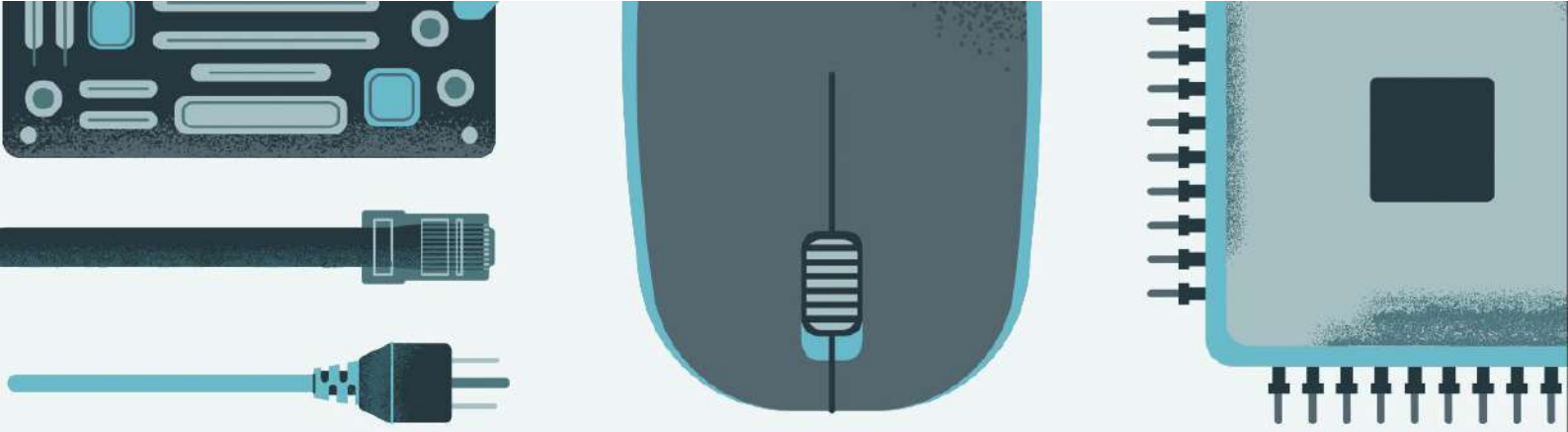
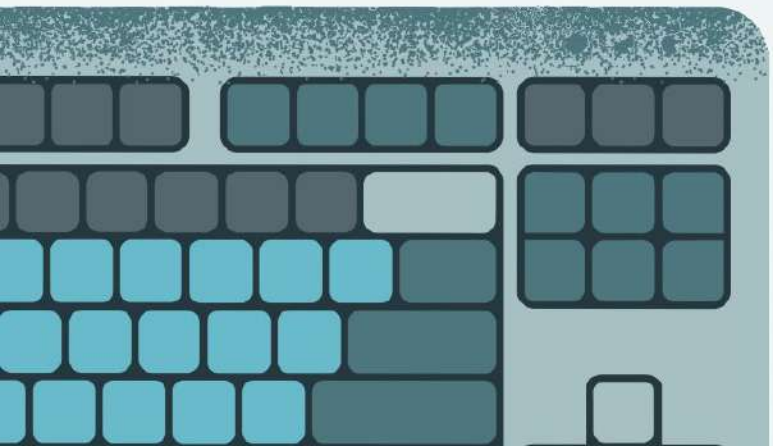


Juan Antonio Pineda Amador

PULUMI CON OPENSTACK Y KUBERNETES



kubernetes openstack Pulumi

Índice

1. Objetivos **2**
2. Escenarios **4**
2.1 Openstack 4
2.2 Kubernetes 5
3. Fundamentos teóricos y conceptos. **7**
3.1 Pulumi **7**
3.1.1 Pulumi vs Terraform 8
3.2 Openstack 9
3.2.1 Componentes de Openstack 10
3.3 Kubernetes **11**
3.3.2 Componentes de Kubernetes 12
3.3.3 Minikube 13
4. Demostraciones **14**
4.1 Instalación de Pulumi 14
4.2 Demo Openstack 16
4.2.1 Creación del proyecto 16
4.2.2 Creación del escenario 23
4.2.2.1 Creacion de la red y Subred 24
4.2.2.2 Creacion del Router 26
4.2.2.3 Creacion de puertos Wordpress y DB 28
4.2.2.4 Creacion de los scripts 30
4.2.2.5 Creacion de las instancias 31
4.2.2.6 Asociación ip flotante 33
4.2.3 Despliegue del escenario 34
4.3 Demo Kubernetes 41
4.3.1 Creación del escenario 41
4.3.1.1 Configuracion general 42
4.3.1.2 Creacion de secrets 43
4.3.1.3 Creacion de volumenes 43
4.3.1.4 Creacion de los Deployments 44
4.3.1.5 Creacion de los Services 47
4.3.1.5 Creacion del Ingress 49
4.3.2 Despliegue del escenario 51
5. Conclusiones y propuestas **55**
6. Bibliografía **57**

1. Objetivos

A lo largo del curso, hemos visto diferentes herramientas de infraestructura como Código (IaC), como Ansible y Vagrant.
Ansible está orientado a la configuración y gestión de software y servicios, mientras que Vagrant se enfoca más en la creación de entornos de infraestructura, especialmente entornos de desarrollo basados en máquinas virtuales.
Al final, ambas herramientas tienen el mismo objetivo: automatizar y facilitar la administración de entornos de sistemas.

Aunque estas herramientas son muy útiles, requieren aprender ciertos lenguajes que no habíamos utilizado antes, lo que puede hacer que su uso y comprensión sean un poco más complicados al principio.
En el caso de Vagrant, se basa en el lenguaje de programación Ruby para definir infraestructuras, por lo que es necesario entender al menos lo básico de su sintaxis.
Por otro lado, Ansible utiliza archivos en YAML, un lenguaje de marcas sencillo pero algo estricto en su formato.
En resumen, aunque ambas herramientas son potentes, requieren cierta curva de aprendizaje relacionada con los lenguajes que utilizan.

Sabiendo todo esto, puede generar cierto rechazo al comenzar en el mundo de Infraestructura como Código (IaC). Sin embargo, en este proyecto se presentará una herramienta de IaC que nos permite crear infraestructura como código, pero con la ventaja de utilizar lenguajes de programación que ya conocemos, como Python, Go, Java y YAML. Esta herramienta se llama Pulumi.

Pulumi es una herramienta relativamente nueva, y debido a la falta de tiempo, no hemos tenido oportunidad de explorarlo en detalle. Sin embargo, esta herramienta es muy potente porque nos permite crear infraestructura como código utilizando lenguajes de programación con los que ya estamos familiarizados. En mi caso, el proyecto se centrará en Python, ya que es el lenguaje de programación que hemos estado trabajando en clase y con el que me siento cómodo.
Aunque Pulumi también podría usarse con YAML, considero que no es necesario explorarlo en este caso, ya que hemos estado trabajando con YAML en Ansible, lo que lo hace menos relevante en este contexto.

El objetivo principal de este proyecto es crear una infraestructura en OpenStack, una plataforma de cloud computing. Esto nos permitirá crear redes, instancias y volúmenes, facilitando así el manejo de infraestructura como código en el ámbito de cloud computing, un tema que no hemos explorado hasta ahora.
También veremos la creación de contenedores con Pulumi utilizando Kubernetes para desplegar WordPress como aplicación. Esto nos permitirá gestionar recursos como pods, deployments, servicios y más, todo a través de Python como lenguaje de programación. Utilizando Pulumi, podremos automatizar la configuración y el despliegue de la infraestructura necesaria para ejecutar WordPress, aprovechando la flexibilidad de la programación para manejar de manera eficiente los recursos dentro de un clúster de Kubernetes.

Para finalizar, también veremos más acerca del panel de control de Pulumi, que nos permite visualizar los despliegues realizados, así como comprobar si se han ejecutado correctamente o han fallado.

2. Escenarios

Este proyecto contempla dos escenarios principales, o más concretamente, los escenarios que se pretende alcanzar. Como se ha mencionado anteriormente, Pulumi permite la creación de infraestructura como código, por lo que en este apartado se describirán los escenarios que se implementarán utilizando esta herramienta.

En primer lugar, se creará un escenario basado en OpenStack. Posteriormente, se desarrollará un segundo escenario utilizando Kubernetes, el cual será más sencillo en comparación con el anterior.

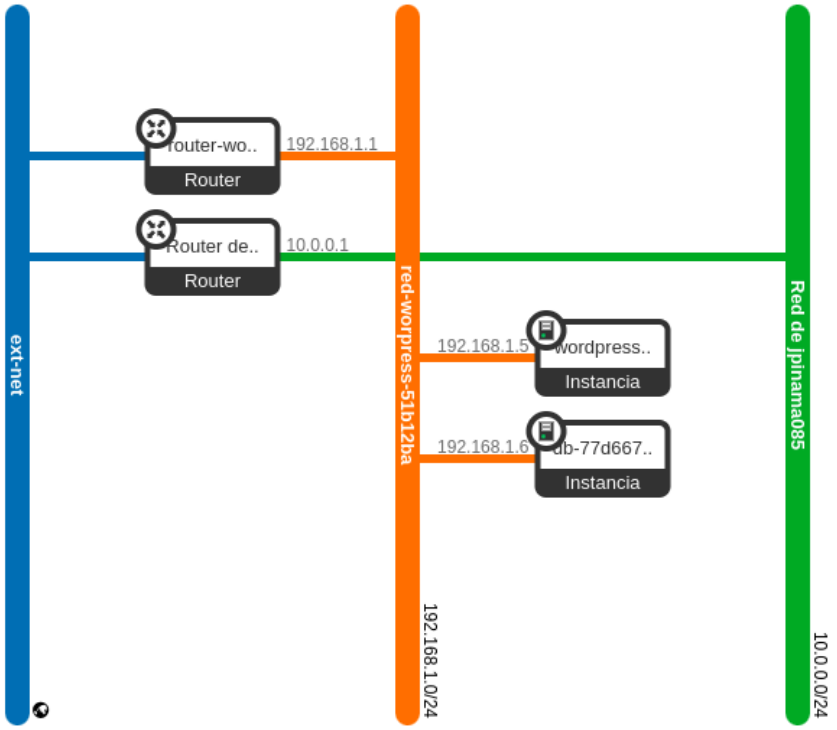
2.1 Openstack

Para este primer escenario, se utilizará Pulumi como herramienta de infraestructura como código. Inicialmente, desarrollaré el código necesario en mi ordenador personal, donde tendré instalado Pulumi. La gestión de Pulumi se realizará mediante línea de comandos, que es su forma de uso más habitual y recomendada.

Una vez creado el proyecto en Pulumi, se procederá a definir la infraestructura en OpenStack. Este escenario consistirá en una red privada con el rango de direcciones 192.168.1.0/24, conectada a un router que, a su vez, estará vinculado a la red externa.
Dentro de esta red se desplegarán dos instancias:

- Una instancia que alojará WordPress.
- Una segunda instancia que contendrá la base de datos necesaria para el funcionamiento de WordPress.

La arquitectura del escenario se visualizaría de la siguiente manera:



2.2 Kubernetes

A continuación, se procederá a la implementación del segundo escenario, esta vez utilizando Kubernetes. Este entorno es considerablemente más sencillo en comparación con el de OpenStack, ya que únicamente se desplegará una instancia de WordPress.
Al igual que en el caso anterior, será necesario crear un nuevo proyecto de Pulumi. A través de este proyecto, se definirá por código toda la infraestructura necesaria para el despliegue en Kubernetes.

En este escenario se crearán los siguientes elementos:

- Dos servicios: Uno para la base de datos, otro para la aplicación WordPress.

- Dos deployments, uno para cada servicio, que garantizarán la disponibilidad de las aplicaciones.

- Un volumen persistente asociado a la base de datos, con el objetivo de evitar la pérdida de datos en caso de reinicios o reinstalaciones.

- Secrets de Kubernetes para gestionar de forma segura las credenciales y parámetros necesarios para la creación y configuración de la base de datos.

Para la creación del escenario basado en Kubernetes se utilizará Minikube, o más concretamente K3s, una distribución ligera de Kubernetes. La elección de esta herramienta responde a un propósito principalmente educativo, ya que el objetivo principal es demostrar el potencial de Pulumi en la gestión de infraestructura mediante código, sin la necesidad de desplegar un clúster completo en un entorno de producción.

3. Fundamentos teóricos y conceptos.

3.1 Pulumi

Pulumi es una herramienta de Infrastructure as Code (IaC) que permite definir y gestionar infraestructura mediante código. Con ella, es posible crear y configurar recursos como redes, instancias, volúmenes, entre otros, de forma automatizada.

La principal diferencia de Pulumi frente a otras herramientas de IaC es que permite utilizar lenguajes de programación convencionales como Python, Go, Java, C#, entre muchos otros. Esto representa una gran ventaja, ya que facilita la adopción de IaC sin necesidad de aprender un nuevo lenguaje de programación o un lenguaje de marcado específico. Como resultado, Pulumi ofrece un aprendizaje más accesible y una mayor flexibilidad a la hora de diseñar y desplegar infraestructuras.

Pulumi ofrece dos formas principales de uso para la gestión y despliegue de infraestructura:

1. Uso mediante línea de comandos (CLI):

Esta es la opción utilizada en este proyecto. Consiste en instalar Pulumi localmente en la máquina del desarrollador. Una vez creado el proyecto, y cuando esté listo para desplegarse, Pulumi realiza una solicitud a la nube de Pulumi (Pulumi Cloud), donde se lleva a cabo el despliegue. Esta plataforma también permite visualizar todos los despliegues realizados a través de una interfaz gráfica accesible desde el navegador web, lo que facilita el seguimiento del estado de la infraestructura.

2. Uso desde Pulumi Cloud directamente:

En esta modalidad, el proyecto se crea y gestiona directamente desde la plataforma web de Pulumi Cloud. La infraestructura se define y despliega desde el entorno en línea, sin necesidad de utilizar la línea de comandos localmente.

3.1.1 Pulumi vs Terraform

La principal diferencia entre Pulumi y Terraform radica en el tipo de lenguaje que utilizan para definir la infraestructura:

- **Terraform** utiliza un lenguaje declarativo llamado HCL (HashiCorp Configuration Language). Este lenguaje está diseñado específicamente para describir la infraestructura, pero requiere que el usuario defina de manera explícita el estado deseado de los recursos.

- **Pulumi**, por otro lado, emplea un enfoque imperativo al utilizar lenguajes de programación convencionales como Python, Go, Java, entre otros. Esto permite a los usuarios aprovechar la familiaridad con lenguajes de programación populares, brindando una mayor flexibilidad y capacidad para usar herramientas y bibliotecas adicionales dentro del mismo código.

En cuanto a la curva de aprendizaje, Terraform presenta una mayor complejidad debido a su lenguaje declarativo y su necesidad de aprender un nuevo conjunto de conceptos y sintaxis. Por el contrario, Pulumi es más accesible para aquellos que ya están familiarizados con lenguajes de programación, lo que facilita su adopción.

Ambas herramientas permiten la gestión de la infraestructura de manera local mediante una interfaz de línea de comandos (CLI) o remota a través de sus respectivas plataformas en la nube: Pulumi Cloud y Terraform Cloud.

En términos de comunidad y documentación, Terraform tiene una base de usuarios más grande y una documentación más extensa, debido a su mayor antigüedad en el mercado. Pulumi, al ser más reciente, tiene una comunidad en crecimiento, aunque aún es más limitada en comparación.

Ambas herramientas están orientadas a los mismos objetivos, pero la diferencia clave es el enfoque en el lenguaje. Terraform se basa en un lenguaje específico, HCL, mientras que Pulumi permite utilizar lenguajes de programación conocidos. Esto hace que Pulumi sea una opción más accesible para desarrolladores que ya tienen experiencia con lenguajes de programación, facilitando su uso y personalización.

3.2 Openstack

OpenStack es un software de código abierto, lo que significa que es gratuito y puede ser modificado y distribuido libremente. Está diseñado para construir y administrar nubes privadas y públicas, proporcionando una plataforma de Infraestructura como Servicio (IaaS).

OpenStack no es una única herramienta, sino un conjunto de componentes que trabajan conjuntamente para ofrecer servicios esenciales en la gestión de entornos en la nube. Entre sus funcionalidades principales se encuentran la creación y administración de máquinas virtuales, redes, almacenamiento y otros recursos necesarios para la operación de infraestructuras virtualizadas.

En cierto sentido, OpenStack puede considerarse una alternativa de código abierto a plataformas comerciales como AWS (Amazon Web Services). Aunque son tecnologías distintas, ambas están orientadas a cumplir un propósito similar: permitir la creación, despliegue y gestión de infraestructura en la nube de forma escalable y automatizada.

3.2.1 Componentes de Openstack

En este apartado se describen los principales componentes de OpenStack disponibles en el entorno del IES Gonzalo Nazareno, que serán utilizados para el desarrollo del proyecto integrado.

- **Nova**: Es el componente encargado de gestionar el ciclo de vida de las máquinas virtuales: su creación, ejecución y eliminación. Es fundamental para el proyecto, ya que permitirá desplegar las instancias necesarias para implementar los distintos escenarios.

- **Keystone**: Se encarga de la autenticación y autorización de usuarios y servicios. En otras palabras, controla quién puede hacer qué dentro del entorno OpenStack. Gestiona usuarios, roles, proyectos y permisos, garantizando la seguridad y el control de acceso.

- **Glance**: Administra las imágenes de sistemas operativos que se utilizan para lanzar nuevas instancias. Estas imágenes pueden estar en formatos como RAW, QCOW2, entre otros. En este proyecto se usará Glance para seleccionar la imagen del sistema operativo que se instalará en cada instancia.

- **Neutron**: Responsable de la gestión de la red en OpenStack. Permite crear redes, subredes, routers, firewalls, y gestionar la conectividad entre instancias. En este proyecto se utilizará para crear la red 192.168.1.0/24 y configurarla según las necesidades del escenario planteado.

- **Cinder**: Proporciona almacenamiento en forma de discos virtuales persistentes que pueden conectarse a las instancias. Estos volúmenes permiten almacenar datos que no se pierdan al apagar o eliminar las máquinas virtuales.

- **Horizon**: Es la interfaz gráfica web de OpenStack, que permite a los usuarios interactuar con el sistema sin necesidad de utilizar la línea de comandos. Aunque su uso en este proyecto será limitado, se empleará ocasionalmente para verificar visualmente que los recursos se han creado correctamente.

3.3 Kubernetes

Kubernetes, también conocido como K8s, es una plataforma de código abierto diseñada para la orquestación de contenedores. Fue desarrollada originalmente por Google y actualmente es mantenida por la Cloud Native Computing Foundation (CNCF).

Su función principal es automatizar el despliegue, la gestión, el escalado y la operación de aplicaciones en contenedores, especialmente en entornos de producción, donde la fiabilidad y la eficiencia son fundamentales.

Existen varias herramientas que permiten instalar y ejecutar Kubernetes de diferentes formas, como:

- **Kind** (Kubernetes IN Docker): Permite ejecutar clústeres de Kubernetes dentro de contenedores Docker, ideal para pruebas y desarrollo local.

- **K3s**: Una versión ligera de Kubernetes diseñada para entornos de recursos limitados o pruebas educativas. Utiliza un solo nodo por defecto y requiere menos recursos que una instalación estándar.

- **K8s** (instalación clásica): Se refiere a la instalación completa y tradicional de Kubernetes en múltiples nodos, adecuada para entornos de producción a gran escala.

En este proyecto se utilizará Minikube, por su sencillez y bajo consumo de recursos, lo que lo hace ideal para fines educativos. Más adelante se explicará con mayor detalle su configuración y uso.

A continuación, se describen los componentes principales que forman parte de la arquitectura de Kubernetes:

3.3.2 Componentes de Kubernetes

Kubernetes se compone de varios elementos que trabajan en conjunto para permitir la orquestación de contenedores. En este proyecto se hará uso de los siguientes componentes clave, los cuales son fundamentales para el despliegue y gestión de las aplicaciones:

- **Pod**: Es la unidad mínima de ejecución en Kubernetes. Un pod puede contener uno o más contenedores, aunque lo más habitual es que contenga solo uno. En los casos en los que haya varios contenedores, estos deben estar estrechamente relacionados y compartir recursos como almacenamiento y red.

- **Deployment**: Se encarga de gestionar el ciclo de vida de los pods. Permite crear nuevas réplicas, actualizar versiones, escalar automáticamente según la demanda y garantizar la alta disponibilidad de los servicios.

- **Service**: Expone los pods para permitir su acceso, tanto desde dentro del clúster como desde el exterior. Se utiliza junto con los deployments para definir cómo deben comunicarse los distintos servicios. Por ejemplo, un servicio de WordPress se expone externamente, mientras que un servicio de base de datos solo es accesible internamente.

- **Ingress**: Gestiona el acceso HTTP/HTTPS a los servicios desde el exterior del clúster. Actúa como un enrutador que dirige el tráfico entrante a los servicios correspondientes, ofreciendo mayor control sobre el acceso y las reglas de red.

Estos son algunos de los componentes más importantes de Kubernetes que se utilizarán en este proyecto. Existen muchos otros, pero se ha considerado que estos son suficientes para entender y llevar a cabo correctamente la implementación propuesta.

3.3.3 Minikube

En este proyecto se utilizará Minikube, una herramienta que permite crear un clúster de Kubernetes de un solo nodo de forma local. Minikube está orientado principalmente a entornos educativos o de desarrollo, ya que facilita la experimentación y el aprendizaje de Kubernetes sin necesidad de recurrir a infraestructuras complejas en la nube.

Lo que hace Minikube es lanzar una máquina virtual en la que se ejecuta todo el entorno de Kubernetes, incluyendo el nodo, el plano de control y los recursos básicos necesarios para operar el clúster. Esto lo convierte en una solución ideal para montar un entorno reducido (mini escenario) y comprobar el funcionamiento de herramientas como Pulumi para la gestión de infraestructura mediante código.

4. Demostraciones

A continuación, se explicarán detalladamente todas las demostraciones prácticas que se llevarán a cabo.

Primero, se incluirá un apartado dedicado a la instalación de Pulumi y a la explicación de sus funciones básicas, para familiarizarse con esta herramienta de infraestructura como código.

Una vez entendido el entorno de trabajo, comenzaremos con la primera demostración, que consistirá en crear un escenario previamente descrito en OpenStack, utilizando Pulumi como herramienta de automatización.

Finalmente, se llevará a cabo una demostración similar, pero esta vez implementando el escenario en un entorno Kubernetes.

Una vez explicados paso a paso todos los procedimientos, se dará comienzo a las demostraciones prácticas.

4.1 Instalación de Pulumi

Para comenzar con el proyecto, lo primero que haremos será instalar Pulumi. Para ello, nos dirigiremos a la documentación oficial de Pulumi, donde se detallan varias formas de instalación.

En este caso, se utilizará la opción más sencilla y rápida, que consiste en ejecutar un script de instalación proporcionado por Pulumi:

```
curl -fsSL https://get.pulumi.com | sh
```

Este comando descargará e instalará automáticamente Pulumi en el sistema.

También es posible realizar la instalación manual descargando los binarios y compilándolos, pero se recomienda el uso del script por su simplicidad y rapidez.

Una vez instalado Pulumi, es necesario añadir su ruta al PATH del sistema de forma permanente, ya que Pulumi no se encuentra en los repositorios oficiales de Debian y se instala localmente en el directorio del usuario.

Cuando se utiliza el script de instalación, Pulumi se instala por defecto en:

```
/home/usuario/.pulumi/bin
```

Para que el sistema reconozca el comando pulumi desde cualquier terminal, es necesario añadir esta ruta al PATH. Para ello, se puede ejecutar el siguiente comando (ajustando "usuario" por el nombre real del usuario del sistema):

```
export PATH=$PATH:/home/usuario/.pulumi/bin
```

Para comprobar que realmente se ha instalado Pulumi y funciona correctamente pondremos el siguiente comando:

```
usuario@triforce:~$ pulumi version
v3.156.0
```

Si nos muestra la versión esto quiere decir que ya tenemos Pulumi completamente instalado y listo para su uso.

4.2 Demo Openstack

4.2.1 Creación del proyecto

A continuación, se procederá con la demostración de infraestructura en OpenStack. Para ello, usaremos Pulumi para crear el escenario previamente descrito en el apartado de escenarios de OpenStack.

El primer paso para comenzar a trabajar con Pulumi es crear una carpeta en nuestro sistema, que actuará como directorio raíz del proyecto. Una vez dentro de esta carpeta, se ejecuta el siguiente comando:

Este comando inicializa un nuevo proyecto de Pulumi. Al ejecutar, el asistente interactivo preguntará si se desea crear el proyecto a partir de una plantilla predefinida o generarlo con la asistencia de IA. En este caso, se optará por la primera opción, eligiendo una plantilla existente.

Pulumi descargará y configurará automáticamente los archivos necesarios, incluyendo el archivo de definición del stack, el archivo principal del programa (index.ts, main.py, etc), y la configuración base del entorno.

```
usuario@triforce:~/openstack-demo$ pulumi new
 Would you like to create a project from a template or using a Pulumi AI
prompt?  [Use arrows to move, type to filter]
> template - Create a new Pulumi project using a template
  ai - Create a new Pulumi project using Pulumi AI
```

Después de ejecutar pulumi new, el asistente nos preguntará qué plantilla deseamos utilizar. Actualmente, Pulumi ofrece más de 230 plantillas oficiales, organizadas por proveedor de infraestructura y lenguaje de programación.

Dado que en esta demostración vamos a trabajar con OpenStack, y existe una plantilla específica para este proveedor, seleccionaremos la plantilla correspondiente a OpenStack con Python.

```
Please choose a template (230 total):
  [Use arrows to move, type to filter]
...
  openstack-go                      A minimal OpenStack Go Pulumi
program
  openstack-javascript              A minimal OpenStack JavaScript
Pulumi program
> openstack-python                  A minimal OpenStack Python Pulumi
program
  openstack-typescript              A minimal OpenStack TypeScript
Pulumi program
...
```

Después de elegir la plantilla, Pulumi nos pedirá que completemos algunos datos para inicializar el proyecto:

- **Nombre del proyecto**: el identificador principal de nuestro proyecto de infraestructura.

- **Descripción**: una breve explicación del propósito del proyecto.

- **Nombre del stack**: nombre del entorno o instancia de despliegue.

En este punto, debemos entender que un stack en Pulumi es un entorno aislado, similar a una rama en GitHub, donde podemos aplicar configuraciones diferentes sin afectar a otros stacks del mismo proyecto.

Esto nos permite, por ejemplo, tener un stack para desarrollo, otro para pruebas y otro para producción, todos compartiendo el mismo código pero con configuraciones independientes.

En este caso, como se trata de una demostración sencilla, se utilizará un único stack.

```
Project name (openstack-demo):
Project description (A minimal OpenStack Python Pulumi program):
Escenario demo openstack
Created project 'openstack-demo'

Please enter your desired stack name.
To create a stack in an organization, use the format
<org-name>/<stack-name> (e.g. `acmecorp/dev`).
Stack name (dev):
Created stack 'dev'
```

Por último, Pulumi nos preguntará qué herramienta deseamos utilizar para gestionar las dependencias del proyecto. Dado que en este caso utilizaremos Python para definir nuestra infraestructura, seleccionaremos pip, el gestor de paquetes estándar de Python.

Pulumi creará automáticamente un entorno virtual en el directorio del proyecto y utilizará pip para instalar todas las dependencias necesarias que figuren en el archivo requirements.txt. Estas dependencias incluyen, por ejemplo, el SDK de Pulumi y el proveedor correspondiente.

De esta forma, el entorno quedará completamente configurado y listo para comenzar a escribir el código de infraestructura.

```
The toolchain to use for installing dependencies and running the program
[Use arrows to move, type to filter]
> pip
  poetry [not found]
  uv [not found]
```

Una vez hayamos respondido todas las preguntas del asistente, el proyecto quedará completamente creado. Si revisamos el contenido del directorio, veremos que se han generado varios archivos y carpetas esenciales:

- **__main__.py**: es el archivo principal del proyecto. Aquí es donde escribiremos el código que define la infraestructura. Es el archivo más importante, ya que en él se implementará todo el escenario de la demostración.

- **Pulumi.yaml**: contiene la configuración básica del proyecto, como el nombre del proyecto, su descripción, y el lenguaje de programación utilizado.

- **requirements.txt**: lista todas las dependencias de Python necesarias para que Pulumi pueda interactuar con la nube, incluyendo el SDK de Pulumi y el proveedor de OpenStack.

- **venv/**: es el entorno virtual de Python creado automáticamente. Aquí se instalan todas las dependencias del proyecto de forma aislada, evitando conflictos con otros entornos del sistema.

Ahora que comprendemos la función de cada uno de estos elementos, estamos listos para comenzar a escribir el código de infraestructura y construir el escenario de la demo en OpenStack utilizando Pulumi.

```
usuario@triforce:~/openstack-demo$ ls
__main__.py  Pulumi.yaml  requirements.txt  venv
```

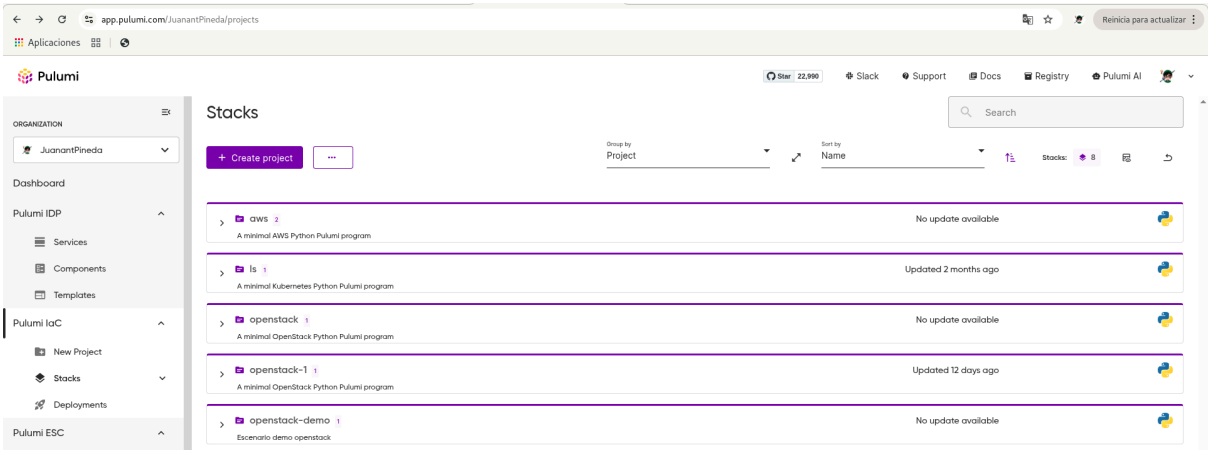
Antes de continuar, es importante mencionar que si ya tenemos una cuenta creada en Pulumi (altamente recomendable), el propio CLI se encargará de vincularla automáticamente al entorno local.

Cuando inicializamos nuestro primer proyecto, Pulumi abrirá una ventana en el navegador para iniciar sesión o vincular la cuenta, lo cual permite asociar el proyecto local con nuestro dashboard en la nube de Pulumi.

Esta integración nos ofrece varias ventajas:

- Visualización centralizada de todos los proyectos y stacks creados.
- Acceso desde cualquier dispositivo al estado de nuestra infraestructura.
- Seguimiento de cambios y despliegues realizados.
- Opcionalmente, colaboración con otros miembros del equipo.

Una vez conectados, podemos acceder al panel de control de Pulumi, y en la sección de Stacks, veremos todos los entornos que hemos creado desde el CLI en nuestro ordenador.



Si le damos al nombre del proyecto que estamos creando podremos ver todos los stack que tienen creado en mi caso solo 1 el stack dev.

Una vez dentro del panel de un stack específico en el dashboard de Pulumi, encontraremos varias secciones que nos permiten gestionar y monitorizar la infraestructura desplegada desde el proyecto. Estas son:

- **Overview**: una visión general del stack, donde se muestra el estado actual, el entorno asociado, el número de recursos desplegados y el historial de actualizaciones.

- **README**: sección destinada a incluir documentación personalizada del stack. Es útil para describir su propósito, cómo se usa o notas relevantes para el equipo.

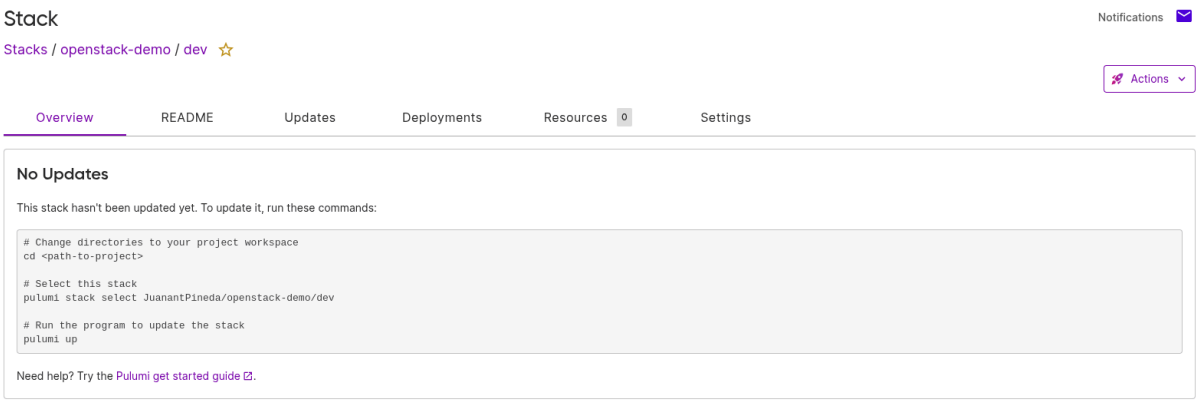
- **Updates**: muestra un historial de actualizaciones y cambios realizados en el stack (cada vez que ejecutamos pulumi up). Incluye quién hizo el cambio, cuándo y qué recursos se modificaron.

- **Deployments**: apartado muy importante. Aquí se listan todos los despliegues realizados, ya sean automáticos (CI/CD) o manuales desde el CLI. Permite analizar los logs de cada ejecución y revertir cambios si es necesario.

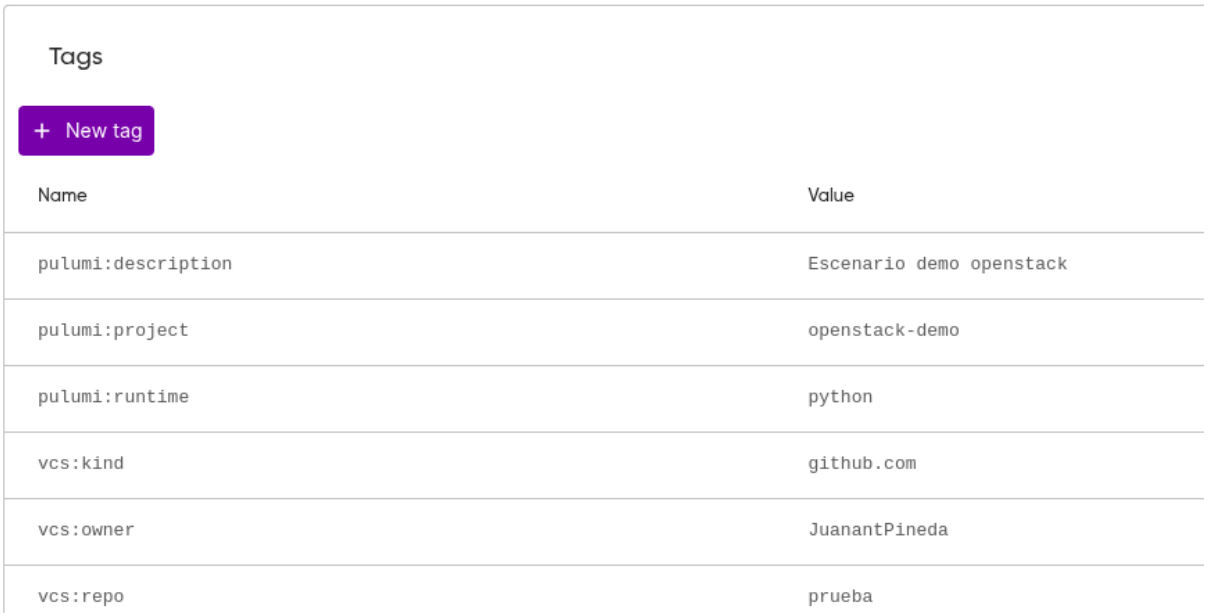
- **Resources**: muestra una lista detallada de todos los recursos creados mediante el código de infraestructura, incluyendo su tipo, estado y propiedades. Esta sección se explicará en más detalle más adelante.

- **Settings**: configuración general del stack, donde se pueden modificar opciones como el nombre, el equipo al que pertenece, eliminar el stack, y gestionar claves o integraciones.

Esta interfaz web es una herramienta muy útil para complementar el trabajo que realizamos desde el terminal y tener control visual de toda nuestra infraestructura como código.



Si nos vamos en el apartado de Tags que está en Overview aquí podemos ver todos los datos que hemos ido introduciendo en la creación del proyecto, que este contiene la misma información que el fichero Pulumi.yaml



Vemos que este fichero contiene la misma información que el panel de control que Pulumi.

```
usuario@triforce:~/openstack-demo$ cat Pulumi.yaml
name: openstack-demo
description: Escenario demo openstack
runtime:
  name: python
  options:
    toolchain: pip
    virtualenv: venv
config:
  pulumi:tags:
    value:
      pulumi:template: openstack-python
```

4.2.2 Creación del escenario

A continuación, se explicará todo el código necesario para la creación del escenario en OpenStack utilizando Pulumi.

Para ello, se trabajará principalmente sobre el fichero __main__.py, que contiene el código principal del despliegue. Iremos analizando y comentando el código que se ha generado para definir todos los recursos necesarios en el entorno de OpenStack.

Todo el código fuente, junto con los archivos de configuración y soporte, está disponible en mi repositorio de GitHub, el cual contiene la estructura completa del proyecto y los ficheros necesarios para su ejecución.

En esta parte del proyecto iré explicando apartado por apartado todos los recursos que se encuentra en el fichero __main__.py .

4.2.2.1 Creacion de la red y Subred

El primer paso en la construcción del escenario es la creación de la red y su correspondiente subred. Para ello, utilizaremos Pulumi con el proveedor de OpenStack.

En Pulumi, cada recurso que queremos gestionar se define como una variable en el código, que representa un componente de infraestructura. Al ejecutar pulumi up, Pulumi se encargará de traducir ese código en una llamada real a la API de OpenStack, desplegando los recursos declarados.

Una vez ya sabemos todo, crearemos la red para ello tendremos que declarar una variable y poner lo siguiente:

```
network = openstack.networking.Network("red-worpress",
    admin_state_up=True,
    description="Red creada con Pulumi en OpenStack")
```

- **openstack.networking.Network**: Es el componente de Pulumi encargado de crear una red en el entorno de OpenStack.

- "**red-wordpress**": Es el nombre lógico que se asignará al recurso dentro de Pulumi.

- **admin_state_up=True**: Indica que la red estará activa desde su creación.

- **description**: Añade una descripción que ayuda a identificar la red en el panel de OpenStack o mediante la CLI.

Una vez creada la red, el siguiente paso es definir la subred que estará asociada a ella. Al igual que con la red, utilizamos una variable para declarar el recurso, que Pulumi gestionará y desplegará automáticamente en OpenStack.

```
subnet = openstack.networking.Subnet("subred-wordpress",
    network_id=network.id,
    cidr="192.168.1.0/24",
    ip_version=4,
    dns_nameservers=["8.8.8.8", "8.8.4.4"],
    gateway_ip="192.168.1.1")
```

- **openstack.networking.Subnet**: Es el recurso de Pulumi para definir una subred en OpenStack.

- "**subred-wordpress**": Nombre lógico del recurso dentro del stack de Pulumi.

- **network_id=network.id**: Se enlaza esta subred a la red previamente creada, utilizando el identificador (id) del recurso network.

- **cidr**: Define el rango de direcciones IP que tendrá la subred.

- **ip_version**: Especifica que la subred será de tipo IPv4.

- **dns_nameservers**: Define los servidores DNS que usarán las instancias conectadas a esta subred.

- **gateway_ip**: IP que funcionará como puerta de enlace predeterminada.

Y con esto ya tendremos creada la red y su correspondiente subred con direccionamiento 192,168.1.0/24.

4.2.2.2 Creacion del Router

El siguiente paso en la configuración del escenario es la creación de un router que permita la comunicación entre la red interna y la red externa de OpenStack. Este router será responsable de enrutar el tráfico hacia el exterior y asignar direcciones IP flotantes (Floating IPs) a las instancias si es necesario.

Para ello, lo primero que necesitamos es obtener el identificador (ID) de la red externa de OpenStack. Esto se realiza mediante la siguiente instrucción:

```
external_network =
openstack.networking.get_network(name="ext-net")
```

- **get_network**: Es una función que obtiene un recurso existente en OpenStack (no lo crea).

- **name="ext-net"**: Especifica el nombre de la red externa configurada en OpenStack. Este nombre puede variar según el proveedor o configuración del entorno, por lo que debe comprobarse previamente en el panel de OpenStack o vía CLI.

Una vez obtenida la red externa, procedemos a crear el router que conectará nuestra red interna (red-wordpress) con el exterior. Este router es fundamental para permitir el acceso a internet desde nuestras instancias y también para enrutar correctamente el tráfico.

El siguiente código crea el router utilizando la red externa obtenida previamente:

```
router = openstack.networking.Router("router-worpress",
    external_network_id=external_network.id,
    admin_state_up=True)
```

- **openstack.networking.Router**: Componente de Pulumi que representa un router en OpenStack.
- "**router-wordpress**": Nombre lógico del recurso dentro del stack de Pulumi.
- **external_network_id=external_network.id**: Se indica el identificador de la red externa que se usará como gateway para este router. Esta red se obtuvo anteriormente con get_network(...).
- **admin_state_up=True**: Indica que el router estará activo desde su creación.

Con el router ya creado y configurado con la red externa, el siguiente paso es conectarlo a la subred interna (subred-wordpress) para permitir el enrutamiento del tráfico entre ambas redes.
Para ello, se utiliza un recurso de tipo RouterInterface, como se muestra a continuación:

```
router_interface =
openstack.networking.RouterInterface("mi_router_interface",
    router_id=router.id,
    subnet_id=subnet.id)
```

- **openstack.networking.RouterInterface**: Es el recurso de Pulumi que permite crear una interfaz de red entre un router y una subred.

- "**mi_router_interface**": Nombre lógico del recurso en Pulumi.

- **router_id=router.id**: Especifica el router que se conectará (en este caso, el que hemos creado anteriormente).

- **subnet_id=subnet.id**: Indica la subred interna a la que se va a conectar el router.

Este paso es clave para establecer la ruta entre la red privada y la red externa. Una vez desplegado, cualquier instancia dentro de la subred podrá comunicarse con el exterior.

4.2.2.3 Creacion de puertos Wordpress y DB

Una vez ya tenemos creada toda nuestra red wordpress que conecta con la red externa para que tenga conexión a internet, tenemos que crear ahora los puertos. Los puertos en Openstack es como una reserva de ip que hace la subred, este paso lo realizo por que de esta manera no se le asigna una ip aleatoria a nuestras instancias y es un poco más lioso, entonces mediante los puertos le indico que ip van a tener tanto la instancia de wordpress como la instancia de db.

Para crear un puerto en pulumi se realiza de la siguiente manera:

```
port_wordpress = openstack.networking.Port("port-wordpress",
    network_id=network.id,
    fixed_ips=[{
        "subnet_id": subnet.id,
        "ip_address": "192.168.1.5"
    }]
)
```

- **openstack.networking.Port**: Recurso que representa un puerto en OpenStack, es decir, una interfaz de red virtual.

- "**port-wordpress**": Nombre lógico del puerto.

- **network_id=network.id**: El puerto se vincula a la red previamente creada (red-wordpress).

- **fixed_ips**: Define una IP estática dentro del rango de la subred.

    - **subnet_id**=subnet.id: Indica en qué subred se reservará la IP.

    - **ip_address="192.168.1.5"**: IP privada que se asignará a la instancia WordPress.

Con esto crearemos un puerto para la instancia wordpress que mas adelante cuando creemos esta instancia le indicaremos que utilice este puerto para que cuando se cree automáticamente se le asigne la ip.

Para crear el puerto de la instancia mariadb haremos el mismo proceso pero con otra dirección ip

```
port_db = openstack.networking.Port("port-db",
    network_id=network.id,
    fixed_ips=[{
        "subnet_id": subnet.id,
        "ip_address": "192.168.1.6"
    }]
)
```

4.2.2.4 Creacion de los scripts

A continuación, se procederá a la creación de un script de inicialización que automatice la configuración de las instancias en el momento de su creación. El objetivo es que, al lanzar las dos instancias desde Pulumi, una se configure automáticamente con WordPress y la otra con la base de datos MariaDB.

Para ello, se utilizará Cloud-Init, una herramienta estándar para inicializar máquinas virtuales en la nube. Se creará un script que contenga todos los pasos necesarios para instalar y configurar WordPress, así como otro para preparar el entorno de MariaDB.

Una vez creado el script, se integrará en el código de Pulumi mediante una variable que contenga el contenido del script. Esta variable se pasará como parámetro durante la creación de las instancias, lo que permitirá que cada máquina se configure de forma automática en su primer arranque.

```
cloud_init_wordpress = system_os.path.join(system_os.getcwd(),
"wordpress.sh")
with open(cloud_init_wordpress, "r") as f:
    cloud_init_data_wordpress = f.read()
```

- **system_os.path.join(system_os.getcwd(), "wordpress.sh")**: Construye la ruta absoluta al archivo wordpress.sh.

- **open(..., "r")**: Abre el archivo en modo lectura.

- **cloud_init_data_wordpress**: Contiene el contenido del script, que será pasado a la instancia como user data para su ejecución automática al arrancar.

Con esta variable creada la utilizaremos más adelante para cuando creemos la instancia de wordpress se la pasaremos y se inicias la instalación del wordpress.

Para la instancia de la base de datos usaremos otro script pero este configura la base de datos, prácticamente es el mismo código pero le pasaremos el script de la base de datos

```
cloud_init_db = system_os.path.join(system_os.getcwd(), "db.sh")
with open(cloud_init_db, "r") as f:
    cloud_init_data_db = f.read()
```

Y con eso ya tendríamos creado el cloud-init que le pasaremos a la instancia.

4.2.2.5 Creacion de las instancias

Después de haber definido toda la infraestructura de red y los scripts de inicialización (cloud-init), el siguiente paso es crear las instancias de máquina virtual en OpenStack: una para WordPress y otra para la base de datos (MariaDB). A continuación, se muestra cómo crear la instancia que contendrá WordPress:

```
instancia_wordpress = os.compute.Instance("wordpress",
    image_name="Debian 12 Bookworm",
    flavor_name="m1.normal",
    networks=[{
        "port": port_wordpress.id
    }],
    key_pair="clave_personal",
    security_groups=["default"],
    user_data=cloud_init_data_wordpress
)
```

- **os.compute.Instance**: Recurso de Pulumi para crear una instancia en OpenStack.

- **"wordpress"**: Nombre lógico de la instancia.

- **image_name="Debian 12 Bookworm"**: Imagen de sistema operativo que se usará para la máquina virtual.

- **flavor_name="m1.normal"**: Especifica el tamaño (CPU/RAM) de la instancia.

- **networks=[{"port": port_wordpress.id}]**: Conecta la instancia al puerto previamente creado con IP fija, este puerto es el que creamos en el apartado de creación de puertos.

- **key_pair="clave_personal"**: Par de claves SSH para acceder a la instancia.

- **security_groups=["default"]**: Grupo de seguridad que define las reglas de acceso (puertos abiertos, etc.).

- **user_data=cloud_init_data_wordpress**: Script cloud-init que se ejecutará en el arranque para instalar y configurar WordPress automáticamente, este es el cloud-init que anteriormente hemos creado.

Con esta definición se nos creará la instancia de wordpress en la red que hemos definido. Ahora vamos a crear la instancia de la base de datos pero hay que tener en cuenta que la variable de port cambia y la del cloud-init.

```
instancia_db = os.compute.Instance("db",
    image_name="Debian 12 Bookworm",
    flavor_name="m1.normal",
    networks=[{
        "port": port_db.id
    }],
    key_pair="clave_personal",
    security_groups=["default"],
    user_data=cloud_init_data_db
)
```

Y con esto ya tendríamos la instancias creada, de esta manera ya podremos desplegar todo este codigo pero falta un pequeño detalle y es que vamos a asociar la ip flotante a la instancia de wordpress para acceder a ella.

4.2.2.6 Asociación ip flotante

Para que la instancia de WordPress sea accesible desde Internet, es necesario asignarle una IP pública mediante una Floating IP de OpenStack. Este tipo de IP permite enrutar el tráfico desde el exterior hacia una instancia ubicada en una red privada.

```
floating_ip = os.networking.FloatingIp("rancher-floating-ip",
    pool="ext-net"
)
```

- **os.networking.FloatingIp**: Recurso que representa una IP pública en OpenStack.
- "**rancher-floating-ip**": Nombre lógico del recurso.
- **pool="ext-net"**: Nombre del pool de red externa (especificado al crear el router anteriormente).

Ahora vamos a asociar la ip flotante que acabamos de crear a la instancia wordpress para eso es necesario saber la ip flotante que acabamos de crear, el id de la instancia.

```
floating_ip_association =
os.compute.FloatingIpAssociate("rancher-ip-association",
    floating_ip=floating_ip.address,
    instance_id=instancia_wordpress.id,
    fixed_ip=instancia_wordpress.access_ip_v4
)
```

- **os.compute.FloatingIpAssociate**: Recurso que asocia la IP flotante creada a la instancia.
- **floating_ip**: Dirección IP pública que has creado con FloatingIp. Es la IP que se asignará externamente.

- **instance_id**: ID de la instancia (máquina virtual) a la que le vas a asignar esa IP pública.

- **fixed_ip**: IP privada interna de la instancia. Aquí le estás diciendo a qué IP interna debe redirigir el tráfico la IP pública.

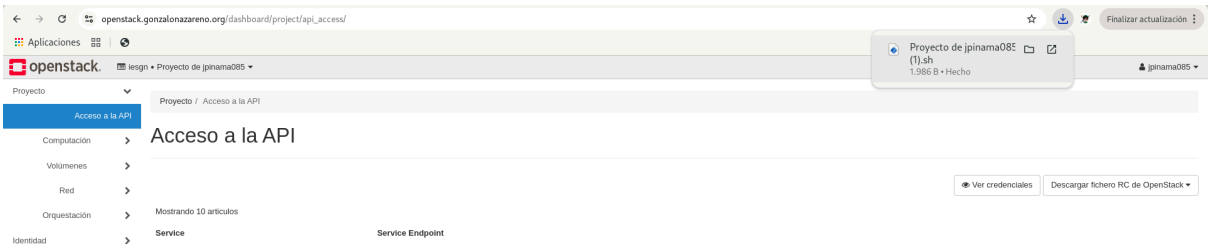
Y para finalizar el código pongo esta linea para que me muestre por pantalla la ip pública que se ha creado.

```
pulumi.export("instance_floating_ip", floating_ip.address)
```

Y con esto ya tendríamos explicado el código y el escenario ya se podría desplegar con toda seguridad.

4.2.3 Despliegue del escenario

Ahora una vez que ya tenemos creado el código y scripts ya podríamos desplegar el escenario, primero tenemos que situarnos en la carpeta donde esta el codigo, luego para el despliegue tenemos que descargas del openstack el fichero RC, para poder tener acceso a la API de openstack de nuestro usuario esto es un script de inicialización.



Lo iniciaremos de la siguiente manera y pondremos nuestra credenciales

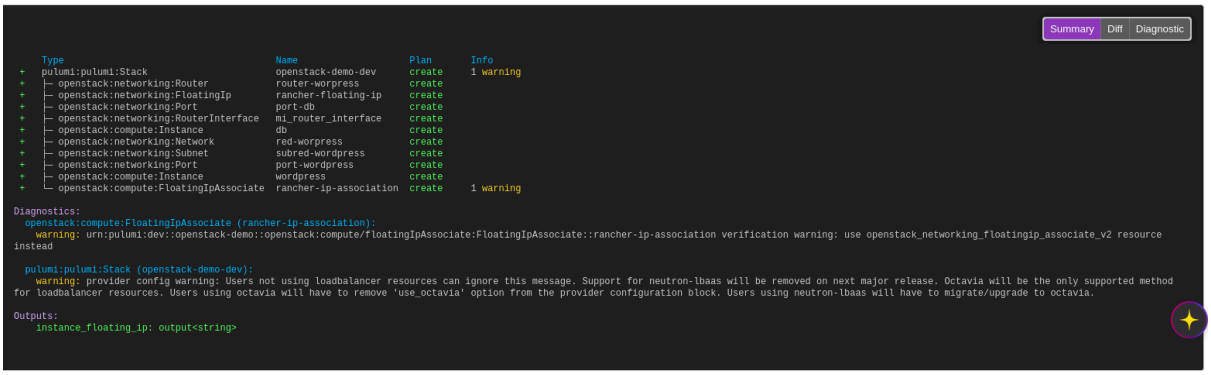
```
source ficheroRC.sh
```

Una vez ya hemos metidos las credenciales podremos ejecutar el comando pulumi up que este se va a encargar de desplegar todo el código que se ha creado, también este despliegue lo podemos ver en el panel de control de forma gráfica en la web de pulumi pulsando Ctrl+O, cuando ejecutamos el comando pulumi up nos mostrará todos los componentes que hemos ido creado cada uno está explicado en el anterior apartado. Y para finalizar le daremos a yes y empezará de forma automática a generar la infraestructura.

```
pulumi up
...
Previewing update (dev)

View in Browser (Ctrl+O):
https://app.pulumi.com/JuanantPineda/openstack-demo/dev/previews/1
9e91085-c380-4bd5-9e72-421a1ea13168

     Type                                    Name
Plan       Info
 +   pulumi:pulumi:Stack                     openstack-demo-dev
create     1 warning
 +   ├─ openstack:networking:Network         red-worpress
create
 +   ├─ openstack:networking:Subnet          subred-wordpress
create
 +   ├─ openstack:networking:Port            port-wordpress
create
 +   ├─ openstack:networking:Router          router-worpress
create
 +   ├─ openstack:networking:FloatingIp      rancher-floating-ip
create
 +   ├─ openstack:networking:Port            port-db
create
 +   ├─ openstack:compute:Instance           db
create
 +   ├─ openstack:networking:RouterInterface mi_router_interface
create
 +   ├─ openstack:compute:Instance           wordpress
create
```

```
 +   └─ openstack:compute:FloatingIpAssociate
rancher-ip-association  create     1 warning
...
Outputs:
    instance_floating_ip: output<string>

Resources:
    + 11 to create

Do you want to perform this update?  [Use arrows to move, type to
filter]
  yes
> no
  details
```

Si nos vamos al enlace que nos ha facilitado Pulumi podremos ver en su panel de control todo los componentes que se van a crear.

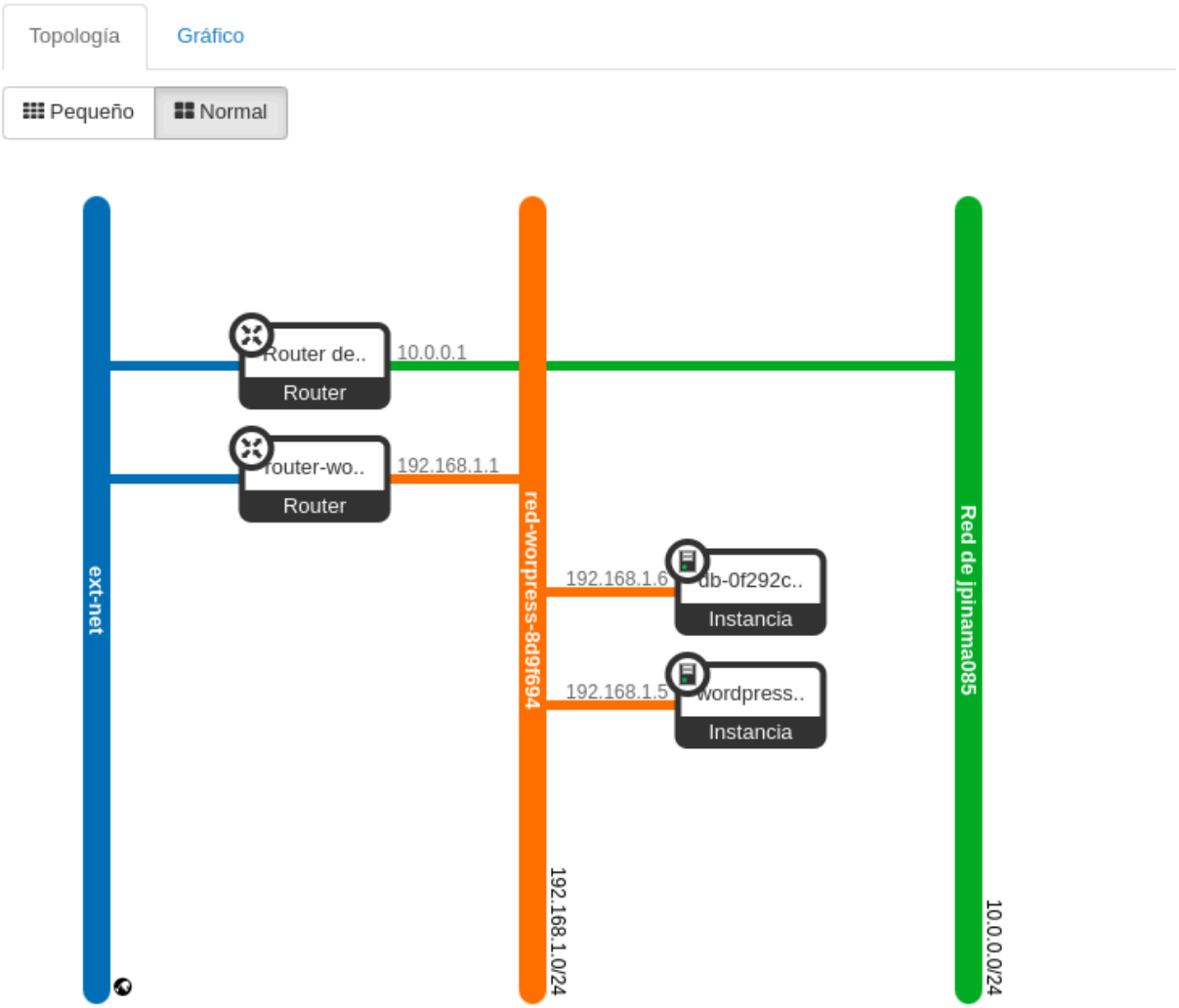


Y ahora empezará a crearse todos los componente si uno da fallo, tendremos que crearlo de nuevo y como pulumi es idempotente pues no vuelve a crear de nuevo los recursos que ya estaban creados.

```
Do you want to perform this update? yes
Updating (dev)

View in Browser (Ctrl+O):
https://app.pulumi.com/JuanantPineda/openstack-demo/dev/updates/1

     Type                                    Name
Status            Info
 +   pulumi:pulumi:Stack                     openstack-demo-dev
created (47s)     1 warning
 +   ├─ openstack:networking:Network          red-worpress
created (7s)
 +   ├─ openstack:networking:Router           router-worpress
created (8s)
 +   ├─ openstack:networking:FloatingIp       rancher-floating-ip
created (8s)
 +   ├─ openstack:networking:Subnet           subred-wordpress
created (6s)
 +   ├─ openstack:networking:RouterInterface  mi_router_interface
created (7s)
 +   ├─ openstack:networking:Port             port-db
created (6s)
 +   ├─ openstack:networking:Port             port-wordpress
created (7s)
 +   ├─ openstack:compute:Instance            db
created (16s)
 +   ├─ openstack:compute:Instance            wordpress
created (20s)
 +   └─ openstack:compute:FloatingIpAssociate
rancher-ip-association  created (3s)      1 warning
Outputs:
    instance_floating_ip: "172.22.201.158"

Resources:
```

```
    + 11 created

Duration: 48s
```

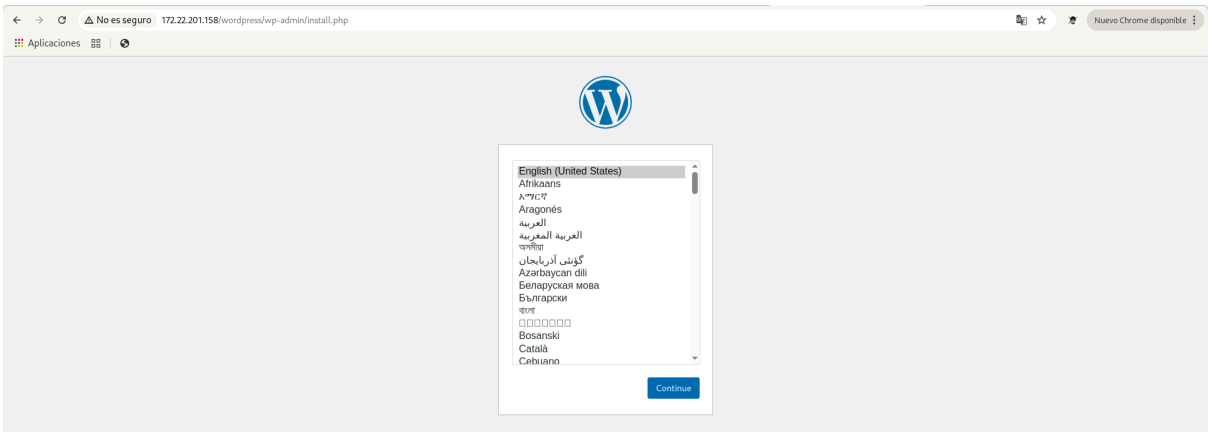
Y con esto ya tendríamos creado el escenario en openstack ahora vamos a comprobar que todo funciona correctamente.



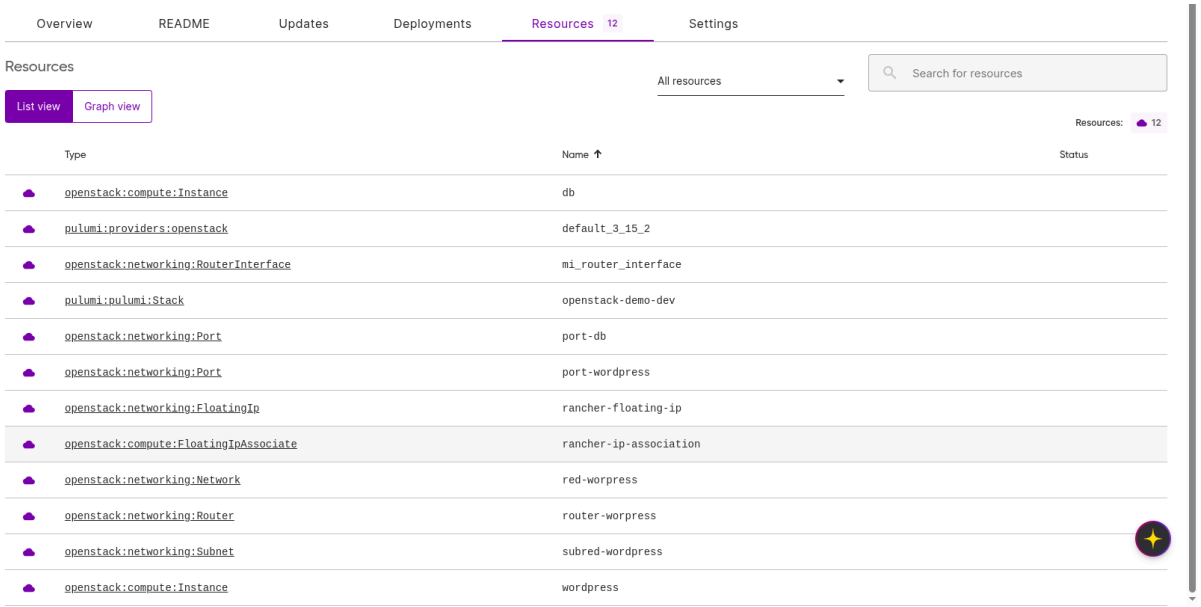
Si nos conectamos a la máquina que le hemos asociado la dirección ip, que en este caso es a la del wordpress podremos ver que se le ha instalado correctamente el wordpress. Ahora vamos a ver si realmente funciona el wordpress para ello tendremos que acceder desde un navegador web al wordpress.

```
usuario@triforce:~/openstack-demo$ ssh debian@172.22.201.158
The authenticity of host '172.22.201.158 (172.22.201.158)' can't
be established.
ED25519 key fingerprint is
SHA256:Y65HAm0XJL9gOn3QXGIVQGIGlIFsM2oHNBktmMtuEJU.
This key is not known by any other names.
Are you sure you want to continue connecting
(yes/no/[fingerprint])? yes
Warning: Permanently added '172.22.201.158' (ED25519) to the list
of known hosts.
Linux wordpress-bfb3d94 6.1.0-28-amd64 #1 SMP PREEMPT_DYNAMIC
Debian 6.1.119-1 (2024-11-22) x86_64

The programs included with the Debian GNU/Linux system are free
software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
debian@wordpress-bfb3d94:~$ ls /var/www/html/
index.html  wordpress
debian@wordpress-bfb3d94:~$
```

Si ponemos la url en el navegador podremos ver que ya tenemos instalado el Wordpress.



Y para finalizar, como curiosidad si nos vamos al panel de control de pulumi, y nos vamos al apartado de Resourses podremos ver que nos muestra todos los recurso que se han creado en openstack, con pulumi , como la red, subred, router,etc.



Y con esto ya tendríamos creado el escenario en openstack con código de python, gracias a pulumi podremos crear diversos escenarios con diferentes lenguajes de programación, ahora pasaremos a la demo con Kubernetes.

4.3 Demo Kubernetes

Para finalizar vamos a crear un escenario en kubernetes, en el cual vamos a crear un wordpress, para ello al igual que en la demo de openstack iré explicando paso por paso la creación del código del escenario con pulumi, utilizare el lenguaje de programación python para desplegar la infraestructura.

4.3.1 Creación del escenario

Ahora vamos a pasar a la creación del escenario para ello al igual que la demo usaremos el comando pulumi new y elegimos una plantilla que tenga que ver con kubernetes.

```
usuario@triforce:~/kubernetes-pulumi$ pulumi new
 Would you like to create a project from a template or using a Pulumi AI
prompt? template
Please choose a template (230 total):
  [Use arrows to move, type to filter]
  kubernetes-azure-csharp            A C# program to deploy a Kubernetes
cluster on Azure
  kubernetes-azure-go                A Go program to deploy a Kubernetes
cluster on Azure
…
> kubernetes-python              A minimal Kubernetes Python Pulumi
program
```

Y este empezar a crear el proyecto de kubernetes el cual lo usaremos para crear el código necesario, este empezará también a crear el entorno virtual el cual se va a descargar todas las dependencias necesarias para que funcione el programa que vamos a crear. Los ficheros que se generan ya se han explicado anteriormente lo cual en esta demo me voy a centrar solamente en explicar el código y el despliegue del escenario.

4.3.1.1 Configuracion general

Para empezar voy a ir explicando paso por paso las partes del código de la infraestructura que se encuentra en mi github, lo primero de todo tenemos que indicar en qué entorno queremos desplegar nuestra infraestructura en mi caso en Minikube ya que es el más simple y para explicar es más sencillo.

```
config = pulumi.Config()
```

Crea un objeto de configuración que permite leer valores definidos en el archivo Pulumi.<stack>.yaml (por ejemplo, Pulumi.dev.yaml).

Y luego debemos crear una variable de configuración para indicar si vamos a usar minikube o no.

```
is_minikube = config.get_bool("isMinikube") or False
```

Lee la variable de configuración isMinikube como un booleano. Si no está definida, asume False por defecto.

Esto lo ponemos por si queremos usar otro tipo de entorno pues pondremos true en vez de false.

4.3.1.2 Creacion de secrets

Ahora vamos a crear los secrets de la base de datos. Los Secrets en Kubernetes permiten almacenar datos sensibles como contraseñas o credenciales, de forma segura. En este caso, creamos un secreto con las variables necesarias para que WordPress y MySQL puedan comunicarse entre sí.

```
mysql_secret = Secret(
    "mysql-secret",
    metadata={"name": "mysql-secret"},
    string_data={
        "MYSQL_ROOT_PASSWORD": "rootpassword",
        "MYSQL_USER": "wordpress",
        "MYSQL_PASSWORD": "wordpresspass",
        "MYSQL_DATABASE": "wordpress"
    }
)
```

Estas variables son las que espera la imagen oficial de MySQL, por ejemplo wordpress tendrá otras variables diferentes.Este Secret será luego referenciado desde el Deployment tanto de MySQL como de WordPress, para configurar sus entornos automáticamente.

4.3.1.3 Creacion de volumenes

Para asegurar que los datos de la base de datos no se pierdan al eliminar el contenedor, vamos a crear un volumen persistente. Esto permite que la información se mantenga incluso si el pod se elimina o reinicia.

Kubernetes lo gestiona mediante un recurso llamado PersistentVolumeClaim (PVC).

```
mysql_pvc = PersistentVolumeClaim(
    "mysql-pvc",
    spec={
        "accessModes": ["ReadWriteOnce"],
        "resources": {"requests": {"storage": "1Gi"}},
    }
)
```

- **ReadWriteOnce**: Sólo puede montarse en modo lectura/escritura por un solo nodo a la vez.

- **storage**: 1Gi: Reservamos 1 GB de almacenamiento para la base de datos.

4.3.1.4 Creacion de los Deployments

Un Deployment en Kubernetes gestiona el ciclo de vida de los Pods, permitiendo escalar, reiniciar y actualizar nuestras aplicaciones de forma controlada.

En este caso, vamos a crear un Deployment específico para la base de datos MySQL. Este se encargará de mantener una instancia del contenedor funcionando y con acceso a sus datos mediante un volumen persistente.

```
mysql_deployment = Deployment(
    mysql_name,
    spec={
        "selector": {"matchLabels": {"app": mysql_name}},
        "replicas": 1,
        "template": {
            "metadata": {"labels": {"app": mysql_name}},
            "spec": {
                "containers": [{
                    "name": mysql_name,
                    "image": "mysql:5.7",
                    "envFrom": [{"secretRef": {"name":
"mysql-secret"}}],
                    "ports": [{"containerPort": 3306}],
                    "volumeMounts": [{"mountPath": "/var/lib/mysql",
"name": "mysql-storage"}]
                }],
```

```
                "volumes": [{"name": "mysql-storage",
"persistentVolumeClaim": {"claimName": mysql_pvc.metadata["name"]}}]
            }
        }
    }
)
```

- **replicas**: 1: Queremos un solo pod para MySQL.

- **envFrom**: Carga las variables de entorno desde el Secret mysql-secret creado anteriormente.

- **volumeMounts**: Monta el volumen persistente en la ruta /var/lib/mysql, que es donde MySQL guarda sus datos.

- **volumes**: Usa el PersistentVolumeClaim previamente creado (mysql_pvc) para asegurar la persistencia de datos.

Repetimos el mismo proceso que hicimos con MySQL, pero esta vez para la aplicación WordPress. La diferencia principal es que aquí definimos directamente las variables de entorno necesarias en el propio Deployment, sin usar un Secret.

```
wordpress_deployment = Deployment(
    app_name,
    spec={
        "selector": {"matchLabels": {"app": app_name}},
        "replicas": 1,
        "template": {
            "metadata": {"labels": {"app": app_name}},
            "spec": {
                "containers": [{
                    "name": app_name,
                    "image": "wordpress:latest",
                    "env": [
                        {"name": "WORDPRESS_DB_HOST", "value":
mysql_service.metadata["name"]},
                        {"name": "WORDPRESS_DB_USER", "value":
```

```
"wordpress"},
                        {"name": "WORDPRESS_DB_PASSWORD", "value":
"wordpresspass"},
                        {"name": "WORDPRESS_DB_NAME", "value":
"wordpress"},
                    ],
                    "ports": [{"containerPort": 80}]
                }]
            }
        }
    }
)
```

- **env**: Definimos las variables de entorno necesarias para que WordPress pueda conectarse a la base de datos.
    - **WORDPRESS_DB_HOST**: apunta al nombre del servicio de MySQL (interno en Kubernetes).
- **containerPort: 80**: WordPress expone su contenido web por el puerto 80.

4.3.1.5 Creacion de los Services

Para que los Pods puedan comunicarse entre sí o ser accesibles desde fuera, Kubernetes utiliza los objetos llamados Services. Cada Deployment debe tener su Service correspondiente.Existen varios tipos de Service:

- **ClusterIP**: (Por defecto) Solo accesible dentro del clúster
- **NodePort**: Accesible desde fuera del clúster a través de un puerto del nodo
- **LoadBalancer**: Crea una IP pública (en cloud) para acceder desde Internet
- **ExternalName**: Redirige a servicios externos (como DNS)

La base de datos no necesita ser accesible desde fuera, solo desde WordPress. Por eso, usamos el tipo ClusterIP o dejamos el tipo por defecto. El valor clusterIP: "None" lo convierte en un Headless Service, útil si necesitas DNS por Pod, pero no es necesario aquí.

```
mysql_service = Service(
    mysql_name,
    spec={
        "ports": [{"port": 3306}],
        "selector": {"app": mysql_name},
        "clusterIP": "None"
    }
)
```

En este caso, sí queremos acceder a WordPress desde fuera del clúster. Por eso, usamos un tipo de Service que cambia en función del entorno:

```
wordpress_service = Service(
    app_name,
    spec={
        "type": "ClusterIP" if is_minikube else "LoadBalancer",
        "ports": [{"port": 80, "targetPort": 80}],
        "selector": {"app": app_name},
    }
)
```

Si estamos usando minikube (is_minikube = True) el type será ClusterIP. Si estamos en un entorno de nube pública (como AWS, GCP, Azure), se usará LoadBalancer, y se asignará automáticamente una IP pública para acceder a WordPress

4.3.1.5 Creacion del Ingress

Para finalizar vamos a crear el ingress para poder acceder al wordpress mediante una URL, un ingress es un proxy inverso que por medio de reglas de encaminamiento que obtiene de la API de Kubernetes, nos permite el acceso a nuestras aplicaciones por medio de nombres. Tendremos que crear un ingress solamente para el Service del wordpress de la siguiente manera:

```
wordpress_ingress = Ingress(
    "wordpress-ingress",
    metadata={
        "name": "wordpress-ingress",
        "annotations": {
            "nginx.ingress.kubernetes.io/rewrite-target": "/"
        }
    },
    spec={
        "rules": [{
            "host": "wordpress.local",
            "http": {
                "paths": [{
                    "path": "/",
                    "pathType": "Prefix",
                    "backend": {
                        "service": {
                            "name":
wordpress_service.metadata["name"],
                            "port": {"number": 80}
                        }
                    }
                }]
            }
        }]
    }
)
```

- **metadata.name:** Asigna el nombre "wordpress-ingress" al recurso Ingress.

- **annotations**: Añade una anotación específica para el Ingress Controller de NGINX, permitiendo reescritura de rutas.

- **host**: Define el dominio interno desde el cual se accederá a WordPress, en este caso wordpress.local.

- **path**: Indica que cualquier ruta (/) será redirigida al backend.

- **backend.service.name**: Hace referencia al nombre del Service de WordPress para enrutar el tráfico HTTP.

- **port.number**: El puerto en el que escucha el servicio (en este caso, 80).

Una vez desplegado este recurso y teniendo un Ingress Controller activo (como NGINX), podremos acceder a WordPress desde un navegador escribiendo:

```
http://wordpress.local
```

4.3.2 Despliegue del escenario

Una vez ya hayamos definido todo el código, vamos a desplegar el escenario con pulumi, pero primero , esto es super importante ponerlo si estamos usando minikube tenemos que poner el siguiente comando:

```
usuario@triforce:~/kubernetes-pulumi$ pulumi config set isMinikube
true
usuario@triforce:~/kubernetes-pulumi$ ls
__main__.py  Pulumi.dev.yaml  Pulumi.yaml  __pycache__
requirements.txt  venv
```

Este comando lo que nos va indicar es que estamos usando minikube con pulumi para que nos cree correctamente el escenario en minikube, como no lo pongamos no se va a crear nos quedaremos en bucle en el despliegue. Además nos creara un archivo **Pulumi.dev.yaml** que este básicamente lo que hace es meter los parametros en un yaml para que se despliegue con minikube.

Ahora con **pulumi up** vamos a desplegar el escenario como lo hicimos con openstack, este empezara a crear los recursos necesario y le damos yes para su creación.

```
usuario@triforce:~/kubernetes-pulumi$ pulumi up
Previewing update (dev)

View in Browser (Ctrl+O):
https://app.pulumi.com/JuanantPineda/kubernetes-pulumi/dev/previews/4481
9d56-0458-478f-82da-42bba7f99218

     Type                                     Name
Plan
 +   pulumi:pulumi:Stack                      kubernetes-pulumi-dev
create
 +   ├─ kubernetes:core/v1:Service            mysql
create
 +   ├─ kubernetes:core/v1:PersistentVolumeClaim  mysql-pvc
create
```

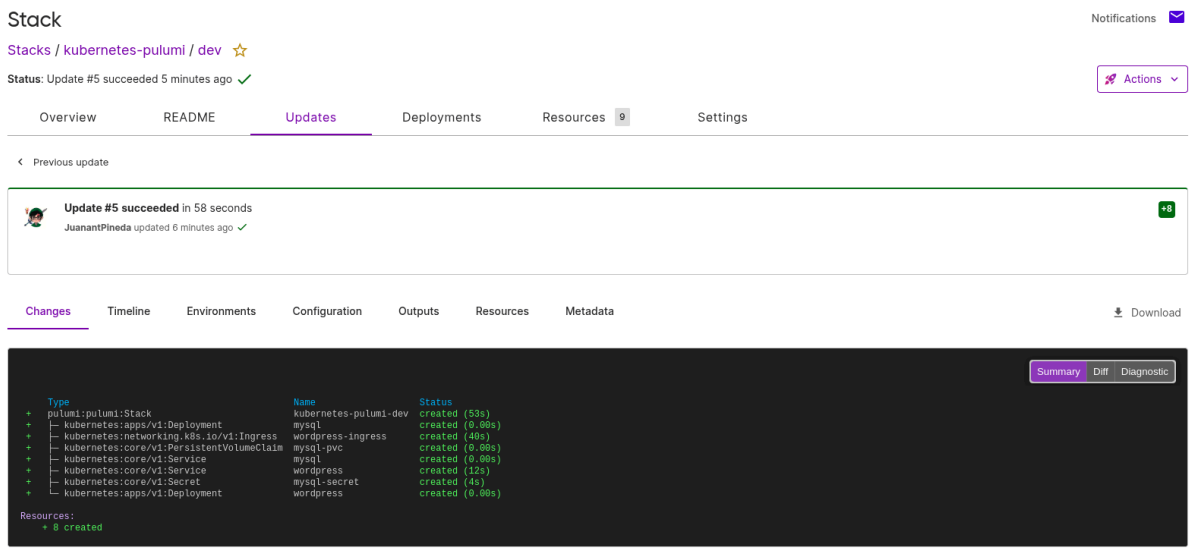
```
 +   ├─ kubernetes:core/v1:Service                wordpress
create
 +   ├─ kubernetes:core/v1:Secret                 mysql-secret
create
 +   ├─ kubernetes:networking.k8s.io/v1:Ingress   wordpress-ingress
create
 +   ├─ kubernetes:apps/v1:Deployment             wordpress
create
 +   └─ kubernetes:apps/v1:Deployment             mysql
create

Resources:
    + 8 to create

Do you want to perform this update?  [Use arrows to move, type to
filter]
  yes
> no
  details
```

Y ahora empezará la creación de los recursos, si queremos verla en el navegador nos meteremos en el enlace que nos da o pulsamos (Ctrl+O).

```
Do you want to perform this update? yes
Updating (dev)

View in Browser (Ctrl+O):
https://app.pulumi.com/JuanantPineda/kubernetes-pulumi/dev/updates
/5

     Type                                        Name
Status
 +   pulumi:pulumi:Stack
kubernetes-pulumi-dev  created (55s)
 +   ├─ kubernetes:core/v1:PersistentVolumeClaim  mysql-pvc
created (0.78s)
 +   ├─ kubernetes:core/v1:Service                mysql
created (0.54s)
 +   ├─ kubernetes:core/v1:Service                wordpress
```

```
created (11s)
 +   ├─ kubernetes:core/v1:Secret                 mysql-secret
created (1s)
 +   ├─ kubernetes:apps/v1:Deployment             wordpress
created (3s)
 +   ├─ kubernetes:apps/v1:Deployment             mysql
created (1s)
 +   └─ kubernetes:networking.k8s.io/v1:Ingress
wordpress-ingress      created (40s)

Resources:
    + 8 created

Duration: 57s
```

Y aqui podemos ver los recursos que se han creado en el navegador.



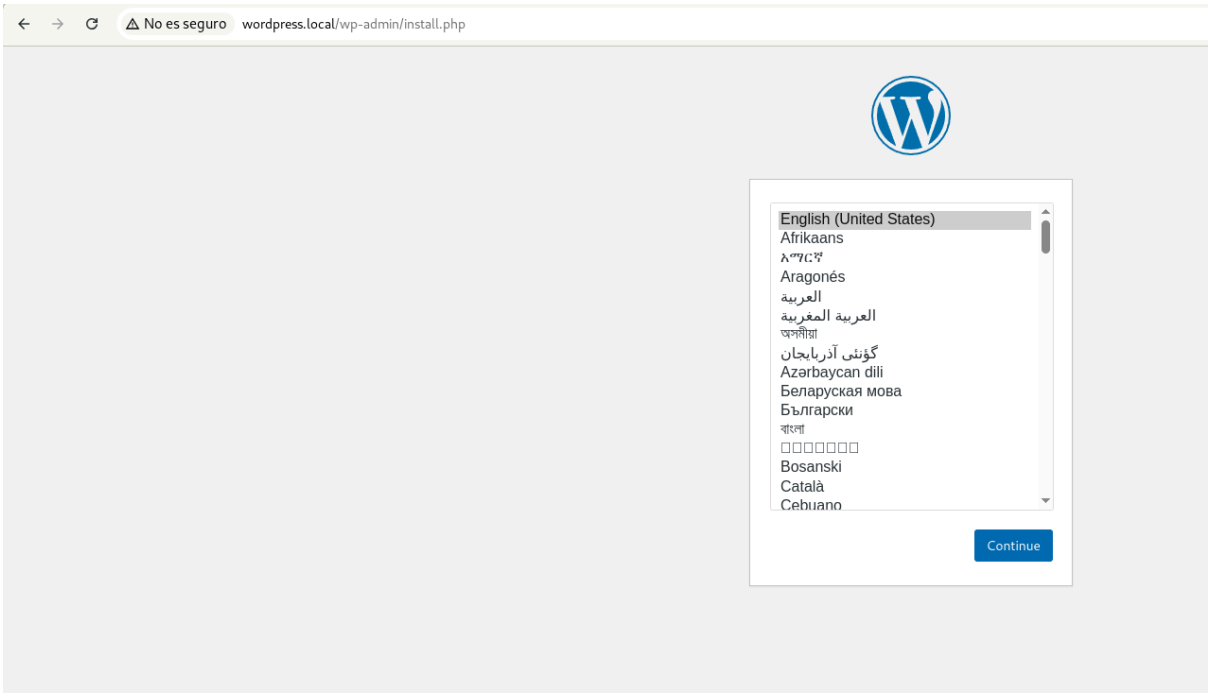
Y para finalizar vamos ver si realmente está funcionando el wordpress para ello tenemos que saber la ip de minikube de la siguiente manera.

```
usuario@triforce:~$ minikube ip
192.168.39.44
```

Y la tenemos que poner en el /etc/hosts y poner el nombre del dominio que le hemos vinculado en el ingress que hemos creado

```
usuario@triforce:~$ cat /etc/hosts
...
192.168.39.44    pruebak8s.pineda.com wordpress.local
...
```

Y si accedemos con el dominio podremos ver que se ha creado correctamente el wordpress y podemos ya usarlo y configurarlo.



Y con esto ya tendríamos finalizado la demo práctica de kubernetes con pulumi

5. Conclusiones y propuestas

A lo largo de este proyecto se ha demostrado que Pulumi es una herramienta potente, flexible y versátil que permite gestionar y desplegar infraestructura mediante código. Su principal ventaja frente a otras herramientas de Infraestructura como Código (IaC), como Terraform, es que permite trabajar con lenguajes de programación tradicionales como Python, JavaScript, Go, entre otros. Esto reduce considerablemente la curva de aprendizaje, ya que no es necesario aprender un lenguaje declarativo específico como HCL (HashiCorp Configuration Language).

Además, Pulumi destaca por su compatibilidad multiplataforma, ya que permite integrarse con múltiples tecnologías y proveedores de infraestructura como OpenStack, AWS, Azure, GCP, Kubernetes, entre otros. Esta capacidad lo convierte en una opción muy atractiva tanto para entornos de desarrollo como de producción.

Durante el desarrollo del proyecto se han creado dos escenarios principales:

- Uno basado en OpenStack, desplegando automáticamente una red, dos instancias y la configuración de servicios como WordPress y MariaDB mediante Cloud-Init.

- Otro usando Kubernetes (minikube), donde se ha automatizado el despliegue de servicios y volúmenes persistentes mediante Pulumi.

Una funcionalidad que no se ha podido desarrollar por limitaciones de tiempo ha sido la integración de Pulumi con AWS, lo cual habría aportado un enfoque más realista y orientado al mundo laboral, ya que actualmente la mayoría de las empresas utilizan proveedores cloud como AWS para su infraestructura.

Asimismo, habría sido interesante ampliar el escenario de Kubernetes desplegándolo en otros entornos como Kind o un clúster multi-nodo real de K8s, lo cual habría permitido comprobar el comportamiento de Pulumi en entornos más complejos y realistas.

En resumen, Pulumi ha resultado ser una herramienta excelente para introducirse en el mundo de la infraestructura como código, permitiendo aprender conceptos fundamentales de forma más accesible, eficiente y práctica. Su uso en este proyecto ha permitido no solo automatizar infraestructura, sino también comprender mejor cómo se despliegan, configuran y conectan los distintos componentes en un entorno cloud moderno.

6. Bibliografía

- https://ualmtorres.github.io/seminario-pulumi/
- https://www.pulumi.com/docs/iac/get-started/
- https://www.pulumi.com/docs/iac/get-started/kubernetes/
- https://www.pulumi.com/registry/packages/openstack/
- https://www.pulumi.com/registry/packages/openstack/api-docs/networking/subnet/
- https://www.pulumi.com/registry/packages/openstack/api-docs/
- https://www.pulumi.com/registry/packages/kubernetes/api-docs/
- https://www.pulumi.com/ai