Proyecto Integrado

Admón. de Sistemas Informáticos en Red

NETWORKING AUTOMATIZADO

Solucionando los problemas de la infraestructura clásica

AUTOR:

Javier Sánchez Páez

TUTOR/ES:

Víctor Montero Malagón

Javier Pastor Cascales

IES Zaidín-Vergeles (Granada), curso 2021-2023

Agradecimientos

A mi familia, porque sin ellos no estaría donde estoy.

A Carlos, por ser mi mentor desde que tengo uso de razón.

A Vicky, por darme la paz y la guerra que necesito para seguir adelante.

A Kyndryl y al equipo de Network & Edge, por enseñarme y confiar en mí.

A los buenos profesores que me han acompañado, por enseñarme y mantenerme motivado.

A mis amigos y amigas, por ser quienes son y por estar siempre ahí.

Abstract

Montar una infraestructura de red "legacy" conlleva varios problemas:

- **Tiempo perdido y dificultad:** Una red es difícil de montar y configurar. Un ingeniero de red tarda aproximadamente un año en abastecer y preparar una red compleja (proxies, routers, compatibilizar infraestructura "legacy") con diferentes localizaciones.

- **Dinero perdido:** No hacer una correcta planificación de los recursos requeridos para nuestra infraestructura puede implicar, entre otras cosas, perder tiempo útil para otras cuestiones y hasta comprar dispositivos que, al tiempo, serán innecesarios.

- **Solución de errores:** Un ingeniero de red pierde alrededor del 50% de su tiempo solucionando problemas que surgen en la red.

- **Métricas perdidas:** El 80% de los ingenieros de red comprueban la efectividad de la red mediante CLI. Además, el 40% de los ingenieros de red afirman que muchos de los problemas que un cliente puede tener con la red no necesariamente es por la red.

- **Escalabilidad:** Ampliar una red ya montada es muy complejo y ésta es propensa a errores que afecten al funcionamiento general de la misma.

- **Seguridad:** Al no tener un control completo de nuestra red podemos estar obviando riesgos de seguridad como, por ejemplo, un ataque que nos inhabilite la red por un firewall mal configurado.

A raíz de estos problemas surge una posible solución, la cual es la **automatización de redes**. De esta forma podemos solucionar los problemas mencionados anteriormente:

- **Tiempo perdido y dificultad**: La red podrá ser abastecida de forma general o específica por dispositivo ahorrando tiempo en tareas repetitivas.

- **Dinero perdido**: Al utilizar una herramienta de inventariado específico de redes (NetBox) podemos ver qué es realmente necesario en nuestra infraestructura y qué podemos omitir, además de poder reparar errores que nos puedan causar pérdidas rápidamente.

- **Solución de errores:** Si tenemos errores podemos hacer "vuelta atrás" (rollbacks), además de poder detectar de dónde viene el error gracias a las **métricas**.

- **Métricas perdidas**: Podemos comprobar la efectividad y flujo de cada uno de los dispositivos de nuestra red y ver si, efectivamente, los fallos que puedan haber pueden ser causados por nuestra infraestructura.

- **Escalabilidad:** Al tener un control de infraestructura centralizado podemos escalar y ampliar nuestra infraestructura fácilmente, ya que aplicaremos una configuración "base" para todos los dispositivos que sean del mismo tipo.

- **Seguridad**: Al aplicar unas configuraciones "base" que incluyan medidas de seguridad podemos tener la certeza de que si lo escribimos bien en la base funcionará bien en entornos de producción.

Palabras clave

(CI, Ansible AWX, Grafana, Kubernetes, GitHub Actions, NetBox)

Contenido

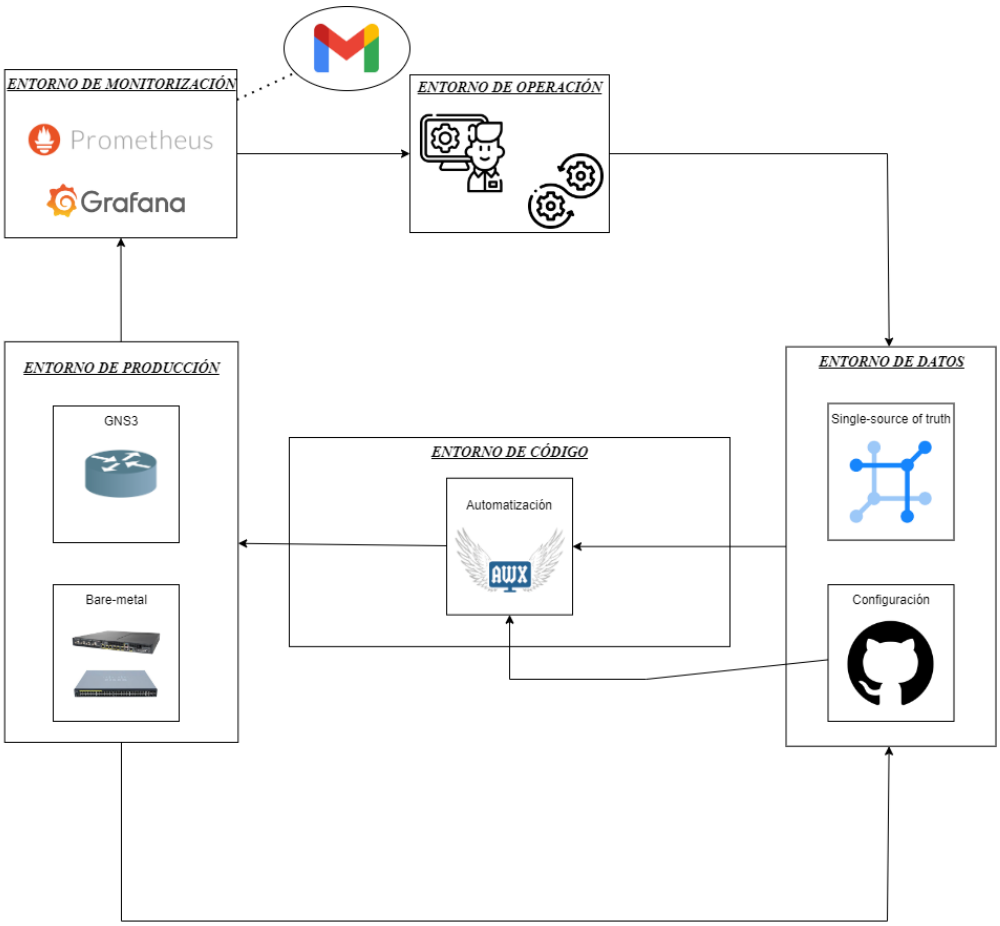
Introducción

A lo largo de los años ha habido una evolución muy marcada en el ámbito de la programación y de la informática en general, aunque siempre se ha obviado la parte más crítica: la **infraestructura de red**.

La infraestructura de red, como ya sabemos, es **difícil** de **operar**, **mantener**, **actualizar** y **reparar**. Lo que se mostrará en este documento es cómo podemos utilizar las novedades del **ámbito de programación** en el campo de las **infraestructuras de redes**.

Objetivos

El objetivo de modernizar la infraestructura de red es ser capaces de montar un entorno **cómodo de operar** para ingenieros de red, a su vez que es lo suficientemente **avanzado** como para añadir cambios seguros. Para ello nos basaremos en el siguiente diagrama:



- **Entorno de operación → entorno de datos**
  El administrador o ingeniero de red se comunicará directa y únicamente con nuestras herramientas de datos (**NetBox** para **inventariado de red** y el repositorio de **GitHub** para **configuraciones y código** que apliquemos). De esta manera **no necesitará involucrarse en todas las fases del despliegue**.

- **Entorno de datos → entorno de código**
  AWX recogerá los dispositivos **guardados** en **NetBox** además de los *playbooks* alojados en **GitHub**.

- **Entorno de código → entorno de producción**
  AWX se **conectará** a los dispositivos de **producción** y le aplicará los **cambios** pertinentes.

- **Entorno de producción → entorno de monitorización**
  Prometheus y **Grafana** recogerán los datos de **monitorización** del entorno de **producción** y de los **equipos** que estemos utilizando para alojar las diferentes herramientas y servicios.

- **Entorno de monitorización → entorno de operación**
  El **administrador** o ingeniero de red **recibirá** los **datos** de monitorización de los dispositivos de red de producción (además de los equipos) para poder **observar fallos** en la infraestructura **fácilmente**.

- **Entorno de producción → entorno de datos**
  En **NetBox** tendremos **inventariados** todos los **dispositivos** de red que utilicemos **en producción**.

Ajustes previos

Para poder desarrollar este proyecto se han utilizado los siguientes equipos:

Hosts	Provider	CPUs	RAM	Versión S.O.	Info extra
ASUS M3500QC	-	8n/16h	16GiB	Windows 11	Ent. Operación
Proliant ML150	-	4n	16GiB	Proxmox	GNS3
K3s-rancher	VirtualBox	4n	8GiB	Ubuntu22.04	AWX/Métricas
Ubuntu SRV	VirtualBox	4n	4GiB	Ubuntu22.04	NetBox
GNS3 VM	Proxmox	2n	8GiB	-	Opcional
Cisco 7200	GNS3	1n	512MiB	IOS 15.2	Virtual

Podemos montar todas las herramientas mencionadas anteriormente de varias formas. En este proyecto utilizaremos **máquinas virtuales** mediante el software **VirtualBox**, aunque podemos montar toda la infraestructura en una máquina local. Los motivos por los que he preferido utilizar máquinas virtuales son los siguientes:

- **Limpieza:** Al utilizar ciertas de estas herramientas en nuestra máquina real podemos tener conflictos.
- **Alternativa más cercana a la realidad**: En un entorno de producción tendremos todas las herramientas montadas en servidores "al uso", no en sistemas operativos domésticos.
- **Protección a errores:** Si por alguna razón se nos corrompe alguno de los entornos/servicios que montaremos podemos utilizar las "*snapshots*" que ofrece VirtualBox para volver a un estado funcional.

Estas máquinas virtuales mencionadas **ejecutarán** el sistema operativo "**Ubuntu Server 22.04**" salvo la de GNS3 que utiliza la versión 20.04:

- **K3s-rancher** ejecutará el **clúster "*mono-nodo*"** de **Kubernetes** (comúnmente llamado **K8s**) junto a AWX, Grafana/Prometheus y Falco.

- **Ubuntu SRV** ejecutará NetBox, PostgreSQL, Apache2 y Redis.

- **GNS3 VM** virtualizará de forma anidada la infraestructura de red que montemos.

En un entorno de producción real omitiremos la máquina virtual de GNS3.

Antes de seguir adelante, es importante mencionar una **diferencia clave** de este clúster de Kubernetes con uno "tradicional". Para ello, vamos a recordar **qué es un clúster de Kubernetes**, qué requisitos necesitamos para ejecutar uno y **qué nos aporta Rancher** (también llamado Rancher K3s o simplemente K3s).

Kubernetes es una **plataforma** "open-source" de **orquestación** y gestión de contenedores que nos servirá para, entre otras cosas, conseguir **alta disponibilidad y escalabilidad**, los cuales son obligatorios para tener un entorno **confiable**.

Kubernetes "**vanilla**" requiere de, como mínimo, **tres hosts**; uno de ellos actuará como el **orquestador**, es decir, organizará las peticiones que le lleguen a través de la API y será con quien nos comunicaremos para agregar, modificar o eliminar los recursos que montemos; los otros dos actuarán como **trabajadores** (**workers**), es decir, serán los encargados de ejecutar los recursos como tal.

La ventaja que nos da Rancher K3s es que **no nos exige un mínimo de equipos**; es decir, podemos montar un clúster en **un solo equipo** (de ahí lo de mono-nodo), el cual actuará de orquestador y de trabajador al mismo tiempo. Este proyecto puede ser lanzado en un clúster "*vanilla*" de Kubernetes si así lo decides, pero para la carga de trabajo que tenemos en este proyecto es **más que suficiente**. Además de esta, otras razones por las que utilizar K3s en lugar de un clúster "*vanilla*" de Kubernetes son:

- **Facilidad** de operabilidad. Dependemos de un único equipo.
- **Ahorro** de recursos. Rancher está altamente optimizado.
- Mismo **resultado**. Podremos acceder a nuestros servicios igual que con K8s.
- **Simplicidad**. Ahorramos posibles problemas en la red del clúster y "troubleshooting".

Rancher es un "**sabor**" de Kubernetes aunque **no el único**, y es que tenemos soluciones como **MicroK8s** (creado por **Canonical**, desarrolladores de Ubuntu), **K0s**, **Minikube**, **Kind**, etc. Estas otras soluciones "**copian**" la misma filosofía de "***single-node cluster***", aunque para mí Rancher lo hace mejor ya que **no necesitas** manejar **comandos específicos** de Rancher: el funcionamiento es exactamente **el mismo** que el de un clúster "vanilla".

Creación del proyecto en GitHub

Para desarrollar este proyecto haremos uso de un repositorio de Git que alojaremos en GitHub. La decisión de utilizar GitHub en lugar de otras opciones como Gitlab o Bitbucket es, principalmente, porque más adelante utilizaremos las llamadas **GitHub Actions**. Además, GitHub es la plataforma más común y utilizada.

Empezaremos yendo a github.com y crearemos una cuenta si no la tenemos. Una vez esté hecha, crearemos un **nuevo repositorio** dándole al botón verde de la columna izquierda llamado "Nuevo Repositorio". Se nos abrirá una nueva página preguntándonos por información como el nombre, propietario, etc. En mi caso lo llamaré "PROYECTO_INTEGRADO", la **rama principal** se llamará "***main***", yo seré el dueño y mi tutor Víctor Montero Malagón tendrá permisos de lectura para poder ver los cambios a medida que vayan surgiendo. Al inicio de este proyecto el repositorio será **privado**, aunque lo podemos cambiar más adelante. El tipo de licencia empleada es la "***GNU General Public License V3***", la cual hace que este proyecto pueda ser modificado, distribuido y utilizado de forma empresarial y privada, siempre y cuando **no se pidan garantías al creador** (en este caso yo) y el código sea **distribuido** con la **misma licencia** que usa este mismo proyecto.

Una vez esté creado, podemos alojarlo en local para hacer los cambios más cómodamente en local con el programa de git. Los pasos a seguir serán los siguientes:

```
# creamos una nueva carpeta en local
$ git init
$ echo "PROYECTO_INTEGRADO" > README.md
# añadiremos un archivo cualquiera para asignar el repositorio remoto
$ git add .
$ git commit -m "First commit"
$ git remote add origin git@github.com:javsanpae/PROYECTO_ASIR.git
$ git push origin main
```

Con esta configuración hecha, **cada vez** que hagamos un "***push***" en el repositorio tan **solo** tendremos que **ejecutar** "`git push`". Hecho esto, podemos seguir adelante con el proyecto.

Configuración de la red de VirtualBox

En este proyecto haremos uso de ciertas "***APIs***" para **conectar** los diferentes **servicios** y entornos que montaremos, por lo que, en caso de que estés utilizando también máquinas virtuales, te **recomiendo** que, en cada de una de las máquinas virtuales, montes **dos tarjetas de red**; una de ellas en tipo "***Adaptador Puente***" con IP **dinámica** (para poder conectarnos a la infraestructura de red) y la otra en tipo "***Adaptador Sólo-Anfitrión***" con direcciones IP **estáticas**. De esta manera, los servidores quedarán con una IP privada fija y **no tendremos que modificar constantemente** los archivos de configuración.

Para hacer estos cambios tendremos que tener nuestras máquinas virtuales **previamente creadas**. Una vez lo estén, iremos al "*Tab*" de "*Red*" de cada uno de los servidores en VirtualBox. Comprobaremos que el **Adaptador 1** (se genera por defecto) esté en modo de red **Adaptador Puente** (en mi caso conectado a una tarjeta inalámbrica Intel AX210). Hecho eso, iremos al **Adaptador 2**, lo **habilitaremos** y lo pondremos en modo de red **Adaptador Sólo-Anfitrión** (en la instalación de VirtualBox se nos debería haber generado una red virtual llamado "*VirtualBox Host-Only Ethernet Adapter*", que tendrá la red *192.168.56.0/24*).

Encendemos ambas máquinas virtuales y **comprobamos** que las dos tarjetas de red están **instaladas** con el comando "`ip a | grep enp0s`" (no hace falta que dé internet, ya que **nosotros nos encargaremos de configurarlas**). En versiones anteriores de VirtualBox y Linux se pueden llamar de distinta manera; por ejemplo, "*eth0*" y "*eth1*".

Empezaremos configurando, por ejemplo, el servidor "K3s-rancher". Para ello, abriremos con el programa "*nano*" el archivo *"/etc/netplan/00-installer-config.yaml"*. Si, **en lugar de Ubuntu Server** estás utilizando **Debian** u otra distribución de Linux, **puede ser** que tengas que modificar otro archivo como el "*/etc/network/interfaces*". Ahí veremos la configuración por defecto que crea el sistema al instalar el sistema operativo. Reemplazaremos el contenido del archivo **por el siguiente**:

```
00-installer-config.yml
---
network:
  ethernets:
    enp0s3:
      dhcp4: true
      nameservers:
        addresses: [8.8.8.8]
    enp0s8:
      dhcp4: false
      addresses: [192.168.56.102/24]
  version: 2
```

En este archivo definimos que, en la interfaz **enp0s3** (la que sale a la red mediante adaptador puente), dejaremos la configuración por defecto con **DHCP habilitado** y las **DNS de Google**. En la interfaz **enp0s8** configuramos que **no vamos a utilizar DHCP** y que la dirección IP será la *192.168.56.102* con máscara *255.255.255.0*. No ponemos valores de DNS porque **esa red no sale a internet**.

Guardamos el archivo y ejecutamos el comando "*sudo netplan try*". Este comando ejecuta un **check sintáctico** al archivo de configuración y nos **confirma** si está **bien compuesto** o si tiene **errores**. Si es correcto no veremos ningún *warning* por pantalla, por lo que ejecutaremos "*sudo netplan apply*" para aplicar los cambios. Podemos volver a listar las tarjetas de red para asegurarnos de que se ha aplicado la configuración.

Ahora pasaremos a la otra máquina virtual (Ubuntu SRV) y copiamos el **mismo archivo** de arriba, esta vez con una **IP distinta** (por ejemplo para esta segunda máquina he configurado la *192.168.56.103*). **Probamos**, **aplicamos** y **hacemos ping** entre las dos máquinas virtuales, deberían tener conexión mutua entre ellas y nuestro host (por defecto viene con la dirección *192.168.56.1*).

Configuración e instalación de Rancher K3s

Como hemos comentado al principio del documento, **Rancher** K3s es una versión de Kubernetes "**single-node**". Esto significa que, aunque Rancher esté muy bien optimizado, **se van a ejecutar más componentes de lo normal en una misma máquina**, por lo que **tendremos en cuenta** los **recursos** de CPU y RAM asignados. Es importante también tener en cuenta el **almacenamiento** que le asignaremos al disco para alojar todos los servicios que montaremos en el clúster (en mi caso con **100GB** he tenido **de sobra** para almacenarlo todo).

Más allá de estos cambios, la **instalación** de **Ubuntu Server 22.04** será la **estándar** mediante el asistente de instalación de la imagen ISO, salvo porque en la configuración de red marcaremos que queremos que la instalación **incluya** un servidor **OpenSSH** para poder acceder **cómodamente** desde nuestro cliente. Llamaré a mi usuario "*jsp*" y el nombre del equipo será "*pi*".

Una vez tengamos instalado Ubuntu Server, **seguiremos** con la **instalación** de **Rancher K3s**. Estando en la terminal de bash del servidor, **ejecutaremos** el comando "`curl -sfL https://get.k3s.io | sh -`". Lo que este comando hará es **almacenar temporalmente** el **script** de instalación y **pasarlo como entrada** a una **terminal de sh**. Cuando la instalación haya finalizado, el output debería salir **parecido** a "`[INFO] systemd: Starting k3s`"

Si intentamos hacer un "`kubectl version -short`" nos **dará error**, puesto que **no tenemos** los **permisos** para poder ejecutar el comando. Para arreglarlo, cambiaremos la **propiedad** del archivo "`k3s.yaml`" con "`sudo chown usuario:grupo /etc/rancher/k3s/k3s.yaml`". Ahora si hacemos un "`kubectl version -short`" nos **deberían** salir las **versiones** del cliente, Kustomize y del server (al momento de redactar este proyecto la versión disponible es la **v1.26.3**).

Una vez hecho esto, vamos a **comprobar** que nuestro Rancher K3s se esté ejecutando con "`kubectl get node`". En caso de que así sea, nos aparecerá nuestro server como **Ready** con el tiempo que lleva activado y su versión.

Para evitar errores que puedan surgir errores más adelante vamos a hacer **dos cambios muy sencillos**; primero, vamos a **copiar** el **contenido** del archivo **k3s.yaml** en **otro archivo** al que llamaremos "***config***" y se **encontrará** en la **carpeta** "***$HOME/.kube***". Esto nos **hará falta más adelante** para evitar errores **al añadir extensiones** de Kubernetes, así que, en una terminal de bash, escribiremos "*cp /etc/rancher/k3s/k3s.yaml $HOME/.kube/config*".

Lo segundo es que en nuestro archivo "$HOME/.bashrc" **vamos a escribir dos nuevas líneas de código**: "*alias k="kubectl"* " y, en la **siguiente línea**, "*export KUBECONFIG=/etc/rancher/k3s/k3s.yaml*".

La primera línea nos servirá para **llamar** al comando **kubectl** únicamente **escribiendo "k"**, lo cual nos ayudará en términos de **eficiencia** (**de aquí en adelante me referiré a kubectl como "k" en los comandos**).

La segunda nos evitará unos **warnings** muy molestos cuando hagamos un "*k get nodes*" o **cualquier comando** en el que **accedamos** a recursos de nuestro **clúster**, ya que por defecto **K3s** toma como archivo de configuración el de **k3s.yaml en lugar del config** que hemos creado hace un momento.

Instalación de Helm

Más adelante en este proyecto **instalaremos un servicio** en nuestro clúster **mediante Helm**. Helm es una **herramienta** de **gestión de paquetes** para Kubernetes que **simplifica** el **despliegue de aplicaciones** en clústeres. Permite instalar, actualizar y personalizar fácilmente aplicaciones **mediante charts** predefinidos. Un chart de Helm es un **paquete** que **contiene** los **recursos** y **configuraciones por defecto** de la aplicación a instalar.

Es ampliamente **utilizado** en entornos de **producción** y empresariales ya que te **simplifica** mucho una instalación que, **de forma manual**, tardarías **mucho tiempo** en asegurar de que todo funcionara bien.

Antes de proceder a la instalación de Helm veremos si lo tenemos instalado con "`sudo snap list`". Lo normal es que no esté instalado, pero en caso de que los estuviera nos saldría en una línea del output. Si no, lo instalaremos con el comando "`sudo snap install helm –classic`". Ejecutaremos un "`helm version`" en la terminal y nos debería salir un output de la versión: en mi caso estamos en la versión v3.10.1. Una vez instalado Helm, podemos continuar.

Instalación de AWX

Vamos a proceder a la **primera instalación** de un **servicio** del proyecto. Esta instalación se hará en el **clúster** de **Rancher**.

Para poder seguir adelante, vamos a hacer un pequeño **repaso** a **qué es Ansible**, cómo funciona y qué son y en qué se **diferencian AWX** y **Ansible Tower**.

Ansible es una **herramienta** de **código abierto** que permite la **gestión** de configuración y la **orquestación** de tareas en entornos distribuidos, lo que la convierte en una solución **eficiente** y muy **potente** para **automatizar** la implementación y **gestión** de **infraestructuras** IT.

Su funcionamiento se divide en cuatro elementos:

- **Playbooks**:
  Un playbook es, básicamente, el "set" de **instrucciones** que queremos ejecutar **contra un determinado host o grupo de hosts**. Un ejemplo de playbook que utilizaremos en este proyecto es, por ejemplo, asignar un "mensaje del día" (o MOTD) en un router Cisco 7200.

- **Inventarios**:
  En un archivo de inventario escribiremos **en qué hosts** vamos a realizar las **tareas** de **automatización**. Dentro de un inventario de Ansible **podemos** establecer **variables**, como por ejemplo con qué **usuario** y qué **clave** entraremos al modo de configuración de un router.
  Un inventario puede estar creado de dos maneras distintas: **Estático** (los hosts se escriben **directamente** en el **archivo**) o **dinámico** (los hosts se reciben **a través** de una API de otro servicio, como por ejemplo **NetBox**)

- **Roles**:
  Los roles en Ansible sirven para **agrupar** los hosts en diferentes **nombres**. Esto puede ser muy útil si, por ejemplo, queremos hacer un cambio en todos los hosts, ya que de otra manera tendríamos que ir uno por uno. También es muy útil para agrupar los hosts útiles (en producción) de los inútiles (desconectados, averiados, en almacén…)

- **Colecciones**:

  Las colecciones de Ansible son **conjuntos** de **roles**, **módulos** y **plugins** que **amplían** y facilitan la reutilización del **contenido** en Ansible. Ofrecen una forma de extender las capacidades de Ansible añadiendo diferentes **plataformas** (por ejemplo, **compatibilidad** para automatizar características de red de Cisco) y hacer una **automatización** más **limpia** y **compatible**.

Teniendo esto en cuenta, **AWX** y **Tower** no son más que "**interfaces gráficas" para Ansible** con ciertas **herramientas** para hacer **CI/CD** (por ejemplo, la asignación de **variables** directamente **en la plataforma** sin necesidad de utilizar el archivo de inventario, guardar contenido sensible en **secrets**, crear flujos de trabajo, tener el **proyecto** de **GitHub** almacenado directamente en él…). La diferencia entre **AWX** y **Tower** es semejante a la que hay entre Fedora/CentOS y Red Hat: esencialmente son **lo mismo**, pero **AWX** es **gratuito** y **libre** de usar; sin embargo, tú eres el responsable de que todo funcione. **Ansible Tower** es de **pago** pero tenemos la **garantía** de que, si algo no funciona, tenemos al **equipo** de **Red Hat** disponible para ayudarnos.

Para este proyecto **no utilizaremos roles** ya que **no es necesario** para una infraestructura tan simple. Sí es importante recalcar cómo vamos a **organizar** los otros tres tipos de código que vamos a utilizar. Para ello, crearemos **tres carpetas** distintas con el nombre de cada tipo: **playbooks** (está en la ruta ***code/awx/***), inventories (estará también en la ruta ***code/awx***) y collections (estará en la **raíz** del proyecto porque si lo almacenamos en una ruta distinta **AWX no** lo **detecta**). En este proyecto son **imprescindibles** las **colecciones** para que el **inventario dinámico** de **NetBox** pueda funcionar (más adelante entraremos en más detalle).

Ahora vamos a lo importante: Levantar el **operador** de **AWX**. El operador es el **set de contenedores** encapsulados en un pod que se encargarán de **ejecutar la instalación**.
Para ello, crearemos un archivo "kustomization.yml" con el siguiente contenido:

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
  - github.com/ansible/awx-
    operator/config/default?ref=2.1.0
    # - awx.yml
images:
  - name: quay.io/ansible/awx-operator
    newTag: 2.1.0
namespace: pi
```

En este archivo le estamos indicando que cree una customización basada en el código de la versión 2.1.0 de AWX, que está alojado en GitHub y que la aplique en el espacio de nombres "pi". Una vez creado el archivo, **levantaremos** el **operador** con el comando "`k apply -k .`". En el output del comando nos **mostrará** que **se han creado muchos tipos de recursos**, así que **esperaremos** unos momentos (alrededor de **dos minutos**).

Vamos a **comprobar** que el **operador** haya sido levantado (importante ahora ejecutar kubectl con "***-n pi***" para referirnos al espacio de nombres que acabamos de crear) con el comando "`k -n pi get pods`". Si nos aparece en **Status: Running podemos seguir adelante**.

Una vez esté ejecutado, vamos a crear un **segundo archivo** llamado "*awx.yml*" con el siguiente **contenido**:

```
apiVersion: awx.ansible.com/v1beta1
kind: AWX
metadata:
     name: pi
spec:
     service_type: nodeport
     nodeport_port: 30080
```

En esta línea estamos creando un **recurso** llamado **AWX** (la herramienta en sí) en el espacio de nombres "***pi***" y que se expondrá al público mediante el puerto *30080* de **nuestro equipo** (en este caso del **servidor**).

Ahora **volvemos** al archivo "*kustomization.yml*" y **descomentaremos** la línea de *awx.yml* en ***metadata***, por lo que se quedaría únicamente en "`- awx.yml`". Cuando lo hayamos **descomentado**, Volvemos a ejecutar "`k apply -k .`"

Para monitorizar cómo se van creando todos los pods necesarios, podemos **ejecutar** el comando "`watch kubectl get pods -n pi`". Este comando, por defecto, **muestra la salida del comando cada 3 segundos**, por lo que podemos ir **viendo** qué **cambios** se van produciendo en los **pods**.

Después de aprox. **siete minutos** (depende de los recursos que tenga el equipo donde hayamos montado el clúster) **veremos** que están **ejecutándose** los siguientes **pods:**

- `awx-operator-controller-manager, 2/2, Running`
- `pi-postgres, 1/1, Running`
- `pi-xxxxxx-xxxxxx, 4/4, Running`

Ahora podemos acceder a la **interfaz** de inicio de sesión **web**. Entramos con la **dirección IP que hayamos configurado** (por ejemplo es *192.168.56.102* en mi caso) y el **puerto** que hayamos configurado en **nodeport**. Para conseguir la **contraseña** (el usuario es admin) introduciremos el **siguiente comando**:

```
K -n pi get secret pi-admin-password -o jsonpath="{.data.password}" | base64 -decode ; echo
```

Este comando nos **decodificará** la clave de admin, que se encuentra **almacenada** en un **secret** llamado "*pi-admin-password*".

Nos saldrá la **contraseña** como **output**. Iniciamos sesión y ya **estaremos dentro** de **AWX**. Para cambiar la contraseña iremos a admin → detalles de usuario → editar → contraseña.

Instalación de Grafana y Prometheus

Pasaremos a la **instalación** del **segundo servicio** en nuestro **clúster de Kubernetes**. Para ello, utilizaremos como base un repositorio ubicado en GitHub llamado "*Kube-Prometheus*". Este **contiene** toda la **configuración** necesaria para **desplegar Grafana y Prometheus** automática y **fácilmente**, además de venir con ciertas **métricas** de **Kubernetes** ya **preconfiguradas**. Para la instalación ejecutaremos los siguientes comandos:

```
$ git clone https://github.com/prometheus-operator/kube-prometheus.git
$ k apply -server-side -f manifests/setup/
$ k apply -f manifests/
$ k -n monitoring delete networkpolicies.networking.k8s.io -all
```

De esta forma **instalaremos** el **operador** de Prometheus y Grafana con todas las **dependencias** necesarias. Al contrario que el de AWX, este se encarga de **levantar** Prometheus y Grafana **autónomamente**, **no necesitamos intervención posterior** a los comandos que ya hemos ejecutado. Sin embargo, **tenemos que sustituir tres archivos** de los que hemos aplicado en el clúster: estos **corresponden** a los **servicios** a los que nos queremos conectar (es decir, alert-manager, Prometheus y Grafana). Podemos o bien **reemplazarlo** en la misma **ejecución** del clúster con el comando "*kubectl edit*" o crear un **nuevo archivo** .yml y **aplicarlo de nuevo** (el clúster identificará y **reconocerá** el nuevo servicio y sabrá que es la **sustitución** de un recurso ya existente). Yo prefiero crear de nuevo los archivos YAML porque, si por algún motivo necesitamos re-desplegar este stack, ya tendremos la configuración del servicio hecha:

Alertmanager-service.yaml

```
--
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/component: alert-router
    app.kubernetes.io/instance: main
    app.kubernetes.io/name: alertmanager
    app.kubernetes.io/part-of: kube-prometheus
    app.kubernetes.io/version: 0.25.0
  name: alertmanager-main
  namespace: monitoring
spec:
  type: LoadBalancer
  ports:
  - name: web
    port: 9093
    targetPort: web
  - name: reloader-web
    port: 8080
  selector:
    app.kubernetes.io/component: alert-router
    app.kubernetes.io/instance: main
    app.kubernetes.io/name: alertmanager
    app.kubernetes.io/part-of: kube-prometheus
```

grafana-service.yaml

```
--
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/component: grafana
    app.kubernetes.io/name: grafana
    app.kubernetes.io/part-of: kube-prometheus
    app.kubernetes.io/version: 9.5.2
  name: grafana
  namespace: monitoring
spec:
  type: LoadBalancer
  ports:
  - name: http
    port: 3000
    targetPort: http
  selector:
    app.kubernetes.io/component: grafana
    app.kubernetes.io/name: grafana
    app.kubernetes.io/part-of: kube-prometheus
```

prometheus-service.yaml

```
--
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/component: prometheus
    app.kubernetes.io/instance: k8s
    app.kubernetes.io/name: prometheus
    app.kubernetes.io/part-of: kube-prometheus
    app.kubernetes.io/version: 2.44.0
  name: prometheus-k8s
  namespace: monitoring
spec:
  type: LoadBalancer
  ports:
  - name: web
    port: 9090
    targetPort: web
  - name: reloader-web
    port: 8080
    targetPort: reloader-web
  selector:
    app.kubernetes.io/component: prometheus
    app.kubernetes.io/instance: k8s
    app.kubernetes.io/name: prometheus
    app.kubernetes.io/part-of: kube-prometheus
```

Estos archivos se **diferencian** de los que vienen por defecto en el apartado de **especificación**: aquí le indicamos que queremos **desplegar** un **servicio** del tipo "**LoadBalancer**" (abre un **puerto externo** en nuestro equipo además de hacer las funciones de balanceador de carga), **a diferencia** de desplegar un servicio "***ClusterIP***" (**expone un puerto internamente**, por lo que **no se puede acceder desde la red**).

Para **aplicar** estas modificaciones utilizaremos el **mismo comando** que utilizamos antes (`k apply -f ./`). Como veremos en la salida del comando, esta vez en lugar de aparecer los recursos como creados aparecerán como **configurados**, ya que mediante los metadatos de los archivos .yaml que hemos creado, Kubernetes ha **reconocido** que ya existían anteriormente y ha **modificado** los que ya había creados. Para ver qué puerto se nos ha asignado, pondremos en la terminal "`k get service -n monitoring`" (en mi caso, por ejemplo, Grafana se ha asignado al puerto del host 31469).

En los comandos que hemos ejecutado anteriormente también **hemos eliminado** las **políticas de seguridad** de Kube-prometheus. Esto es porque, si no, **no nos deja conectarnos a los servicios** desde la red externa al servidor.

Volviendo a Grafana, iniciamos sesión con las credenciales admin@admin y, si queremos, le cambiamos la contraseña. En mi caso dejaré la contraseña admin. Con esto ya tendremos Grafana instalado y preparado para configurar más adelante.

Securizando el clúster de K3s con Falco

En esta sección instalaremos un programa para **securizar** nuestro **clúster** llamado ***Falco***. En resumen, Falco es un "**runtime**" **de seguridad** que se ejecuta en nuestro servidor en segundo plano y que, **mediante logs**, nos **anunciará** de **accesos** externos al servidor, así como de intentos de conexión y extracción de datos. Para instalar Falco utilizaremos **Helm**, programa que instalamos al principio del documento.

Lo primero que haremos será **crear** un **namespace** para Falco con el comando "`k create namespace falco`". El output de este comando debería indicarnos que se ha creado con la salida "`namespace/falco created`". Una vez creado el namespace, **añadimos** el **chart** con el comando "`Helm repo add falcosecurity https://falcosecurity.github.io/charts`", **actualizamos** los repositorios con "`Helm repo update`" y lo **instalamos** con "`helm install falco falcosecurity/falco –namespace falco`".

Una vez instalado haremos un "`k get pod -n falco`" para ver si los pods de Falco se están ejecutando correctamente. Debería aparecernos en estado "*Running*" y con 2 de 2 contenedores ejecutándose.

Vamos a hacer una **prueba** haciendo un "*kubectl get pod*" en el mismo servidor del clúster. Al cabo de unos segundos, **ejecutamos** "***kubectl logs falco-xxxxx -n falco***" y veremos la **hora** de la **última solicitud** y su **dirección IP**. Gracias a esta simple herramienta tendremos nuestro clúster **securizado** y con capacidad de generarnos **logs** con posibles **errores de seguridad**.

Instalación de NetBox

Esta instalación será ejecutada en la máquina virtual "Ubuntu SRV". Como mencionamos anteriormente, esta máquina virtual está **basada en Ubuntu Server 22.04**. Al igual que con el clúster de Rancher K3s, también han sido **modificadas** las **características** de CPU y RAM para evitar cuelgues e inestabilidad.

Para empezar nos conectaremos por SSH y **clonaremos** el repositorio de **NetBox**. Utilizaremos la **imagen** de **Docker** en lugar de la instalación "legacy" por **simplicidad**, **velocidad** y **estabilidad**; si queremos modificar algo es muy **sencillo**. Al cabo de un **momento** si entramos en nuestra ***IP:puerto*** veremos la **interfaz** de NetBox.

Para poder acceder necesitamos ejecutar en el contenedor de NetBox el archivo "manage.py" para **crear** un **superusuario**, por lo que ejecutaremos "*`docker compose exec NetBox /opt/NetBox/NetBox/manage.py createsuperuser`*"

Nos pedirá el **nombre de usuario** (admin en mi caso), **email** y **contraseña**. Una vez terminado, tendremos el **usuario creado** y **podremos acceder** más tarde.

Esto se puede **automatizar fácilmente** con un **script** de bash, y así he hecho: en el repositorio de GitHub podemos ver un script que, **automáticamente**, te **instala Docker**, **Docker Compose**, clona **NetBox** y crea un **servicio** de **systemd** para que el servicio de NetBox se **ejecute automáticamente** cada vez que encendamos el servidor. Se encuentra en *infra/netbox*.

Instalación de GNS3

Durante lo que llevamos de documento hemos visto cómo preparar los equipos para crear un flujo de trabajo enfocado a hacer cambios en la red, pero todavía no hemos respondido a la pregunta principal: ¿Cómo lo vamos a hacer si no tenemos ningún dispositivo físico? Aquí es donde entra **GNS3**:

GNS3 es una **plataforma** que nos va a ayudar **virtualizando** la **infraestructura de redes** que vayamos a montar en nuestro **proyecto**. Podemos montar desde **firewalls** hasta **routers**, **switches**, **VPNs**… Incluso máquinas virtuales y contenedores con sistemas operativos de escritorio.

Voy a utilizar GNS3 porque no cuento con dispositivos para aprovecharlos en este proyecto. Esta herramienta es una parte del proyecto **opcional** y que en un entorno de trabajo real **no sería necesario**.

Podemos instalarlo de **dos formas**: o bien con una **imagen ISO** (instalación desde cero, aunque no es compleja ya que **GNS3 está basado en Ubuntu Server 20.04**) o con una **imagen** con formato **OVA** (incluye una máquina virtual con el sistema instalado y preparado para correr). Para montar la máquina virtual con **OVA** en un entorno local **es necesario instalar VMware**, el cual tiene ciertas **incompatibilidades** con WSL y VirtualBox, por lo que lo **instalaré** en un servidor **aparte** con **Proxmox** ya que esta plataforma de virtualización "*bare-metal*" sí es completamente **compatible**. El servidor utilizado para este propósito fue proporcionado por el instituto, concretamente por el subdirector José Luis Navarro, así que le mando un agradecimiento por la ayuda otorgada en este aspecto.

Tan sólo tendremos que **importar** la **OVA** a nuestro **virtualizador** (ya sea VMware o Proxmox), y **asignar** los **discos** a la **máquina** virtual. GNS3 cuenta con una **aplicación local** para escritorio pero tiene muchos **conflictos** con el sistema operativo, hasta el punto de dejar **Windows inutilizable**.

Configuración de GNS3

Vamos a empezar el apartado de configuraciones con GNS3 ya que es la más **rápida** y **simple** de toda esta sección. Para empezar, entraremos a la **interfaz web** de GNS3 (por defecto ProxMox ya crea un adaptador puente para todas las máquinas virtuales) y crearemos un **nuevo proyecto**: en mi caso le llamaré Proyecto Integrado.

Cuando hayamos creado el proyecto, se nos abrirá una **interfaz de diagrama** muy similar a la que utilizan otros programas como Diagrams.net o Cisco Packet Tracer.

Por defecto GNS3 **no trae** ningún **dispositivo** de red virtualizable por tema de **licencias**, por lo que tendremos que buscarnos una **imagen de dispositivo** por nuestra cuenta. En mi caso me descargué una imagen de un router ***Cisco 7200 Series*** que Cisco otorga gratuitamente para **pruebas** y estudiantes, aunque **podemos utilizar** otros tipos de **dispositivos**, modelos y marcas (**no recomiendo** utilizar el router ***Cisco 3600 Series*** ya que **no incluye SSHv2** y no podremos acceder a él mediante AWX, ya que SSHv1 está obsoleto en Linux desde hace años).

Una vez que tengamos nuestra imagen de dispositivo la **importamos** en GNS3 (Preferencias → Importar…) y la **colocamos** en el diagrama. También colocaremos un recurso de GNS3 llamado "**Cloud**", que nos hará la función de "***pass-through***" para que podamos **acceder** el **router desde la red**. Conectamos ambos recursos mediante un cable **RJ-45** (preferentemente en el **puerto *FastEthernet0/0*** ya que es **el más rápido**) y, ahora sí, **encendemos** el router. Sobre él haremos click derecho, le daremos a encender y a la **Terminal** integrada. Después de unos instantes, tendremos nuestro **router operativo** y listo para configurar.

Configurando el router para el primer acceso

Antes de seguir adelante con la configuración de los servicios vamos a **configurar** el **router** para que quede mínimamente **operativo** y **funcional para cuando sea necesario**.

Como hemos visto hace un momento, entraremos a la terminal integrada del router que nos proporciona GNS3 y configuramos la dirección IP (*192.168.1.170*), además de añadir ciertos **criterios de seguridad** que el router requiere para poder **habilitar SSH**.

```
R1(config)# ip address 192.168.1.170 255.255.255.0
R1(config)# hostname Router1
Router1(config)# ip domain name pi.local
Router1(config)# crypto key generate rsa
How many bits in the modulus [512]: 2048
SSH1.5 has been enabled
# ssh1.5 incluye compatibilidad con ssh v1 y v2
Router1(config)# line vty 0 4
Router1(config-line)# transport input ssh
Router1(config-line)# login local
Router1(config)# username admin password admin
Router1(config)# enable password cisco
Router1# copy run start
```

Estos criterios de seguridad que hemos comentado hace un momento son **añadir** el **nombre del dominio** (en este caso, por ejemplo, se llamará "*pi.local*"), añadir un **par de claves RSA** y establecer **SSH** como el **protocolo** de acceso por defecto (**de serie viene con Telnet**, el cual además de ser **muy inseguro** no es compatible con AWX, por lo que no nos serviría).

Además de estos criterios de seguridad hemos añadido un **usuario** con credenciales 'admin'@'admin' y su contraseña para el "modo enable" es 'cisco'. De esta manera podemos acceder al **router** desde la **red** y, por consecuencia, desde **AWX**.

Configuración del repositorio de GitHub con AWX

Para que AWX pueda acceder a los playbooks e inventarios que vamos escribiendo necesitamos **clonar** el **repositorio** de GitHub **en local**. AWX **incluye** un apartado llamado "**Proyectos**" en el que podemos clonar el repositorio fácilmente.

Para añadir un nuevo proyecto en AWX iniciaremos sesión con las credenciales que hayamos configurado en la instalación y lo primero es **añadir una organización**.

Crear una **organización** nos va a servir para **unir** todos los **recursos del proyecto** en un mismo sitio y tenerlo todo **limpio** y **organizado**. Para crear una organización iremos a la columna izquierda, pincharemos en Organizaciones y le daremos al botón azul de Añadir. Pondremos el nombre pertinente (en mi caso Proyecto Integrado) y el resto de valores en defecto.

Una vez añadida la organización añadiremos unas **credenciales** para que AWX **pueda acceder al repositorio** (esto es obligatorio si tenemos el repositorio privado como es el caso. Si fuera público no es necesario). Creamos un **par de claves** en nuestra máquina local con el comando `ssh-keygen`. La clave **pública** la añadiremos a nuestro perfil de **GitHub** y la **privada** a **AWX**.

Ahora sí, crearemos el proyecto. Como hemos comentado antes, es la base sobre la que funcionará todo el workflow de AWX y donde pondremos el repositorio con el que estamos trabajando. Pondremos la URL en formato SSH y la rama "*main*" en mi caso.

También vamos a **activar** una casilla llamada "*Revisión de actualización durante el lanzamiento*". Lo que hace esta opción es revisar si ha habido cambios en el repositorio **cada vez** que se lance un nuevo playbook o se recojan los datos de NetBox.

Al añadirlo veremos que se ejecuta un trabajo que se encargará de comprobar la conexión al repositorio y descargar los últimos valores. Si todo fue correctamente, veremos un mensaje en verde con el asunto "*Correctamente*".

Poblando el inventario de NetBox con dispositivos GNS3

Antes de poblar el inventario en NetBox vamos a **crear** un **token** para poder utilizar la **API**, de donde extraeremos el contenido en AWX. Para ello, iremos a la interfaz web de NetBox, entraremos en detalles del usuario (admin en este caso) y nos dirigiremos a "API Tokens". Allí generaremos un token que tenga permisos para todo y le pondremos la duración que nosotros veamos necesaria (por ejemplo hasta el 31 de diciembre de 2023). Le damos a Guardar y veremos la clave, vamos a **guardarla** ya que **la necesitaremos** en el siguiente punto.

Volviendo al punto de este apartado, para que podamos tener el inventario dinámico funcionando en AWX tenemos que **crear** las **especificaciones** de la infraestructura en NetBox. De esa manera podemos saber si la conexión funciona o no.

Vamos a guardar en NetBox los **datos** del router ***Cisco 7200*** que hemos configurado anteriormente, aunque podemos almacenar **tantos dispositivos como queramos**.

Para poder crear un dispositivo único tendremos que crear los **siguientes requisitos**:

- Fabricante (Manufacturer)
- Tipo de dispositivo (Device Type)
- Sitio (Site)
- Rol de dispositivo (Device Role)

Veremos cómo crear cada uno de los requisitos necesarios.

Primero empezaremos creando el **fabricante** (en este caso Cisco). En este requisito se nos piden únicamente dos valores: el **Nombre** y el **Slug** (**identificador** en minúsculas). En ambos campos pondremos Cisco. Todos los datos que meteré será mediante archivos YAML, así que para fabricante se nos quedaría este archivo:

```
cisco.yml
---
name: Cisco
slug: cisco
```

Importaremos el archivo accediendo a NetBox y abriendo la pestaña perteneciente a cada tipo de configuración, en este caso Fabricante. Podemos hacer uso de archivos .csv para importar datos masivamente, pero en este caso como sólo almacenaremos un dispositivo vamos a utilizar el formato YAML.

En **Device Type** nos va a pedir que demos información sobre el **dispositivo** en sí, sus **componentes** y **factor de forma**. El archivo resultante en mi caso es el siguiente:

Cisco-catalyst-3600.yml

```
---
manufacturer: Cisco
model: Catalyst 3600
slug: cisco-catalyst-3600
part_number: CATALYST-3600
is_full_depth: true
u_height: 1
interfaces:
  - name: FastEthernet0/0
    type: 100BASE-TX (10/100ME)
  - name: Ethernet1/0
  - type: 100BASE-TX (10/100ME)
```

El router que utilizaré es un Cisco 7200, aunque la configuración de componentes es la misma que el 3600, por lo que reutilizaré el archivo YAML del router Cisco 3600.

Ahora vamos a añadir un **sitio** y un **rol de dispositivo**. En esta sección configuraremos la ubicación donde se va a encontrar el router y su rol (es decir, si está en uso, inventariado, obsoleto, en proceso de retirada…)

proyecto_integrado.yml

```
---
name: Proyecto Integrado
slug: proyecto-integrado
status: active
```

lab.yml

```
---
name: Lab
slug: lab
color: 1f54ab
vm_role: false
```

Una vez creados los requisitos, vamos a añadir **dispositivos específicos** (en este caso el Router1 que hemos configurado antes):

router1.yml

```
---
name: Router1
device_role: Lab
manufacturer: Cisco
device_type: Catalyst 3600 Series
status: active
site: Proyecto Integrado
```

Ya sólo nos falta añadir las direcciones IP del dispositivo. Podemos añadir tantas direcciones IP como puertos le hayamos asignado al modelo mediante el archivo YAML del principio, aunque sólo almacenaremos la dirección IP que configuramos al principio.

router1-ip.yml

```
---
address: 192.168.1.170/32
status: active
device: Router1
interface: FastEthernet0/0
is_primary: true
```

Configuración del inventario de NetBox con AWX

Para utilizar los dispositivos almacenados en NetBox con AWX **configuraremos** un tipo de **inventario dinámico**. Este funcionará gracias a la **API** que NetBox proporciona y con ella podemos **extraer** todos los datos que veamos necesarios, además de poder **manejar** esos **datos** y **gestionarlos** a nuestra conveniencia.

Para crear el inventario en AWX iremos a la **interfaz web** de nuevo y seleccionaremos la sección de **Inventarios**. En él configuraremos únicamente el nombre del inventario, porque la fuente de datos se configurará una vez esté creado el inventario.

Como hemos mencionado en la instalación de AWX, la ruta del inventario de NetBox es code/awx/, por lo que en ese directorio crearemos un archivo (por ejemplo NetBox.yml) con el siguiente contenido:

```
plugin: NetBox.NetBox.nb_inventory
api_endpoint: XXX.XXX.XXX.XXX:puerto
token: XXXXXXXX
validate_certs: false
interfaces: true
group_names_raw: true
group_by:
  - device_roles
  - sites
  - device_types
  - manufacturers
  - platforms
device_query_filters:
  - has_primary_ip: false
```

Este archivo YAML **define** que **utilizaremos** el plugin de **NetBox** (de ahí que las colecciones en este proyecto son obligatorias, las añadiremos justo después), la dirección **IP** y el **puerto** del servidor NetBox, así como el **token** que hayamos generado previamente.

En "*api_endpoint*" y "*token*" pondremos la URL:puerto y el token de NetBox respectivamente.

Además de esos parámetros tenemos el de "*validate_certs*" (permite **recoger** los **datos** de NetBox **a pesar de que no utilice el protocolo HTTPS**), "*interfaces*" (añade la **información** de las **interfaces** configuradas en las **variables** de **inventario**), "*group_names_raw*" (añade los **valores** de **filtrado group_by** como **grupos**), "*group_by*" (**agrupa** los dispositivos recogidos **según** los **criterios** que pongamos) y "*device_query_filters*" (en este caso **filtra** que el dispositivo que estamos leyendo en ese momento tenga una dirección IP primaria. **Si no** la tiene, el dispositivo **no se guardará en AWX**).

Añadimos el archivo a la fuente de nuestro inventario y, si intentamos ejecutarlo, **dará fallo** porque todavía **no hemos importado** las **colecciones** que necesitamos. Para ello, en la raíz del proyecto dentro de la carpeta "collections" crearemos un archivo llamado "requirements.yml". Este archivo **listará** todas las **colecciones** que necesitemos, AWX lo entenderá y las **instalará automáticamente**. El contenido de este fichero, en mi caso, es el siguiente:

requirements.yml

```
---
collections:
  - NetBox.NetBox
  - community.general
  - cisco.ios
```

La **colección** NetBox es **necesaria** para el **inventario dinámico**, la colección *community.general* incluye unas **funciones básicas** que podemos utilizar para otro tipo de dispositivos y la colección *cisco.ios* incluye la **compatibilidad** para que podamos escribir **playbooks** de forma más limpia.

Cuando hayamos subido nuestro fichero de colecciones así como la fuente de nuestro inventario dinámico **actualizaremos** el proyecto en AWX. Si hemos escrito todos los ficheros correctamente, veremos que en la salida del trabajo se mostrarán las colecciones que hemos instalado, por lo que, ahora sí, iremos a Inventarios → Fuente → Actualizar. Veremos que, en unos momentos, **AWX se comunica con la API del servidor de NetBox** y va **almacenando** los dispositivos que hayamos creado previamente.

Cambiando parámetros del router desde AWX

Llegados a este punto, vamos a hacer los **primeros cambios** en **nuestro router** directamente **desde AWX**. Para ello escribiremos "***playbooks***" que este ejecutará y **accederán** al **router**. Como ya hemos instalado la colección de Cisco.IOS podemos seguir adelante sin repasar nada.

Lo primero que haremos será **crear** una **credencial** en AWX. Para ello, iremos al desplegable de la parte izquierda y pincharemos en **Credenciales**. El tipo de credencial será **tipo Máquina** con nombre de usuario "*admin*", contraseña "*admin*" y contraseña para elevación de privilegios "*cisco*". Es muy importante que ajustemos el **método** de **escalación** de **permisos** en "***enable***", ya que es el modo que utilizan los routers Cisco y de otra manera no funcionaría.

Una vez hecho vamos a crear nuestro **primer playbook**, el cual será uno muy simple que nos **creará un banner** (MOTD). De esta manera podremos ver que se efectúan los cambios en nuestro workflow y más adelante crearemos un playbook más completo y "realista".

Ejecución de un playbook simple

Como comentamos, ahora **vamos a ejecutar** un playbook **simple** pero **suficiente** como para ver que AWX es capaz de conectarse a nuestro router virtual y cambiarle la configuración. En este caso le pondremos un MOTD:

Code/awx/playbooks/motd.yml

```
---
- name: Configurar banner en Routers Cisco
  hosts: cisco
  tasks:

  - name: Añadir Banner
    ios_banner:
      banner: login
      text: |
        Hola Mundo! Si estás
        leyendo esto significa que
        la ejecución del playbook
        fue correcta y podemos
        seguir adelante.
      state: present
```

Una vez hayamos creado el archivo YAML podemos **crear** la **plantilla** en **AWX**, a la cual llamaremos de igual manera. Es muy importante al crear la **plantilla** asegurarnos de que se **usan las credenciales** que creamos en el punto anterior y que es capaz de detectar nuestro playbook; **si no**, tendremos que volver a **sincronizar** el **proyecto**.

Procedemos a **ejecutar** el playbook y, si todo ha salido bien, nos saldrá un mensaje de *OK[1]* en color verde.

Para **comprobar** que esto ha funcionado, saltaremos al router y podremos visualizarlo de **dos formas** distintas:

- **El router nos muestra el banner nada más conectarnos**. En mi caso no lo va a hacer porque ya he configurado el router en mi anfitrión e inicia sesión automáticamente.
- **Accediendo a la configuración** que se está ejecutando en el momento. Es decir, ejecutaremos un *show running-config* (o *sh run*).

Voy a ejecutarlo de la segunda manera. **Accedemos al router y veremos en la configuración lo siguiente:**

```
!
banner login ^C
Hola Mundo! Si estC!s
leyendo esto significa que
la ejecuciC3n del playbook
fue correcta y podemos
seguir adelante.
^C
```

Las **tildes** no se muestran correctamente por temas de **codificación**, aunque lo importante es que ahora podemos ver cómo la **funcionalidad** básica del *workflow* **hace lo esperado**. Vamos a hacer un **repaso** de lo que tenemos hecho en el proyecto:

- **ENTORNO DE OPERACIÓN -> ENTORNO DE DATOS**
  El ingeniero de redes puede acceder tanto a NetBox para cambiar la información de los routers como a GitHub para cambiar la configuración de estos.

- **ENTORNO DE OPERACIÓN -> ENTORNO DE CÓDIGO**
  AWX recibe el código de GitHub y los datos de NetBox y los guarda para su posterior uso.

- **ENTORNO DE CÓDIGO/OPERACIÓN -> ENTORNO DE PRODUCCIÓN**
  AWX aplica los playbooks alojados en el repositorio en los routers y dispositivos que configuramos en NetBox.

- **ENTORNO DE PRODUCCIÓN -> ENTORNO DE DATOS**
  NetBox almacena los datos de los dispositivos que tenemos en producción.

Haciendo playbooks más complejos

Como el playbook que hemos ejecutado es demasiado simple, **vamos a hacer** un ejemplo de **playbook** que sí podríamos utilizar en un **entorno real**. El que os muestro a continuación le **cambia** el **nombre** al router, le crea un **banner personalizado**, manda un **ping** al servidor de **ProxMox**, crea **dos VLANs**, **lista** la **configuración actual** y **guarda la modificada**. A este playbook lo llamaremos "*revision.yml*", ya que **no es una configuración "como tal"**, sino que **lo ejecutaremos periódicamente** para que el **router tenga siempre la misma configuración**.

revision.yml

```
---
- name: Revisión de configuración de los routers Cisco
  hosts: cisco
  tasks:

    - name: Poner nombre y dominio por defecto
      become: true
      ios_system:
        hostname: RouterAWX
        domain_name: pi.local
        domain_search:
          - ansible.com
          - cisco.com

    - name: Añadir banner personalizado
      become: true
      ios_banner:
        banner: login
        text: |
           YOU ARE ACCESING A RESTRICTED SYSTEM!
           IF YOU ARE NOT A NETWORK ENGINEER AND
           YOU DO NOT BELONG TO THIS ORGANIZATION
```

```
        LEAVE NOW OR LEGAL ACTIONS WILL BE
        TAKEN AGAINST YOU.
        FOR ANY ISSUES, PLEASE CONTACT AN ADMIN
        IMMEDIATELY.

- name: Crear usuario "ansible"
  become: true
  ios_user:
    name: ansible
    configured_password: ansible
    password_type: password
    state: present

- name: Crear VLAN
  become: true
  ios_vlans:
    config:
    - name: printers
      state: active
      vlan_id: 155
      shutdown: disabled

- name: Lista la configuración actual
  become: true
  ios_command:
    commands: show running config

- name: Guarda los cambios cuando haya modificaciones
  become: true
  ios_config:
    save_when: modified
```

Este **playbook** nos **configura** el **nombre** del router como "**RouterAWX**", lo introduce al **dominio** "***pi.local***" y **habilita** "***ansible.com***" y "***cisco.com***" como **dominios** de **búsqueda**. Además de estos cambios también añadirá un **banner personalizado** que se lanzará al iniciar sesión, **creará** un **usuario** con credenciales '*ansible'@'ansible'*, una **VLAN** con ID **155** llamada ***printers*** que estará habilitada y no se apagará, lista la configuración que se esté ejecutando y, finalmente, guardará los cambios modificados.

La principal **razón** por la que **Ansible** es tan ampliamente **utilizado** es por ser una herramienta basada en el ***lenguaje declarativo***. Esto significa que, en un **archivo YAML**, **definiremos qué configuración queremos aplicar**. Ansible va a ir **comprobando** una a una las **tareas** que tenemos configuradas con la configuración que se esté ejecutando en el dispositivo (router en este caso). **Si la configuración es distinta, Ansible entra en acción y aplica los cambios necesarios**. **En caso contrario** (es decir, la configuración es idéntica), Ansible **omitirá la ejecución** de esa tarea **para evitar perder tiempo y conflictos** ("*overlapping*", que nos dé errores porque un recurso ya existe, etc.)

Notificación de cambios del repositorio con GitHub Actions

En este apartado configuraremos una **acción** de *GitHub Actions* para que nos mande un **correo electrónico** cada vez que haya un **cambio** en el **repositorio**, aunque primero vamos a ver en qué consiste *GitHub Actions*.

GitHub Actions es una **característica** de *GitHub* desarrollada para que funcione como un **desencadenador** de **acciones** (***trigger***). Es decir, **cada vez** que en nuestro repositorio haya, por ejemplo, un ***push*** (subidas nuevas de datos) se **ejecutará** el trigger que nosotros hayamos configurado en un archivo *.yml*. Lo más interesante de Actions es que, **internamente**, **se ejecuta** en una máquina virtual con **Ubuntu Server**, entonces podemos mandarle también que **ejecute scripts** con todas las opciones que se nos ocurran.

En este caso utilizaremos **nuestro correo electrónico** para que nos mande **notificaciones** de actualización del repositorio. Para ello empezaremos creando el archivo *.yml*, que tendrá un contenido **parecido al siguiente**:

```
name: Notificaciones Correo
on:
    push:
    delete:
    create:
jobs:
  send-push:
    name: Mandar e-mail al detectar un nuevo Push en el repositorio
    runs-on: ubuntu-latest
    steps:
      - name: Checkout código fuente
        uses: actions/checkout@v2

      - name: Enviando correo electronico
        uses: dawidd6/action-send-mail@v3.6.1
```

```
        with:
          server_address: smtp.gmail.com
          server_port: 465
          username: ${{ secrets.EMAIL_ADDRESS }}
          password: ${{ secrets.EMAIL_PASSWD }}
          subject: "Se detectó un ${{ github.event_name }}
          en el repositorio ${{ github.repository }}."
          from: github <noreply@github.com>
          to: javsanpae@gmail.com
          body: |
            Acción: ${{ github.event_name }}
            Repo: ${{ github.repository }}
            Commit: ${{ github.sha }}
            Autor: ${{ github.actor }}
            Se han detectado nuevos cambios en el
            repositorio. Para comprobarlos, ve a
            https://github.com/javsanpae/PROYECTO_ASIR .
```

Para hacer uso de este disparador tendremos que crear antes dos ***secrets*** en *GitHub*: el que **alojará** el **correo** electrónico y el que **contendrá** la **contraseña de aplicación** para que funcione. Para ello, iremos al repositorio de *GitHub*, entraremos en *Settings* y en *Secrets -> Actions*.

En mi caso utilizaré el **correo electrónico personal** proporcionado por el instituto (***jsanchez155@ieszaidinvergeles***.org) como encargado de **mandar los correos** electrónicos y mi **correo electrónico personal** (***javsanpae@gmail.com***) como **receptor de estos correos**.

El correo electrónico del instituto está establecido en Google Workspace (Gmail, Drive, etc), por lo que, para habilitar esta función con correos electrónicos de Google, necesitaremos utilizar una **clave de aplicación,** la cual sólo estará disponible si habilitamos la **verificación en dos pasos**.

La verificación en dos pasos, como ya sabemos, es un **método de seguridad** que agrega una capa adicional de **protección** a una cuenta en línea. En lugar de solo ingresar una contraseña, la verificación en dos pasos requiere que el usuario proporcione **dos elementos** de **verificación** diferentes (en este caso, un **código** que nos llegará a nuestro **número de teléfono**).

Para generar la clave de aplicación, iremos a nuestra cuenta de Google -> Gestionar tu cuenta de Google y, en el buscador, introduciremos "contraseñas de aplicación". Entramos en la opción con ese nombre y generamos una clave personalizada con el nombre que queramos (en este caso "*Gmail-RepoPI*"). Copiamos la clave de 16 letras y la pegamos en el *secret* que hayamos creado previamente.

Subiremos el archivo en la ruta ".github/" para que GitHub entienda que se trata de una acción y, como previamente hemos configurado que la acción se ejecute cuando haya un push, se lanzará automáticamente. Comprobamos que haya llegado un correo electrónico a nuestra dirección parecido a este:

```
Se detectó un push en el repositorio javsanpae/PROYECTO_ASIR.


Acción: push

Repo: javsanpae/PROYECTO_ASIR

Commit: 349799a563d12b97284209beeef7a4a0889fcc84

Autor: javsanpae


Se han detectado nuevos cambios en el repositorio. Para comprobarlos, ve a https://github.com/javsanpae/PROYECTO_ASIR.
```

Conclusión

Este proyecto ha abordado de forma completa la automatización de redes y ha logrado alcanzar los objetivos planteados al inicio. Mediante el análisis de diferentes herramientas para la entrega continua de código, así como el aprovisionamiento de infraestructura, se ha obtenido un cierto manejo sobre el área de "NetDevOps", permitiendo así identificar los puntos débiles y haciéndolos fáciles de operar y modificar para los ingenieros de red.

Los resultados obtenidos demuestran que la idea de utilizar la Infraestructura como Código (*IaC*) se puede aplicar también en un entorno de redes. Estos resultados son de gran importancia en el campo de las redes tanto locales como de mayor escala y contribuyen al avance del conocimiento en la automatización de tareas y procesos.

Durante el desarrollo de este Proyecto Integrado, se han enfrentado problemas y obstáculos que han sido superados mediante la investigación y tomando diferentes caminos y soluciones para un único problema. Esto ha demostrado la capacidad de adaptación y resolución de problemas, así como la rigurosidad y la dedicación invertida en esta investigación.

No obstante, existen oportunidades para futuras investigaciones que podrían profundizar en la protección de la red a prueba de errores o expandir este estudio al levantamiento de un entorno de operación de una empresa real, mezclando automatización de redes con despliegue de aplicaciones y viceversa. Estas posibles líneas de investigación podrían ampliar aún más el conocimiento en este campo y brindar nuevas perspectivas y posibles mejoras.

En conclusión, este Proyecto Integrado ha logrado cumplir con sus objetivos, aportando nuevos conocimientos y perspectivas en el área de "networking". Este trabajo sienta las bases para un mayor avance en la metodología "DevOps" y representa un paso significativo en mi formación académica y profesional.

Bibliografía

1. **Jonathan Wijaya (2015).** Network Automation using Ansible for Cisco Routers Basic Configuration.

2. **Cisco ENSA.** Module 14: Network Automation.

3. **Gerardo Ocampos (2022).** Despliegue de aplicaciones con Ansible + AWX.

4. **Ivan Pepelnjak.** Network Automation 101.

5. **Matteo Canzari (2021).** A brief introduction to Kubernetes.