rke --version
```

Figura 41



Como se muestra en la imagen, el funcionamiento de RKE es el siguiente:

1) El usuario define la configuración del clúster en un fichero en formato YAML.

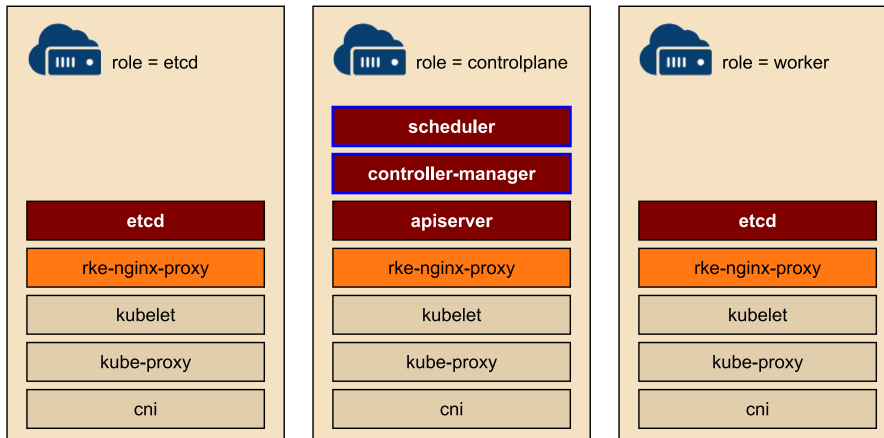
2) RKE se conecta a los *host* Linux por SSH, a través de usuario/password o claves SSH.

3) RKE aprovisiona los servicios de Kubernetes a través de *containers* de Docker.

4) Finalmente, RKE genera el archivo kube_config_cluster.yml para que podamos acceder a la API de Kubernetes e interaccionar con el clúster.

Ahora, crearemos el archivo cluster.yml y construiremos nuestro clúster de Kubernetes de prueba:

Figura 42



Como habíamos creado tres instancias en EC2, cada una de ellas tendrá un rol diferente de los tres necesarios para el funcionamiento de Kubernetes. También podemos asignar los tres roles a cada una de las instancias y tener así nuestro clúster en alta disponibilidad (HA).

Los diferentes **roles** son los siguientes:

- **Etcd**: mantiene el estado del clúster y es el componente más importante. Podemos ejecutar etcd en un solo nodo, aunque normalmente se necesitan 3, 5 o más nodos para crear una configuración de alta disponibilidad. Etcd es un almacén *key-value* distribuido que almacena todo el estado de Kubernetes. <https://coreos.com/etcd/>

- **Controlplane**: ejecutar el servidor de API, el *scheduler* y los controladores.

- **Worker**: en estos nodos, se crearán las cargas de trabajo y el despliegue de los diferentes *pods*.

Cada componente de Kubernetes se ejecutará en un contenedor Docker. Los componentes específicos requeridos dependerán de la función del nodo dentro del clúster: como miembro de etcd, miembro del controlplane, worker, o una combinación de estos. RKE también implementará el complemento de red especificado CNI (Canal, Flannel o Calico son los soportados actualmente), el servidor de métricas y el controlador de *ingress* Nginx como Daemonset en cada nodo.

Para este ejemplo utilizaremos una configuración básica de RKE, pero podemos consultar todas las opciones disponibles en la siguiente URL:

<https://rancher.com/docs/rke/v0.1.x/en/config-options/>

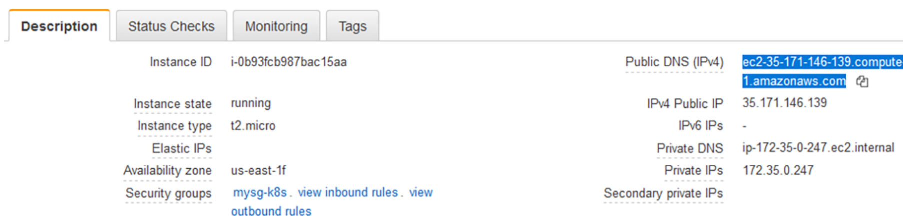
O consultar algunos ejemplos más en:

<https://rancher.com/docs/rke/v0.1.x/en/example-yamls/>

```
nodes:
  - address: ec2-35-171-146-139.compute-1.amazonaws.com
    user: ubuntu
    ssh_key_path: k8s-key.pem
    role:
    - etcd
  - address: ec2-3-80-232-112.compute-1.amazonaws.com
    user: ubuntu
    ssh_key_path: k8s-key.pem
    role:
    - controlplane
  - address: ec2-3-80-5-11.compute-1.amazonaws.com
    user: ubuntu
    ssh_key_path: k8s-key.pem
    role:
    - worker
```

En nuestro caso tan solo especificaremos la IP pública de nuestra instancia de EC2, que podemos obtener desde la consola web de AWS. El nombre de usuario, por defecto EC2, crea el usuario ubuntu en la instancia, la ruta a la clave SSH que generamos en AWS y el rol que le asignamos al *host*.

Figura 43



Una vez finalizada la configuración, el despliegue del clúster ya es cosa de RKE y tan solo debemos ejecutar el comando:

```
$ rke up --config cluster.yml
```

El instalador de RKE nos irá mostrando por consola todos los pasos que va realizando sobre cada una de las instancias, como se puede apreciar en la siguiente imagen:

Figura 44

```
INFO[0136] [remove/rke-log-linker] Successfully removed container on host [54.86.111.112]
INFO[0136] [healthcheck] service [kube-proxy] on host [54.162.202.115] is healthy
INFO[0137] [worker] Successfully started [rke-log-linker] container on host [54.162.202.115]
INFO[0138] [remove/rke-log-linker] Successfully removed container on host [54.162.202.115]
INFO[0139] [healthcheck] service [kubelet] on host [18.234.109.13] is healthy
INFO[0140] [worker] Successfully started [rke-log-linker] container on host [18.234.109.13]
INFO[0141] [remove/rke-log-linker] Successfully removed container on host [18.234.109.13]
INFO[0141] [worker] Successfully started [kube-proxy] container on host [18.234.109.13]
INFO[0141] [healthcheck] Start Healthcheck on service [kube-proxy] on host [18.234.109.13]
INFO[0143] [healthcheck] service [kube-proxy] on host [18.234.109.13] is healthy
INFO[0144] [worker] Successfully started [rke-log-linker] container on host [18.234.109.13]
INFO[0145] [remove/rke-log-linker] Successfully removed container on host [18.234.109.13]
INFO[0145] [worker] Successfully started Worker Plane..
INFO[0145] [sync] Syncing nodes Labels and Taints
INFO[0148] [sync] Successfully synced nodes Labels and Taints
INFO[0148] [network] Setting up network plugin: canal
INFO[0148] [addons] Saving addon ConfigMap to Kubernetes
INFO[0149] [addons] Successfully Saved addon to Kubernetes ConfigMap: rke-network-plugin
INFO[0149] [addons] Executing deploy job..
INFO[0155] [addons] Setting up KubeDNS
INFO[0155] [addons] Saving addon ConfigMap to Kubernetes
INFO[0155] [addons] Successfully Saved addon to Kubernetes ConfigMap: rke-kubedns-addon
INFO[0155] [addons] Executing deploy job..
INFO[0167] [addons] KubeDNS deployed successfully..
INFO[0167] [addons] Setting up Metrics Server
INFO[0167] [addons] Saving addon ConfigMap to Kubernetes
INFO[0167] [addons] Successfully Saved addon to Kubernetes ConfigMap: rke-metrics-addon
INFO[0167] [addons] Executing deploy job..
INFO[0173] [addons] KubeDNS deployed successfully..
INFO[0173] [ingress] Setting up nginx ingress controller
INFO[0173] [addons] Saving addon ConfigMap to Kubernetes
INFO[0174] [addons] Successfully Saved addon to Kubernetes ConfigMap: rke-ingress-controller
INFO[0174] [addons] Executing deploy job..
INFO[0185] [ingress] ingress controller nginx is successfully deployed
INFO[0185] [addons] Setting up user addons
INFO[0185] [addons] no user addons defined
INFO[0185] Finished building Kubernetes cluster successfully
[root@node ~]#
```

Si todo ha funcionado correctamente, el instalador finalizará con el mensaje:

```
$ Finished building Kubernetes cluster successfully
```

y nos habrá generado el archivo que utilizaremos para acceder a nuestro clúster de Kubernetes a través de la API.

Después de crear el clúster, RKE genera un archivo de credenciales de clúster (kubeconfig), por defecto con el nombre de kube_config_cluster.yml, que se puede usar con kubectl para interactuar con su clúster. Si se desea agregar o eliminar nodos, actualizar la versión de Kubernetes o cambiar la configuración del clúster de Kubernetes, se puede usar el comando RKE nuevamente. Solo hay que modificar el archivo de configuración del clúster RKE (cluster.yml) y ejecutar el comando rke up una vez más.

2.5. Administrar nuestro clúster de Kubernetes

Una de las formas de interaccionar con el clúster de Kubernetes es mediante la utilidad de línea de comandos llamada kubectl para comunicarse con el servidor de la API del clúster.

Para instalar kubectl en Linux (Ubuntu, Debian):

```
$ sudo apt-get update && sudo apt-get install -y apt-transport-https
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo $ apt-key add -
```

```
$ echo "deb https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee -a /etc/apt/
  sources.list.d/kubernetes.list
$ sudo apt-get update
$ sudo apt-get install -y kubectl
```

Si necesitamos instalar kubectl en otro sistema operativo, podemos consultar la documentación de Kubernetes:

<https://kubernetes.io/docs/tasks/tools/install-kubectl/>

Configuraremos kubectl para que utilice el fichero de configuración que nos ha generado RKE con las credenciales para acceder al clúster ejecutando los siguientes comandos:

```
$ mkdir -p $HOME/.kube;
$ cp -i kube_config_cluster.yml $HOME/.kube/config;
$ chown $(id -u):$(id -g) $HOME/.kube/config;
```

Podemos comprobar el estado de los nodos de nuestro clúster con el siguiente comando:

```
$ kubectl get nodes
```

Figura 45

```
[root@node ~]# kubectl get nodes
NAME                                         STATUS   ROLES          AGE   VERSION
ec2-3-80-232-112.compute-1.amazonaws.com     Ready    controlplane   2m    v1.11.6
ec2-3-80-5-11.compute-1.amazonaws.com        Ready    worker         2m    v1.11.6
ec2-35-171-146-139.compute-1.amazonaws.com   Ready    etcd           2m    v1.11.6
```

Si lo que queremos es consultar todos los *pods* que se generan nada más crear un clúster de Kubernetes podemos usar el comando:

```
$ kubectl get pods --all-namespaces
```

Alternativamente, también podemos indicarle directamente nuestro archivo de configuración con el parámetro --kubeconfig:

Figura 46

```
[root@node ~]# kubectl --kubeconfig kube_config_cluster.yml get pods --all-namespaces
NAMESPACE       NAME                                      READY   STATUS      RESTARTS   AGE
ingress-nginx   default-http-backend-797c5bc547-8hmwb     1/1     Running     0          4m
ingress-nginx   nginx-ingress-controller-p9ssc            1/1     Running     0          4m
kube-system     canal-27sjn                               3/3     Running     0          4m
kube-system     canal-2vgxh                               3/3     Running     0          4m
kube-system     canal-n9kjr                               3/3     Running     0          4m
kube-system     kube-dns-7588d5b5f5-sk5f5                 3/3     Running     0          4m
kube-system     kube-dns-autoscaler-5db9bbb766-gjtzb      1/1     Running     0          4m
kube-system     metrics-server-97bc649d5-b7pv2            1/1     Running     0          4m
kube-system     rke-ingress-controller-deploy-job-4ph8h   0/1     Completed   0          4m
kube-system     rke-kubedns-addon-deploy-job-77m7v        0/1     Completed   0          4m
kube-system     rke-metrics-addon-deploy-job-hbdk6        0/1     Completed   0          4m
kube-system     rke-network-plugin-deploy-job-5pmkd       0/1     Completed   0          4m
```

Ahora nuestro clúster ya está preparado para poder lanzarle cargas de trabajo, *workloads,* de Kubernetes.

Enlace de interés

Para conocer más sobre la sintaxis de kubectl es interesante consultar la guía *kubectl Cheat Sheet* de la página oficial de Kubernetes:
<https://kubernetes.io/docs/reference/kubectl/cheatsheet/>

2.5.1. Administrar Kubernetes desde el Dashboard

Una forma más intuitiva y fácil de administrar Kubernetes es a través de su Dashboard. Se trata de una interfaz de usuario basada en web. Además, desde aquí también vamos a poder ver claramente los *pods* que tenemos, los nodos, los estados de estos, varias estadísticas y demás.

Lo primero que haremos entonces será la instalación del Dashboard a través del fichero YAML de la web oficial de Kubernetes:

```
$ kubectl create -f https://raw.githubusercontent.com/kubernetes/dashboard/master/aio/deploy/
  recommended/kubernetes-dashboard.yaml
```

Una vez desplegado el YAML que nos ofrece el Dashboard, verificaremos que se ha desplegado correctamente:

Figura 47

```
[root@node ~]# kubectl get pods -n kube-system
NAME                                      READY   STATUS      RESTARTS   AGE
canal-27sjn                               3/3     Running     0          7m
canal-2vgxh                               3/3     Running     0          7m
canal-n9kjr                               3/3     Running     0          7m
kube-dns-7588d5b5f5-sk5f5                 3/3     Running     0          7m
kube-dns-autoscaler-5db9bbb766-gjtzb      1/1     Running     0          7m
kubernetes-dashboard-5dd89b9875-qlqd7     1/1     Running     0          58s
metrics-server-97bc649d5-b7pv2            1/1     Running     0          7m
rke-ingress-controller-deploy-job-4ph8h   0/1     Completed   0          7m
rke-kubedns-addon-deploy-job-77m7v        0/1     Completed   0          7m
rke-metrics-addon-deploy-job-hbdk6        0/1     Completed   0          7m
rke-network-plugin-deploy-job-5pmkd       0/1     Completed   0          7m
```

Para acceder al Dashboard de Kubernetes utilizaremos un *service* de tipo **NodePort**, y para ello editaremos el *service* por defecto, que utiliza el tipo ClusterIP a NodePort, con el comando:

```
$ kubectl -n kube-system edit service kubernetes-dashboard
```

Figura 48

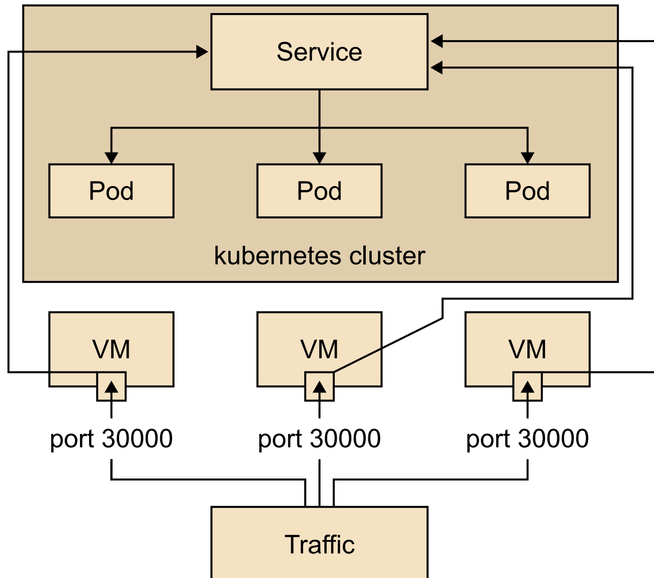


Figura 49

```
metadata:
  creationTimestamp: 2019-02-10T14:22:41Z
  labels:
    k8s-app: kubernetes-dashboard
  name: kubernetes-dashboard
  namespace: kube-system
  resourceVersion: "2086"
  selfLink: /api/v1/namespaces/kube-system/services/kubernetes-dashboard
  uid: 53508362-2d3f-11e9-b77c-16052ff274b0
spec:
  clusterIP: 10.43.177.59
  externalTrafficPolicy: Cluster
  ports:
  - nodePort: 32319
    port: 443
    protocol: TCP
    targetPort: 8443
  selector:
    k8s-app: kubernetes-dashboard
  sessionAffinity: None
  type: NodePort
status:
  loadBalancer: {}
```

Ahora si consultamos nuevamente el *service* del Dashboard de Kubernetes, vemos que está publicado en el puerto **32319/TCP** de nuestro *host*.

```
$ kubectl -n kube-system get service kubernetes-dashboard
```

Figura 50

```
[root@node ~]# kubectl -n kube-system get service kubernetes-dashboard
NAME                   TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)         AGE
kubernetes-dashboard   NodePort   10.43.177.59   <none>        443:32319/TCP   9m
```

Antes de poder acceder al Dashboard, debemos revisar nuestro Security Group y añadir ese puerto a la lista de puertos permitidos:

Figura 51

Edit inbound rules

Inbound rules control the incoming traffic that's allowed to reach the instance.

Type	Protocol	Port Range	Source	
Custom TCP Rule	TCP	6443	Custom	0.0.0.0/0
Custom TCP Rule	TCP	6443	Custom	::/0
All traffic	All	All	Custom	sg-0fa73185737906d8b
SSH	TCP	22	Custom	0.0.0.0/0
SSH	TCP	22	Custom	::/0
Custom TCP Rule	TCP	32319	Anywhere	0.0.0.0/0, ::/0

Add Rule

Ahora ya podemos acceder al Dashboard a través del DNS público de nuestra instancia de EC2, por HTTPS y en el puerto indicado:

<https://ec2-3-80-5-11.compute-1.amazonaws.com:32319/#!/login>

Figura 52

Kubernetes Dashboard

Authentication method:

- Kubeconfig
- Token

Token *

SIGN IN SKIP

El último paso para acceder al Dashboard será modificar el *service account* de kubernetes-dashboard para que puede administrar el clúster:

```
$ kubectl create clusterrolebinding kubernetes-dashboard -n kube-system --clusterrole=cluster-
  admin --serviceaccount=kube-system:kubernetes-dashboard
```

y obtener el *token* de acceso:

```
$ kubectl -n kube-system describe secret kubernetes-dashboard-token-XXXXX
```

Figura 53


```
Data
====
ca.crt:     1017 bytes
namespace:  11 bytes
token:      eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3
VudC9uYW1lc3BhY2UiOiJrdWJlLXN5c3RlbSIsImtlYmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VjcmV0Lm5hbWUiOiJrdWJlcm5ldGVzLWRhc2hib2FyZC10b
2tlbilweno0ciIsImtlYmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VydmljZS1hY2NvdW50Lm5hbWUiOiJrdWJlcm5ldGVzLWRhc2hib2FyZCIsImtlYmVybmV0
ZXMuaW8vc2VydmljZWFjY291bnQvc2VydmljZS1hY2NvdW50LnVpZCI6IjUyZGY3N2NlLTJkM2YtMTFlOS1iNzdjLTE2MDUyZmYyNzRiMCIsInN1YiI6InN5c3RlbTp
zZXJ2aWNlYWNjb3VudDprdWJlLXN5c3RlbTprdWJlcm5ldGVzLWRhc2hib2FyZCJ9.q0bWB2NMfJRnHSbbyDTdc-h38Op8-VTvoluqkTU98tf4klHOUYpRvscDz7wnv
3QWN7JrEkytZm-QrDIo3XlghA79CCNaZuWY8leokqxJShJR3PAGqxWxJalc6qpmzJ9Q2lTacAmnLqqPhllcmVU6IW8iVb_IMcHE4ZvPGNFxw20iC_tPwgnplWqwHCtY
i5P33hoL-QreEnxM3825f49w3kwC7h3LnCcpOelWR-Ia2wJ5Pxu2GQ9yFgn3pwh7cFE4y8MFvFqIotF7Acfejup8c_6FKbYdQ38EPfwjYCSE8z8Rtr-eSAt55IFbc_m
ZVFEwyVHHUdLkAWWn90qwD5PN7A
```

Figura 54

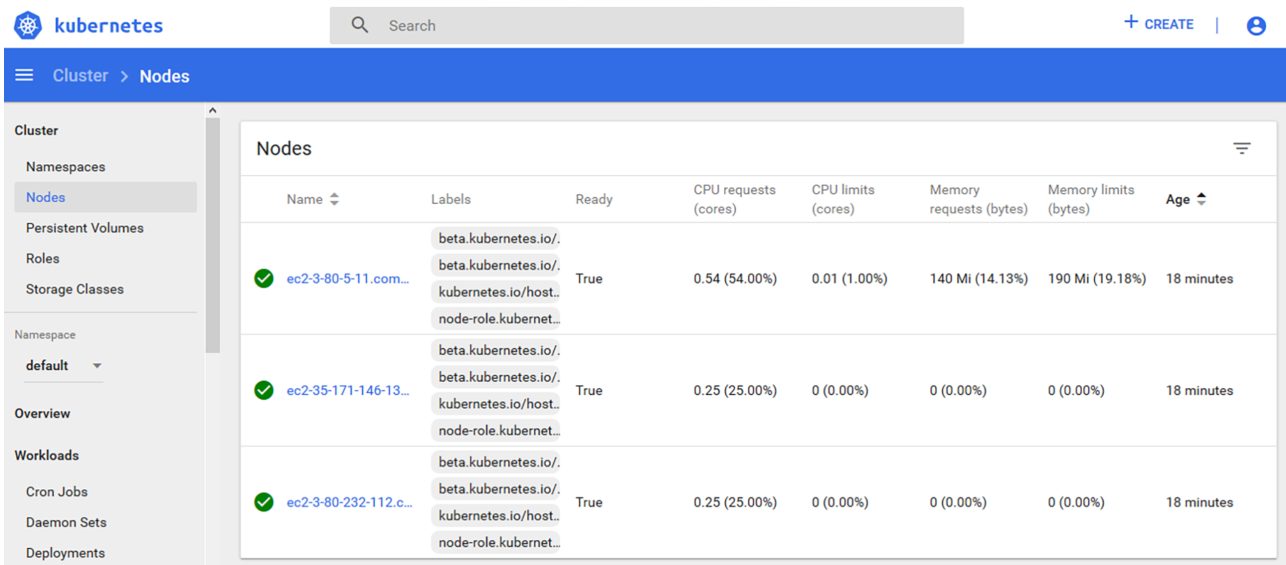


Figura 55

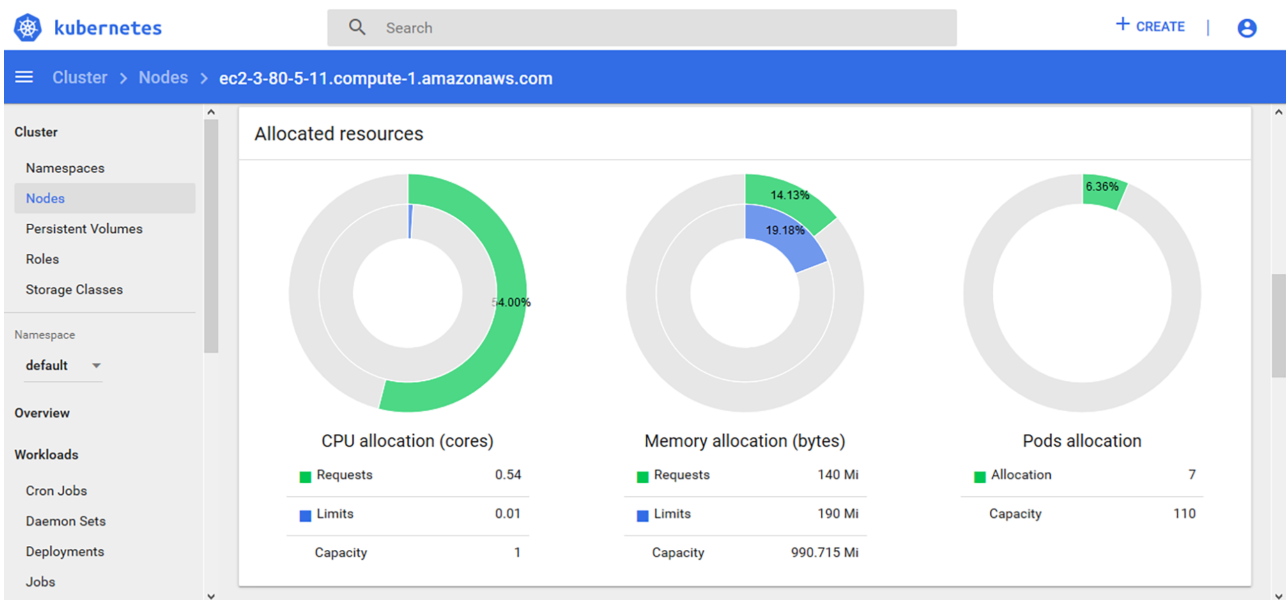


Figura 56

kubernetes
Search
+ CREATE
Workloads > Pods

Persistent Volumes
Roles
Storage Classes
Namespace
kube-system
Overview
Workloads
Cron Jobs
Daemon Sets
Deployments
Jobs
Pods
Replica Sets
Replication Controllers
Stateful Sets
Discovery and Load Balancing
Ingresses
Services

Pods

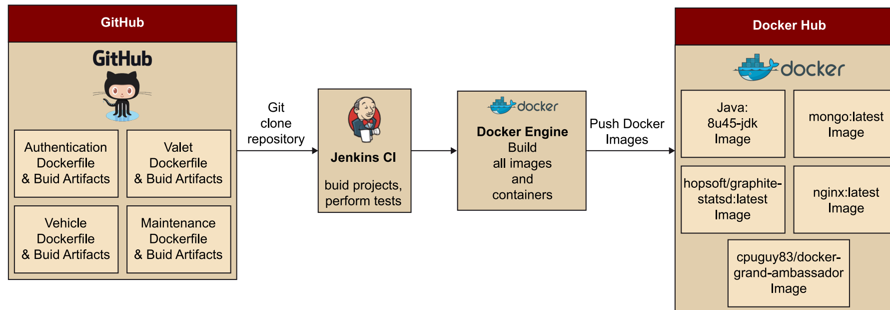
Name	Node	Status	Restarts	Age
kubernetes-dashboard-5dd89b9875-qlqd7	ec2-3-80-5-11.compute-1.amazonaws.com	Running	0	12 minutes
rke-ingress-controller-deploy-job-4ph8h	ec2-3-80-232-112.compute-1.amazonaws.com	Terminated: Completed	0	19 minutes
metrics-server-97bc649d5-b7pv2	ec2-3-80-5-11.compute-1.amazonaws.com	Running	0	19 minutes
rke-metrics-addon-deploy-job-hbdk6	ec2-3-80-232-112.compute-1.amazonaws.com	Terminated: Completed	0	19 minutes
kube-dns-autoscaler-5db9bbb766-gjtzb	ec2-3-80-5-11.compute-1.amazonaws.com	Running	0	19 minutes
kube-dns-7588d5b5f5-sk5f5	ec2-3-80-5-11.compute-1.amazonaws.com	Running	0	19 minutes
rke-kubedns-addon-deploy-job-77m7v	ec2-3-80-232-112.compute-1.amazonaws.com	Terminated: Completed	0	19 minutes
canal-27sjn	ec2-3-80-5-11.compute-1.amazonaws.com	Running	0	19 minutes
canal-2vgxh	ec2-35-171-146-139.compute-1.amazonaws.com	Running	0	19 minutes
canal-n9kjr	ec2-3-80-232-112.compute-1.amazonaws.com	Running	0	19 minutes

1 - 10 of 11

https://ec2-3-80-5-11.compute-1.amazonaws.com:32319/#!/log/kube-system/canal-27sjn/pod?namespace=kube-system

3. Herramientas de automatización completa y *testing*: Jenkins

Figura 57. Diagrama relacional de Git, los diferentes componentes de Docker y OpenNebula



Requerimientos previos:

- **Java:** Lenguaje de programación multiplataforma.
- **Maven:** Herramienta de gestión y construcción de software, encargada de proveer vía online de las dependencias necesarias.
- **Git:** Software de control de versiones
- **Tomcat:** Funciona como un contenedor de servlets que implementa las especificaciones de Java Servlet y JavaServer Pages (JSP).
- **Docker Engine:** Nos permite crear los contenedores, gestionarlos e instanciar-los.
- **Jenkins:** Software dedicado a la integración continua que nos permitirá automatizar los procesos de test, construcción y despliegue entre otros.
- **Registry Docker Hub:** Nos permite crear, compartir y utilizar imágenes creadas por nosotros o por terceros.

En este caso de uso partiremos de un proyecto web desarrollado en Java cumpliendo con las especificaciones de JavaServlet Pages. Por ello, se deberá instalar **Java Development Kit** (JDK) y descargar la librería **JUnit 4** para ejecutar los test; por último, solo quedará instalar el contenedor de servlets **Apache Tomcat**.

3.1. Servlet Demo

El servlet utilizado durante la fase de automatización es un ejemplo sencillo de una aplicación web Java ubicada en un sistema de versionado público como GitHub.

Para trabajar con el proyecto de forma individual se deberá hacer un *fork* del repositorio, lo que permitirá a cada alumno poder tener una copia remota con permisos para hacer cualquier cambio requerido.

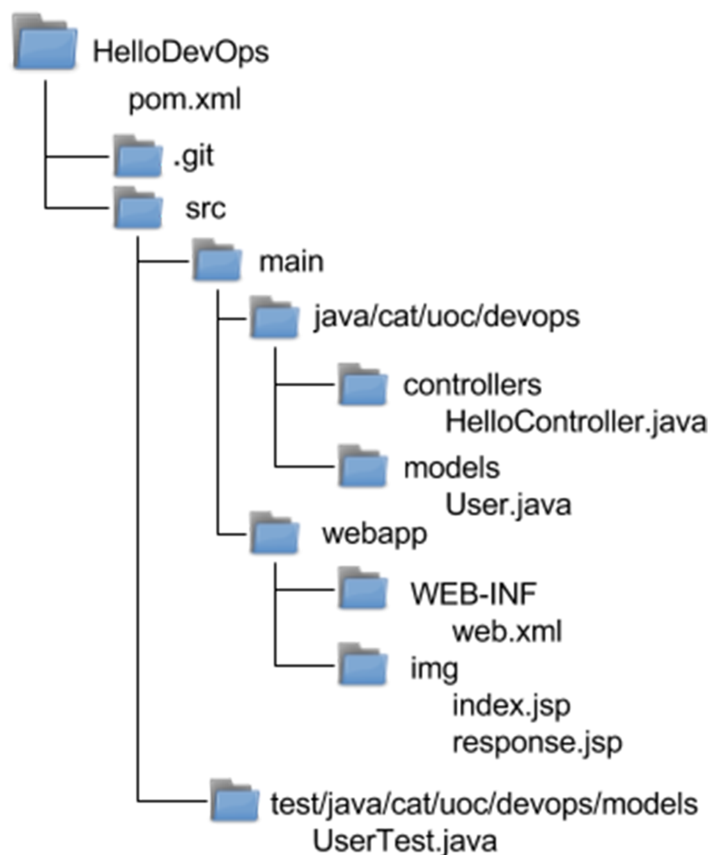
Una vez disponemos del *fork* del repositorio en nuestra cuenta de GitHub, procederemos al clonado en local mediante el comando:

```
git clone https://github.com/<usuario>/HelloDevOps.git
```

El proyecto demo está basado en un servlet JSP con una clase principal User junto con su controlador y los test JUnit, que verificarán el correcto funcionamiento de este.

La estructura completa del proyecto será la siguiente:

Figura 58. Detalle de las estructura del proyecto



Fuente: Joan Caparrós.

El proyecto HelloDevOps contiene los siguientes subdirectorios:

- **.git:** Esta carpeta se crea automáticamente en proyectos versionados bajo Git.

- **src/main:** Contendrá los modelos y los controladores junto con los jsp que construirán las páginas web de la aplicación.

- **src/test:** Contendrá los diferentes test desarrollados bajo la librería JUnit.

3.2. Ejecución de test del proyecto

Para ejecutar las pruebas unitarias, no es necesario configurar nada. La configuración predeterminada de Maven escaneará el directorio ${basedir}/src/test/java en busca de ficheros con el patrón *Test.java.

Para efectuar las pruebas unitarias desarrolladas, ejecutaremos:

```
$ mvn test
```

En el caso de superar los test, verificaremos que todos los test se han superado y que el resultado completo de la ejecución de las pruebas unitarias ha sido satisfactorio.

Figura 59. Detalle de una ejecución satisfactoria de los test

```
...
-------------------------------------------------------
T E S T S
-------------------------------------------------------
Running cat.uoc.devops.models.UserTest
Testing getName
Testing setName
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.044 sec

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
...
```

Fuente: Joan Caparrós.

Existen otros tipos de test, llamados **test de integración continua**, que se diferencian de los unitarios en que no necesitan disponer de todo el entorno de la aplicación para ejecutarse, sino que analizan las aplicaciones a partir de sus contextos (partes independientes en las que se divide una aplicación).

Maven incorpora la posibilidad de poder realizar estos test independientes sin que afecten a la continuidad de la construcción del proyecto. Imaginad que uno de los contextos de nuestra aplicación formase parte de un sistema externo susceptible a sufrir inestabilidades, los test pararían el proceso de construcción cada vez que este no respondiera de forma adecuada. Para estos casos, Maven –mediante el uso de su *plugin* Failsafe– introduce la posibilidad de incorporar en el ciclo de vida de construcción las fases de «pre-integration-test», «integration-test» y «post-integration-test».

Para el uso del *plugin* Failsafe se deberá añadir al fichero pom.xml el siguiente código:

```
<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>2.19.1</version>
        <configuration>
            <testFailureIgnore>false</testFailureIgnore>
       <configuration>
        <executions>
          <execution>
            <goals>
              <goal>integration-test</goal>
              <goal>verify</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  [...]
</project>
```

Según convención, los test que queramos integrar en los test de integración continua deberán incorporar «IT» o «IntegrationTest» como sufijo en el nombre del fichero.

Definir si el proceso de *build* debe o no pararse al fallar un test de integración continua vendrá indicado por la variable *testFailureIgnore* de nuestro fichero pom.xml, «false» (valor por defecto) para detener el proceso ante fallos de test de integración continua y «true» para continuar pese el fallo.

3.3. Construcción del proyecto

Para compilar el proyecto de forma manual tendremos que ejecutar el siguiente comando:

```
$ mvn package
```

Mediante el uso de Maven y dado el contenido del fichero principal **pom.xml**, el comando resolverá todas las dependencias, procederá a la ejecución de los test desarrollados para finalmente empaquetar el proyecto dentro de un fichero WAR en el directorio target.

Los archivos WAR (Web Application Archive) son ficheros utilizados para la distribución de JavaServer Pages, servlets, clases Java, archivos XML, librerías y otros ficheros necesarios para el despliegue de aplicaciones en contenedores web como el proporcionado por Tomcat.

3.3.1. Despliegue del proyecto

Una vez realizada la construcción del proyecto y para hacer el servlet accesible, deberemos:

- Copiar el archivo WAR generado en la carpeta webapps de nuestro tomcat.
- Reiniciar el servicio de tomcat.

A partir de este momento la aplicación estará disponible bajo el subdirectorio con nombre igual al fichero WAR desplegado.

3.3.2. Página inicial

La página inicial se compone de un formulario simple que requiere un nombre en forma de cadena de caracteres.

Figura 60. Detalle de la página inicial

Fuente: Joan Caparrós.

3.3.3. Página resultado

La página resultado utiliza la cadena anteriormente introducida para saludar al programador en cuestión.

Figura 61. Detalle de la página resultado

Ejemplo JSP - Automatización y testing

Hola DevOp

Fuente: Joan Caparrós.

3.4. Dockerización del proyecto

En este momento disponemos de una aplicación web funcional, que requiere un contenedor de aplicaciones Apache Tomcat para hacerla accesible.

¿Cómo hacerla portable y que pueda ser ejecutada en cualquier máquina con docker? La respuesta es la dockerización de la aplicación.

Docker nos permitirá introducir en una «caja» todas aquellas cosas que la aplicación necesita para ser ejecutada, sin tener que preocuparse por la versión de software que la máquina *host* tiene instalada, por si tiene instalados todos los módulos necesarios o por si son compatibles o no con nuestra aplicación.

El despliegue de la imagen dockerizada resultante ofrecerá a los desarrolladores, *testers* y administradores de sistemas confianza y seguridad en el producto resultante, ya que asegurará el entorno de ejecución, permitiendo a los desarrolladores centrarse en el código sin preocuparse de posibles fallos dados por las máquinas donde se ejecutarán.

3.4.1. Construcción de la imagen docker

Para construir una imagen docker podemos partir de cualquier contenedor disponible en el registro de imágenes de Docker (Docker Hub Registry), donde se encuentran disponibles aplicaciones funcionales, tales como bases de datos, servidores de aplicaciones, servidores web, etc.; lo único necesario será construir un archivo Dockerfile que contenga la composición de nuestro contenedor.

Nota

Docker Hub dispone de más de cien mil imágenes disponibles de forma pública y listas para ser usadas.

Nuestra imagen partirá de la imagen oficial dockerizada del contenedor de aplicaciones web tomcat en su versión 8.5. El funcionamiento del contenedor de aplicaciones es sencillo y solo requerirá copiar en su carpeta webapps el fichero WAR resultante de la construcción del proyecto para que de esta forma al ejecutar la aplicación se realice el despliegue de la aplicación de manera automática.

El fichero Dockerfile pondrá por escrito todo aquello necesario para el despliegue de nuestra aplicación, ejecutando todas aquellas instrucciones para preparar el entorno encapsulado. Este fichero puede construirse en cualquier directorio de nuestra máquina.

Las instrucciones más importantes disponibles para la construcción de un fichero Dockerfile son:

1) FROM: establece la imagen base a partir de la cual crearemos la imagen que construirá el Dockerfile. Un Dockerfile válido debe situar la instrucción FROM como primera línea (excluyendo comentarios).

```
#Inclusión de la última versión de la imagen (por defecto tag:latest).
FROM <imagen>

#Definición de la versión específica de la imagen a utilizar mediante tag.
FROM <imagen>:<tag>

#Definición de la versión específica de la imagen a utilizar mediante el uso del código sha256
generado durante la construcción de la imagen.
FROM <imagen>@<digest>
```

2) MAINTAINER: permite establecer el campo autor de la imagen generada.

```
MAINTAINER <autor>
```

3) ENV: establece variables de entorno que podrán ser interpretadas por las instrucciones mediante el formato $variable o ${variable}.

```
ENV <variable> <valor>
ENV <variable>=<valor>
```

4) RUN: permite ejecutar una instrucción en el contenedor, para instalar o persistir cambios en el contenedor. Resulta útil para la instalación de paquetería mediante gestores de paquetes o ejecución de *scripts* que afecten a la construcción de la imagen.

```
#Ejecución del comando en una shell, por defecto /bin/sh -c en Linux y cmd /S /C em Windows
RUN <command>

#Ejecución del comando en una shell en formato parametrizado (formato exec)
RUN ["ejecutable", "param1", "param2"]
```

5) ADD: permite añadir ficheros, directorios o ficheros remotos al sistema de ficheros del contenedor.

```
#Copia ficheros con origen local o remoto en un destino dentro del contenedor
ADD <origen>... <destino>

#Versión parametrizada de la instrucción
ADD ["<origen>",... "<destino>"]
```

6) COPY: similar a la instrucción ADD, permite añadir ficheros y directorios pero no permite la inclusión de ficheros remotos.

```
COPY <origen>... <destino>
COPY ["<origen>",... "<destino>"]
```

7) VOLUME: crea un punto de montaje en el sistema de ficheros del contenedor. Los volúmenes permiten externalizar un determinado directorio y así proporcionar persistencia a los datos depositados (las imágenes de docker no almacenan datos en el contenedor entre diferentes ejecuciones), lo que permite a su vez que estos directorios sean compartidos por otros contenedores o por la máquina anfitrión.

```
VOLUME ["/data"]
```

8) EXPOSE: permite exponer los puertos TCP/IP por los que se pueden acceder a los servicios del contenedor. Por ejemplo: puerto 22 para SSH, 80 para HTTP o 3306 para MySQL. Esta instrucción expone los puertos para ser usados por el propio contenedor. En el caso de querer hacerlos accesibles desde fuera del contenedor, se deberá utilizar el *flag* -p (publish) durante la ejecución.

```
EXPOSE <puerto> [<puerto>...]
```

9) CMD: similar al RUN, permite la ejecución de instrucciones pero con la diferencia de que este no se ejecuta al construir la imagen, sino que lo hace en el momento de ejecución. Para que el fichero Dockerfile sea válido, solo puede contener una única instrucción CMD.

```
#Formato exec parametrizado
CMD ["ejecutable","param1","param2"]

#Definición de los parámetros utilizados por ENTRYPOINT
CMD ["param1","param2"]

#Formato shell
CMD comando param1 param2
```

10) ENTRYPOINT: permite sobreescribir el binario por defecto (/bin/sh -c) que ejecutará CMD.

```
#Formato exec parametrizado
ENTRYPOINT ["ejecutable", "param1", "param2"]

#Formato shell
ENTRYPOINT comando param1 param2
```

3.4.2. Fichero Dockerfile

Volviendo al ejemplo, la definición del fichero Dockerfile se ha establecido dentro de la carpeta principal **HelloDevOps** del proyecto, por tanto, las referencias a ficheros partirán de esta carpeta, y quedarán de la siguiente forma:

```
#Imagen de partida tomcat en su versión 8.5
FROM tomcat:8.5-jre8

#Copiamos el contenedor WAR generado de nuestra aplicación dentro la carpeta webapps.
COPY target/hellodevops.war  /usr/local/tomcat/webapps/
```

En este momento tenemos todo lo necesario para la creación de una imagen docker.

Para crear una imagen a partir del fichero Dockerfile, nos situaremos en el directorio donde se encuentra el dichero y ejecutaremos:

```
#Instrucción de creación de imágenes:
#docker build -t <nombre_de_la_imagen> <ruta_al_fichero_Dockerfile>

$ docker build -t hellodevops:v1.
```

- **docker build:** Es el comando utilizado para la creación de una imagen docker.

- **-t hellodevops:v1:** Indicamos mediante el *flag* **-t** el nombre de la imagen y opcionalmente un *tag* en formato «nombre:tag».

- .: Indica que el fichero Dockerfile se encuentra en el directorio donde nos encontramos.

Al finalizar la construcción de nuestra imagen docker se mostrará el resultado del proceso, y en el caso de ser exitoso acompañará con el identificador de nuestra nueva imagen.

```
Successfully built 94af2f7c9151
```

Para verificar que la imagen se ha creado correctamente, usaremos la instrucción **docker images**:

```
$ docker images
REPOSITORY   TAG     IMAGE ID     CREATED      SIZE
hellodevops  v1   94af2f7c9151  4 minutes ago 332.8 MB
tomcat  8.5-jre8  fd9c13f14ae6  11 hours ago  332.6 MB
```

Dentro del resultado deberíamos encontrar la imagen creada junto con la dependencia de la imagen del Apache Tomcat 8.5, que podremos incluir en futuros proyectos sin la necesidad de volver a descargarlo.

3.4.3. Ejecución de la aplicación mediante contenedor Docker

Para iniciar un contenedor es tan sencillo como usar el comando **docker run**. Si la imagen no existe en el equipo local, Docker intentará buscarla en el registro público de Docker. Es importante tener en cuenta que los contenedores están diseñados para detenerse cuando el comando ejecutado en su interior termina.

```
Instrucción para la ejecución de contenedores Docker:
#docker run [OPCIONES] IMAGEN[:TAG|@DIGEST] [COMANDO] [ARG...]

$ docker run -p 8080:8080 -d hellodevops:v1
```

- **docker run:** Comando utilizado para la ejecución de contenedores a partir de una imagen.

- **-p 8080:8080:** Indicamos mediante el *flag* -p <puerto_local>:<puerto_en_el_contenedor> la relación de puertos que se establecerá para que nuestra aplicación sea accesible desde la máquina anfitrión. En este caso, la relación será directa y el puerto 8080 local referenciará al puerto 8080 dentro del Apache Tomcat del contenedor docker.

- **-d:** Indicamos mediante el *flag* -d que el contenedor se ejecutará en segundo plano; de no ser así, la duración de la aplicación estaría unida a la sesión de nuestro terminal.

- **hellodevops:v1:** Es el nombre de la imagen con la cual queremos crear el contenedor. También podemos usar el id pero el nombre es más fácil de recordar.

En este momento, mediante el comando docker ps verificaremos que nuestro contenedor está activo, con la asignación de puertos establecida.

```
$ docker ps
CONTAINER ID       IMAGE             COMMAND            CREATED           STATUS
f37315e8cf14       hellodevops:v1    "catalina.sh run"   9 minutes ago     Up 9 minutes

PORTS                   NAMES
0.0.0.0:8080->8080/tcp  pensive_jennings
```

Comprobamos el correcto funcionamiento de la aplicación web dockerizada mediante la URL local: http://<my.servlet.host>:8080/hellodevops/.

3.4.4. Detención de la aplicación mediante contenedor Docker

Para detener un contenedor docker disponemos de dos maneras distintas:

1) docker stop: detiene un contenedor en ejecución enviando primero una señal SIGTERM a la aplicación principal dentro del contenedor para luego, después de un tiempo de gracia, enviar una señal SIGKILL:

```
$ docker stop f37315e8cf14
f37315e8cf14
```

2) docker kill: detiene un contenedor en ejecución enviando directamente la señal SIGKILL o la señal indicada mediante --signal a la aplicación principal dentro del contenedor.

```
$ docker kill f37315e8cf14
f11f5cc90e84
```

La ejecución de docker stop intenta acabar con la aplicación de manera amistosa mediante una señal estándar POSIX, mientras que docker kill mata el proceso. La recomendación de uso será siempre dentro de lo posible la utilización del comando docker stop.

3.5. Jenkins

Esta herramienta nos ofrece un componente fiable para ejercer de servidor de integración continua. El propósito principal para su uso no es solo por su capacidad de automatización de procesos, sino que ofrece una plataforma orientada al seguimiento del proceso de ejecución de todas estas tareas.

3.5.1. Instalación mediante imagen dockerizada

Conociendo el funcionamiento de los contenedores, podemos optar por la utilización de la imagen oficial dockerizada de Jenkins suministrada por Docker Hub; este contenedor nos proporcionará el servicio de automatización de una manera sencilla y rápida.

```
#docker run [OPCIONES] IMAGEN [COMANDO] [ARGUMENTOS...]
$ docker run -p 8090:8080 -p 50000:50000   \
-v <ubicación_fichero_docker.sock>:/var/run/docker.sock -v $(which docker):/usr/bin/docker jenkins
```

- **docker run:** Comando utilizado para la ejecución de contenedores a partir de una imagen.

- **-p 8090:8080 -p 50000:50000:** Esta aplicación será accesible a través de los puertos 8090 ofreciendo la interfaz gráfica web y 50000 exponiendo la API, para ejecución de operaciones remotas.

- **-v <ubicación_fichero_docker.sock>:/var/run/docker.sock -v $(which docker):/usr/bin/docker:** Dado que este contenedor requiere el uso de los comandos docker, mapearemos el socket y el propio binario docker para que este pueda ser usado por el contenedor.

- **jenkins:** Nombre de la imagen del contenedor en Docker Hub.

En el caso de precisar persistencia de nuestras configuraciones, mapearemos mediante la opción «-v» el *path* del *home* de Jenkins dentro del contenedor con una ubicación conocida de nuestro ordenador.

```
$ docker run -p 8090:8080 -p 50000:50000 -v <ubicación_local>:/var/jenkins_home  \
-v <ubicación_fichero_docker.sock>:/var/run/docker.sock -v $(which docker):/usr/bin/docker jenkins
```

Nota

La ubicación del fichero docker.sock en la mayoría de las distribuciones Linux es /var/run/docker.sock.

3.5.2. Instalación mediante fichero WAR oficial

La instalación de Jenkins puede realizarse mediante el despliegue del fichero WAR que encontraremos disponible para su descarga en la página oficial de Jenkins. Para ello, disponemos de dos formas:

- Ubicando el fichero WAR en el directorio webapps de nuestro tomcat.

- Ejecutando **java -jar jenkins.war--httpPort=8090 start** donde hayamos guardado nuestro fichero oficial.

En ambos casos la aplicación será accesible mediante la URL http:/ <my.jenkins.host>:8090.

3.5.3. Configuración de Jenkins

A la hora de montar un entorno de integración continua se requerirá dotar a la plataforma de las herramientas necesarias para la ejecución de todos los procesos automatizados. Jenkins ya incorpora muchos mecanismos que tan solo deberán ser configurados, como el servidor de correo, pero otros deberán instalarse (Maven y *plugins*).

Configuración servidor de correo saliente

Jenkins notificará a los desarrolladores los sucesos importantes producidos durante cada una de sus operaciones.

Para configurar el servidor de correo saliente a través de la interfaz gráfica de Jenkins, nos dirigiremos utilizando los diferentes menús a *Administrar Jenkins* -> *Configurar el Sistema* y al apartado dedicado a la Notificación por correo electrónico.

Figura 62. Detalle de menú de configuración de Jenkins

Fuente: Joan Caparrós.

El servidor utilizado puede ser local o hacer referencia a un servidor smtp ofrecido por terceros. A continuación mostramos un ejemplo de configuración de la plataforma Jenkins para la utilización de Gmail como plataforma de envío de correos.

Figura 63. Detalle de la configuración del servidor de correo saliente de Jenkins

Notificación por correo electrónico

Servidor de correo saliente (SMTP)	smtp.gmail.com
sufijo de email por defecto	@gmail.com
☑ Usar autenticación SMTP	
Nombre de usuario	<Nombre Usuario/Servidor>
Contraseña	••
Usar seguridad SSL	☑
Puerto de SMTP	465
Dirección para la respuesta	<email respuesta>
Juego de caracteres	UTF-8

☐ Probar la configuración enviando un correo de prueba

Fuente: Joan Caparrós.

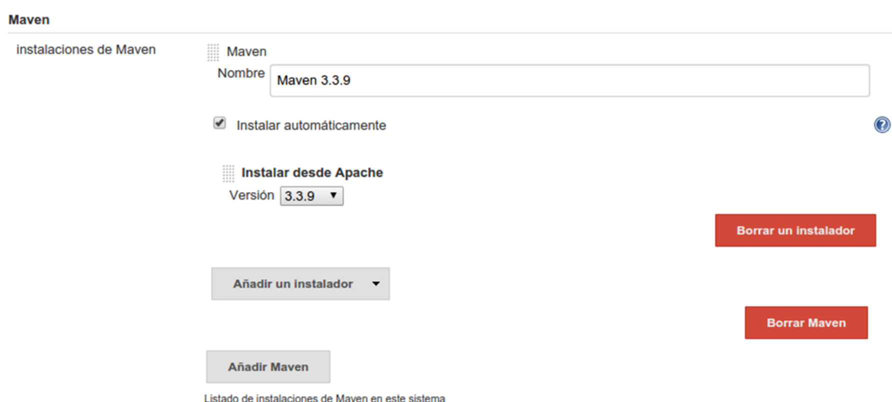
Una vez configurado, podremos enviar un correo de prueba para asegurar la correcta configuración del servidor.

Instalación Maven en Jenkins

Debido a que Jenkins utilizará los comandos mvn para la compilación y construcción de nuevas imágenes, será un requisito que Maven esté instalado dentro de nuestro contenedor Jenkins.

Para instalar Maven a través de la interfaz gráfica de Jenkins, nos dirigiremos utilizando los diferentes menús a *Administrar Jenkins -> Administrar Plugins* y en el apartado dedicado a la configuración de Maven pondremos un nombre para la versión elegida (por ejemplo: «Maven 3.3.9») y clicaremos en *Instalar automáticamente*.

Figura 64. Detalle de la configuración de Maven en Jenkins



Fuente: Joan Caparrós.

A partir de este momento, Maven ya estará disponible para ser incorporado en la *pipeline* del proceso de construcción de nuestros proyectos.

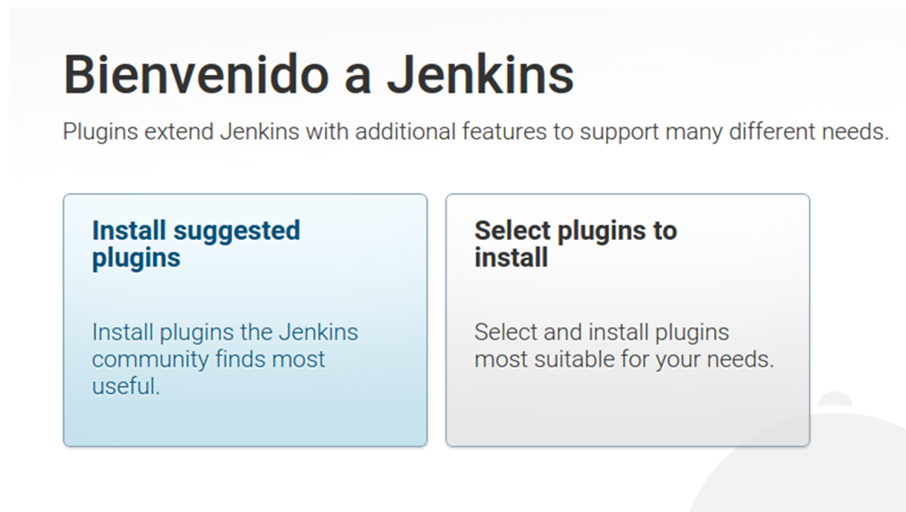
Instalación de *plugins* necesarios

Para poder realizar las acciones propuestas en este caso de uso, se requerirá la instalación de distintos *plugins* de Jenkins.

Jenkins pone a disposición la capacidad de instalar los *plugins* más usados por la comunidad durante la fase de configuración de la aplicación.

Así pues, podremos adquirir la mayoría de los *plugins* necesarios clicando sobre «Install suggested plugins» dentro de la página de bienvenida de Jenkins.

Figura 65. Página de bienvenida de Jenkins



Fuente: Joan Caparrós.

En el caso de poseer una instalación previa de Jenkins o de no querer instalar la totalidad de los *plugins* sugeridos, se deberán instalar como mínimo los siguientes:

- **Maven Plugin:** Necesario para dotar a nuestro servidor de integración continua de las herramientas para construir y testar nuestro proyecto.
- **Git Plugin:** Permite la interacción con Git, por tanto, con plataformas como GitHub.
- **Mailer Plugin:** Permite configurar las notificaciones de correo electrónico que la plataforma enviará con los resultados de la compilación y testeo.
- **Workspace Cleanup Plugin:** Elimina el espacio de trabajo antes de la construcción asegurando que no haya ninguna interferencia de archivos entre construcciones.

Instalación del *plugin* CloudBees Docker Build and Publish en Jenkins

Construir una imagen docker en Jenkins requiere el uso de los binarios de docker. Hemos aprendido cómo construir la imagen durante la dockerización de la aplicación, y estos podrían ser perfectamente incluidos en una de las diferentes etapas que podríamos definir en un *pipeline* de nuestro proceso completo de automatización, pero Jenkins ofrece herramientas para facilitar la construcción y publicación de imágenes docker en un Registry Docker Hub, sin tener que escribir ninguna línea de código. En este caso de uso utilizaremos el *plugin* CloudBees Docker Build and Publish.

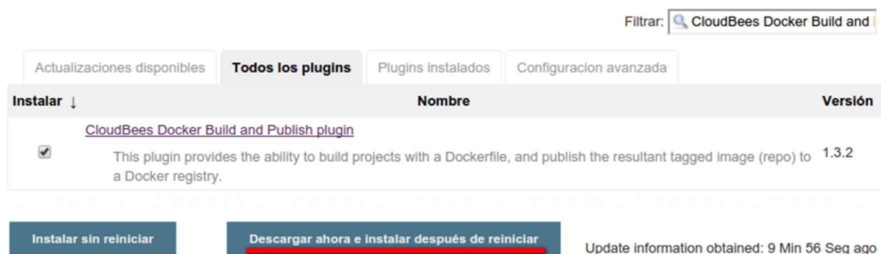
Para instalar el *plugin* nos dirigiremos utilizando los diferentes menús a *Administrar Jenkins -> Administrar Plugins*.

Figura 66. Detalle del menú de administración de Jenkins

Fuente: Joan Caparrós.

Dentro de la administración de *plugins* nos situaremos en la pestaña *Todos los plugins*, utilizaremos el filtro para indicar el *plugin* en cuestión, *CloudBees Docker Build and Publish*, marcaremos el *checkbox* de instalación y procederemos clicando sobre *Descargar ahora e instalar después de reiniciar*.

Figura 67. Detalle de la instalación y configuración del *plugin* CloudBees Docker Build and Publish



Fuente: Joan Caparrós.

3.6. Registry Docker Hub

Para completar el ciclo de nuestras imágenes docker, construiremos nuestro propio registro de imágenes de Docker. Así, seremos capaces de enviar nuestras aplicaciones que se encuentran en contenedores a la nube, teniendo un punto donde almacenar y compartir estas aplicaciones con otros usuarios.

Docker ofrece dentro de sus productos un punto centralizado de almacenamiento seguro de imágenes, que hasta el momento lo hemos usado para construir nuestras imágenes, Docker Hub.

Aunque Docker Hub nos permite disponer de solo un repositorio privado y debe ser considerado un servicio de pago para el almacenamiento de aplicaciones privadas, como sucede con GitHub, Docker cubre las necesidades de los desarrolladores entregando las herramientas necesarias para que cualquiera pueda construirse su propio repositorio privado en la nube.

3.6.1. Registro mediante imagen dockerizada

Podemos optar por la utilización de la imagen oficial dockerizada de Registry Docker Hub suministrada por Docker Hub. Este contenedor nos proporcionará el registro de nuestras propias imágenes generadas mediante docker.

Iniciaremos nuestro *registry*:

```
$ docker run -p 5000:5000 -v <ubicación_local>:/tmp/registry-dev registry
```

- **docker run:** Comando utilizado para la ejecución de contenedores a partir de una imagen.
- **-p 5000:5000:** Esta aplicación será accesible a través del puerto 5000, no dispone de interfaz gráfica web.
- **-v <ubicación_local>:/tmp/registry-dev:** Con el uso del *flag* «-v» dotamos de persistencia al contenedor ubicando de forma local las imágenes subidas a nuestra aplicación.
- **registry:** Nombre de la imagen del contenedor en Docker Hub.

Para probar y entender el funcionamiento del registro de imágenes de Docker, realizaremos una pequeña prueba donde descargaremos una imagen desde Docker Hub a nuestra máquina local para luego enviarla a nuestro propio Registry Docker Hub.

Descargamos la imagen hello-world https://hub.docker.com/_/hello-world/ a nuestra máquina:

```
#docker pull [OPCIONES] NOMBRE[:ETIQUETA|@DIGEST]
$ docker pull hello-world
```

- **docker pull:** Comando utilizado para la descarga de imágenes desde un registro de imágenes de Docker.
- **hello-world:** Nombre de la imagen del contenedor en Docker Hub.

Para que nuestra imagen local pueda ser subida a otro registro de imágenes, deberemos etiquetarla.

```
#docker tag IMAGEN[:TAG] IMAGEN[:ETIQUETA]
$ docker tag hello-world <my.registry.host>:5000/myhello-world
```

- **docker tag:** Comando utilizado para etiquetar aplicaciones en contenedores indicando el nuevo registro de imágenes, nombre y etiqueta del propio contenedor.

- **hello-world:** Nombre de la imagen del contenedor en Docker Hub.

- **<my.registry.host>:5000/myhello-world:** Asignación del nombre y registro de contenedores para nuestra imagen local hello-world.

Comprobamos que la imagen con su nueva etiqueta aparece dentro de nuestras imágenes locales.

```
$ docker images
REPOSITORY                                    TAG              IMAGE ID
hello-world                                   latest           c54a2cc56cbb
<my.registry.host>:5000/myhello-world         latest           c54a2cc56cbb
```

Si la imagen aparece, procederemos a subirla hacia nuestro propio registro de imágenes Docker.

```
#docker push [OPCIONES] NOMBRE[:ETIQUETA]
$ docker push <my.registry.host>:5000/myhello-world
The push refers to a repository [<my.registry.host>:5000/myhello-world]
a02596fdd012: Pushed
latest: digest: sha256:a18ed77532f6d6781500db650194e0f9396ba5f05f8b50d4046b294ae5f83aa4 size: 524
```

- **docker push:** Comando utilizado para la subida de imágenes hacia un registro de imágenes de Docker.

- **<my.registry.host>:5000/myhello-world:** Nombre y ubicación de la imagen del contenedor en Docker Hub.

Para asegurar que la imagen se encuentra dentro del registro de imágenes creado, verificaremos que esta se encuentre en el registro y procederemos a descargarla.

```
#Existen distintos comandos para hacer llamadas a la API del registro de imágenes Docker,
dependiendo de la versión de este estarán o no disponibles, es por eso que mostraremos dos
métodos para recuperar las imágenes remotas:
#Mediante instrucciones docker
$ docker search <my.registry.host>:[PUERTO]/library

#Mediante llamadas a la API
$ curl <my.registry.host>:5000/v2/_catalog
{"repositories":["myhello-world"]}
```

```
$ docker pull <my.registry.host>:5000/myhello-world
Using default tag: latest
latest: Pulling from myhello-world
Digest: sha256:a18ed77532f6d6781500db650194e0f9396ba5f05f8b50d4046b294ae5f83aa4
Status: Image is up to date for <my.registry.host>:5000/myhello-world:latest
```

3.6.2. Contribuir con nuestra imagen a la comunidad

Hemos hablado anteriormente de la plataforma Docker Hub Registry como un punto de almacenamiento de imágenes públicas de Docker listas para su uso. Detrás de todas estas imágenes hay un trabajo duro por parte de los desarrolladores para que nosotros podamos disfrutar de todos estos contenedores con tan solo invocar su ejecución.

Para compartir nuestro trabajo con la comunidad, podemos hacer accesible nuestras imágenes dentro de dicha plataforma mediante los siguientes pasos:

1) Visitar la página de Docker Hub.

2) Registrarnos dentro de la plataforma mediante el formulario de ***Sign Up*** y verificar la cuenta.

3) Entrar en la plataforma (***Log In***).

4) Crear nuestro repositorio, indicando nombre, descripción y visibilidad (pública o privada).

5) A partir de este momento tendremos disponible nuestro primer repositorio en la nube.

6) Para subir nuestro contenedor al repositorio creado en Docker Hub, deberemos autenticar nuestra máquina local mediante la instrucción:

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID,
head over to https://hub.docker.com to create one.
Username: <usuario>
Password: <contraseña>
Login Succeeded
```

7) Etiquetamos nuestra imagen local <*imagen.local*>, para ser albergada dentro del repositorio con nombre <*usuario*>/<*nombre.repositorio*>.

```
#docker tag IMAGEN[:TAG] IMAGEN[:ETIQUETA]
$ docker tag <imagen.local> <usuario>/<nombre.repositorio>
```

8) Subimos la imagen al repositorio de Docker Hub:

```
#docker push [OPCIONES] NOMBRE[:ETIQUETA]
$ docker push <usuario>/<nombre.repositorio>
The push refers to a repository [docker.io/<usuario>/<nombre.repositorio>]
a02596fdd012: Mounted from library/<imagen.local>
latest: digest: sha256:a18ed77532f6d6781500db650194e0f9396ba5f05f8b50d4046b294ae5f83aa4 size:
```

En este momento nuestra imagen ya estará disponible para todos los usuarios de la comunidad Docker simplemente con ejecutar la respectiva instrucción *docker pull*.

Vale la pena pararse e investigar todas las opciones que la plataforma Docker Hub nos ofrece, ya que no se limita solo a albergar sino que permite la creación de organizaciones con sus respectivos equipos de trabajo y derivar la automatización de construcción de imágenes mediante cuentas GitHub o Bitbucket relacionadas.

3.7. Proceso de automatización completo (*pipeline*)

El proceso de generación de contenedores funcionales con nuestra aplicación será el objetivo principal de nuestro *pipeline*. Hasta el momento hemos realizado cada uno de los pasos de forma manual y faltaría dotar a nuestro proceso de creación de imágenes de una cierta automatización para realizar cada una de las acciones.

Jenkins aparece en nuestro esquema como plataforma encargada de la automatización y del monitoreo, pasando a ser nuestro **sistema de integración continua**, que se encargará de todas las tareas definidas en nuestro proceso de automatización.

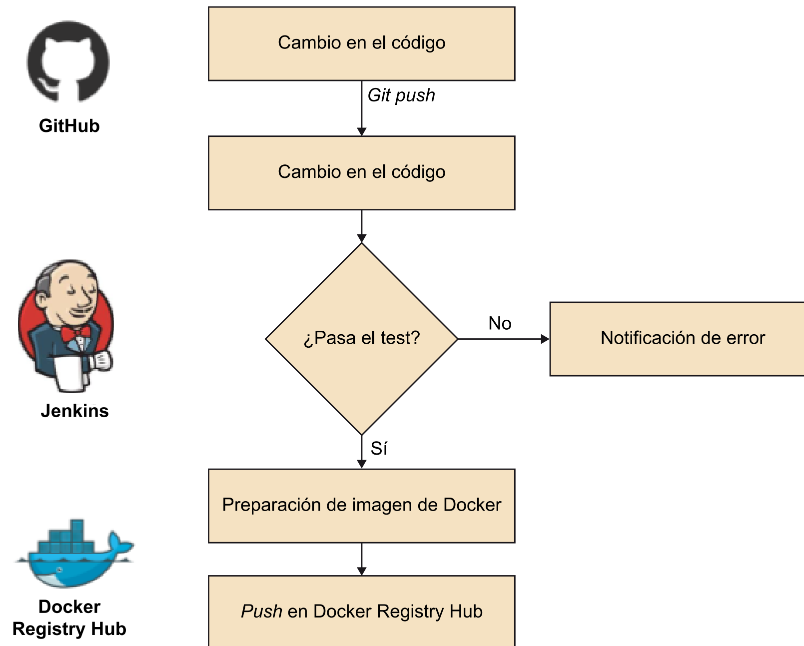
Para nuestro *pipeline* definido detectaremos las siguientes tareas:

- Compilar el código cada vez que este sufra un cambio.
- Realizar las pruebas unitarias definidas.
- Construir mediante Maven nuestra aplicación.
- Realizar la imagen del contenedor con nuestra aplicación.

- Registrar la imagen generada en nuestro registro de imágenes de Docker.

- Avisar a los desarrolladores, en el caso de que alguno de los pasos establecidos no finalice correctamente, para efectuar los cambios en el código que sean necesarios.

Figura 68. Tareas definidas en la Jenkins *pipeline*

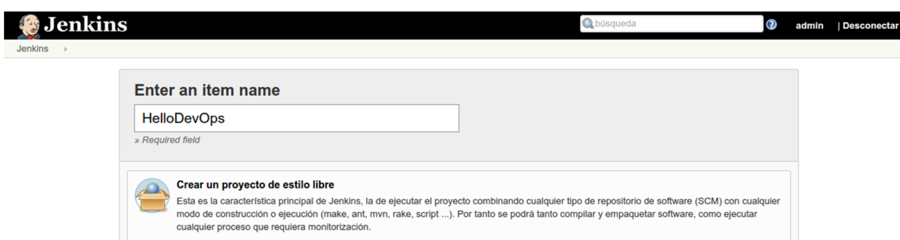


Fuente: Joan Caparrós.

Para iniciar nuestro proyecto en Jenkins, se deberá crear una **Nueva Tarea**, disponible en el menú lateral derecho de la plataforma. En el caso de no disponer de ninguna tarea anterior, Jenkins indicará en su parte central un enlace directo para empezar con la creación de la primera tarea.

Definiremos un nombre de proyecto interno para la plataforma (*"HelloDevOps"*) y elegiremos el método de *Crear un proyecto de estilo libre*, ideal para la ejecución de proyectos provinientes de cualquier tipo de repositorio de software, con cualquier método de construcción o ejecución (make, ant, mvn, scripts...).

Figura 69. Creación de la tarea *HelloDevOps*



Fuente: Joan Caparrós.

Una vez dentro de la pantalla de configuración del proceso de automatización definiremos las propiedades **Generales**, que definirán aquellas propiedades transversales a todas las operaciones realizadas. Para nuestro ejemplo dejaremos todos estos campos en blanco.

A continuación definiremos la configuración para cada una de las acciones descritas durante la definición del *pipeline*.

3.7.1. Compilar el código cada vez que este sufra un cambio

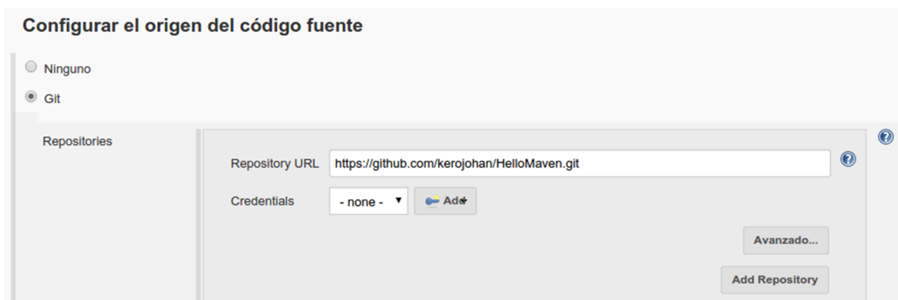
Dado que nuestro proyecto debe generar una nueva imagen cada vez que se incorporen cambios a la rama principal (*master*) de nuestro proyecto, deberemos establecer nuestro repositorio Git como fuente de origen.

Configurar el origen del código fuente

- Seleccionar **Git**.

- Introducir en **Repository URL** https://github.com/<usuario>/HelloDevOps.git manteniendo las credenciales en **none** (si vuestro repositorio es público).

Figura 70. Configuración de la tarea HelloDevOps



Fuente: Joan Caparrós.

Existen diferentes acciones que pueden hacer que nuestro proyecto Jenkins se ejecute de manera automática. Estos representan los **Disparadores de ejecuciones**, y son los siguientes:

- **Lanzar ejecuciones remotas:** Permite la ejecución del *pipeline* accediendo a una URL especial (JENKINS_URL/job/HelloDevOps/build?token=TOKEN_NAME o /buildWithParameters?token=TOKEN_NAME), útil para integrar dentro de *scripts*. Un uso habitual es la llamada de esta URL desde los *scripts* (*hooks*) ejecutados cuando se realizan cambios dentro de un repositorio de versiones.

- **Construir después de la construcción de otros proyectos:** Permite definir como disparadores de ejecución la finalización de construcción de otros proyectos.

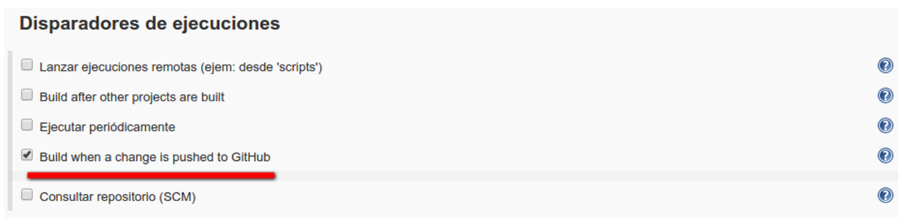
- **Ejecutar periódicamente:** Permite la definición de tiempos de ejecución del proyecto definidos en formato cron con algunas pequeñas adaptaciones (ejemplo: H/15 * * * * - dispara la ejecución del *pipeline* cada 15 minutos).

- **Construir cuando un cambio es subido (*pushed*) al GitHub:** Permite relacionar la ejecución del proyecto Jenkins a cada comando PUSH recibido por el repositorio de versiones GitHub definido en la sección «Configurar el origen del código fuente».

- **Consultar repositorio (SCM):** Esta opción complementa a la anterior y permite la ejecución de comprobaciones del repositorio de versiones en el caso de que no se disponga de *scripts* capaces de disparar ejecuciones. Se definen en el formato cron adaptado de Jenkins.

Para nuestro proyecto estableceremos la opción **Construir cuando un cambio es subido (*pushed*) al GitHub**.

Figura 71. Configuración de la tarea HelloDevOps



Fuente: Joan Caparrós.

Realizar las pruebas unitarias definidas

El lanzamiento de las pruebas unitarias define un proceso de ejecución, que lanzará un comando maven con parámetro «test» (mvn test). Los resultados de la ejecución definirán si el proceso continúa y esto indicará que nuestro proyecto funciona adecuadamente.

Ejecutar:

- En el seleccionable elegir **Ejecutar tareas 'maven' de nivel superior**.

- **Versión de Maven**: Elegiremos mediante el listado desplegable la versión de Maven definida en el apartado Instalación Maven en Jenkins de este caso de uso.

- **Goals**: test.

Figura 72. Configuración de la tarea HelloDevOps

Fuente: Joan Caparrós.

Construir mediante Maven nuestra aplicación

El proceso de construcción de nuestro proyecto es indispensable para la generación del contenedor WAR en la carpeta target, que alimentará las siguientes fases de nuestro *pipeline*. De modo similar a la ejecución de las pruebas unitarias, lanzarán un proceso maven con parámetro «package».

Ejecutar:

- En el seleccionable elegir **Ejecutar tareas 'maven' de nivel superior**.
- **Versión de Maven**: Elegiremos mediante el listado desplegable la versión de Maven definida en el apartado Instalación Maven en Jenkins de este caso de uso.
- **Goals**: package.

Figura 73. Configuración de la tarea HelloDevOps

Fuente: Joan Caparrós.

3.7.2. Construcción y publicación de la imagen

Realizar la imagen del contenedor y posterior registro en cualquier registro de imágenes de Docker será una tarea fácil mediante el *plugin* instalado **Docker Build and Publish**. Este ejecutará todos los comandos docker indicados en el apartado de *Dockerización y Registry Docker Hub* simplemente indicando el repositorio destino y la URL del repositorio de imágenes docker.

Para la subida en nuestro repositorio de imágenes Docker, hemos de ejecutar:

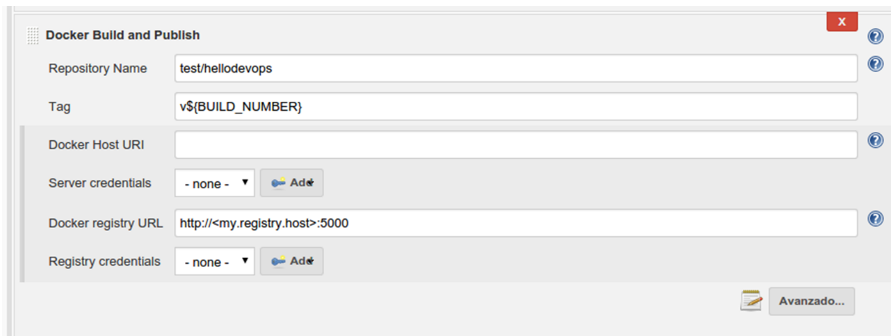
- En el seleccionable elegir **Docker Build and Publish**.
- **Repository Name:** «test/hellodevops», definiendo un registro «test» donde se ubicará nuestra imagen con nombre «hellodevops».

- **Tag:** La etiqueta de nuestra imagen definirá la versión del contenedor; para ello concatenamos «v» con una variable interna con la versión actual de construcción de Jenkins (${BUILD_NUMBER}).

- **Docker Host URI:** Puede dejarse en blanco para utilizar los valores de entorno docker por defecto (normalmente unix:///var/run/docker.sock o tcp://127.0.0.1:2375).

- **Docker registry URL:** http://<my.registry.host>:5000/.

Figura 74. Configuración de la tarea HelloDevOps



Fuente: Joan Caparrós.

Para la subida en la nube (Docker Hub), hemos de ejecutar:

- En el seleccionable elegir **DockerBuild and Publish**.

- **Repository Name:** «<usuario.docker.hub>/hellodevops», definiendo el repositorio remoto que previamente habremos creado.

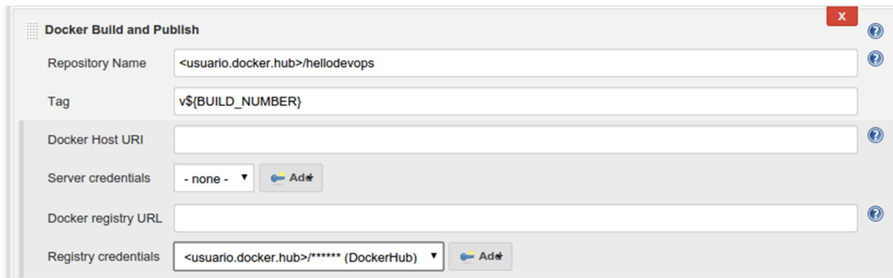
- **Tag:** La etiqueta de nuestra imagen definirá la versión del contenedor; para ello concatenamos «v» con una variable interna con la versión actual de construcción de Jenkins (${BUILD_NUMBER}).

- **Docker Host URI:** Dejaremos los valores en blanco para el uso de las variables por defecto de la plataforma.

- **Docker registry URL:** Podemos dejar el campo en blanco, ya que por defecto este contendrá el valor que nos interesa https://index.docker.io/v1/.

- **Registry credentials:** Indicaremos un nuevo registro con las credenciales de nuestro usuario dentro de la plataforma Docker Hub.

Figura 75. Configuración de la tarea HelloDevOps



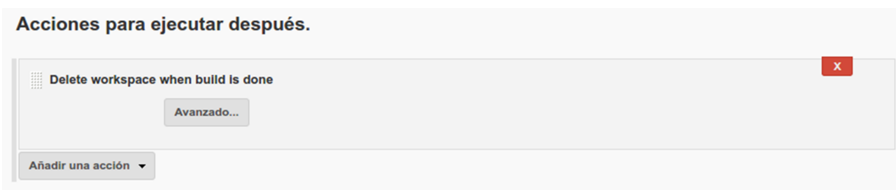
Fuente: Joan Caparrós.

Una vez compilado el proyecto y publicada la imagen en el registro de imágenes, liberaremos el espacio mediante la definición de un último proceso de ejecución.

Acciones para ejecutar después:

- en el seleccionable hemos de elegir **Delete workspace when build is done.**

Figura 76. Configuración de la tarea HelloDevOps



Fuente: Joan Caparrós.

Esto eliminará la carpeta con el proyecto al terminar el proceso.

3.7.3. Notificación de errores durante el proceso de automatización

Si se ha configurado, Jenkins enviará un correo electrónico a los destinatarios especificados cuando se produzca un determinado evento importante.

Es importante denotar que cada construcción fallida enviará un nuevo correo electrónico y que también se enviarán en casos de superación de problemas tras una construcción fallida y construcciones inestables.

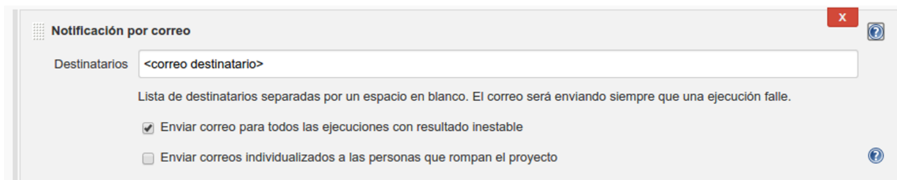
Acciones para ejecutar después:

- En el seleccionable, elegir **Notificación por correo**.

- **Destinatarios**: Permite especificar múltiples destinatarios a los cuales se les enviará la copia completa de la salida por consola de procesos fallidos. Las direcciones de correo serán separadas por un espacio en blanco.

- Se deberá especificar qué usuarios deberán recibir los errores, o todos los desarrolladores o aquellos que hayan realizado el último commit.

Figura 77. Configuración de la tarea HelloDevOps



Fuente: Joan Caparrós.

Para ejecutar por primera vez nuestro proceso de integración continua, podremos o bien introducir algún cambio en nuestro código y hacer *push* en el repositorio de versiones, o bien clicar en la opción del menú lateral *Construir ahora*.

Figura 78. Ejecución de la tarea HelloDevOps

Fuente: Joan Caparrós.

¡Ya tenemos nuestro servidor de integración continua preparado y funcionando!