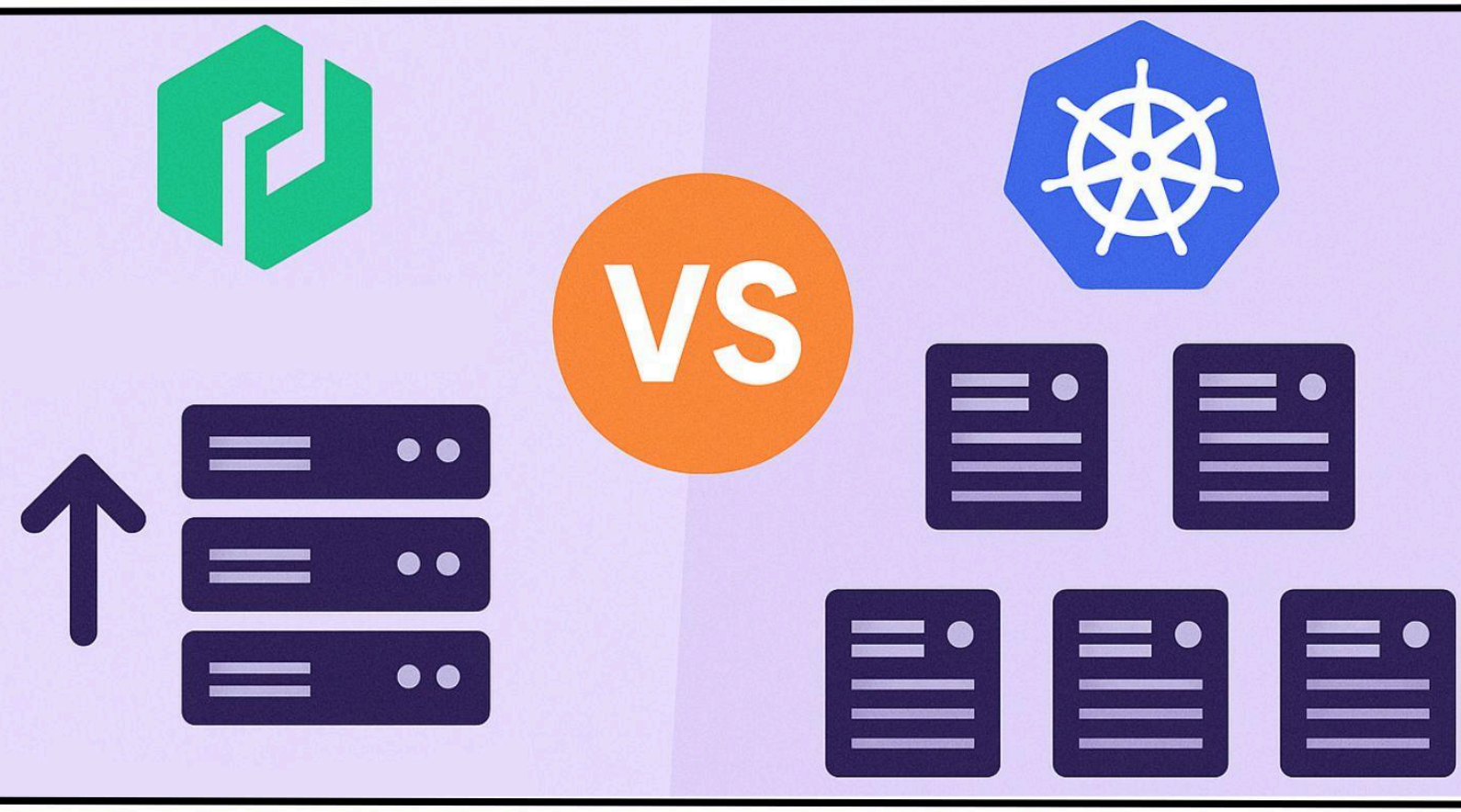
# Nomad VS Kubernetes.




# Autoescalado en Nomad

**Proyecto integrado final de grado**

**Alejandro Albaladejo Gago**

**IES Gonzalo Nazareno**

**Curso 2024/2025**

# Nomad vs Kubernetes. Autoescalado en Nomad

## 1.- Introducción

En este proyecto se dará a conocer una nueva herramienta de orquestación como es **Nomad Hashicorp**, que se presenta como una alternativa a **Kubernetes.**
También se va a realizar un autoescalado en esta nueva herramienta y se propondrán las principales diferencias en cuanto a su uso, frente a Kubernetes.

### 1.1.- Objetivos del proyecto

- Familiarizarse con Nomad, como alternativa a Kubernetes, mostrando sus características.
- Conocer conceptos importantes de Nomad y aprender a definir configuraciones básicas para ejecutar aplicaciones.
- Integrar otras herramientas del entorno de Hashicorp, como Consul o Prometheus.
- Realizar distintos despliegues en Nomad.
- Conocer el funcionamiento del autoescalado y aplicarlo en Nomad en función de métricas.

### 1.2.- Escenario

Este proyecto está realizado sobre una máquina virtual Linux (Debian 12), pero se puede realizar en otras distribuciones Linux sin problemas y también en la misma máquina Host.

## 2.- Fundamentos teóricos previos necesarios

En este apartado se describen los fundamentos teóricos que van a ser necesarios para entender el proyecto.

### 2.1- ¿Qué es la orquestación de contenedores?

La orquestación de contenedores es el proceso mediante el cual se automatiza la gestión, despliegue, programación, escalado, eliminación de contenedores… en entornos productivos. Esto permite a las organizaciones ejecutar aplicaciones de forma eficiente y escalable, sobre todo en arquitecturas nuevas que están basadas en microservicios, que pueden estar compuestas por cientos o miles de contenedores.

La contenerización, con herramientas como **Docker**, permite “empaquetar” una aplicación con todo lo necesario para funcionar como las librerías y configuraciones en un solo

contenedor. Esto hace que sea fácil moverla y ejecutarla en cualquier sitio, como un ordenador, un servidor, en la nube…
Pero cuando hay muchas aplicaciones divididas en muchos contenedores y estos repartidos en varios servidores, ya Docker no es capaz de organizarlos. Aquí es donde entran las plataformas de orquestación, como son **Kubernetes** o **Nomad**, que se encargan de gestionar automáticamente todos esos contenedores, ayudando a que las aplicaciones se mantengan disponibles, se reinicien si fallan, se escalen… sin tener que hacerlo manualmente.

### 2.1.1- ¿Por qué es necesaria la orquestación de contenedores?

En sus inicios, el despliegue de contenedores a gran escala requería scripts personalizados para coordinar nodos, gestionar versiones y mantener configuraciones distribuidas, lo que generaba numerosos problemas de mantenimiento, escalabilidad y control.
Las herramientas de orquestación **resuelven estos desafíos automatizando todo el ciclo de vida** del contenedor, desde su creación hasta su destrucción, simplificando significativamente la operación de sistemas complejos.

### 2.1.2- Casos de uso

La orquestación de contenedores es esencial cuando se requiere:

- Escalar contenedores dinámicamente en múltiples servidores.
- Ejecutar múltiples versiones de una misma app (p. ej., pruebas y producción).
- Asegurar alta disponibilidad con instancias replicadas.
- Desplegar contenedores en múltiples regiones geográficas.
- Maximizar el uso de recursos de infraestructura disponibles.
- Gestionar sistemas compuestos por miles de microservicios.

### 2.1.2- Beneficios

- **Automatización completa** del ciclo de vida de los contenedores.
- **Equilibrio de carga** entre servicios distribuidos.
- **Alta disponibilidad y resiliencia** ante fallos: los contenedores se reinician automáticamente o se replican para evitar interrupciones.
- **Escalabilidad automática** basada en demanda, red o recursos disponibles.
- **Supervisión y gestión del estado de los contenedores** en tiempo real.
- **Seguridad y control de acceso** en entornos contenerizados.
- **Optimización de costos e infraestructura**, al aprovechar al máximo cada instancia activa y crear nuevas bajo demanda.

## 2.2- Autoescalado

El autoescalado (Auto-Scaling), es una característica que se utiliza para asignar automáticamente recursos informáticos en función de la demanda del sistema. Garantiza que las aplicaciones tengan los recursos necesarios para mantener una disponibilidad constante y cumplir con los objetivos de rendimiento.

### 2.2.1- Escalado horizontal

El escalado horizontal (**scaling out**) implica añadir más máquinas o nodos a un entorno de cloud computing. También existe el **scaling in** que consiste en reducir el número de nodos del entorno.

### 2.2.2- Escalado vertical

El escalado vertical (**scaling up**) es el proceso de añadir más potencia, como RAM, CPU o almacenamiento, a los nodos existentes en el entorno.

### 2.2.3- Escalado dinámico

En el escalado dinámico, se reacciona a las necesidades de recursos a medida que se producen, ajustando la asignación de recursos en función de la utilización en tiempo real.

*Por ejemplo, tenemos una página web dinámica que no tiene mucho movimiento normalmente, pero ha tenido una publicidad y de repente se han triplicado las visitas. Si tenemos un autoescalado dinámico, proveerá automáticamente los recursos necesarios para que nuestra página web responda a las peticiones de los visitantes. En el momento que la página deje de tener tantas visitas, volverá reducir los recursos.*

### 2.2.4- Escalado predictivo

Las políticas de escalado predictivo utilizan la **IA** y el **machine learning** para anticiparse a las necesidades futuras de recursos antes de que se produzcan, basándose en el historial de uso.

*Por ejemplo, se entiende que en fechas navideñas, habrá más compras por internet, por lo que escalaría en esas fechas lo necesario. Esto ayuda a minimizar el tiempo de inactividad.*

### 2.2.5- Escalado programado

El escalado programado, asigna los recursos según un calendario predeterminado.

*Por ejemplo, si una página web tiene más visitas por la tarde que por la mañana, se puede establecer una política para adaptarse a esta demanda.*

### 2.2.6- Beneficios del autoescalado

- Minimiza la configuración manual de la infraestructura.
- Aumenta la escalabilidad.
- Proporciona un rendimiento consistente.
- Mejora la experiencia del usuario.
- Reduce los costes del cloud computing.

## 2.3.- Kubernetes

Como ya sabemos, Kubernetes es una plataforma de orquestación de contenedores de código abierto.

### 2.3.1.- Arquitectura

Kubernetes se basa en una arquitectura **cliente-servidor** donde los nodos y los componentes del clúster trabajan juntos para mantener la disponibilidad y el estado de las aplicaciones.

**Nodo Master:** Es el cerebro del clúster, donde se gestionan las decisiones sobre el estado de las aplicaciones, como la planificación de contenedores, el control de la configuración, la monitorización, la supervisión… Este nodo no ejecuta los contenedores directamente, su función es la de coordinar y gestionar todos los aspectos del clúster.Los componentes principales de un nodo master son:

- **API Server**: Punto central para todas las comunicaciones del clúster. Recibe solicitudes y las comunica al resto de componentes.
- **Scheduler**: Se encarga de decidir en qué nodo worker se ejecutarán los contenedores (pods).
- **Controller Manager**: Gestiona los controladores que se encargan de mantener el estado deseado del sistema.
- **etcd**: Base de datos distribuida que guarda el estado y la configuración de todo el clúster.

**Nodo Worker:** Es donde realmente se ejecutan los contenedores. Cada nodo worker tiene los recursos necesarios para ejecutar aplicaciones en contenedores. Los componentes principales de un nodo worker son:

- **Kubelet**: Asegura que los contenedores dentro de un pod estén funcionando como se espera.
- **Kube Proxy**: Gestiona el balanceo de carga y la red dentro del clúster.
- **Container Runtime**: Es la herramienta que se encarga de ejecutar los contenedores (generalmente Docker o containerd).

## 3.- Nomad Hashicorp

Nomad es un programador que sirve para organizar y ejecutar trabajos o aplicaciones en distintos ordenadores o servidores. Por ejemplo, nosotros le decimos lo que queremos ejecutar y Nomad se encargará de decidir en qué servidor se hace, cuándo y de qué forma. Esto es fundamental, ya que en una empresa puede haber un grupo de ordenadores en el que unos estén muy ocupados y otros más libres, eligiendo así a los que estén más libres.

***Ejemplo***
*Suponemos que tenemos 5 servidores en nuestra empresa, 2 de ellos están muy ocupados, mientras los otros 3 están libres. Si mandamos a ejecutar una aplicación, Nomad la colocará en uno de los que están libres, para así optimizar los recursos.*

### 3.1.- Características

Nomad tiene unas características destacadas que hacen que sea una opción interesante tanto para proyectos personales como para entornos empresariales.

- **Simplicidad y Fiabilidad:** Se ejecuta como un único binario auto-contenido, sin dependencias externas. Gestiona automáticamente fallos de aplicaciones y nodos, siendo distribuido y teniendo una alta disponibilidad.
- **Soporte para Dispositivos (GPU, etc.):** Ofrece soporte nativo para cargas de trabajo de ML/IA con GPU, y utiliza plugins de dispositivo para detectar y utilizar automáticamente recursos de hardware como GPU, FPGA y TPU.
- **Federación Multiregión:** Soporte integrado para federar clústeres a través de múltiples regiones, facilitando la implementación de trabajos en cualquier clúster federado y replicando automáticamente políticas (ACLs, cuotas, etc.).
- **Escalabilidad Comprobada:** Demostrada escalabilidad a más de 10.000 nodos en producción, ofreciendo alto rendimiento y baja latencia.
- **Integración con el Ecosistema HashiCorp:** Se integra fluidamente con Terraform, Consul y Vault para provisión, descubrimiento de servicios y gestión de secretos.

## 3.2. Arquitectura

Nomad sigue una estructura distribuida y altamente disponible, compuesta por dos tipos principales de nodos. **clientes y servidores**. Esta estructura permite escalar horizontalmente, resistir a fallos y mantener la disponibilidad del clúster.

### 3.2.1 Componentes principales

- **Nodo servidor (servers):** Son los encargados de gestionar el estado del clúster. Reciben los trabajos (jobs) que se quieren ejecutar, planifican y reparten entre los nodos clientes. Nomad usa un sistema de consenso basado en Raft para garantizar la consistencia. De los servidores, uno actúa como líder , y los demás como seguidores. Si el líder fallara, se elige uno nuevo automáticamente.
- **Nodo cliente (clients):** Son las máquinas donde se ejecutan realmente los trabajos. Estos reciben las instrucciones del servidor y ejecutan las tareas asignadas. Informan constantemente de su estado (carga, disponibilidad, recursos…) al servidor para que la planificación sea precisa.
- **Agentes de Nomad**: Cada máquina que forma parte del clúster ejecuta un agente Nomad. Este agente puede estar en modo cliente o servidor, y se configura según su rol. Los agentes manejan la comunicación, ejecución de trabajos y la supervisión.
- **Jobs:** Son las definiciones de trabajo que los usuarios crean. Un job describe qué se va a ejecutar, con cuantas réplicas, qué recursos… Estos jobs se escriben en formato HCL (HashiCorp Configuration Language).

### 3.2.2 Flujo de trabajo de un clúster

1. Un usuario envía un job al clúster, especificando que quiere ejecutar.
2. Los servidores de Nomad evalúan el job y planifican dónde se debe ejecutar.
3. El líder del clúster decide qué cliente ejecutará cada tarea, basándose en la disponibilidad de recursos y otras condiciones.
4. Los clientes ejecutan el trabajo y reportan su estado.
5. Los servidores monitorean constantemente a los clientes, reprogramando tareas en caso de fallos.

## 3.3.- Recursos

Nomad organiza sus recursos a través de una entidad central que se llama **job**. Todo lo que se ejecuta en un clúster de Nomad se debe definir mediante un job, escrito en formato *HCL* o *JSON*.

### 3.3.1.- Job

Un *job* es un archivo que contiene las instrucciones necesarias para que Nomad ejecute aplicaciones o tareas dentro del clúster. Se compone de múltiples elementos que especifican desde el tipo de ejecución hasta los recursos asignados y comportamiento ante fallos.

**Componentes principales de un Job**

1. **Tipo de job (type)**
   Nomad admite tres tipos de trabajos:
   - **Service:** Para aplicaciones que deben estar siempre disponibles, como APIs o microservicios.
   - **Batch:** Para tareas que se ejecutan una vez y terminan, como scripts.
   - **System:** Para tareas que se ejecutan en todos los nodos, como servicios de fondo para monitoreo.
2. **Datacenters**
   - Lista de datacenters en los que se puede ejecutar el job. Deben coincidir con la definida en la configuración del clúster
3. **Grupo de tareas (group)**
   - Es un conjunto de tareas que se ejecutan en el mismo nodo y que comparten recursos comunes. Es una agrupación dentro de un job y se utiliza cuando necesitamos que varias tareas se desplieguen juntas.
4. **Tareas (task)**
   - Una tarea es una unidad individual de trabajo dentro del grupo. Aquí se define el driver (docker, java, exec…), la configuración del contenedor o script, recursos, variables de entorno, puertos ….
5. **Recursos**
   - Definen cuánta CPU, memoria y red necesita la tarea. Nomad usa esta información para ubicarla correctamente en el clúster.
6. **Network**
   - Configura los puertos y el modo de red para la aplicación, es especialmente útil para exponer los servicios.
7. **Restart y update**
   - Controla cómo se comportan las tareas en caso de fallo y cómo se actualizan las instancias.

**Ejemplo básico de un job**

```
job "nginx" {
  datacenters = ["dc1"]
  type = "service"

  group "nginx" {
    count = 1 #Número de contenedores que se va a usar

    network { #Puertos que se van a exponer
      port "http" {
        static = 8086 #Puerto expuesto en el host
estáticamente
        to = 80  #Puerto del contenedor al que se redirige.
      }
    }

    task "nginx" {
      driver = "docker" #Usará docker como motor de ejecución

      config {
        image = "nginx:latest" #Imagen docker que se usará
        ports = ["http"] #Asocia la tarea con el puerto
definido
      }

      service {
        provider = "nomad"
        name = "nginx"
        port = "http"
      }
    }
  }
}
```

## 3.4.- Comandos básicos

Para trabajar con Nomad desde la línea de comandos, disponemos de una serie de comandos que permiten gestionar el ciclo de vida de los jobs.

Podemos obtener una lista general de comandos disponibles con:

```
nomad --help

Usage: nomad [-version] [-help] [-autocomplete-(un)install]
<command> [args]

Common commands:
    run         Run a new job or update an existing job
    stop        Stop a running job
    status      Display the status output for a resource
    alloc       Interact with allocations
    job         Interact with jobs
    node        Interact with nodes
    agent       Runs a Nomad agent
```

- **Levantar un entorno de Nomad en modo desarrollo**. Este modo ejecuta tanto el servidor como el cliente en un solo proceso. Para pruebas locales viene bien.

```
nomad agent -dev
```

- **Lanzar un job**. Permite iniciar o actualizar un archivo con la definición del job.

```
nomad job run ejemplo1.hcl
```

- **Consultar el estado de todos los jobs**. Muestra una lista con todos los jobs desplegados y su estado actual.

```
nomad job status

ID             Type     Priority  Status   Submit Date
nginx-service  service  50        running
2025-04-30T16:01:05+02:00

==> View and manage Nomad jobs in the Web UI:
http://127.0.0.1:4646/ui/jobs
```

➢ **Obtener detalles de un job en ejecución.** Permite ver información más detallada sobre un job específico, incluyendo sus asignaciones, estado, versiones…

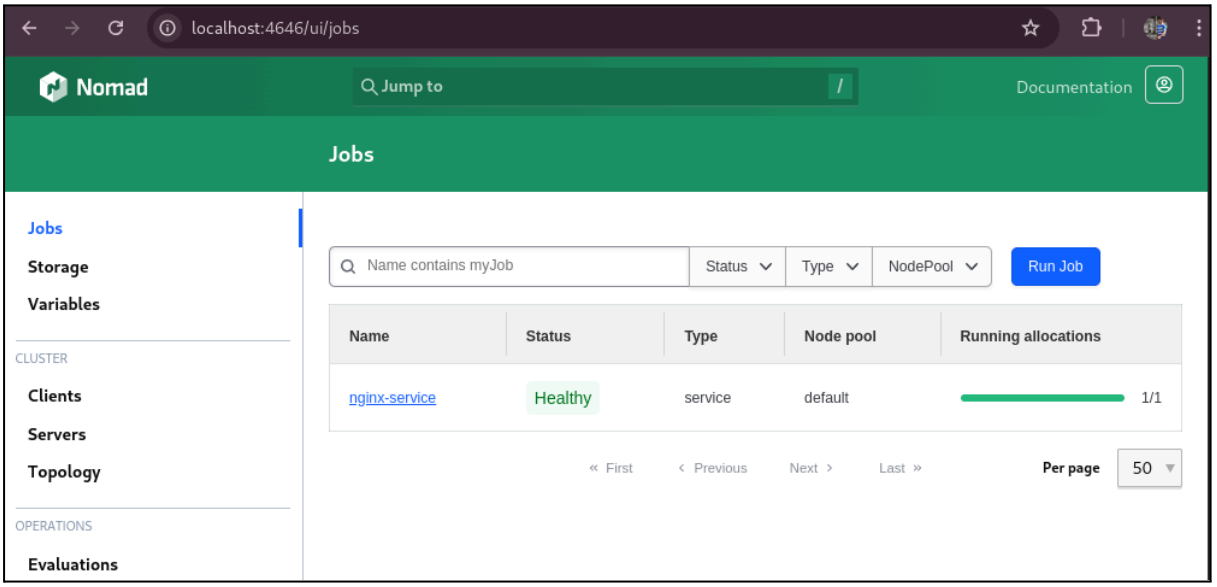
```
nomad job status nginx-service
ID            = nginx-service
Name          = nginx-service
Submit Date   = 2025-04-30T16:04:10+02:00
Type          = service
Priority      = 50
Datacenters   = dc1
Namespace     = default
Node Pool     = default
Status        = running
Periodic      = false
Parameterized = false

Summary
Task Group   Queued  Starting  Running  Failed  Complete
Lost  Unknown
nginx-group  0       0         1        0       0         0
0

Latest Deployment
ID          = 20068f9f
Status      = successful
Description = Deployment completed successfully

Deployed
Task Group   Auto Revert  Desired  Placed  Healthy  Unhealthy
Progress Deadline
nginx-group  true         1        1       1        0
2025-04-30T16:14:22+02:00

Allocations
ID        Node ID   Task Group   Version  Desired  Status
Created  Modified
94186641  81762823  nginx-group  0        run      running
39s ago  26s ago
```

➢ **Detener un job**. Finaliza la ejecución de un job.

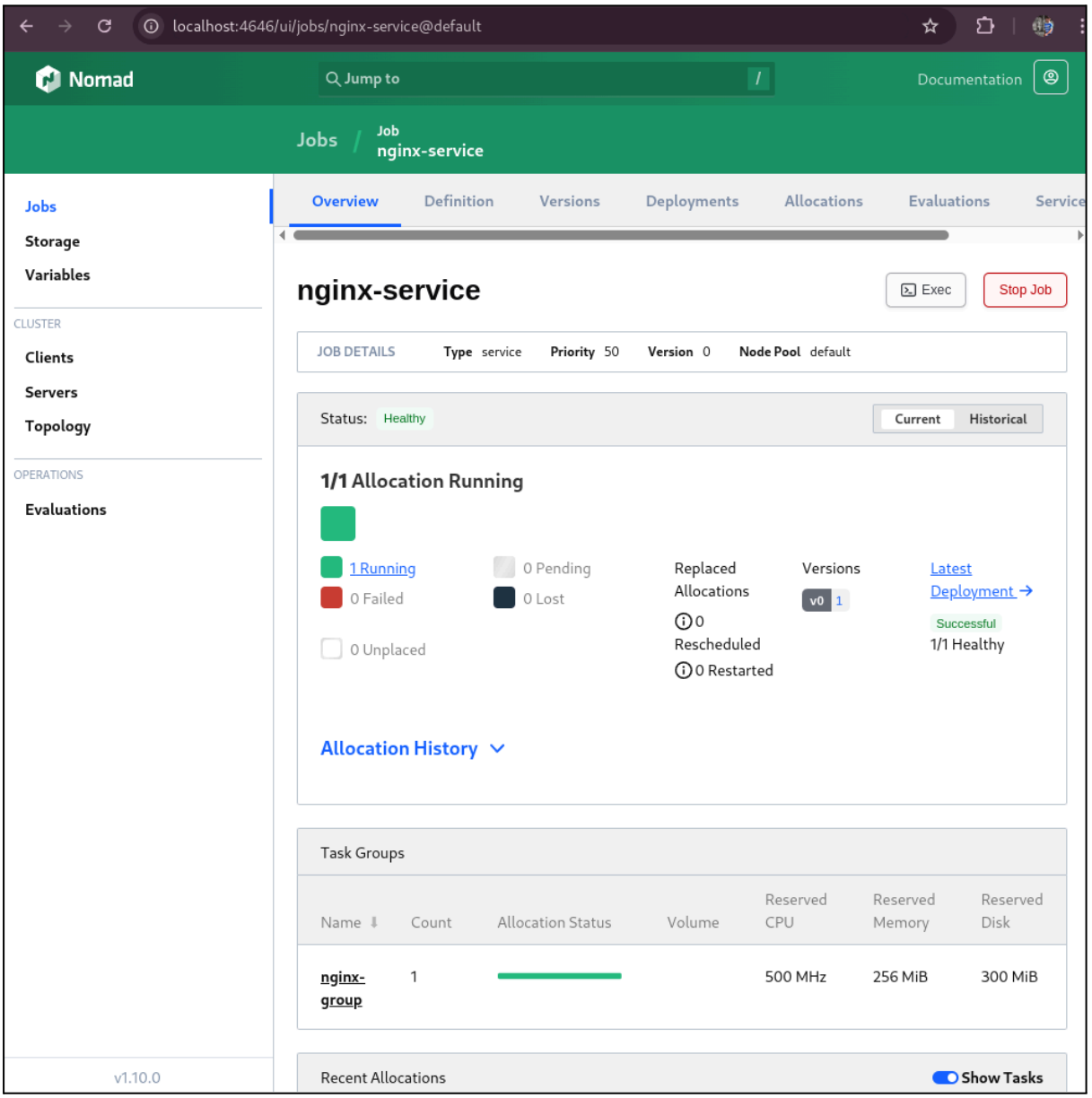
```
nomad job stop nginx-service
```

## 3.5.- Interfaz gráfica

Para acceder a la interfaz gráfica de Nomad, podemos acceder a través del navegador http://localhost:4646
En caso de tenerlo en una máquina distinta se accede con la IP en lugar de localhost.
Al entrar en la URL, veremos directamente la pestaña de **jobs**.
Aquí vemos el nombre del servicio, el estado, el tipo…




Si accedemos al job, podemos ver información más detallada acerca de este job, como su grupo, la definición… También podemos pararlo desde aquí, ver los despliegues realizados y un largo etc de cosas.


localhost:4646/ui/jobs/nginx-service@default
Nomad
Jump to
/
Documentation
Jobs
Job
nginx-service
Jobs
Storage
Variables
CLUSTER
Clients
Servers
Topology
OPERATIONS
Evaluations
Overview
Definition
Versions
Deployments
Allocations
Evaluations
Service
nginx-service
Exec
Stop Job
JOB DETAILS
Type service
Priority 50
Version 0
Node Pool default
Status: Healthy
Current
Historical
1/1 Allocation Running
1 Running
0 Pending
0 Failed
0 Lost
0 Unplaced
Replaced Allocations
0 Rescheduled
0 Restarted
Versions
v0 1
Latest Deployment
Successful
1/1 Healthy
Allocation History
Task Groups
Name
Count
Allocation Status
Volume
Reserved CPU
Reserved Memory
Reserved Disk
nginx-group
1
500 MHz
256 MiB
300 MiB
Recent Allocations
Show Tasks
v1.10.0

## 4.- Instalaciones de herramientas necesarias

A continuación se detallan los pasos que se deben seguir para instalar todas las herramientas necesarias para posteriormente realizar las demos.

### 4.1.- Instalación Docker

1. Actualizar los paquetes del sistema

```
sudo apt update
```

2. Instalar paquetes necesarios.

```
sudo apt install ca-certificates curl
```

3. Crear directorio para guardar la clave GPG de Docker.

```
sudo install -m 0755 -d /etc/apt/keyrings
```

4. Descargar y guardar la clave GPG oficial de Docker

```
sudo curl -fsSL https://download.docker.com/linux/debian/gpg
-o /etc/apt/keyrings/docker.asc
```

```
sudo chmod a+r /etc/apt/keyrings/docker.asc
```

5. Añadir el repositorio de Docker

```
echo \
  "deb [arch=$(dpkg --print-architecture)
signed-by=/etc/apt/keyrings/docker.asc]
https://download.docker.com/linux/debian \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" |
\
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

6. Actualizar e instalar Docker

```
sudo apt update
```

```
sudo apt-get install docker-ce docker-ce-cli containerd.io
docker-buildx-plugin docker-compose-plugin
```

7. Permitir ejecutar docker sin sudo (Opcional).

```
sudo usermod -aG docker $USER
```

```
su - $USER
```

```
docker --version
```

8. Probar Docker con una imagen de prueba.

```
docker run hello-world
```

## 4.2.- Instalación Nomad

1. Actualizar paquetes e instalar dependencias necesarias.

```
sudo apt update &&  sudo apt install wget gpg coreutils
```

2. Añadir la clave de Hashicorp

```
wget -O- https://apt.releases.hashicorp.com/gpg | \
  sudo gpg --dearmor -o
/usr/share/keyrings/hashicorp-archive-keyring.gpg
```

3. Añadir el repositorio oficial de Hashicorp

```
echo "deb
[signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg]
https://apt.releases.hashicorp.com $(lsb_release -cs) main" \
| sudo tee /etc/apt/sources.list.d/hashicorp.list
```

4. Actualizar e instalar Nomad

```
sudo apt update && sudo apt install nomad
```

5. Comprobar la instalación

```
nomad --version
```

## 4.3.- Instalación Consul

1. Descargar clave GPG de Hashicorp

```
wget -O - https://apt.releases.hashicorp.com/gpg | sudo gpg
--dearmor -o
/usr/share/keyrings/hashicorp-archive-keyring.gpg
```

2. Añadir el repositorio de Consul.

```
echo "deb [arch=$(dpkg --print-architecture)
signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg]
https://apt.releases.hashicorp.com $(lsb_release -cs) main" |
sudo tee /etc/apt/sources.list.d/hashicorp.list
```

3. Actualizar paquetes e instalar Consul.

```
sudo apt update && sudo apt install consul
```

4. Crear fichero de configuración de Consul

```
cat /etc/consul.d/consul.hcl

datacenter = "dc1"
data_dir = "/opt/consul"
log_level = "INFO"

server = true
bootstrap_expect = 1
ui_config{
  enabled = true
}

client_addr = "0.0.0.0"
bind_addr = "192.168.122.66"
advertise_addr = "192.168.122.66"

addresses {
  http = "0.0.0.0"
}

connect {
  enabled = true
}
```

5. Crear unidad systemd para Consul

```
cat /usr/lib/systemd/system/consul.service

[Unit]
Description="HashiCorp Consul - A service mesh solution"
Documentation=https://developer.hashicorp.com/consul
Requires=network-online.target
After=network-online.target
ConditionFileNotEmpty=/etc/consul.d/consul.hcl

[Service]
Type=simple
EnvironmentFile=-/etc/consul.d/consul.env
User=consul
Group=consul
ExecStart=/usr/bin/consul agent -config-dir=/etc/consul.d/
ExecReload=/bin/kill --signal HUP $MAINPID
KillMode=process
KillSignal=SIGTERM
Restart=on-failure
LimitNOFILE=65536

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

6. Habilitar e iniciar el servicio

```
sudo systemctl daemon-reexec
sudo systemctl daemon-reload
sudo systemctl enable consul
sudo systemctl start consul
```

## 4.4.- Instalación Prometheus

1. Actualizar paquetes.

```
sudo apt update
```

2. Crear usuario prometheus.

```
sudo useradd --no-create-home --shell /bin/false prometheus
```

3. Cambiar al directorio tmp.

```
cd /tmp
```

4. Descargar prometheus

```
curl -LO
https://github.com/prometheus/prometheus/releases/download/v2
.26.0/prometheus-2.26.0.linux-amd64.tar.gz
```

5. Descomprimir el fichero descargado.

```
tar -xvf prometheus-2.26.0.linux-amd64.tar.gz
```

6. Mover la carpeta a /etc/prometheus

```
sudo mv prometheus-2.26.0.linux-amd64 /etc/prometheus
```

7. Cambiar propietario

```
sudo chown -R prometheus:prometheus /etc/prometheus
```

8. Cambiar permisos

```
sudo chmod -R 755 /etc/prometheus
```

9. Crear el servicio de systemd

```
sudo cat /etc/systemd/system/prometheus.service

[Unit]
Description=Prometheus Monitoring
Wants=network-online.target
After=network-online.target

[Service]
User=prometheus
Group=prometheus
Type=simple
ExecStart=/etc/prometheus/prometheus
--config.file=/etc/prometheus/prometheus.yml
--storage.tsdb.path=/etc/prometheus/data

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

10. Habilitar y arrancar el servicio.

```
sudo systemctl daemon-reload
```

```
sudo systemctl enable prometheus
```

```
sudo systemctl start prometheus
```

## 4.5.- Instalación Nomad-Autoscaler

1. Descargar y descomprimir **go**

```
wget https://go.dev/dl/go1.24.0.linux-amd64.tar.gz
```

```
sudo tar -C /usr/local -xzf go1.24.0.linux-amd64.tar.gz
```

2. Configurar las variables de entorno para Go.

```
echo 'export PATH=$PATH:/usr/local/go/bin' >> ~/.bashrc
```

```
echo 'export PATH=$PATH:$HOME/go/bin' >> ~/.bashrc
```

```
source ~/.bashrc
```

```
go version
```

3. Clonar y compilar Nomad Autoscaler

```
sudo apt install git
```

```
git clone https://github.com/hashicorp/nomad-autoscaler.git
```

```
cd nomad-autoscaler
```

```
sudo apt install make
```

```
make dev
sudo cp bin/nomad-autoscaler /usr/local/bin/
```

## 5.- Demos

Los ficheros que se verán a continuación se pueden ver en la URL de GitHub del proyecto: https://github.com/alealbaladejo/Proyecto-Final-Nomad

## 5.1.- Despliegue nginx

```
cat nginx.hcl
job "nginx" {
  datacenters = ["dc1"]
  type = "service"

  group "nginx" {
    count = 1

    network {
      port "http" {
        static = 8086
        to = 80
      }
    }

    task "nginx" {
      driver = "docker"

      config {
        image = "nginx:latest"
        ports = ["http"]
      }

      service {
        provider = "nomad"
        name = "nginx"
        port = "http"
      }
    }
  }
}
```

A continuación se explica detalladamente el fichero anterior:
**job "nginx"**: Define un job nuevo que se va a llamar *nginx.*
**datacenters = ["dc1"]:** Indica que el job se desplegará en el datacenters *dc1* que por defecto es el que viene configurado.

**type = "service":** Dice que este job es un servicio, por lo que debe mantenerse corriendo.
**group "nginx":** Define un grupo de tareas con el nombre *nginx*.
**count = 1:** Se lanzará una única instancia del grupo.

**network:** Define la configuración de red para el grupo
**port "http":** Declara un puerto que se llama *http*
**static = 8086:** Reserva el puerto 8086 del host
**to = 80:** Se redirige al puerto 80 del contenedor.

**task "nginx":** Define una tarea llamada *nginx*.
**driver = "docker":** Usará docker como driver para ejecutar la tarea.

**image:** Será la imagen docker que usará
**ports:** Lista con los puertos que se expondrán desde el contenedor. Debe coincidir con los definidos en "network"

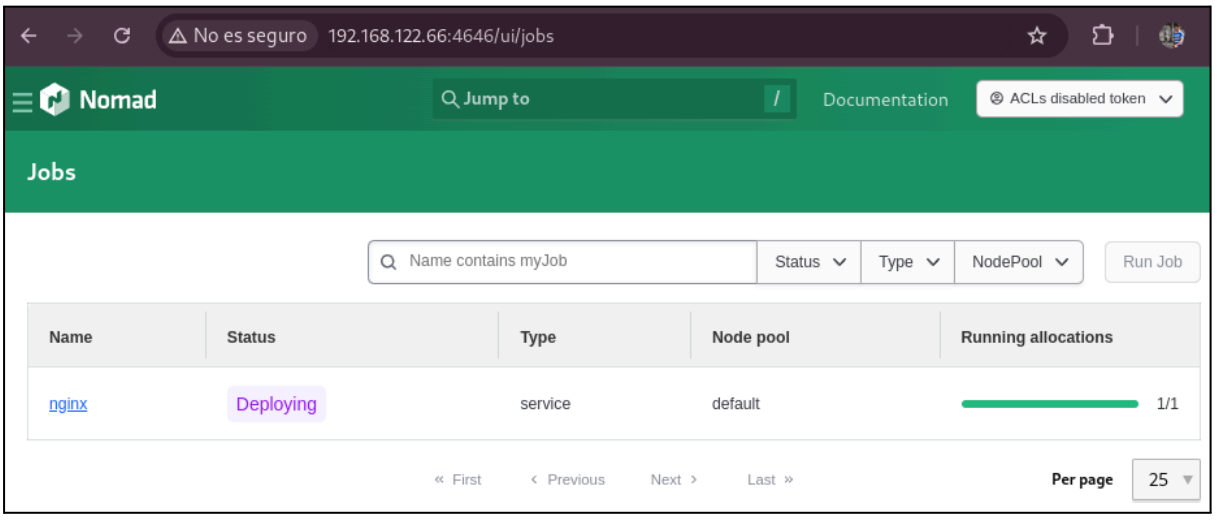
**service:** Permite que Consul o Nomad conozcan este servicio.
**provider = "nomad":** Indica que se usa el proveedor de servicios interno de Nomad
**name = "nginx":** Nombre del servicio.
**port = "http":** Puerto definido como *http*

Sabiendo ya el significado de cada apartado anterior, ejecutamos los comandos necesarios para desplegarlo con Nomad:

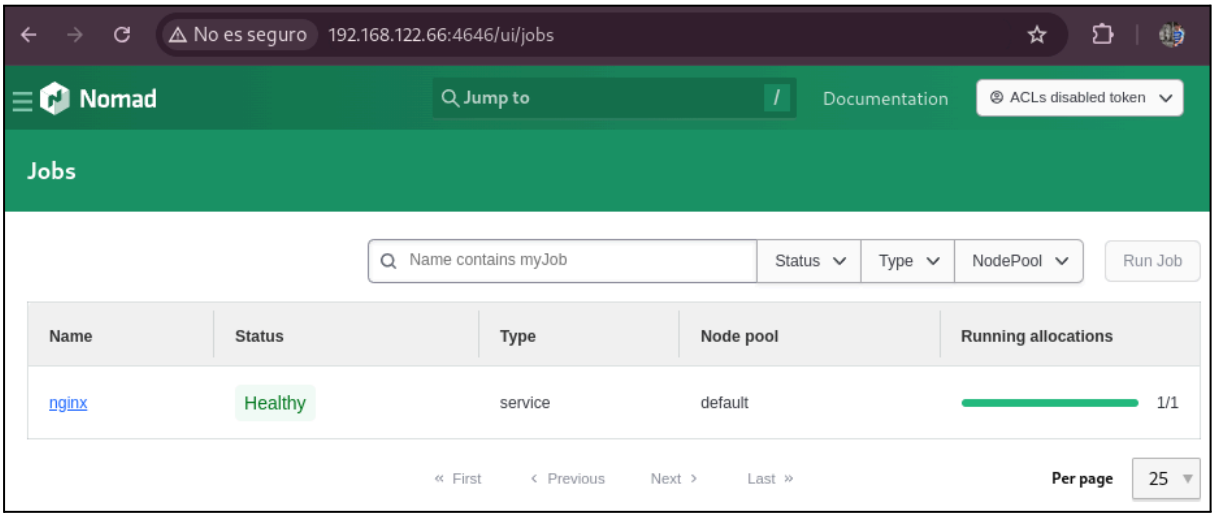
Primero desplegamos el job con:

```
nomad job run nginx.hcl
```

Y mientras se despliega podemos ver como se ve en la interfaz gráfica (192.168.122.66:4646):



Una vez que termina de desplegarse:




Vemos que el estado ahora ha pasado de **Deploying** a **Healthy**

Podemos ver información desde la propia terminal, como por ejemplo, conocer detalles sobre una instancia (allocation):

```
nomad job status nginx
ID            = nginx
Name          = nginx
Submit Date   = 2025-05-19T16:01:02+02:00
Type          = service
Priority      = 50
Datacenters   = dc1
Namespace     = default
Node Pool     = default
Status        = running
Periodic      = false
Parameterized = false

Summary
Task Group  Queued  Starting  Running  Failed  Complete  Lost
Unknown
nginx       0       0         1        0       0         0
0

Latest Deployment
ID          = ed4d7023
Status      = successful
Description = Deployment completed successfully
```

```
Deployed
Task Group  Desired  Placed  Healthy  Unhealthy  Progress
Deadline
nginx       1        1       1        0
2025-05-19T16:11:15+02:00

Allocations
ID        Node ID   Task Group  Version  Desired  Status
Created   Modified
7ade50b5  73516670  nginx       0        run      running
6m3s ago  5m50s ago
```

Ahora ya conocemos el id (7ade50b5) y podemos especificar un poco más la información:

```
nomad alloc status 7ade50b5
…
Created             = 9m40s ago
Modified            = 9m27s ago
Deployment ID       = ed4d7023
Deployment Health   = healthy

Allocation Addresses:
Label  Dynamic  Address
*http  yes      192.168.122.66:8086 -> 80

Task "nginx" is "running"
Task Resources:
CPU        Memory           Disk     Addresses
0/100 MHz  4.7 MiB/300 MiB  300 MiB

Recent Events:
Time                       Type        Description
2025-05-19T16:01:05+02:00  Started     Task started by client
2025-05-19T16:01:03+02:00  Driver      Downloading image
2025-05-19T16:01:02+02:00  Task Setup  Building Task
Directory
2025-05-19T16:01:02+02:00  Received    Task received by
client
```

Aquí podemos ver información como el tiempo desde que se creó o modificó, el estado, la dirección en la que se está ejecutando…

Para ver los logs podemos hacerlo con el comando:

```
nomad alloc logs 7ade50b5

/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty,
will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in
/docker-entrypoint.d/
/docker-entrypoint.sh: Launching
/docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum
of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6
in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Sourcing
/docker-entrypoint.d/15-local-resolvers.envsh
/docker-entrypoint.sh: Launching
/docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching
/docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for
start up
```

Y si queremos entrar en la máquina podemos hacerlo con:

```
nomad alloc exec -task nginx -t 7ade50b5 /bin/bash
root@077304aba3d6:/#
```

Ahora pasamos a comprobar que nos está desplegando Nginx en el puerto indicado:

```
curl http://192.168.122.66:8086

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
```

```
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully
installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Por último podemos comprobar que se ha creado un contenedor docker con un Nginx que sirve en el puerto 8086:

```
docker ps

CONTAINER ID   IMAGE          COMMAND
CREATED          STATUS          PORTS
NAMES
2f39465254cc   nginx:latest   "/docker-entrypoint...."   41
seconds ago   Up 40 seconds   192.168.1.136:8086->80/tcp,
192.168.1.136:8086->80/udp
nginx-ed07c964-174f-cb75-814f-13246f96dadc
```

## 5.2.- Apache con un fichero local

El objetivo de esta demo es desplegar un contenedor Apache con un fichero *index.html* personalizado desde la máquina local.
Para empezar la estructura del proyecto sería la siguiente:

```
tree
.
├── apache.hcl
└── web
    └── index.html

2 directories, 2 files
```

Donde el fichero que vamos a montar como index.html es el siguiente:

```
cat web/index.html

<h1>Proyecto Nomad Hashicorp</h1>
<h4>Alejandro</h4>
```

En Nomad, para especificar un volumen en un job, antes tendremos que configurar el fichero de configuración, añadiendo el volumen en el cliente:

```
cat /etc/nomad.d/nomad.hcl
…
client {
  enabled = true
  servers = ["127.0.0.1"]

  host_volume "webapache" {
    path =
"/home/ale/Proyecto-Final-Nomad/apache-fichero/web"
    read_only = false
  }
}
```

En este caso habrá que **añadir la ruta absoluta** dónde se encuentra el fichero *index.html* que vamos a utilizar.
Ahora reiniciamos el servicio:

```
sudo systemctl restart nomad.service
```

Y ahora nos fijamos en el job creado:

```
cat apache.hcl

job "apache" {
  datacenters = ["dc1"]
  type = "service"

  group "apache" {
    count = 1
    volume "webapache" {
      type = "host"
      read_only = false
      source = "webapache"
    }
    network {
```

```
      port "http" {
        static = 8085
        to = 80
      }
    }
    task "apache" {
      driver = "docker"
      config {
        image = "httpd:2.4"
        ports = ["http"]
      }
      volume_mount {
        volume = "webapache"
        destination = "/usr/local/apache2/htdocs"
        read_only = false
      }
      resources {
        cpu    = 500
        memory = 256
      }
      service {
        name     = "apache"
        port     = "http"
        provider = "nomad"
      }
    }
  }
}
```

Lo nuevo añadido aquí sería**:**
**volume "webapache":** Especifica el nombre que va a tener este volumen.
**type = "host":** Indica que va a ser una ruta en el sistema de archivo del host.
**read_only = false:** El contenedor tendrá permiso de escritura y lectura sobre el volumen.
**source = "webapache":** Debe coincidir con el nombre que hemos registrado en el cliente en */etc/nomad.d/nomad.hcl*.

Y por otra parte:
**volume_mount:** Esta parte decide dónde se va a montar el volumen dentro del contenedor.
**volume = "webapache":** Especifica el nombre del volumen (declarado arriba) que se va a montar en esta tarea.
**destination = "/usr/local/apache2/htdocs":** Indica la ruta dentro del contenedor donde se va a montar el volumen.
**read_only = false:** El contenedor podrá modificar los archivos en esa ruta.

Una vez que tengamos esto claro y bien configurado, podemos pasar a lanzar el job a Nomad:

```
nomad job run apache.hcl
…
  ✓ Deployment "81ae82e1" successful

    2025-05-19T17:30:43+02:00
    ID          = 81ae82e1
    Job ID      = apache
    Job Version = 6
    Status      = successful
    Description = Deployment completed successfully

    Deployed
    Task Group  Desired  Placed  Healthy  Unhealthy  Progress
Deadline
    apache      1        1       1        0
2025-05-19T17:32:49+02:00
```

Y podemos comprobar que se ha configurado correctamente si accedemos a la url:

```
curl 192.168.122.66:8085

<h1>Proyecto Nomad Hashicorp<h1>
<h4>Alejandro</h4>
```

Si ahora añadimos una línea a este fichero desde el host, veremos como automáticamente cambia en el contenedor:

```
echo "<h5>Línea añadida</h5>" >> web/index.html
curl 192.168.122.66:8085

<h1>Proyecto Nomad Hashicorp<h1>
<h4>Alejandro</h4>
<h5>Línea añadida</h5>
```

## 5.3.- Job con variables MariaDB

A continuación vamos a desplegar una base de datos MariaDB, para entender el funcionamiento de variables en los jobs de Nomad.
Lo primero vamos a comentar las cosas nuevas del job:

```
cat mariadb.hcl

job "mariadb" {
  datacenters = ["dc1"]
  type        = "service"

  group "db" {
    count = 1

    network {
      port "db" {
        to = 3306
     static = 3306
      }
    }

    task "mariadb" {
      driver = "docker"

      config {
        image = "mariadb:latest"
        ports = ["db"]
      }

      template {
        destination = "${NOMAD_SECRETS_DIR}/db.env"
        env         = true
        change_mode = "restart"

        data = <<EOT
MYSQL_ROOT_PASSWORD={{ with nomadVar
"nomad/jobs/mariadb/db/mariadb" }}{{ .MYSQL_ROOT_PASSWORD
}}{{ end }}
MYSQL_DATABASE={{ with nomadVar
"nomad/jobs/mariadb/db/mariadb" }}{{ .MYSQL_DATABASE }}{{ end
}}
MYSQL_USER={{ with nomadVar "nomad/jobs/mariadb/db/mariadb"
```

```
}}{{ .MYSQL_USER }}{{ end }}
MYSQL_PASSWORD={{ with nomadVar
"nomad/jobs/mariadb/db/mariadb" }}{{ .MYSQL_PASSWORD }}{{ end
}}
EOT
      }
      service {
        provider = "nomad"
        name     = "mariadb"
        port     = "db"
        tags     = ["db", "mariadb"]
      }

      resources {
        cpu    = 500
        memory = 512
      }
    }
  }
}
```

La parte nueva que hemos añadido aquí que no hemos visto antes es la opción de **template**, que básicamente define un bloque de plantilla que Nomad procesa antes de iniciar la tarea. Normalmente se usa para inyectar variables de entorno.

En este bloque:
**destination = "${NOMAD_SECRETS_DIR}/db.env":** Es la ruta dentro del contenedor donde se escribirá el contenido.
**env = true:** Indica que cada línea del archivo generado se interpretará como una variable de entorno con el formato clave - valor y será inyectada al contenedor.
**data = <<EOT … EOT>>:** Aquí se define el contenido de la plantilla.
**{{ with nomadVar "nomad/jobs/mariadb/db/mariadb" }}:** El bloque **with** carga el mapa de variables almacenado en la ruta del sistema de variables de Nomad.
**{{ .MYSQL_ROOT_PASSWORD }}{{ end }}:** Aquí se accede a una clave específica dentro del mapa cargado por nomadVar, que en este caso es MYSQL_ROOT_PASSWORD. {{end}} cierra el bloque with.

En definitiva, lo que hace es cargar las variables que previamente se han añadido con el comando:

```
nomad var put nomad/jobs/mariadb/db/mariadb \
MYSQL_ROOT_PASSWORD=root_password \
MYSQL_DATABASE=database \
MYSQL_USER=usuario \
MYSQL_PASSWORD=my_password
```

Ahora ejecutamos el job:

```
nomad job run mariadb.hcl
```

Si entramos en el contenedor podemos ver que se han añadido estas variables:

```
nomad alloc exec -task mariadb -t 449a6cbd /bin/bash

root@2d62addf2eb7:/# echo $MYSQL_DATABASE
database
```

Para verlas con un comando nomad:

```
nomad var get nomad/jobs/mariadb/db/mariadb

Namespace   = default
Path        = nomad/jobs/mariadb/db/mariadb
Create Time = 2025-05-20T16:26:00+02:00
Check Index = 669

Items
MYSQL_DATABASE      = database
MYSQL_PASSWORD      = mypassword
MYSQL_ROOT_PASSWORD = root_password
MYSQL_USER          = usuario
```

Podemos ver estas variables con la interfaz gráfica también pero antes tendríamos que configurar varias cosas, ya que por defecto no aparece la opción de ver las variables en la interfaz gráfica.

Lo primero que haremos es añadir una política de acl en la configuración de nomad:

```
cat /etc/nomad.d/nomad.hcl
…
acl {
  enabled = true
  token_ttl = "30s"
  policy_ttl = "30s"
  replication_token = ""
}
```

Una vez cambiado el fichero de configuración, reiniciamos el servicio.

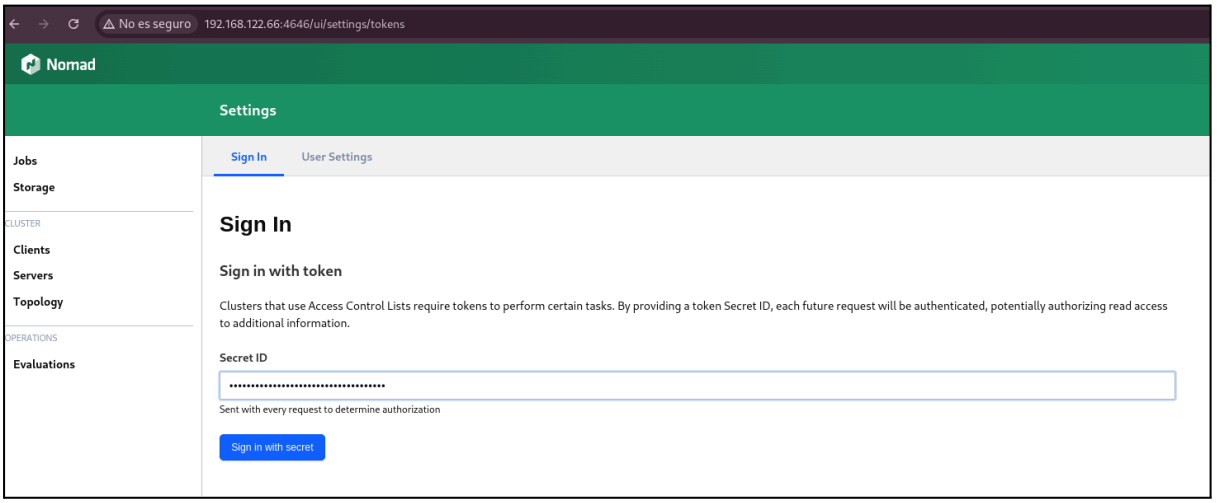
```
sudo systemctl restart nomad.service
```

Y ejecutamos:

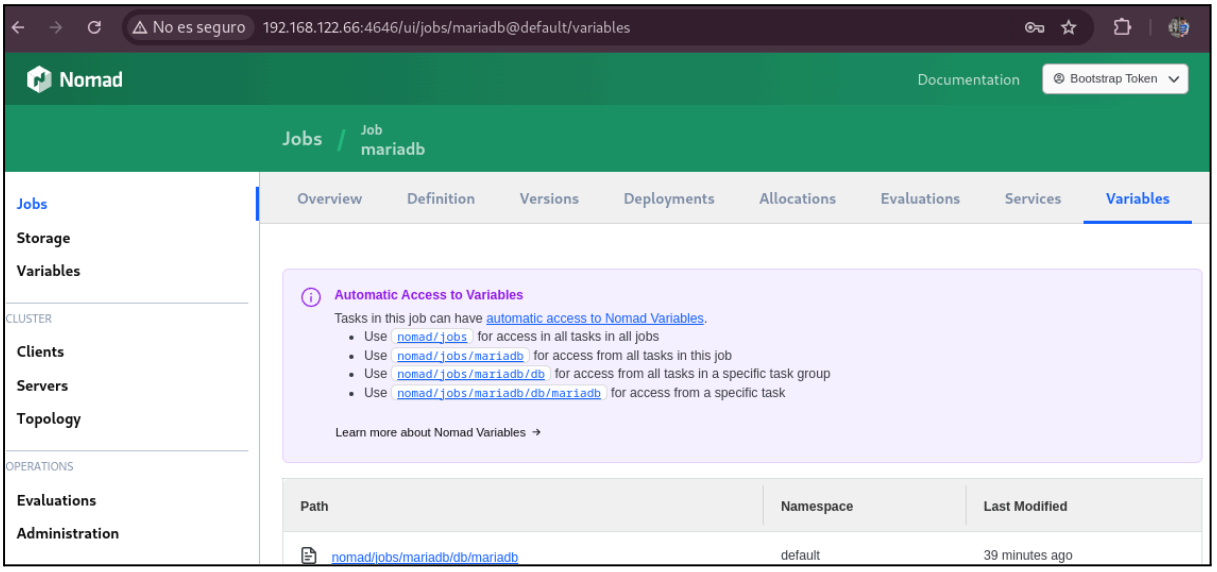
```
nomad acl bootstrap

Accessor ID  = 6214f5e8-0700-7f31-ed68-ff59553be74d
Secret ID    = e981364f-79a7-763f-627a-4c347947bd1d
...
```

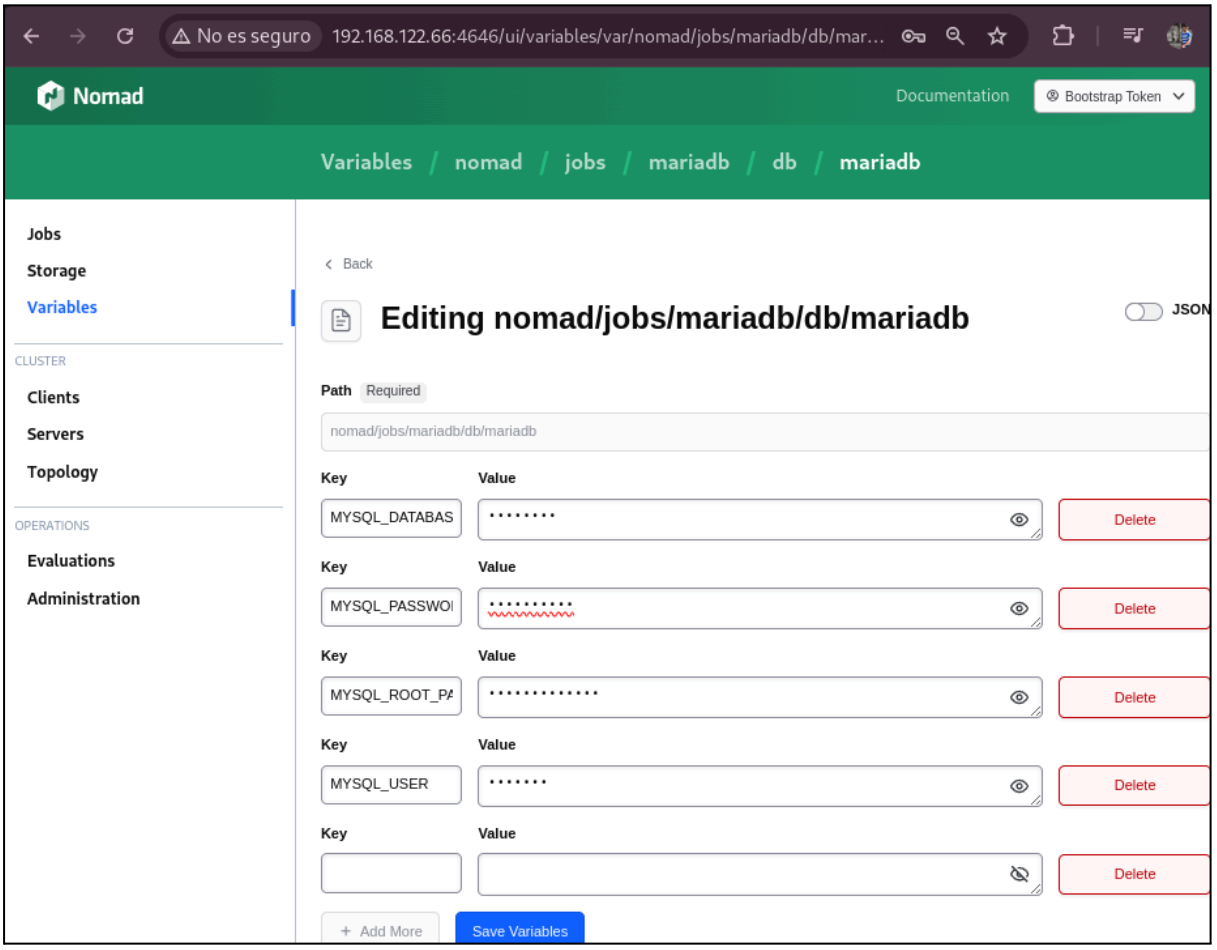
```
export NOMAD_TOKEN="e981364f-79a7-763f-627a-4c347947bd1d"
```

Ahora entramos en la interfaz gráfica y una vez añadida el secret ID como token:




Podremos ver las variables en la interfaz gráfica accediendo a la ruta nomad/jobs/mariadb/db/mariadb, una vez autenticados con el token generado anteriormente.






Ahora podemos acceder a esta base de datos desde fuera, por ejemplo:

```
mysql -u usuario -h 192.168.122.66 -D database -p

Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 10
Server version: 11.7.2-MariaDB-ubu2404 mariadb.org binary
distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and
others.

Type 'help;' or '\h' for help. Type '\c' to clear the current
input statement.

MariaDB [database]>
```

## 5.4.- Guestbook con Redis en el mismo job

Lo primero que haremos será añadir un volumen para que redis guarde sus datos ahí y así hacerlo persistente.
Creamos el directorio donde se guardará la información

```
sudo mkdir -p /opt/nomad/redis
```

Ahora modificamos el fichero de configuración para declarar el volumen:

```
cat /etc/nomad.d/nomad.hcl
...
client {
  enabled = true
  servers = ["127.0.0.1"]
…
  host_volume "redis_volume" {
    path = "/opt/nomad/redis"
    read_only = false
  }
}
```

Reiniciamos el servicio

```
sudo systemctl restart nomad.service
```

A continuación declaramos un único job, que va a contener tanto la aplicación de Guestbook, como la base de datos Redis.

```
cat guestbook.hcl

job "guestbook" {
  datacenters = ["dc1"]

  group "guestbook-group" {
    network {
      port "http" {
        to = 5000
        static = 8083
      }
      port "redis" {
     to = 6379
     static = 6379
      }
```

```
    }
    volume "redis_data" {
      type      = "host"
      read_only = false
      source    = "redis_volume"
    }

    task "redis" {
      driver = "docker"

      config {
        image = "redis"
        command = "redis-server"
        args = ["--appendonly", "yes"]
        ports = ["redis"]
      }

      volume_mount {
        volume      = "redis_data"
        destination = "/data"
        read_only   = false
      }

      resources {
        cpu    = 200
        memory = 256
      }
    }

    task "app" {
      driver = "docker"

      config {
        image = "iesgn/guestbook"
        ports = ["http"]
      }
      env {
        REDIS_SERVER = "${NOMAD_IP_redis}"
      }

      resources {
        cpu    = 200
```

```
        memory = 256
      }

      service {
        provider = "nomad"
        name = "guestbook"
        port = "http"
      }
    }

  }
}
```

En este job definimos dos puertos, uno para la aplicación que se ejecuta en el puerto 5000 y se mapea al 8083 y otro para redis que se ejecuta en el puerto 6379.

En la tarea de la app, no hace falta que pongamos la IP del contenedor donde corre redis para que se conecte a la base de datos, sino que especificamos con la variable que nos proporciona Nomad para inyectar la IP.

Ahora desplegamos el job:

```
nomad job run guestbook.hcl
```

Y podemos ver desde el navegador en el puerto especificado (8083) la aplicación corriendo.

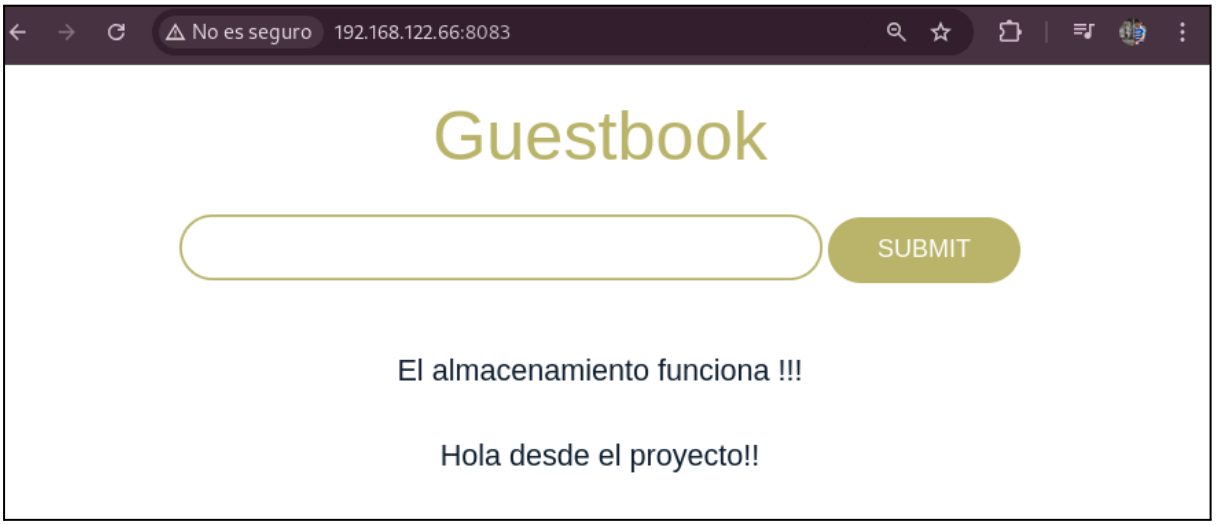
Ahora para comprobar la persistencia, borramos por completo el job.

```
nomad job stop -purge guestbook

==> 2025-05-22T13:33:33+02:00: Monitoring evaluation
"abc51f5c"
    2025-05-22T13:33:33+02:00: Evaluation triggered by job
"guestbook"
    2025-05-22T13:33:33+02:00: Evaluation status changed:
"pending" -> "complete"
==> 2025-05-22T13:33:33+02:00: Evaluation "abc51f5c" finished
with status "complete"
```

Y una vez borrado, volvemos a desplegar la aplicación:

```
nomad job run guestbook.hcl
```

Veremos cómo el mensaje que enviamos antes, sigue estando presente.




## 5.5.- Guestbook con Redis en dos jobs distintos

Ahora pasamos a dividir el job anterior en dos jobs distintos. Esto nos va a causar un problema, ya que por defecto Nomad no proporciona un mecanismo para el descubrimiento de servicios entre jobs distintos, es decir no es capaz de averiguar la IP de un servicio, que está desplegado en otro job distinto y tendríamos que poner en el propio job de Guestbook, la dirección IP de la máquina donde corre redis, por lo que va en contra de las buenas prácticas. Lo podemos ver en el siguiente ejemplo:

A continuación vemos el fichero de redis:

```
cat redis.hcl

job "redis" {
  datacenters = ["dc1"]

  group "redis" {
    count = 1

    network {
      port "db" {
        static = 6379
        to = 6379
      }
    }

    task "redis" {
      driver = "docker"

      config {
        image = "redis:7"
        ports = ["db"]
      }

      resources {
        cpu    = 100
        memory = 128
      }

      service {
     provider = "nomad"
        name = "redis"
        port = "db"
      }

    }
  }
}
```

Y ahora vemos el de guestbook

```
cat guestbook.hcl
job "guestbook" {
  datacenters = ["dc1"]

  group "guestbook-group" {
    network {
      port "http" {
        to     = 5000
        static = 8087
      }
    }

    task "app" {
      driver = "docker"

      config {
        image = "iesgn/guestbook"
        ports = ["http"]
      }

      template {
        data = <<EOF
REDIS_SERVER=192.168.122.66
EOF
        destination = "secrets/env"
        env         = true
      }

      resources {
        cpu    = 200
        memory = 256
      }

      service {
        provider = "nomad"
        name     = "guestbook"
        port     = "http"
      }
    }
  }
}
```

Como hemos dicho esta no es la manera recomendable, ya que la IP puede ser dinámica y nos puede dar un quebradero de cabeza.

Para evitar esto, Hashicorp proporciona una herramienta llamada **Consul** que nos ayuda al descubrimiento de servicios.

En el apartado de instalaciones de esta misma memoria se puede ver como instalar Consul en Debian 12.
Así que pasamos directamente a iniciar Consul.

```
sudo systemctl start consul.service
```

```
sudo systemctl status consul.service
● consul.service - "HashiCorp Consul - A service mesh
solution"
     Loaded: loaded (/lib/systemd/system/consul.service;
disabled; preset: enabled)
     Active: activating (start) since Thu 2025-05-22 16:51:12
CEST; 14s ago
       Docs: https://developer.hashicorp.com/
   Main PID: 19791 (consul)
      Tasks: 11 (limit: 5710)
     Memory: 27.9M
        CPU: 361ms
     CGroup: /system.slice/consul.service
             └─19791 /usr/bin/consul agent
-config-dir=/etc/consul.d/
```

Otra forma en desarrollo podría ser con el comando siguiente en una pestaña distinta

```
consul agent -dev
```

Una vez tengamos Consul bien configurado, pasamos a modificar los jobs para el despliegue. En el de redis.hcl únicamente vamos a eliminar la línea de **provider = "nomad"**, para que Consul lo pueda registrar.
Por lo que quedaría:

```
job "redis" {
  datacenters = ["dc1"]

  group "redis" {
    count = 1

    network {
      port "db" {
        static = 6379
        to = 6379
      }
    }

    task "redis" {
      driver = "docker"

      config {
        image = "redis:7"
        ports = ["db"]
      }

      resources {
        cpu    = 100
        memory = 128
      }

      service {
        name = "redis"
        port = "db"
      }
    }
  }
}
```

Y en el job de guestbook quitaremos la IP como hemos dicho y añadiremos:
**REDIS_SERVER={{ with service "redis" }}{{ (index . 0).Address }}{{ end }}**

Esta línea lo que hace es consultar en Consul un servicio que se llame "*redis*" y accede a la primera instancia (**index . 0**) para recibir la dirección IP (**.Address**)

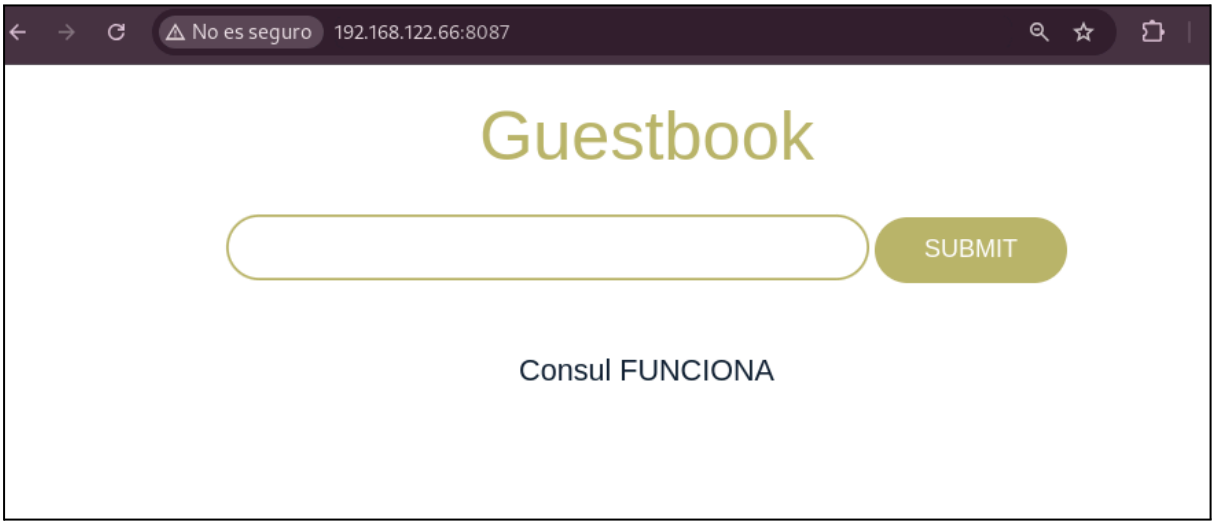
Quedaría así el fichero

```
job "guestbook" {
  datacenters = ["dc1"]

  group "guestbook-group" {
    network {
      port "http" {
        to     = 5000
        static = 8087
      }
    }

    task "app" {
      driver = "docker"

      config {
        image = "iesgn/guestbook"
        ports = ["http"]
      }

      template {
        data = <<EOF
REDIS_SERVER={{ with service "redis" }}{{ (index . 0).Address
}}{{ end }}
EOF
        destination = "secrets/env"
        env         = true
      }

      resources {
        cpu    = 200
        memory = 256
      }

      service {
        provider = "nomad"
        name     = "guestbook"
        port     = "http"
      }
    }
  }
}
```

Ahora podemos desplegar los dos servicios

```
nomad job run redis.hcl
nomad job run guestbook.hcl
```

Y podemos comprobar que funciona correctamente accediendo al navegador web.




## 5.6. Wordpress con MariaDB

Ahora vamos a desplegar un Wordpress que se conecte a una base de datos MariaDB, usando variables, Consul y volumen persistente.

Lo primero que hacemos es crear un directorio para el volumen

```
sudo mkdir -p /opt/nomad/bd-wp
```

Ahora modificamos el fichero de configuración de Nomad para añadir este volumen al cliente

```
sudo cat /etc/nomad.d/nomad.hcl
...
client {
...
  host_volume "mariadb_volume" {
    path = "/opt/nomad/bd-wp"
    read_only = false
  }
}
```

Una vez hecho esto reiniciamos el servicio de Nomad.

```
sudo systemctl restart nomad
```

Ahora vamos a pasar a crear las variables necesarias que necesitarán tanto MariaDB como Wordpress.

```
nomad var put --force nomad/jobs  \
MYSQL_ROOT_PASSWORD="root_password" \
MYSQL_DATABASE="database" \
MYSQL_USER="usuario" \
MYSQL_PASSWORD="my_password"
```

Y pasamos a desplegar los jobs tanto de MariaDB como de Wordpress.

```
cat mariadb.hcl

job "mariadb" {
  datacenters = ["dc1"]

  group "db" {
    network {
      port "db" {
        static = 3306
      }
    }

    volume "db-wp" {
      type      = "host"
      read_only = false
      source    = "mariadb_volume"
    }

    task "mariadb" {
      driver = "docker"

      config {
        image = "mariadb:10.11"
        ports = ["db"]
      }

      volume_mount {
        volume      = "db-wp"
        destination = "/var/lib/mysql"
        read_only   = false
      }
```

```
      template {
        data = <<EOF
MYSQL_ROOT_PASSWORD={{ with nomadVar "nomad/jobs" }}{{
.MYSQL_ROOT_PASSWORD }}{{ end }}
MYSQL_DATABASE={{ with nomadVar "nomad/jobs" }}{{
.MYSQL_DATABASE }}{{ end }}
MYSQL_USER={{ with nomadVar "nomad/jobs" }}{{ .MYSQL_USER
}}{{ end }}
MYSQL_PASSWORD={{ with nomadVar "nomad/jobs" }}{{
.MYSQL_PASSWORD }}{{ end }}
EOF
        destination = "secrets/env"
        env         = true
      }

      service {
        name = "mariadb"
        port = "db"
        tags = ["db"]
        check {
          type     = "tcp"
          interval = "10s"
          timeout  = "2s"
        }
      }

      resources {
        cpu    = 300
        memory = 256
      }
    }
  }
}
```

Y el fichero del wordpress

```
cat wordpress.hcl

job "wordpress" {
  datacenters = ["dc1"]

  group "web" {
```

```
    network {
      port "http" {
        static = 8080
        to = 80
      }
    }

    task "wordpress" {
      driver = "docker"

      config {
        image = "wordpress:6.8"
        ports = ["http"]
      }

      template {
        data = <<EOF
WORDPRESS_DB_NAME={{ with nomadVar "nomad/jobs" }}{{
.MYSQL_DATABASE }}{{ end }}
WORDPRESS_DB_USER={{ with nomadVar "nomad/jobs" }}{{
.MYSQL_USER }}{{ end }}
WORDPRESS_DB_PASSWORD={{ with nomadVar "nomad/jobs" }}{{
.MYSQL_PASSWORD }}{{ end }}
WORDPRESS_DB_HOST={{ with service "mariadb" }}{{ (index .
0).Address }}:{{ (index . 0).Port }}{{ end }}
EOF
        destination = "secrets/env"
        env         = true
      }
      service {
        name = "wordpress"
        port = "http"
        tags = ["web"]
        check {
          type     = "http"
          path     = "/"
          interval = "10s"
          timeout  = "2s"
        }
      }

      resources {
```

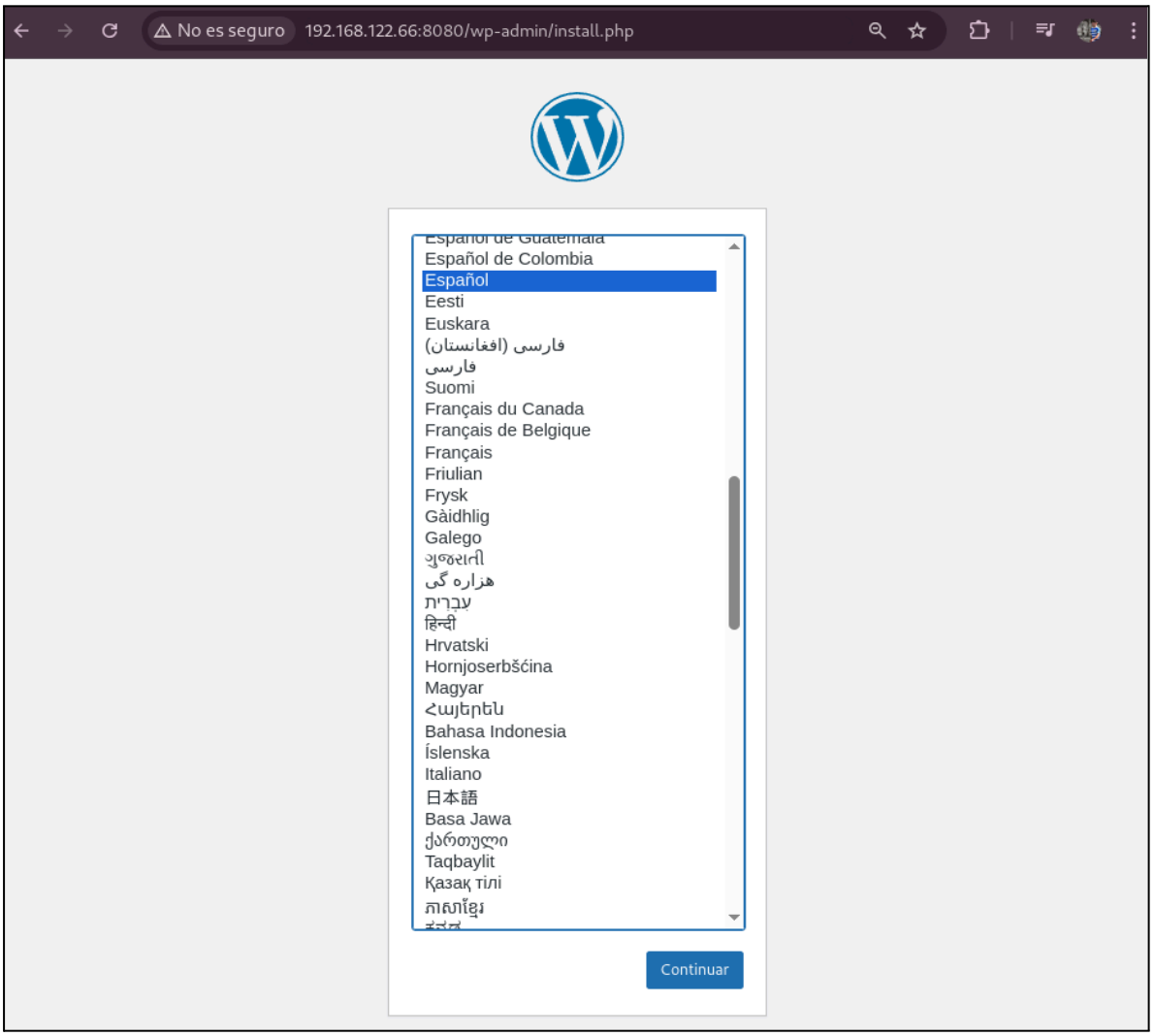
```
        cpu    = 500
        memory = 512
      }
    }
  }
}
```

Donde hemos añadido tanto variables de Nomad, como servicios de Consul. Cogemos las variables que ya creamos anteriormente para la base de datos, no tenemos que crear nuevas variables para Wordpress, ya que el contenido de estas debe ser el mismo.

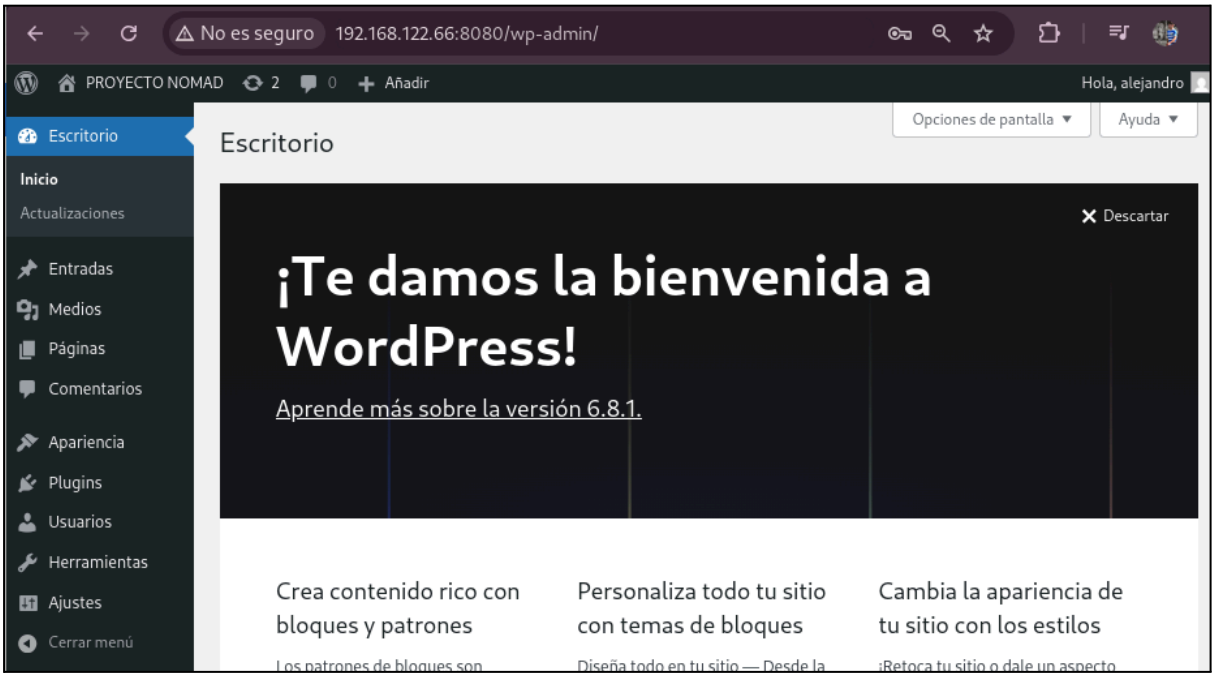
Desplegamos la base de datos y la aplicación.

```
nomad job run mariadb.hcl
nomad job run wordpress.hcl
```

Y accedemos al navegador por el puerto 8080 del servidor para ver el Wordpress



Una vez terminada la instalación podemos ver que ya tenemos el Wordpress desplegado.




## 6.- Escalado manual Nomad

A continuación vamos a ver cómo realizar un escalado manual de una aplicación en Nomad Hashicorp, utilizando Consul como servicio de descubrimiento y Nginx como balanceador de carga.

Lo primero que tendremos es el fichero de la aplicación que será la que escalaremos.

```
cat webapp.hcl

job "demo-webapp" {
  datacenters = ["dc1"]

  group "demo" {
    count = 3

    network {
      port "http" {
        to = -1
      }
    }

    service {
      name = "demo-webapp"
```

```
      port = "http"

      check {
        type     = "http"
        path     = "/"
        interval = "2s"
        timeout  = "2s"
      }
    }

    task "server" {
      driver = "docker"

      env {
        PORT    = "${NOMAD_PORT_http}"
        NODE_IP = "${NOMAD_IP_http}"
      }

      config {
        image = "hashicorp/demo-webapp-lb-guide"
        ports = ["http"]
      }
    }
  }
}
```

Hemos usado una imagen de prueba que proporciona Nomad Hashicorp en su documentación oficial.
Esto lanzará 3 contenedores con la misma app.
Lo nuevo que se ve aquí es el port = -1 que lo que hace es asignar un puerto aleatorio para cada instancia.
El servicio se registra en Consul como demo-webapp

Esto lo lanzamos

```
nomad job run webapp.hcl
```

Pero para poder acceder a la aplicación desde el navegador web y ver las distintas instancias, tendremos que usar un balanceador, que va a ser un nginx configurado como job de Nomad.

```
cat nginx.hcl

job "nginx" {
  datacenters = ["dc1"]

  group "nginx" {
    count = 1

    network {
      port "http" {
        static = 8080
      }
    }

    service {
      name = "nginx"
      port = "http"
    }

    task "nginx" {
      driver = "docker"

      config {
        image = "nginx"
        ports = ["http"]

        volumes = [
          "local:/etc/nginx/conf.d",
        ]
      }

      template {
        data = <<EOF
upstream backend {
{{ range service "demo-webapp" }}
  server {{ .Address }}:{{ .Port }};
{{ else }}server 127.0.0.1:65535; # force a 502
{{ end }}
}

server {
   listen 8080;
```

```
    location / {
        proxy_pass http://backend;
    }
}
EOF

        destination   = "local/load-balancer.conf"
        change_mode   = "signal"
        change_signal = "SIGHUP"
      }
    }
  }
}
```

Se usa **template** para que Nomad rellene dinámicamente los upstreams con las IPs/puertos registrados por Consul del servicio demo-webapp.
Si algún servicio cambia, se vuelve a generar el fichero load-balancer.conf y NGINX se reinicia con SIGHUP.
Y lo lanzamos

```
nomad job run nginx.hcl
```

Ahora para ver el funcionamiento del escalado, podemos abrir una terminal nueva, donde monitoreamos los contenedores en docker cada 1 segundo.

```
watch -n 1 docker ps

CONTAINER ID   IMAGE                                  COMMAND
CREATED          STATUS          PORTS

NAMES
78e7dcae5296   nginx
"/docker-entrypoint.…"   3 minutes ago    Up 3 minutes
80/tcp, 192
.168.122.66:8080->8080/tcp, 192.168.122.66:8080->8080/udp
nginx-8ab3ca0b-0f2c-9875-3dca-e04e5c04503d

4b6a06077685   hashicorp/demo-webapp-lb-guide   "./main"
23 minutes ago   Up 23 minutes   192.168.122
.66:21228->21228/tcp, 192.168.122.66:21228->21228/udp
server-9764b087-61f3-3fe5-42f7-f1d281fa52ce
```

```
1b82b6be9e24   hashicorp/demo-webapp-lb-guide   "./main"
23 minutes ago   Up 23 minutes   192.168.122
.66:22749->22749/tcp, 192.168.122.66:22749->22749/udp
server-8239e750-ea37-0760-6c68-6497b1784ba1

26b71f8f8b64   hashicorp/demo-webapp-lb-guide   "./main"
23 minutes ago   Up 23 minutes   192.168.122
.66:24365->24365/tcp, 192.168.122.66:24365->24365/udp
server-55366a27-b110-5d57-8aae-30d30ddc88aa
```

Tenemos 3 instancias con la aplicación y 1 con nginx que corre por el puerto 8080 donde va balanceando la carga entre los 3 contenedores. Podemos comprobarlo accediendo a esa ip varias veces.

```
curl http://192.168.122.66:8080
Welcome! You are on node 192.168.122.66:21228

curl http://192.168.122.66:8080
Welcome! You are on node 192.168.122.66:24365

curl http://192.168.122.66:8080
Welcome! You are on node 192.168.122.66:22749
```

Como vemos nos balancea correctamente. Ahora pasamos a modificar el fichero de webapp, donde vamos  a poner que haya 6 contenedores.

```
cat webapp.hcl
...
  group "demo" {
    count = 6
...
```

Y lanzamos el job mientras en otra terminal, estamos monitoreando los contenedores. Veremos cómo se añaden 3 más aparte de los 3 que ya hay creados.

```
nomad job run webapp.hcl
```

```
CONTAINER ID   IMAGE                                COMMAND
CREATED          STATUS          PORTS
NAMES
da0a27d59ec5   hashicorp/demo-webapp-lb-guide   "./main"
12 seconds ago   Up 11 seconds   192.168.122.66:22409->22409/tcp,
192.168.122.66:22409->22409/udp
server-0cde9e8d-9758-9eeb-365a-368a438061ac

c485f35e9281   hashicorp/demo-webapp-lb-guide   "./main"
12 seconds ago   Up 11 seconds   192.168.122.66:20371->20371/tcp,
192.168.122.66:20371->20371/udp
server-bfe8c928-76f5-f43c-8510-57397a93dc62

54e53fc854ab   hashicorp/demo-webapp-lb-guide   "./main"
12 seconds ago   Up 11 seconds   192.168.122.66:27702->27702/tcp,
192.168.122.66:27702->27702/udp
server-4d1cccc2-0b94-9bad-c11d-c8202c61f9d6

78e7dcae5296   nginx
"/docker-entrypoint...."   8 minutes ago    Up 8 minutes
80/tcp, 192.168.122.66:8080->8080/tcp,
192.168.122.66:8080->8080/udp
nginx-8ab3ca0b-0f2c-9875-3dca-e04e5c04503d

4b6a06077685   hashicorp/demo-webapp-lb-guide   "./main"
28 minutes ago   Up 28 minutes   192.168.122.66:21228->21228/tcp,
192.168.122.66:21228->21228/udp
server-9764b087-61f3-3fe5-42f7-f1d281fa52ce

1b82b6be9e24   hashicorp/demo-webapp-lb-guide   "./main"
28 minutes ago   Up 28 minutes   192.168.122.66:22749->22749/tcp,
192.168.122.66:22749->22749/udp
server-8239e750-ea37-0760-6c68-6497b1784ba1

26b71f8f8b64   hashicorp/demo-webapp-lb-guide   "./main"
28 minutes ago   Up 28 minutes   192.168.122.66:24365->24365/tcp,
192.168.122.66:24365->24365/udp
server-55366a27-b110-5d57-8aae-30d30ddc88aa
```

Y si comprobamos de nuevo accediendo al balanceador, veremos como los puertos de los contenedores nuevos también nos aparecerán:

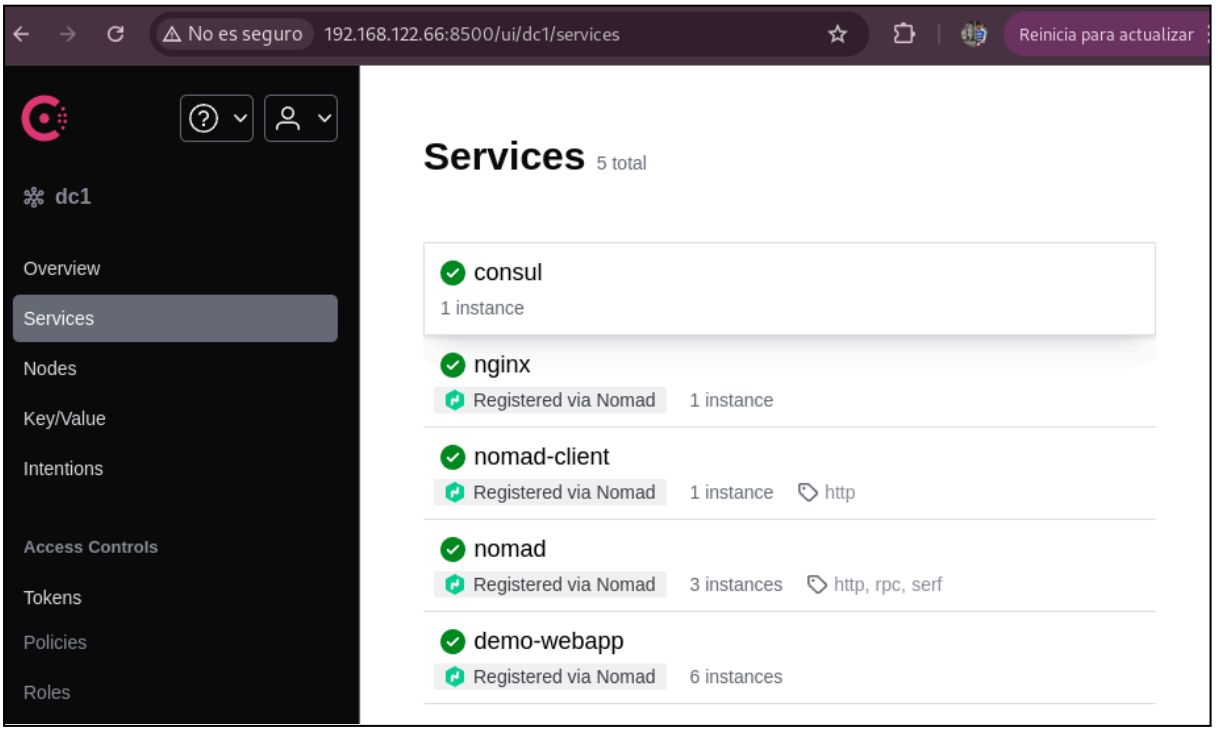
```
curl http://192.168.122.66:8080
Welcome! You are on node 192.168.122.66:21228

curl http://192.168.122.66:8080
Welcome! You are on node 192.168.122.66:20371

curl http://192.168.122.66:8080
Welcome! You are on node 192.168.122.66:22409

curl http://192.168.122.66:8080
Welcome! You are on node 192.168.122.66:27702
```

Con esto ya hemos conseguido un escalado manualmente, además de añadir un balanceador de carga de contenedores, que aún no lo habíamos visto.

Como ampliación podemos conocer el comando de Consul para saber que servicios tiene registrados:

```
consul catalog services
consul
demo-webapp
nginx
nomad
nomad-client
```

O incluso a través de la interfaz web de consul que corre por el puerto 8500.



## 7.- Autoescalado Nomad

A continuación realizaremos una demostración del autoescalado en Nomad. Para ello necesitaremos tener instalado nomad-autoscaler, como anteriormente se mostró.

Primero tenemos que tener un fichero de configuración para el nomad-autoscaler. Esto lo podemos tener de dos maneras.

1. Creamos el fichero de configuración autoscaler.hcl y ejecutamos el comando siguiente

```
nomad-autoscaler agent -config autoscaler.hcl
```

   De esta manera tendremos que tener una pestaña abierta todo el tiempo, por lo que en este caso, optaremos por crear un servicio de systemd.
2. La segunda manera sería creando el servicio de systemd, que será la que veremos a continuación.

### 7.1.- Servicio de systemd nomad-autoscaler

Como vemos en el **ExecStart** indicamos el binario de nomad-autoscaler y configuramos un ruta, que es donde se encontrará el fichero de configuración del autoscaler.
Pasamos a crear un directorio donde guardaremos este fichero:

```
mkdir -p /opt/nomad/autoscaler
```

Aquí dentro crearemos el siguiente fichero:

```
cat /opt/nomad/autoscaler/autoscaler.hcl

nomad {
  address = "http://192.168.122.66:4646"
  token   = "e981364f-79a7-763f-627a-4c347947bd1d"
}

telemetry {
  prometheus_metrics = true
  disable_hostname   = true
}

apm "prometheus" {
  driver = "prometheus"
  config = {
    address = "http://192.168.122.66:9090"
```

```
  }
}

strategy "target-value" {
  driver = "target-value"
}
```

El token que vemos lo podemos generar si tenemos las acl activadas, como es en este caso con:

```
nomad acl bootstrap
```

En el caso de que ya lo tengamos, podemos ejecutar:

```
echo $NOMAD_TOKEN
```

A continuación ejecutamos lo siguientes comandos:

```
sudo systemctl daemon-reexec
```

```
sudo systemctl daemon-reload
```

```
sudo systemctl start nomad-autoscaler
```

```
sudo systemctl enabled nomad-autoscaler
```

Y comprobamos que está activo:

```
sudo systemctl status nomad-autoscaler

● nomad-autoscaler.service - Nomad Autoscaler
     Loaded: loaded
(/etc/systemd/system/nomad-autoscaler.service; enabled;
preset: enabled)
     Active: active (running) since Fri 2025-05-30 17:10:43
CEST; 15min ago
   Main PID: 4489 (nomad-autoscale)
      Tasks: 11 (limit: 5710)
     Memory: 15.1M
        CPU: 1.204s
     CGroup: /system.slice/nomad-autoscaler.service
             └─4489 /usr/bin/nomad-autoscaler agent -config
/home/ale/autoescalado/autoscaler.hcl
```

Con esto ya tenemos configurado el autoscaler.

## 7.2.- Configuración Prometheus y Nomad

Ahora necesitamos también configurar **Prometheus** con **Nomad**, que será de donde coja las métricas para realizar el autoescalado.

En el fichero de Nomad añadiremos la telemetría al final del todo:

```
 cat /etc/nomad.d/nomad.hcl
...

telemetry {
  prometheus_metrics         = true
  publish_allocation_metrics = true
  publish_node_metrics       = true
  disable_hostname           = true
  collection_interval        = "5s"
}
```

Ahora en el fichero de **Prometheus** añadiremos el job_name de nomad:
El fichero completo quedaría así:

```
cat /etc/prometheus/prometheus.yml

# my global config
global:
  scrape_interval:     15s
  evaluation_interval: 15s

# Alertmanager configuration
alerting:
  alertmanagers:
  - static_configs:
    - targets:
      # - alertmanager:9093

rule_files:

scrape_configs:
  - job_name: 'prometheus'
    static_configs:
    - targets: ['192.168.122.66:9090']

  - job_name: 'nomad'
    metrics_path: /v1/metrics
```

```
    params:
      format: [prometheus]
    static_configs:
    - targets: ['192.168.122.66:4646']
```

Una vez tengamos los dos ficheros, reiniciamos ambos servicios:

```
sudo systemctl restart nomad.service
sudo systemctl restart prometheus
```

## 7.3.- Demo Autoscaler

Una vez que tengamos los servicios activos, podemos pasar a la parte de la demostración del autoescalado en Nomad.

Para ellos vamos a levantar una aplicación, que nos dice en qué instancia estamos, por lo que vamos a necesitar un balanceador de carga, por lo que usaremos nginx como balanceador.

Los ficheros que necesitaremos será uno para el balanceador y otro para la aplicación.

```
cat webapp.hcl

job "webapp" {
  datacenters = ["dc1"]
  type = "service"

  group "webapp" {
    count = 3

    network {
      port "http" {
        to = -1
      }
    }

    scaling {
      enabled = true
      min     = 1
      max     = 5
      policy {
        cooldown = "30s"
```

```
        check "avg_sessions" {
          source = "prometheus"
          query =
"avg(nomad_client_allocs_cpu_total_percent{exported_job=\"web
app\", task_group=\"webapp\", task=\"server\"})"
          strategy "target-value" {
            target = 30.0
         lookback = "30s"
         interval = "10s"
          }
        }
      }

    service {
      name     = "webapp"
      port     = "http"
      check {
     type = "http"
     path = "/"
     interval = "2s"
     timeout = "2s"
      }
    }

    task "server" {
      driver = "docker"

      env {
     PORT = "${NOMAD_PORT_http}"
     NODE_IP = "${NOMAD_IP_http}"
      }
      config {
        image = "hashicorp/demo-webapp-lb-guide"
        ports = ["http"]
      }
    }
  }
}
```

Y el fichero del balanceador nginx:

```
cat balanceador.hcl

job "nginx-lb" {
  datacenters = ["dc1"]

  group "nginx" {
    count = 1

    network {
      port "http" {
        static = 8050
      }
    }

    service {
      name = "nginx-lb"
      port = "http"
    }

    task "nginx" {
      driver = "docker"

      config {
        image = "nginx:latest"
        ports = ["http"]

        volumes = [
          "local:/etc/nginx/conf.d",
        ]
      }

      template {
        data = <<EOF
upstream backend {
{{ range service "webapp" }}
  server {{ .Address }}:{{ .Port }};
{{ else }}server 127.0.0.1:65535; # force a 502
{{ end }}
}

server {
   listen 8050;
```

```
    location / {
       proxy_pass http://backend;
    }
}
EOF
        destination   = "local/load-balancer.conf"
        change_mode   = "signal"
        change_signal = "SIGHUP"
      }

    }
  }
}
```

Desplegamos estos ficheros con nomad:

```
nomad job run webapp.hcl
nomad job run balanceador.hcl
```

Como el nomad-autoscaler está funcionando, esto escala tanto hacia arriba como hacia abajo, por lo que aunque hayamos puesto que se desplieguen 3 instancias del webapp, a los 20 segundos aproximadamente se dará cuenta que la métrica no supera los 30, por lo que escalará hacia abajo y nos quedaremos con 1 solo contenedor de cada uno.

Lo podemos ver:

```
docker ps
CONTAINER ID   IMAGE                                COMMAND
CREATED          STATUS          PORTS
NAMES
28761a8e6344   hashicorp/demo-webapp-lb-guide   "./main"
17 seconds ago   Up 17 seconds
192.168.122.66:24976->24976/tcp,
192.168.122.66:24976->24976/udp
server-022b4301-a204-93a8-e51a-a999496614a9

b4a08e34847a   hashicorp/demo-webapp-lb-guide   "./main"
17 seconds ago   Up 17 seconds
192.168.122.66:26125->26125/tcp,
192.168.122.66:26125->26125/udp
server-6b670643-b283-5cb7-d150-5c83096ddb3b
```

```
2c58d9685436   hashicorp/demo-webapp-lb-guide   "./main"
17 seconds ago   Up 17 seconds
192.168.122.66:24554->24554/tcp,
192.168.122.66:24554->24554/udp
server-3b3f41d7-7bef-60f7-5b66-29774708df5e

a96d17664a8c   nginx:latest
"/docker-entrypoint.…"   2 minutes ago    Up 2 minutes
80/tcp, 192.168.122.66:8050->8050/tcp,
192.168.122.66:8050->8050/udp
nginx-15bc0796-ed15-6963-6437-cdfee86ee035


docker ps
CONTAINER ID   IMAGE                                COMMAND
CREATED              STATUS              PORTS
NAMES
b4a08e34847a   hashicorp/demo-webapp-lb-guide   "./main"
About a minute ago   Up About a minute
192.168.122.66:26125->26125/tcp,
192.168.122.66:26125->26125/udp
server-6b670643-b283-5cb7-d150-5c83096ddb3b

a96d17664a8c   nginx:latest
"/docker-entrypoint.…"   3 minutes ago        Up 3 minutes
80/tcp, 192.168.122.66:8050->8050/tcp,
192.168.122.66:8050->8050/udp
nginx-15bc0796-ed15-6963-6437-cdfee86ee035
```

Si accedemos a la aplicación por el puerto del balanceador:

```
curl  http://192.168.122.66:8050/
Welcome! You are on node 192.168.122.66:26125
```

Ahora vamos a simular un aumento de la carga para que balancee hacia arriba el autoscaler. Lo haremos con un comando de **hey**:

```
sudo apt install hey
```

```
hey -n 500000 -c 500 http://192.168.122.66:8050/
```

Mientras podemos monitorear los contenedores en una pestaña nueva para ver si se crean los nuevos:

```
watch -n 1 docker ps
```

```
CONTAINER ID   IMAGE                              COMMAND
CREATED          STATUS          PORTS
NAMES
ce41cb2f89cd   hashicorp/demo-webapp-lb-guide   "./main"
22 seconds ago   Up 22 seconds   192.168.122.66:25294->25294/tcp,
192.168.122.66:25294->25294/udp
server-e875dad9-8906-447b-1b63-6bad9ae7ab3b

cb586a01ebc5   hashicorp/demo-webapp-lb-guide   "./main"
22 seconds ago   Up 22 seconds   192.168.122.66:23756->23756/tcp,
192.168.122.66:23756->23756/udp
server-61e38e50-52b7-61f1-5a05-5da4f9d78647

edcf2f7ca90a   hashicorp/demo-webapp-lb-guide   "./main"
22 seconds ago   Up 22 seconds   192.168.122.66:27534->27534/tcp,
192.168.122.66:27534->27534/udp
server-098d9639-9da9-315b-db4e-07e45be469c7

b4a08e34847a   hashicorp/demo-webapp-lb-guide   "./main"
7 minutes ago    Up 7 minutes    192.168.122.66:26125->26125/tcp,
192.168.122.66:26125->26125/udp
server-6b670643-b283-5cb7-d150-5c83096ddb3b

a96d17664a8c   nginx:latest
"/docker-entrypoint...."   9 minutes ago    Up 9 minutes
80/tcp, 192.168.122.66:8050->8050/tcp,
192.168.122.66:8050->8050/udp
nginx-15bc0796-ed15-6963-6437-cdfee86ee035
```

Y vemos como nos aparecen nuevas instancias

```
Welcome! You are on node 192.168.122.66:27534
Welcome! You are on node 192.168.122.66:23756
Welcome! You are on node 192.168.122.66:26125
Welcome! You are on node 192.168.122.66:25294
```

Una vez que termina, nos quedamos únicamente con 1 contenedor.

## 8.- Comparación entre Nomad y Kubernetes

A continuación vamos a describir las diferencias que hay entre Nomad y Kubernetes.

### 8.1.- Filosofía y arquitectura

| NOMAD | KUBERNETES |
|---|---|
| Filosofía Unix. Herramienta con propósito específico (solo orquestación y scheduling). | Arquitectura monolítica con muchas funcionalidades integradas. |
| Binario único que no requiere servicios externos. | Conjunto de múltiples servicios (API server, scheduler, controller manager, etcd…). |
| Sencillo de desplegar, operar y mantener | En entornos de alta disponibilidad es bastante complejo. |
| Se integra con Consul y Vault para descubrimiento de servicios y gestión de secretos. | Ofrece servicios para descubrimiento, monitoreo, secretos… |

### 8.2.- Carga de trabajo (Workloads)

| NOMAD | KUBERNETES |
|---|---|
| Admite contenedores, MV, aplicaciones (Java, IIS en Windows, Qemu…) | Exclusivamente para contenedores Linux (principalmente Docker y ahora containerd). |
| Basado en controladores extensibles para nuevos tipos de carga | Centrado en contenedores con una abstracción uniforme, pero limitada fuera de este modelo. |

### 8.3.- Despliegue

| NOMAD | KUBERNETES |
|---|---|
| Mismo binario en todos los entornos (desarrollo, edge, producción…) | Distintas variantes como minikube, kubeadm, k3s… Varían en capacidades y complejidad. |
| Despliegue simple y uniforme, sin diferencia entre entornos. | Fragmentación en herramientas de instalación y mantenimiento. Despliegue más pesado y complejo |

### 8.4.- Escalabilidad

| NOMAD | KUBERNETES |
|---|---|
| Soporta más de 10.000 nodos en producción real. | Soporta hasta 5.000 nodos y 300.000 contenedores. |
| Diseñado para escalabilidad horizontal y multicluster nativo. | La federación de clústeres es inmadura, y gestionar varios clústeres es complejo. |

## 9. Conclusiones

Para terminar este proyecto, vamos con las conclusiones.
En definitiva, habiendo visto Kubernetes durante el curso, puedo decir que, teniendo los conceptos más claros, me ha sido más sencillo entender cómo funciona Nomad y cómo utilizarlo.

Nomad es una herramienta más básica y al no depender de tantas dependencias, se hace más sencillo el empezar a conocerlo.

Por otra parte, si necesitamos algo más avanzado, necesitamos integrar otras herramientas como por ejemplo Consul o Vault, mientras que Kubernetes ya dispone íntegramente de estos servicios, sin necesidad de otras herramientas.

Podría acabar diciendo que Nomad es una herramienta simple, pero que se le puede sacar bastante, quizá para empresas algo más pequeñas o que estén empezando en el mundo de la orquestación. Kubernetes al tener muchas más cosas, es más usada en el mundo laboral, pero hace falta un conocimiento avanzado sobre Kubernetes, para sacar todo lo que tiene.

## 10. Bibliografía

*¿Qúe es la orquestación de contenedores? AWS.*
https://aws.amazon.com/es/what-is/container-orchestration/#:~:text=La%20orquestaci%C3%B3n%20de%20contenedores%20es,para%20ejecutarse%20en%20cualquier%20infraestructura.

*Autoscaling. IBM.*
https://www.ibm.com/es-es/topics/autoscaling

*Kubernetes. José Domingo.*
https://fp.josedomingo.org/sri/#unidad-8-kubernetes

*Introducción a Nomad. Hashicorp.*
https://developer.hashicorp.com/nomad/intro

*Arquitectura de Nomad. Hashicorp*
https://developer.hashicorp.com/nomad/docs/concepts/architecture

*Job en Nomad. Hashicorp.*
https://developer.hashicorp.com/nomad/docs/concepts/job

*Instalación Docker.*
https://docs.docker.com/engine/install/debian/

*Instalación Nomad.*
https://developer.hashicorp.com/nomad/docs/install

*Instalación Consul*
https://developer.hashicorp.com/consul/install

*Plantilla para jobs*
https://developer.hashicorp.com/nomad/docs/job-specification/template

*Comparación Nomad vs Kubernetes.*
https://developer.hashicorp.com/nomad/docs/nomad-vs-kubernetes