TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA

Trabajo Integrador Arquitectura y Sistemas Operativos

Virtualización

Alumnos:

Giardini Silvia

Cordero Marina

Materia:

Arquitectura y Sistemas Operativos

Profesor:

Mauricio Gabriel Pasti

Tutor:

Jonathan Zarate

Fecha de entrega:

05 de junio de 2025

Índice

Introducción

La **virtualización** es un pilar esencial en la informática moderna. Permite crear entornos aislados, como **máquinas virtuales** y **contenedores**, optimizando el uso de recursos y facilitando el desarrollo, despliegue y mantenimiento de aplicaciones. Seleccionamos este tema por su relevancia en la implementación de soluciones tecnológicas eficientes: la virtualización posibilita ejecutar múltiples sistemas operativos o aplicaciones en un mismo hardware físico, mejorando la **eficiencia**, **escalabilidad** y **flexibilidad**. Las máquinas virtuales, que emulan sistemas operativos completos, y los contenedores, que ofrecen entornos ligeros y portátiles, son fundamentales para gestionar eficazmente entornos de desarrollo y producción.

Para un técnico en programación, comprender la virtualización es crucial. Nos permite configurar y administrar estos entornos usando herramientas como **VMware**, **VirtualBox**, **Hyper-V** para máquinas virtuales robustas, y **Docker** para la creación ágil de contenedores.

El objetivo de este trabajo es explorar los conceptos fundamentales de la virtualización, implementando un entorno con Docker que ejecute un programa Python simple. Así, analizaremos su funcionamiento para comprender las ventajas y desafíos de estas tecnologías en un entorno práctico.

Marco Teórico

En el contexto del desarrollo en programación, la virtualización ofrece diversas herramientas y entornos que facilitan la creación, prueba y despliegue de aplicaciones. Algunos ejemplos incluyen:

1. **Máquinas virtuales (VM)**: Herramientas como VMware, VirtualBox o Hyper-V permiten crear entornos completos con sistemas operativos virtualizados. Los desarrolladores pueden usarlas para probar aplicaciones en diferentes sistemas operativos (Windows, Linux, macOS) o configuraciones sin necesidad de hardware adicional.

2. **Contenedores**: Tecnologías como Docker y Kubernetes proporcionan entornos ligeros y aislados para ejecutar aplicaciones y sus dependencias. Los contenedores son ideales para el desarrollo, ya que permiten replicar entornos de producción, garantizar consistencia entre desarrollo y despliegue, y facilitar la portabilidad.

3. **Entornos de desarrollo virtualizados**: Herramientas como Vagrant permiten crear y gestionar entornos de desarrollo reproducibles, configurando máquinas virtuales o contenedores con scripts para instalar dependencias automáticamente, lo que agiliza el trabajo en equipo.

4. **Entornos de virtualización de aplicaciones**: Soluciones como Wine o Parallels Desktop permiten ejecutar aplicaciones diseñadas para un sistema operativo en otro, útil para probar software en entornos heterogéneos.

5. **Cloud Computing**: Plataformas como AWS, Azure o Google Cloud ofrecen máquinas virtuales y servicios de contenedores que los desarrolladores utilizan para escalar aplicaciones, realizar pruebas en entornos distribuidos o implementar microservicios.

Estas herramientas y entornos de virtualización son esenciales en el desarrollo de software, ya que permiten a los programadores trabajar en entornos controlados, reproducibles y escalables, optimizando el proceso de desarrollo, prueba y despliegue de aplicaciones.

En este trabajo práctico, nos centraremos exclusivamente en dos de las tecnologías más relevantes en este ámbito: **las máquinas virtuales (VM)** y **los contenedores**. A continuación, exploraremos en detalle estas tecnologías y, posteriormente, realizaremos una comparación entre ambas para destacar sus características, ventajas y casos de uso.

Máquinas Virtuales (VM)

Las **Máquinas Virtuales (VM)** son entornos de computación que emulan un sistema informático completo, replicando tanto el hardware virtualizado (como CPU, memoria, almacenamiento y red) como un sistema operativo completo. Este enfoque permite ejecutar múltiples sistemas operativos de manera independiente en un solo hardware físico, cada uno en su propia VM, lo que las hace ideales para casos como pruebas de software, entornos de desarrollo, aislamiento de aplicaciones o ejecución de sistemas legacy (o sistemas heredados).

Según VMware, una empresa líder en tecnologías de virtualización, las máquinas virtuales se definen como:

"Una máquina virtual (VM) es una representación digital de un sistema físico, que ejecuta un sistema operativo completo y aplicaciones como una computadora real. Un hipervisor permite que múltiples VMs se ejecuten en un solo host físico, proporcionando aislamiento y optimización de recursos."

Características y funcionamiento:

1. **Emulación de hardware**: Las VMs simulan componentes físicos completos, lo que permite que cada máquina virtual funcione como si fuera un ordenador independiente. Esto incluye procesadores virtuales, discos duros, interfaces de red, etc.

2. **Sistema operativo completo**: Cada VM ejecuta su propio sistema operativo (por ejemplo, Windows, Linux, etc.), lo que proporciona un entorno totalmente funcional, pero también implica un mayor uso de recursos, ya que el sistema operativo consume memoria, CPU y almacenamiento.

3. **Hipervisor**: Las VMs son gestionadas por un hipervisor (como VMware ESXi, Microsoft Hyper-V, Oracle VirtualBox o KVM), que actúa como una capa intermedia entre el hardware físico y las máquinas virtuales. Hay dos tipos principales de hipervisores:

   - **Tipo 1 (nativos o bare-metal)**: Se ejecutan directamente sobre el hardware físico, ofreciendo mejor rendimiento y eficiencia (ejemplo: VMware ESXi, XenServer).

   - **Tipo 2 (hospedados)**: Se ejecutan sobre un sistema operativo anfitrión, siendo más fáciles de configurar, pero menos eficientes (ejemplo: VirtualBox, VMware Workstation).

4. **Aislamiento robusto**: Cada VM opera de forma aislada, lo que significa que un fallo o compromiso en una VM no afecta a otras VMs ni al sistema anfitrión. Esto las hace ideales para entornos que requieren alta seguridad o separación de cargas de trabajo.

Ventajas:

- **Flexibilidad**: Permiten ejecutar diferentes sistemas operativos (por ejemplo, Windows y Linux) en el mismo hardware físico.
- **Aislamiento**: Proporcionan un entorno seguro y aislado, útil para pruebas, desarrollo o ejecución de aplicaciones en entornos controlados.
- **Portabilidad**: Las VMs pueden trasladarse entre diferentes hipervisores o servidores con facilidad, siempre que sean compatibles.
- **Snapshots y backups**: Se pueden crear instantáneas (snapshots) para guardar el estado de una VM en un momento dado, facilitando la recuperación ante fallos o cambios no deseados.
- **Compatibilidad**: Ideales para sistemas legacy o aplicaciones que requieren entornos específicos.

Desventajas:

- **Consumo de recursos**: La emulación de hardware y la ejecución de un sistema operativo completo por cada VM generan un mayor uso de CPU, memoria y almacenamiento en comparación con otras tecnologías como contenedores.
- **Rendimiento**: El overhead del hipervisor y la emulación de hardware pueden reducir el rendimiento en comparación con sistemas nativos o contenedores.
- **Tiempo de arranque**: Las VMs requieren iniciar un sistema operativo completo, lo que puede ser más lento que otras soluciones ligeras.

- **Gestión compleja**: Configurar y mantener múltiples VMs puede ser más complicado, especialmente en entornos con muchas máquinas virtuales.

Casos de uso:

- **Entornos de prueba y desarrollo**: Los desarrolladores pueden probar aplicaciones en diferentes sistemas operativos sin necesidad de hardware dedicado.
- **Servidores consolidados**: En centros de datos, las VMs permiten ejecutar múltiples aplicaciones en un solo servidor físico, optimizando recursos.
- **Seguridad**: Aislar aplicaciones críticas o sospechosas en VMs para minimizar riesgos.
- **Migración de sistemas**: Facilitan la migración de servidores físicos a entornos virtualizados o a la nube.
- **Recuperación ante desastres**: Las VMs pueden replicarse o respaldarse fácilmente para garantizar la continuidad del negocio.

Contenedores

Los **contenedores** son entornos virtualizados ligeros que permiten **empaquetar** y **ejecutar** aplicaciones junto con sus dependencias (bibliotecas, configuraciones, herramientas específicas) de manera aislada, compartiendo el kernel del sistema operativo anfitrión.

A diferencia de las máquinas virtuales, que incluyen un sistema operativo completo y consumen más recursos, los contenedores son mucho **más eficientes en términos de memoria, CPU y almacenamiento, ya que no duplican el sistema operativo base**. Esto los hace ideales para entornos donde se requiere alta portabilidad, escalabilidad y rapidez en el despliegue.

Según la documentación oficial de Docker, un contenedor es "una unidad estándar de software que empaqueta el código y todas sus dependencias, de modo que la aplicación se ejecuta de manera rápida y confiable en diferentes entornos de cómputo. Una imagen de contenedor de Docker es un paquete de software ligero, independiente y ejecutable que incluye todo lo necesario para ejecutar una aplicación: código, runtime, herramientas del sistema, bibliotecas y configuraciones".

Los contenedores se basan en tecnologías como namespaces y cgroups en Linux, que garantizan el aislamiento de procesos y la gestión eficiente de recursos. Esto permite que múltiples contenedores se ejecuten simultáneamente en el mismo host sin conflictos, compartiendo recursos de manera óptima. Herramientas como **Docker** y **Kubernetes** han popularizado su uso al simplificar la creación, gestión y orquestación de contenedores, facilitando su adopción en entornos de desarrollo, pruebas y producción.

Los contenedores son especialmente valiosos en arquitecturas de microservicios, donde cada componente de una aplicación puede ejecutarse en un contenedor independiente, lo que mejora la modularidad y la facilidad de actualización. Por ejemplo, un contenedor puede alojar una aplicación web, mientras otro maneja la base de datos, y ambos se comunican sin necesidad de entornos monolíticos complejos. Su portabilidad asegura que una aplicación funcione de manera consistente en diferentes entornos, desde un portátil de desarrollo hasta un clúster en la nube, siempre que el motor de contenedores (como Docker) esté presente.

En términos prácticos, los contenedores ofrecen ventajas como:

- **Eficiencia de recursos**: Al compartir el kernel, consumen menos memoria y CPU que las máquinas virtuales.
- **Portabilidad**: Los contenedores pueden ejecutarse en cualquier sistema compatible con el motor de contenedores, independientemente del entorno subyacente.
- **Escalabilidad**: Herramientas como Kubernetes permiten escalar contenedores horizontalmente (añadiendo más instancias) o verticalmente (asignando más recursos) con facilidad.
- **Despliegue rápido**: Los contenedores se inician en segundos, a diferencia de las máquinas virtuales, que requieren minutos.
- **Ecosistema robusto**: Imágenes preconfiguradas en repositorios como **Docker Hub** facilitan el acceso a aplicaciones y servicios listos para usar.

En resumen, los contenedores representan una solución eficiente y moderna para el desarrollo y despliegue de aplicaciones, siendo una tecnología clave en la informática actual, especialmente en entornos de nube y DevOps.

Estructura de Docker

Las estructuras de Docker son los componentes fundamentales que permiten construir, ejecutar y gestionar aplicaciones en contenedores. Incluyen:

1. **Imágenes**:

- **Definición**: Las imágenes de Docker son archivos inmutables y ligeros que encapsulan todo lo necesario para ejecutar una aplicación, incluyendo el código fuente, bibliotecas, dependencias, configuraciones y el sistema operativo base (como una versión mínima de Linux o Windows).

- **Características**:

  - Son portátiles, lo que permite su uso en diferentes entornos (desarrollo, pruebas, producción) sin modificaciones.

  - Se versionan mediante etiquetas (tags), como nginx:latest o nginx:1.21, para identificar diferentes versiones de la misma imagen.

  - Se construyen en capas, lo que optimiza el almacenamiento y la transferencia, ya que las capas comunes entre imágenes se reutilizan.

- **Almacenamiento**: Las imágenes se almacenan en registros, como Docker Hub (público), Amazon ECR, Google Container Registry o registros privados configurados por las organizaciones.

- **Ejemplo**: Una imagen de una aplicación web puede incluir un servidor como Nginx, el código de la aplicación y las dependencias necesarias, listas para ejecutarse en cualquier máquina con Docker.

2. **Contenedores**:

- **Definición**: Un contenedor es una instancia ejecutable de una imagen. Es un entorno aislado que contiene todo lo necesario para que la aplicación funcione, pero comparte el kernel del sistema operativo anfitrión, lo que lo hace más ligero que una máquina virtual.

- **Características**:
  - **Aislamiento**: Utilizan tecnologías del sistema operativo como namespaces (para aislar procesos, red, sistema de archivos) y cgroups (para limitar recursos como CPU y memoria).
  - **Efímero**: Los contenedores son desechables; pueden crearse, detenerse, eliminarse y recrearse sin pérdida de funcionalidad, siempre que se base en la misma imagen.
  - **Estado**: Pueden ser stateless (sin datos persistentes) o stateful (con almacenamiento persistente usando volúmenes o bind mounts).
- **Ciclo de vida**: Un contenedor puede estar en estados como creado, en ejecución, pausado, detenido o eliminado, gestionados mediante comandos como docker run, docker stop, o docker rm.

3. **Dockerfiles**:
   - **Definición**: Un Dockerfile es un archivo de texto con un conjunto de instrucciones que especifican cómo construir una imagen de Docker. Actúa como un script automatizado para la creación de imágenes.
   - **Instrucciones principales**:
     - **FROM**: Define la imagen base desde la cual se construye (por ejemplo, FROM ubuntu:20.04).
     - **RUN**: Ejecuta comandos durante la construcción de la imagen, como instalar dependencias (RUN apt-get install python3).
     - **COPY** y **ADD**: Copian archivos o directorios desde el host a la imagen.

- **WORKDIR**: Establece el directorio de trabajo dentro de la imagen.
- **EXPOSE**: Indica los puertos que el contenedor expondrá.
- **ENTRYPOINT** y **CMD**: Definen el comando predeterminado que se ejecuta al iniciar el contenedor. ENTRYPOINT especifica el ejecutable principal, mientras que CMD define argumentos o comandos predeterminados que pueden sobrescribirse.

- **Buenas prácticas**:
  - Minimizar el número de capas combinando comandos (por ejemplo, usando && en una sola instrucción RUN).
  - Usar imágenes base ligeras, como alpine, para reducir el tamaño.
  - Evitar incluir datos innecesarios para optimizar la imagen.

4. **Volúmenes y Bind Mounts**:

- **Volúmenes**: Espacios de almacenamiento gestionados por Docker para persistir datos generados por contenedores o compartir datos entre ellos. Son independientes del ciclo de vida del contenedor y se almacenan en el host.
  - Tipos: Volúmenes nombrados, anónimos o gestionados por el sistema de archivos del host.
- **Bind Mounts**: Mapean directorios o archivos específicos del host al contenedor, útiles para desarrollo o configuraciones específicas.
- **Uso**: Los volúmenes son ideales para bases de datos (por ejemplo, persistir datos de MySQL), mientras que los bind mounts son útiles para montar código fuente en desarrollo.

5. **Redes (Networking)**:

   - **Definición**: Docker proporciona redes virtuales para permitir la comunicación entre contenedores y con el mundo exterior.
   - **Tipos de redes**:
     - **Bridge**: Red predeterminada para contenedores en el mismo host, que permite comunicación interna mediante nombres de contenedores.
     - **Host**: El contenedor usa la red del host directamente, eliminando el aislamiento de red.
     - **Overlay**: Permite comunicación entre contenedores en diferentes hosts, comúnmente usado en Docker Swarm o Kubernetes.
     - **None**: Desactiva la red para el contenedor, aislando completamente su conectividad.

6. **Docker Compose**:

   - **Definición**: Herramienta para definir y gestionar aplicaciones multi-contenedor usando un archivo YAML (docker-compose.yml).
   - **Funcionalidad**: Permite configurar servicios, redes y volúmenes en un solo archivo, facilitando el despliegue de aplicaciones complejas.
   - **Uso**: Ideal para entornos de desarrollo y pruebas, o para aplicaciones que requieren múltiples contenedores coordinados.

7. **Registros (Docker Registries)**:

- **Definición**: Repositorios para almacenar y distribuir imágenes de Docker. Docker Hub es el registro predeterminado, pero se pueden configurar registros privados.
- **Operaciones**: Subir imágenes (docker push), descargarlas (docker pull) o buscarlas (docker search).
- **Seguridad**: Los registros privados permiten autenticación y control de acceso para imágenes sensibles.

8. **Seguridad**:

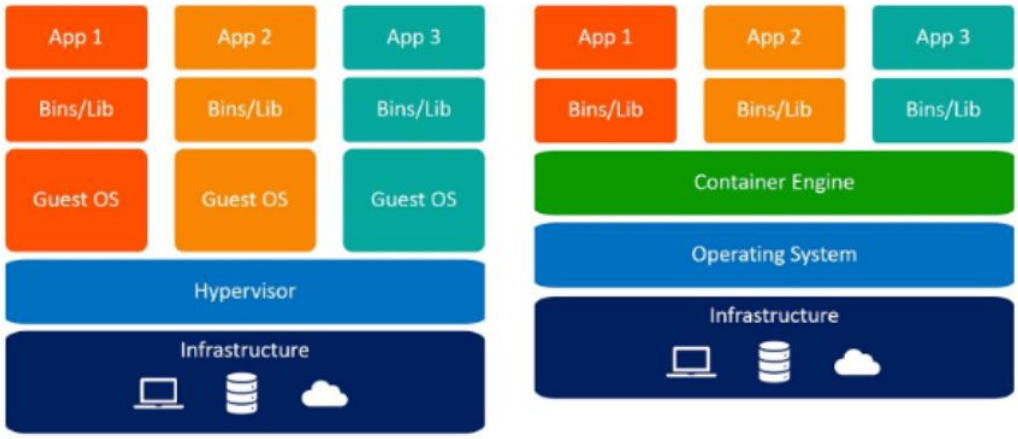
- **Aislamiento**: Los contenedores están aislados por defecto, pero vulnerabilidades en el kernel o configuraciones incorrectas pueden comprometer la seguridad.
- **Prácticas recomendadas**:
  - Ejecutar contenedores con usuarios no root.
  - Limitar recursos con cgroups.
  - Escanear imágenes en busca de vulnerabilidades (por ejemplo, con herramientas como Trivy o Docker Scan).
  - Usar imágenes oficiales y verificadas desde registros confiables.

Comparación Máquinas Virtuales vs contenedores:

A diferencia de los contenedores (como Docker), que comparten el kernel del

sistema operativo anfitrión y son más ligeros, las VMs incluyen un sistema operativo completo y emulan hardware, lo que las hace más pesadas pero también más aisladas y versátiles para ciertos casos. Mientras que los contenedores son ideales para aplicaciones modernas y escalables, las VMs son preferibles cuando se requiere un aislamiento completo o compatibilidad con sistemas operativos distintos.

En resumen, las máquinas virtuales ofrecen una solución robusta y flexible para virtualización, pero su mayor consumo de recursos las hace menos eficientes en comparación con tecnologías más ligeras como los contenedores. Su elección depende de las necesidades específicas de aislamiento, compatibilidad y rendimiento del entorno.



Diferencias clave:

***VMs**:* Mayor aislamiento, pero mayor consumo de recursos (memoria, CPU). Cada VM ejecuta un sistema operativo completo.

***Contenedores**:* Menor aislamiento (comparten el kernel), pero más ligeros y rápidos. Ideales para microservicios.

Comparación entre VMs y Contenedores

Característica	Máquinas Virtuales	Contenedores
Aislamiento	Alto (hardware y SO completos)	Moderado (comparte kernel)
Uso de recursos	Alto (SO completo por VM)	Bajo (sin SO completo)
Tiempo de inicio	Minutos	Segundos
Portabilidad	Moderada (depende del hipervisor)	Alta (estándar Docker)
Ejemplo	VMware, VirtualBox	Docker, Podman

Diagrama Comparativo Referencias

- Tanenbaum, A. S., & Bos, H. (2015). *Modern Operating Systems*. Pearson.
- Docker Inc. (2025). Documentación oficial de Docker. Recuperado el 30/05/2025 de https://docs.docker.com.

Caso Practico:

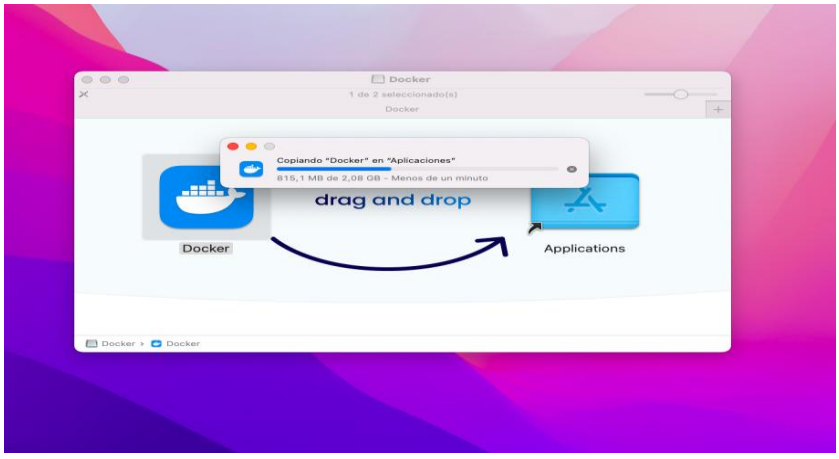
Primeros Pasos con Docker: Aplicación Python en Contenedores

Breve descripción del problema a resolver

El objetivo de este caso práctico elegido, es implementar un entorno de desarrollo robusto y portátil para una aplicación Python utilizando Docker. El desafío consiste en garantizar que la aplicación se ejecute en un entorno aislado, con todas sus dependencias correctamente configuradas, y que permita a los desarrolladores realizar modificaciones al código de manera eficiente sin necesidad de reconstruir imágenes repetidamente. Esto asegura consistencia en el entorno de desarrollo, facilidad para compartir el proyecto entre equipos y portabilidad para su despliegue en diferentes sistemas. Además, se busca optimizar el flujo de trabajo para pruebas rápidas y desarrollo iterativo, minimizando errores relacionados con configuraciones locales.

Instalación de Docker:

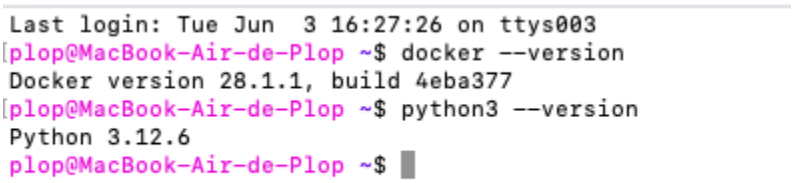
1. **Descarga e Instalación**: Descargamos e instalamos Docker en nuestra computadora desde el sitio oficial.



2. **Verificación**: Abrimos una terminal (por ejemplo, PowerShell en Visual Studio Code) y verificamos la instalación ejecutando:

docker --version

python –version



Esto confirma que Docker y Python están disponibles.

3. **Docker Hub**: Creamos una cuenta y nos registramos si queremos aprovechar al máximo las capacidades de Docker Hub, como por ejemplo trabajar con imágenes privadas o necesitamos subir nuestras propias imágenes, o si somos parte de un equipo y necesitamos colaborar de forma segura.

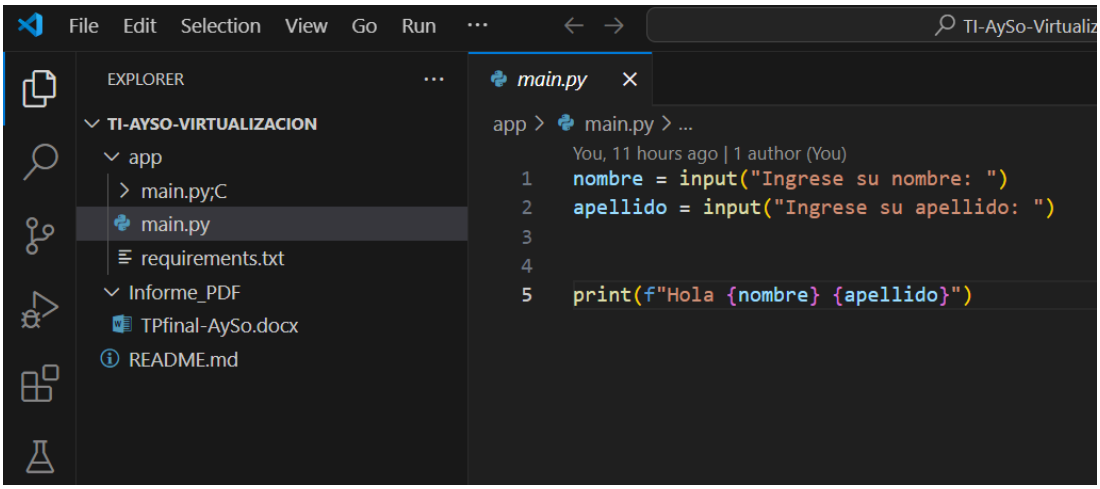
Situación:

Tenemos un programa Python (main.py) que se desea ejecutar en otro entorno, sin la necesidad de tener que configurarlo para que funcione correctamente, se elige el entorno de un contenedor docker para lograr el objetivo.

La estructura del proyecto es:

- **Carpeta principal**: TI-AYSO-VIRTUALIZACION
  - **Carpeta** app:

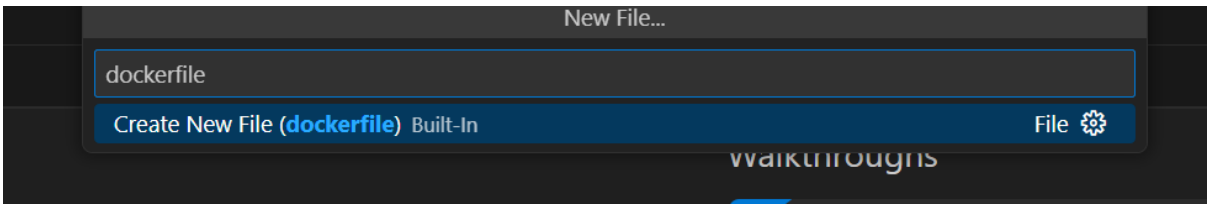
- main.py: Aplicación desarrollada.
- requirements.txt: Lista de librerías y dependencias del programa.



Paso 1: Crear una Imagen Docker

1. **Crear un Dockerfile**:

- En la carpeta app, creamos un archivo llamado Dockerfile.



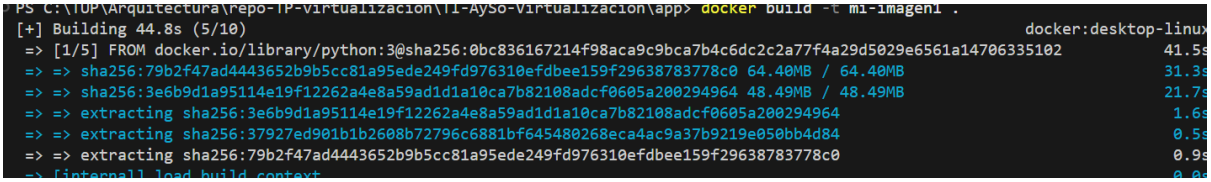
- Buscamos en Docker Hub la imagen oficial de Python y copiamos el contenido sugerido para la creación de el Dockerfile.
- Ajustamos el WORKDIR (ruta de trabajo) y el CMD (comando para ejecutar main.py).

2. **Generar la Imagen**:
   - En la terminal, nos posicionamos en la carpeta app con el comando cd
   - Ejecutamos el comando para crear la imagen:

```
docker build -t mi-imagen1 .
```

El parámetro -t asigna un nombre a la imagen

   - Se genera la imagen:



3. **Verificar la Imagen**:
   - Confirmamos que la imagen se creó correctamente con:

```
docker image ls
```

```
Deleted: sha256:10816c3097c89f625f24f9b23a54f2f1c270c4724fb0b276bd797ac166caac4c
PS C:\TUP\Arquitectura\repo-TP-virtualización\TI-AySo-Virtualizacion\app> docker image ls
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
mi-imagen1    latest    7380de74f305   6 minutes ago   1.49GB
PS C:\TUP\Arquitectura\repo-TP-virtualización\TI-AySo-Virtualizacion\app>
```

Paso 2: Crear y Ejecutar un Contenedor

1. **Crear el Contenedor**:

   ○ Usamos el comando docker run con la opción --rm para eliminar el contenedor automáticamente tras su ejecución:

```
docker run -it --rm --name mi-contenedor1 mi-imagen1
```

- --rm: Elimina el contenedor al finalizar.
- -it: Permite interacción con el contenedor.

2. **Resultado**:

El contenedor ejecuta main.py

```
You, 15 hours ago | 1 author (You)
1  nombre = input("Ingrese su nombre: ")
2  apellido = input("Ingrese su apellido: ")
3
4
5  print(f"Hola {nombre} {apellido}")        You,
```

y muestra la salida correspondiente.

```
Ingrese su nombre: marina
Ingrese su apellido: cordero
Hola marina cordero
```

Paso 3: Uso de Volúmenes para Modificaciones

Para evitar recrear imágenes tras cada cambio en el código, usamos volúmenes para vincular el código local con el contenedor.

1. **Ejecutar con Volumen**:

- Usa el parámetro -v para montar la carpeta local en el contenedor: docker run --rm -it -v $(pwd):/app mi-aplicacion-python (copiado de la imagen oficial Python en Docker Hub)
  - -v $(pwd):/app: Sincroniza la carpeta actual (pwd) con /app en el contenedor.

En la terminal power shell que estamos usando de Visual Studio Code:

```
docker run -it --rm --name mi-app1 -v "${PWD}:/usr/src/app" mi-imagen1
```

2. **Modificar y Probar**:

- Realizamos cambios en main.py localmente.

```
1  nombre = input("Ingrese su nombre: ")
2  apellido = input("Ingrese su apellido: ")
3  edad = input("Ingrese su edad: ")
4
5  print(f"Hola {nombre} {apellido} tu edad es {edad} años")
```

- Ejecutamos el comando anterior y los cambios se reflejarán automáticamente en el contenedor sin necesidad de crear otra imagen.

```
Ingrese su nombre: marina
Ingrese su apellido: cordero
Ingrese su edad: 40
Hola marina cordero tu edad es 40 años
```

Conclusión

Con este flujo de trabajo, logramos:

- Crear una imagen Docker para una aplicación Python con todas sus dependencias configuradas.
- Ejecutar contenedores de manera interactiva y desechable, garantizando un entorno limpio en cada ejecución.
- Implementar volúmenes para reflejar cambios en el código en tiempo real, optimizando el ciclo de desarrollo.
- Validar exhaustivamente el funcionamiento de la aplicación, asegurando consistencia, portabilidad y facilidad para realizar modificaciones. Este enfoque proporciona un entorno de desarrollo robusto, reproducible y eficiente, ideal para proyectos Python que requieren escalabilidad y colaboración en equipo.

Metodología Utilizada:

1. **Investigación previa (fuentes utilizadas)**

- **Documentación oficial de Docker**: Consultamos la documentación oficial de Docker para entender los conceptos básicos de contenedores, imágenes, volúmenes y la sintaxis del Dockerfile.
- **Docker Hub**: Exploramos Docker Hub para identificar la imagen oficial de Python (python:3.12-slim) y revisar ejemplos de Dockerfile para aplicaciones Python.
- **Tutoriales y guías**: Utilizamos recursos como el proporcionado en el curso, como así videos tutoriales y documentación oficial que se detallan a a continuación:

https://xentic.com.pe/virtualizacion-servidores-vmware/#:~:text=La%20virtualizaci%C3%B3n%20de%20servidores%20con,demandas%20tecnol%C3%B3gicas%20actuales%20y%20futuras.

https://docker-curriculum.com/

https://www.youtube.com/watch?v=9eTVZwMZJsA

https://tup.sied.utn.edu.ar/mod/resource/view.php?id=6657

https://www.youtube.com/watch?v=Ne-lS7m9HEw

2. **Etapas de diseño y prueba del código**

a) **Diseño del entorno**:
   - Definimos la estructura del proyecto (TI-AYSO-VIRTUALIZACION/app) con un archivo main.py para una aplicación simple y un requirements.txt para gestionar dependencias.
   - Creamos un Dockerfile basado en la imagen python:3.12-slim, optimizando para un tamaño reducido y configurando el directorio de trabajo (/app) y el comando de ejecución (CMD ["python", "main.py"]).

b) **Desarrollo del código**:
   - Implementamos una aplicación básica en main.py que devuelve un mensaje en la ruta raíz (/).
   - Generamos requirements.txt especificando versiones exactas de las librerías (en el ejemplo, por la simplicidad del programa, está vacío) para garantizar reproducibilidad.
   -

c) **Pruebas iniciales**:
   - Ejecutamos la aplicación localmente (sin Docker) para verificar su funcionalidad: python main.py.
   - Construimos la imagen Docker con docker build -t mi-aplicacion-python . y verificamos su creación con docker image ls.
   -

d) **Pruebas en contenedor**:
   - Ejecutamos el contenedor con docker run --rm -it - y confirmamos que la aplicación era accesible.
   - Probamos con volúmenes (-v $(pwd):/app) para reflejar cambios en main.py sin reconstruir la imagen, verificando que las modificaciones se reflejaban en tiempo real.

e) **Iteraciones y ajustes**:
   - Resolvimos errores iniciales, como problemas de permisos en volúmenes o dependencias faltantes, ajustando el Dockerfile y los comandos que se ejecutaron.

Herramientas y recursos utilizados:

- **IDE**: Visual Studio Code con la extensión de Docker y Python.
- **Control de versiones**: Utilizamos Git con un repositorio en GitHub para versionar el código y el Dockerfile. Comandos típicos:
- git init
- git add .
- git commit -m "Inicial: Configuración de aplicación Python con Docker"

git push origin mail

https://github.com/Marigi84/TI-AySo-Virtualizacion.git

- **Terminal**: PowerShell integrado en Visual Studio Code para ejecutar comandos de Docker (docker build, docker run, etc.).
- **Navegador**: Google Chrome.
- **Trabajo colaborativo (reparto de tareas en el grupo de trabajo)**
- **Equipo**: El proyecto fue desarrollado por un equipo de 2 personas.

Reparto de tareas:

- **Desarrollador 1 (Silvia Giardini):**
  - **Investigación y redacción del marco teórico:** Responsable de investigar y redactar las secciones relacionadas con las máquinas virtuales (VM), incluyendo conceptos, características, ventajas, desventajas y casos de uso.
  - **Diseño del entorno y Dockerfile:** Creación y configuración inicial del Dockerfile, incluyendo la selección de la imagen base (python:3.12-slim) y la definición de las instrucciones principales (FROM, WORKDIR, CMD, etc.).
  - **Pruebas iniciales:** Ejecución de pruebas locales del código Python (main.py) y construcción de la imagen Docker, verificando su correcto funcionamiento con comandos como docker build y docker image ls.
  - **Documentación de metodología y resultados:** Redacción de las secciones de metodología utilizada (etapas de diseño, pruebas, herramientas) y resultados obtenidos, incluyendo casos de prueba y errores corregidos.
  - **Coordinación de recursos visuales:** Selección y organización de las imágenes incluidas en el documento (capturas de pantalla de la terminal, estructura del proyecto, etc.) para ilustrar el caso práctico.

- **Desarrollador 2 (Marina Cordero):**
  - **Investigación y redacción del marco teórico:** Responsable de investigar y redactar las secciones relacionadas con los contenedores, incluyendo conceptos, estructura de Docker (imágenes, contenedores, volúmenes, redes, Docker Compose) y comparación con máquinas virtuales.
  - **Desarrollo del código Python:** Creación y edición del archivo main.py, asegurando que la aplicación sea funcional y cumpla con los requisitos del caso práctico.
  - **Pruebas con volúmenes:** Configuración y pruebas del uso de volúmenes (-v $(pwd):/app)

para reflejar cambios en el código en tiempo real, así como resolución de problemas relacionados con permisos o configuraciones.

- **Documentación del caso práctico:** Redacción de los pasos detallados del caso práctico (instalación de Docker, creación de la imagen, ejecución del contenedor).
- **Gestión del control de versiones:** Configuración del repositorio en GitHub, incluyendo git init, git add, git commit y git push para versionar el proyecto.

Revisión cruzada:

Cada miembro revisó el trabajo del otro para garantizar calidad y consistencia. Por ejemplo, Silvia revisó el código Python y las secciones de contenedores redactadas por Marina, mientras que Marina revisó el Dockerfile y las secciones de máquinas virtuales redactadas por Silvia. Esta revisión cruzada incluyó verificaciones de la funcionalidad del contenedor, la claridad de la documentación y la coherencia del formato del documento final.

Esta distribución asegura que ambas desarrolladoras contribuyan de manera equitativa en las áreas de investigación, desarrollo técnico, pruebas, documentación y gestión del proyecto, aprovechando sus habilidades y garantizando un trabajo colaborativo eficiente.

Conclusiones

El desarrollo de este trabajo ha sido una experiencia enriquecedora que nos permitió profundizar en los conceptos y aplicaciones prácticas de la virtualización.

A continuación, se detallan los aprendizajes, posibles mejoras y dificultades enfrentadas durante el proceso.

Qué se aprendió al hacer el trabajo:

- **Comprensión de la virtualización:** A través de la investigación y la práctica, adquirimos un entendimiento claro de las diferencias entre máquinas virtuales y contenedores, así como sus casos de uso específicos. Aprendimos cómo las máquinas virtuales ofrecen un aislamiento robusto pero consumen más recursos, mientras que los contenedores, como los gestionados por Docker, son ligeros y optimizados para el desarrollo ágil.
- **Manejo de Docker:** Dominamos el ciclo de vida de los contenedores, desde la creación de un Dockerfile hasta la construcción de imágenes y la ejecución de contenedores. Comprendimos la importancia de conceptos como volúmenes, imágenes ligeras (como python:3.12-slim).
- **Desarrollo colaborativo:** El uso de herramientas como Git, GitHub nos permitió coordinar tareas de manera eficiente, reforzando la importancia de la gestión de proyectos y la revisión cruzada para garantizar la calidad del trabajo.
- **Resolución de problemas técnicos:** Aprendimos a diagnosticar y solucionar errores comunes en Docker, como problemas de permisos en volúmenes o configuraciones incorrectas en el Dockerfile, lo que fortaleció nuestras habilidades técnicas y de depuración.

Posibles mejoras o extensiones futuras:

- **Integración de Docker Compose:** Para proyectos más complejos, podríamos incorporar Docker Compose para gestionar aplicaciones multi-contenedor, como una aplicación web con un

contenedor para el frontend, otro para el backend y uno para la base de datos.

- **Automatización de pruebas:** Implementar un flujo de integración continua (CI/CD) con herramientas como GitHub Actions para automatizar la construcción, prueba y despliegue de la imagen Docker, mejorando la eficiencia del desarrollo.
- **Optimización de imágenes:** Explorar imágenes base aún más ligeras (por ejemplo, python:3.12-alpine) y técnicas avanzadas de construcción de imágenes para reducir el tamaño y mejorar el rendimiento.
- **Despliegue en la nube:** Extender el caso práctico para desplegar el contenedor en una plataforma como AWS, Azure o Google Cloud, evaluando su comportamiento en un entorno de producción real.
- **Seguridad mejorada:** Incorporar prácticas de seguridad avanzadas, como el uso de usuarios no root en el contenedor y el escaneo de imágenes con herramientas como Trivy para detectar vulnerabilidades.

Dificultades que surgieron y cómo se resolvieron:

- **Problemas de permisos con volúmenes:** Inicialmente, enfrentamos errores al montar volúmenes con -v $(pwd):/app, ya que el comando tiene ligeras modificaciones, según la terminal que estemos usando.
- **Errores en el Dockerfile:** Durante la creación del Dockerfile, surgieron problemas por configuraciones incorrectas, como rutas de trabajo mal definidas o comandos CMD mal formateados. Estos se corrigieron consultando la documentación oficial de Docker y probando iterativamente con comandos como docker build y docker run.
- **Coordinación inicial del equipo:** La asignación inicial de tareas presentó desafíos debido a la falta de experiencia previa con Docker. Esto se superó mediante comunicación fluida vía mensajería instantánea, donde discutimos los avances, aclaramos dudas y ajustamos las actividades realizadas para un seguimiento más claro.

- **Curva de aprendizaje de Docker:** La familiarización con los comandos y conceptos de Docker (como imágenes, contenedores y volúmenes) fue inicialmente un desafío. Lo resolvimos consultando recursos adicionales, como la documentación oficial de Docker, tutoriales en video y guías prácticas recomendadas en el curso.

En conclusión, este trabajo nos permitió consolidar conocimientos teóricos y prácticos sobre la virtualización, con un enfoque particular en Docker. Logramos implementar un entorno de desarrollo robusto, portátil y eficiente para una aplicación Python, superando desafíos técnicos mediante la investigación, la colaboración y la iteración. Las lecciones aprendidas y las posibles mejoras identificadas nos preparan para abordar proyectos más complejos en el futuro, aprovechando al máximo las ventajas de la virtualización en el desarrollo de software.