ACTIVIDAD 2

A. Infraestructura como Código

1. Introducción a IaC

- **Definición:** IaC consiste en gestionar la configuración y la administración mediante el uso de scripts (archivos de texto). Esto conlleva a tener toda la parte administrativa o de configuración bajo indicaciones ya escritas. Comparándolo con la configuración manual, el fundamento IaC nos facilita las configuraciones de las aplicaciones que usemos, ahorrandonos bastante tiempo, así como también reduciendo errores al momento de configurarlas.
- **Beneficios:**
  - **Consistencia:** Al estar todo escrito en archivos, cada vez que se configure algo, se configurará de la misma manera en cualquier servidor o sistema.
  - **Control de versiones:** Se puede guardar los cambios que se haga en la configuración, volver a versiones anteriores si algo falla y saber quién modificó qué.
  - **Automatización:** No hay necesidad de configurar todo manualmente cada vez; basta con solo ejecutar un archivo y la configuración se hace sola.
  - **Menos errores humanos:** Como no se tiene que configurar todo manualmente, se reduce la posibilidad de fallo en la configuración.

2. Escritura de IaC

- **Herramientas populares:**
  - **Terraform:** Permite crear y administrar infraestructura mediante archivos de configuración. Mayormente usado en servicios en la nube como AWS, Azure o Google Cloud.
  - **Ansible:** Usado para instalar programas, configurar servidores y hacer modificaciones en sistemas de manera automatizada.
  - **Pulumi:** Comparándolo con Terraform, Pulumi permite escribir los archivos de configuración en lenguaje conocidos como Python o C#.
- **Buenas prácticas:**
  - **Nombres claros:** Usar nombres que se entiendan bien en los servidores, bases de datos y redes para reconocerlos fácilmente.
  - **Uso de variables:** En vez de usar valores fijos, usar variables hace que sea más fácil cambiar configuraciones sin modificar todo el código.

- **Modularización de código:** Dividir el código en partes pequeñas ayuda a que sea más ordenado y fácil de manejar.
- **Uso de Git:** Guardar los archivos en un repositorio permite ver los cambios, además de volver a versiones anteriores y potenciar el trabajo en equipo.

3. Patrones para modulos

- **Modularización:** En lugar de escribir toda la configuración en solamente un archivo, se deben crear módulos específicos para cada parte del sistema, por ejemplo, un módulo para redes, otro para bases de datos y otro para servidores de aplicaciones. Esto permite reutilizar dichos módulos en otros proyectos.
- **Estructura:**

```
infraestructura/
|— main.tf                # Archivo principal que une los módulos
|— variables.tf           # Variables generales del proyecto
|— outputs.tf             # Valores de salida del proyecto
|— modules/
|   ├— red/
|   |   ├— main.tf        # Configuración de la red
|   |   ├— variables.tf   # Variables especificas de la red
|   |   ├— outputs.tf     # Valores de salida de la red
|   ├— base_datos/
|   |   ├— main.tf        # Configuración de la base de datos
|   |   ├— variables.tf   # Variables especificas de la base de datos
|   |   ├— outputs.tf     # Valores de salida de la base de datos
|   ├— servidores/
|   |   ├— main.tf        # Configuración de los servidores de aplicación
|   |   ├— variables.tf   # Variables especificas de los servidores
|   |   ├— outputs.tf     # Valores de salida de los servidores
```

4. Patrones para dependencias

- **Gestión de dependencias:** Cuando un módulo necesita información de otro módulo para su configuración, se hace uso del enfoque de paso de información entre módulos (en vez de definir los valores manualmente). Esto hace que los módulos sean independientes, reutilizables y también con la capacidad de comunicarse.
- **Outputs e inputs:** Cuando un módulo intercambia información con otro, se usan outputs e inputs.
  1. **Outputs:** Valores generados por un módulo que otros pueden usar.
  2. **Inputs:** Valores que un módulo necesita y que pueden provenir de otro módulo.

B. Contenerización y despliegue de aplicaciones modernas

1. Contenerización de una aplicación con Docker

- **¿Qué son los contenedores?:** Una máquina virtual crea un sistema operativo completo dentro de otro. En cambio, un contenedor comparte el mismo sistema, pero mantiene sus procesos separados, volviéndolos más rápidos y menos consumidores de recursos.
- **Dockerfile:** Es un archivo donde se escriben las instrucciones para crear un contenedor. Un ejemplo de la estructura básica de un Dockerfile puede ser:
  1. **FROM:** Define la imagen base sobre la que se construirá el contenedor.
  2. **WORKDIR:** Define la carpeta donde se ejecutarán los comandos dentro del contenedor.
  3. **COPY:** Copia archivos del sistema al contenedor.
  4. **RUN:** Permite ejecutar comandos durante la construcción del contenedor.
  5. **CMD/ENTRYPOINT:** Define el comando que se ejecutará cuando se inicie el contenedor.
     Ejemplo:

```
FROM python:3.9
WORKDIR /app
COPY . .
RUN pip install -r requerimientos.txt
CMD ["python", "app.py"]
```

- **Imagen vs Contenedor:**