



Trabajo Práctico Integrador

Virtualización con Docker

Arquitectura y Sistemas Operativos

Alumnos - Grupo

Agustín De Armas, Hugo Adrián Isaurralde.

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional

Docente Titular

Martín Aristiaran

Docente Tutor

Andrés Odiard

22 de Octubre de 2025

Tabla de Contenidos

Introducción.....	3
Objetivos.....	4
Marco Teórico.....	5
1. Virtualización y Contenedores.....	5
Virtualización Completa (Máquinas Virtuales - VMs).....	5
Virtualización a Nivel de Sistema Operativo (Contenedores).....	5
2. Docker Como Proyecto.....	6
Concepto General: Construcción de Imágenes Optimizadas.....	6
Estructura por Capas (Modelo OSI Simplificado).....	7
3. Redes Virtualizadas en Docker.....	8
Arquitectura y Zonificación de Redes Virtuales en Docker.....	8
Direccionamiento IP y DNS Interno.....	9
Comunicación Entre Contenedores.....	10
Diagrama de Red y Comunicación.....	11
4. Docker Compose y la Orquestación.....	12
Definición y Configuración de Servicios.....	12
Gestión de Dependencias y Tiempos de Arranque.....	13
Configuración de Variables de Entorno.....	13
Persistencia de Datos (Volúmenes).....	14
Caso Práctico: Configuración e Implementación.....	15
1. Instalación de Docker y Docker Compose.....	15
2. Estructura del proyecto.....	16
3. Archivo .env.....	17
4. Ejecución.....	17
5. Pruebas.....	18
6. Resultado esperado.....	18
Dificultades y Soluciones.....	19
Reflexión y Conclusiones Personales.....	20
Bibliografía.....	21
Anexo.....	21

Introducción

El presente Trabajo Práctico Integrador (TPI) aborda la **implementación y orquestación** de una arquitectura de servicios multi-contenedor utilizando la plataforma **Docker** bajo un enfoque de **virtualización ligera**. El proyecto se centra en demostrar los principios fundamentales de la **contenedorización moderna**, la **segregación de red** y la **continuidad operativa** mediante un sistema de healthcheck automatizado.

La solución implementada se articula en una **arquitectura de microservicios** compuesta por tres elementos clave, todos gestionados de forma **declarativa** a través del stack de **Docker Compose**:

1. **API Flask** (`app`): Un servicio de backend en Python que actúa como interfaz de aplicación, exponiendo el endpoint `/healthcheck/mongo` para validar de forma transaccional el estado de la conexión con la capa de datos.
2. **Base de datos MongoDB** (`mongo`): El servicio de persistencia, desplegado en un contenedor que reside exclusivamente en una **zona de red aislada** (`backend_net`). Esta configuración replica el principio de **Defensa en Profundidad**, protegiendo la capa de datos de accesos directos desde el host o la red pública.
3. **Cliente Tester** (`cliente`): Un contenedor liviano dedicado a la ejecución de **monitoreo sintético**, enviando peticiones HTTP periódicas a la API.

Este diseño arquitectónico facilita un despliegue **ágil y portable**, contrastando con el alto consumo de recursos de la virtualización tradicional basada en Máquinas Virtuales (VM). Además, establece un modelo robusto de **zonificación de red** donde la comunicación interna está estrictamente controlada mediante **redes bridge**.

Objetivos

Los objetivos de este TPI están orientados a la demostración de competencias técnicas en el ámbito de la virtualización de infraestructura, la orquestación de servicios y la gestión de la continuidad operativa:

- **Implementación de Infraestructura Contenerizada:** Desplegar un entorno virtualizado, altamente **reproducible y portable**, utilizando el motor Docker y la herramienta de orquestación **Docker Compose**.
- **Diseño de Arquitectura de Red Zonificada:** Configurar una **topología de red multi-interfaz** mediante **bridge networks** (**public_net** y **backend_net**) para lograr la **segregación de tráfico** y aislar la capa de persistencia (MongoDB) del exterior.
- **Integración de Aplicaciones Desacopladas:** Establecer una comunicación segura y funcional entre el servicio **API Flask** y la **Base de Datos MongoDB**, aprovechando el mecanismo de **resolución DNS interno** de Docker Compose (ej. uso del hostname **mongo**).
- **Validación de Continuidad Operativa (Monitoreo Sintético):** Implementar un mecanismo de **monitoreo automatizado** (**cliente**) para verificar la disponibilidad en tiempo real y el estado funcional de la cadena completa de servicios (Cliente > API > DB).
- **Análisis Comparativo:** Evaluar y documentar las ventajas operativas, de rendimiento que ofrece la **contenedorización** frente a la virtualización completa tradicional.

Marco Teórico

1. Virtualización y Contenedores

La **virtualización** consiste en la abstracción de los recursos físicos del *hardware* para permitir la ejecución simultánea de múltiples entornos aislados sobre un mismo sistema anfitrión (*host*). Históricamente, se diferencia en dos paradigmas:

Virtualización Completa (Máquinas Virtuales - VMs)

En este modelo, el **hipervisor** (ej. VMware, VirtualBox) emula el hardware de la máquina física. Cada **Máquina Virtual (VM)** debe instalar un **Sistema Operativo (SO) completo** e independiente (SO Invitado) que incluye su propio *kernel*.

- **Ventajas:** Proporciona un **aislamiento robusto**, ya que el SO Invitado es totalmente independiente del SO Anfitrión.
- **Desventajas:** Genera un **gran footprint** o huella de recursos (RAM, CPU, Disco) debido a la redundancia de tener múltiples *kernels* operativos.

Virtualización a Nivel de Sistema Operativo (Contenedores)

El enfoque de **contenedorización** (como Docker) es un tipo de virtualización **ligera** o de SO, donde todos los entornos aislados **comparten el *kernel*** del sistema operativo host.

Los contenedores logran este aislamiento sin el overhead de un *kernel* propio, basándose en dos primitivas esenciales del sistema operativo Linux:

1. **Namespaces (Espacios de Nombres):** Proveen el **aislamiento** de la vista del contenedor. Un *Namespace* asegura que el contenedor solo pueda ver sus propios procesos, red, sistema de archivos, hostname y usuarios. Es decir, restringe lo que el contenedor puede ver del sistema.

2. **Control Groups (cgroups):** Proveen la **limitación y el control** de los recursos. Un *cgroup* impone límites a los recursos de hardware que el contenedor puede consumir (CPU, RAM, E/S de disco), evitando que un contenedor acapare recursos y afecte a otros.

Esta arquitectura resulta en un **mínimo footprint**, **tiempos de arranque y despliegue casi instantáneos** y una **portabilidad** superior, lo que hace de los contenedores la tecnología preferida para las arquitecturas de microservicios modernas.

2. Docker Como Proyecto

Docker es una plataforma de **virtualización ligera** que permite empaquetar aplicaciones y sus dependencias en imágenes portables. A diferencia de la virtualización completa (VM), los contenedores comparten el *kernel* del sistema anfitrión, lo que resulta en un **bajo overhead** de recursos, arranque instantáneo y alta portabilidad.

Cada contenedor ejecuta un proceso principal aislado dentro de su propio entorno, con un sistema de archivos, red y variables independientes.

Concepto General: Construcción de Imágenes Optimizadas

La imagen del servicio `app` (API Flask) se construye a partir de un `Dockerfile`. El diseño de este archivo es crucial para la **eficiencia operativa** y la **optimización del footprint**:

- **Optimización del Tamaño (Footprint):** La imagen base utilizada es `FROM python:3.10-slim`. La etiqueta `slim` asegura que la imagen de Python contenga solo los paquetes mínimos necesarios para ejecutar la aplicación, lo que reduce drásticamente el tamaño final de la imagen y, consecuentemente, los tiempos de pull y despliegue.

- **Aprovechamiento del *Layer Caching*:** El `Dockerfile` está estructurado para maximizar el caché de capas:
 - Primero, se copian e instalan las dependencias (`COPY requirements.txt .` y `RUN pip install...`).
 - Luego, se copia el código de la aplicación (`COPY . .`).
 - Si solo se modifica el código de la aplicación (los archivos `.py`), Docker **reutilizará la capa de dependencias en caché**, sin necesidad de reinstalarlas, acelerando la reconstrucción.
- **Gestión de Logs:** La directiva `ENV PYTHONUNBUFFERED=1` desactiva el buffering de la salida estándar de Python, garantizando que los logs generados por la API se transmitan en **tiempo real** (sin latencia) al log driver de Docker.
- **Entrada de Contenedor (*Entrypoint*):** El comando `CMD ["python", "run.py"]` define el proceso que se ejecutará como PID 1 dentro del contenedor, iniciando la aplicación Flask.

Estructura por Capas (Modelo OSI Simplificado)

En una implementación tradicional con VirtualBox, la virtualización de red se logra creando adaptadores virtuales que asignan interfaces de red virtuales a cada máquina invitada. En Docker, el principio es el mismo, pero se gestiona a nivel de **Capa 2 (Enlace)** y **Capa 3 (Red)** dentro del motor de red de Docker. Cada contenedor actúa como un **host virtual** con su propia pila TCP/IP y una interfaz de red virtual.

Capa	Docker equivalente	Descripción
7– Aplicación	API Flask, MongoDB, Tester	Procesan la lógica de negocio y los datos.
4 – Transporte	TCP / UDP	Comunicación entre servicios (ej., TCP:5000 para Flask, TCP:27017 para Mongo).
3 – Red	Redes virtuales (bridge, overlay)	Asignan subredes, direccionamiento IP, y tablas de ruteo internas.
2 – Enlace	Interfaces y bridge docker0	Enlaces virtuales entre contenedores y el host.
1 – Física	Interfaz de red del host	Conexión real del sistema operativo base.

3. Redes Virtualizadas en Docker

Arquitectura y Zonificación de Redes Virtuales en Docker

El entorno Docker, en este proyecto, utiliza dos redes definidas en el `docker-compose.yml`, equivalentes a **zonas de seguridad** en una arquitectura de red tradicional, en las cuales se componen **tres contenedores** interconectados entre sí:

Contenedor	Rol	Red	Puerto interno	Puerto externo (host)
app	API Flask (core + healthcheck)	backend_net public_net	5000	5000
mongo	Base de datos MongoDB	backend_net	27017	No expuesto
cliente	Cliente de prueba (simula usuario)	public_net	—	—

- `public_net`: Esta red externa representa la “zona pública” o DMZ. API (`app` servicio) está **multihomed** (conectada a dos redes) , lo que le permite recibir tráfico del **Host local** (vía mapeo de puertos `5000:5000`) y del contenedor `cliente`.
- `backend_net`: Esta red privada es un ejemplo de **Defensa en Profundidad**, ya que aísla la capa de datos (`mongo` no expone puertos al host). Solo el servicio `app` (API) está conectado, actuando como un proxy o interfaz de acceso a la DB.

Cada red es de tipo `bridge`, lo que significa que Docker crea un switch virtual con un rango de IPs privado (por defecto `172.x.x.x`). Los contenedores conectados a una misma red pueden comunicarse usando resolución DNS interna.

Direccionamiento IP y DNS Interno

Al iniciar los servicios, Docker asigna **IPs internas automáticas** a cada contenedor dentro de su red.

Ejemplo:

```
app    → 172.18.0.3 (public_net y backend_net)
mongo  → 172.18.0.2 (solo backend_net)
cliente → 172.19.0.3 (solo public_net)
```

Docker Compose utiliza el nombre del servicio (ej. `mongo`, `app`) como **hostname** para la resolución DNS interna. Esto es crucial para la conexión.

Ejemplo: `http://mongo:27017` o `http://app:5000`.

Esto elimina la necesidad de direcciones IP estáticas o manuales: Docker administra el ruteo y resolución automáticamente.

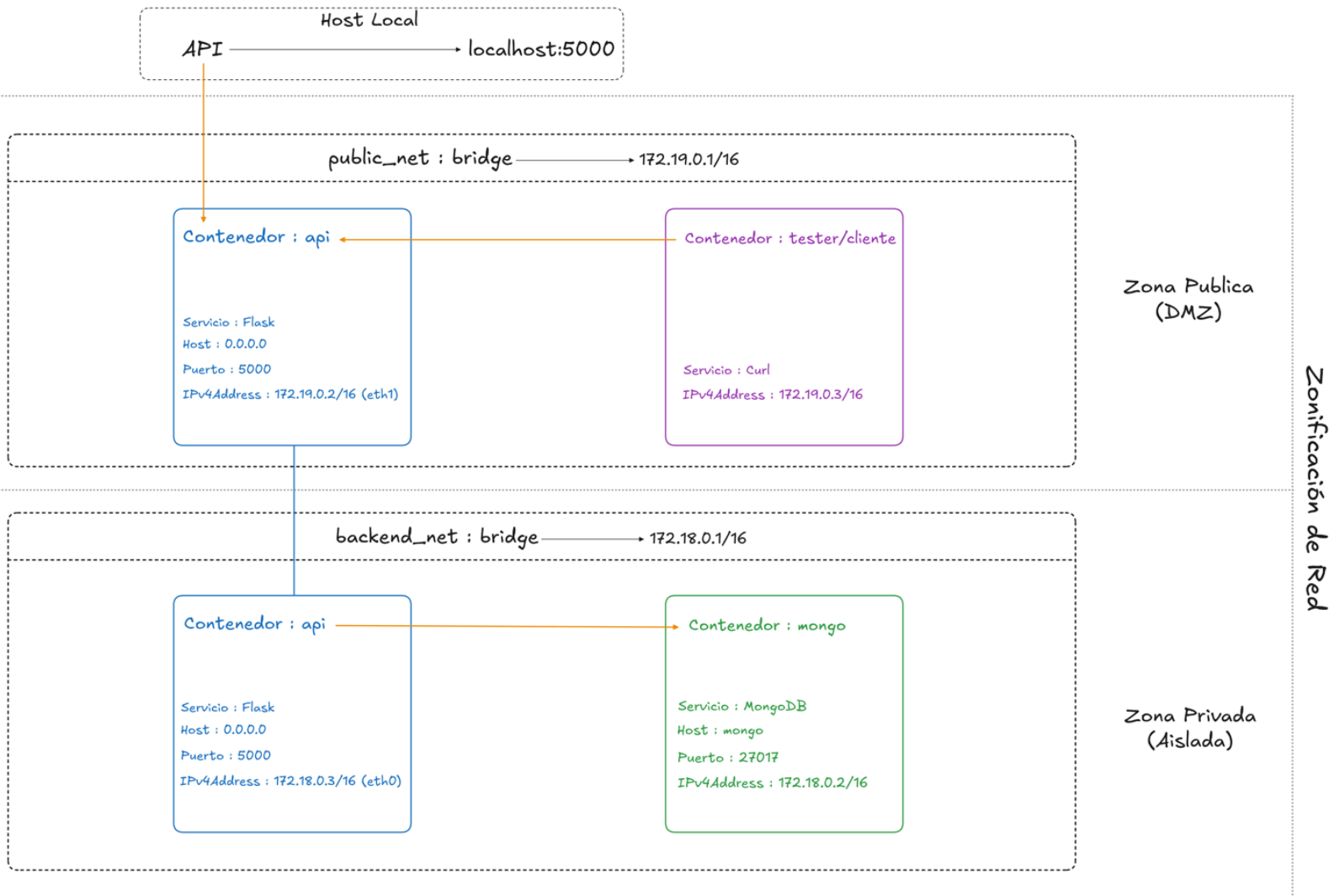
```
1  >Run All Services
1  services:
2  >Run Service
2  mongo:
3  networks:
4  - backend_net
5
6  >Run Service
6  api:
7  networks:
8  - backend_net
9  - public_net
10
11 >Run Service
11 cliente:
12 networks:
13 - public_net
14
15 networks:
16 backend_net:
17 driver: bridge
18 public_net:
19 driver: bridge
20
```

Comunicación Entre Contenedores

1. El contenedor `api` levanta Flask en el puerto `5000` (`APP_PORT=5000` del `.env`).
2. El servicio core de Flask establece conexión con MongoDB usando los valores: `mongodb://mongo:27017/mongo_db_default`
(El hostname `mongo` se resuelve dentro de `backend_net` gracias a Docker DNS.)
3. La ruta `/healthcheck/mongo` prueba la conexión y devuelve un JSON con estado.
4. El contenedor `cliente` envía un curl a `http://api:5000/healthcheck/mongo` por `public_net`.

5. El host local también puede acceder al mismo endpoint desde `localhost:5000`.

Diagrama de Red y Comunicación



- **MongoDB** no expone puertos al host, sólo es accesible por la API en red interna (`backend_net`).
- **API Flask** es el único servicio expuesto externamente (puerto `5000`).
- **Cliente** representa tráfico de usuario o cliente externo dentro de la red pública.
- Esta separación garantiza el aislamiento del servicio de datos, replicando el principio de defensa en profundidad en entornos virtualizados.

4. Docker Compose y la Orquestación

Docker Compose es una herramienta esencial para definir y ejecutar aplicaciones Docker multi-contenedor. En lugar de gestionar contenedores, redes, y volúmenes de forma individual a través de comandos `docker` separados, Compose utiliza un archivo de configuración declarativo, `docker-compose.yml`, para orquestar todo el entorno como una unidad.

El uso de Compose en este proyecto ejemplifica una arquitectura de microservicios básica, garantizando que todos los componentes (API, DB, Cliente) se desplieguen, se comuniquen y se gestionen de manera eficiente.

Definición y Configuración de Servicios

El archivo `docker-compose.yml` define tres servicios principales, cada uno con un propósito específico y su propia configuración de red y dependencia:

Servicio	Función	Base de Imagen / Construcción
<code>mongo</code>	Persistencia de datos (Capa de Datos)	Imagen oficial <code>mongo:latest</code>
<code>app</code>	Lógica de negocio (API Flask)	Construcción a partir de <code>Dockerfile</code>
<code>cliente</code>	Monitoreo sintético (Cliente curl)	Imagen <code>curlimages/curl:8.6.0</code>

Gestión de Dependencias y Tiempos de Arranque

La orquestación efectiva requiere asegurar que los servicios dependientes se inicien en el orden correcto.

El servicio `app` (API Flask) requiere que el servicio `mongo` (MongoDB) esté activo para poder establecer una conexión. Esta dependencia se gestiona mediante la directiva `depends_on`:

```
app:
  # ...
  depends_on:
    - mongo      # Asegurar MongoDB inicie antes que la app
```

Aunque `depends_on` asegura el orden de inicio de los contenedores, no garantiza que el servicio interno de MongoDB esté completamente operacional y listo para aceptar conexiones. Sin embargo, para este proyecto, el healthcheck periódico del contenedor `cliente` y la lógica de retry implícita en la API suelen ser suficientes para manejar la eventual ventana de tiempo donde la DB aún está inicializándose.

Configuración de Variables de Entorno

Docker Compose gestiona de forma centralizada la configuración inyectando variables de entorno en los contenedores. Para el servicio `app`, se utiliza un mecanismo dual:

1. **Archivo `.env` (Directiva `env_file`):** La línea `env_file: - ../.env` en el servicio `app` inyecta todas las variables definidas en el archivo `.env`. Esto incluye claves como `APP_SECRET_KEY` o variables de configuración de la DB.
2. **Bloque `environment`:** Este bloque permite sobrescribir variables de entorno o inyectar aquellas que son críticas y se resuelven dinámicamente, como la conexión de la DB, que utiliza valores ya cargados del `.env`:

```
environment:
  DB_DEFAULT_HOST: ${DB_DEFAULT_HOST}
  DB_DEFAULT_USER: ${DB_DEFAULT_USER}
  # ...
```

Para el servicio `mongo`, las variables son inyectadas directamente a través del bloque `environment` para inicializar el administrador y la base de datos por defecto (ej. `MONGO_INITDB_ROOT_USERNAME`, `MONGO_INITDB_DATABASE`).

Persistencia de Datos (Volúmenes)

Para asegurar que los datos de MongoDB persistan a pesar de que el contenedor sea eliminado o reiniciado, Docker Compose define un volumen persistente llamado `mongo-data`:

```
volumes:
  mongo-data:  # Define el volumen persistente para la DB
  # ...
  mongo:
  # ...
  Volumes:
    - mongo-data:/data/db  # Mapeo del volumen persistente
```

Esto mapea el volumen `mongo-data` del host a la ruta `/data/db` dentro del contenedor, garantizando que el estado de la base de datos se mantenga intacto entre despliegues.

Caso Práctico: Configuración e Implementación

1. Instalación de Docker y Docker Compose

Para el desarrollo y ejecución del entorno virtualizado se utilizó **Docker Desktop** sobre **Windows 10/11**, empleando **Git Bash** como terminal principal para la ejecución de comandos.

- **Descarga e instalación de Docker Desktop:**

Desde el sitio oficial:

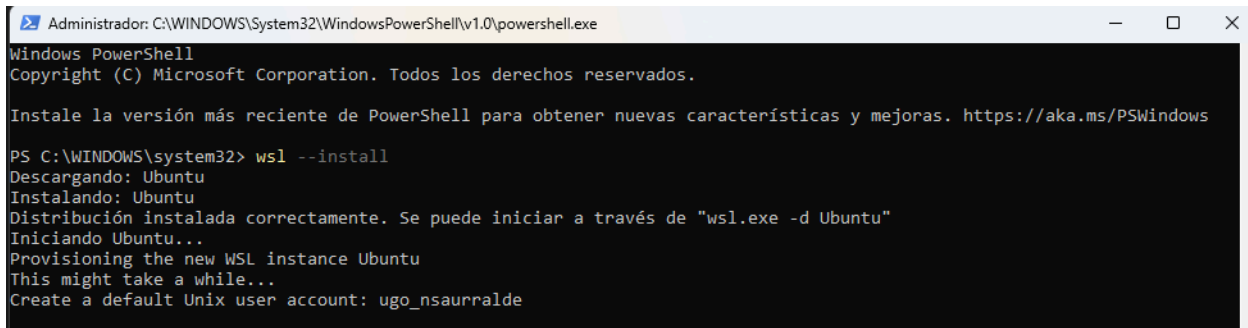
<https://www.docker.com/products/docker-desktop>

Se ejecutó el instalador con las opciones por defecto.

- **Activación de WSL 2 (Subsistema de Windows para Linux):**

Docker Desktop requiere WSL 2 como backend.

En caso de no estar habilitado, se configura mediante PowerShell con permisos de administrador, y luego se reinicia el sistema.



```
Administrador: C:\WINDOWS\System32\WindowsPowerShell\v1.0\powershell.exe
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Instale la versión más reciente de PowerShell para obtener nuevas características y mejoras. https://aka.ms/PSWindows

PS C:\WINDOWS\system32> wsl --install
Descargando: Ubuntu
Instalando: Ubuntu
Distribución instalada correctamente. Se puede iniciar a través de "wsl.exe -d Ubuntu"
Iniciando Ubuntu...
Provisioning the new WSL instance Ubuntu
This might take a while...
Create a default Unix user account: ugo_nsaurralde
```

- **Verificación de instalación:**

Una vez iniciado Docker Desktop, se comprobó el correcto funcionamiento desde Git Bash (Cabe destacar que **Docker Compose** ya viene integrado dentro de Docker Desktop, por lo cual no requiere instalación adicional).

```
docker --version
docker compose version
```

- **Uso de Git Bash:**

Todos los comandos de construcción, ejecución y administración de contenedores se realizaron desde **Git Bash**, aprovechando su compatibilidad con comandos de shell de Linux.

```
docker compose up --build
docker ps
docker logs flask_api
```

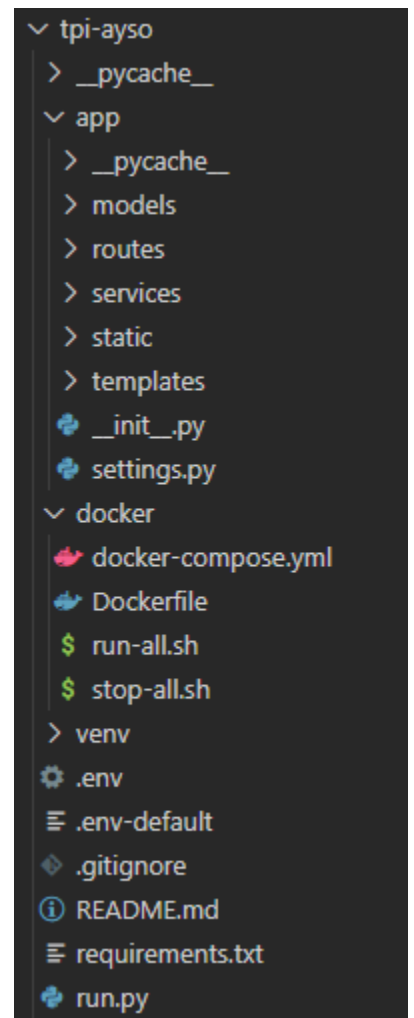
2. Estructura del proyecto

El proyecto se estructura bajo una arquitectura modular que permite separar el código de aplicación de la infraestructura virtualizada.

Dentro de la carpeta `app/` se organiza la lógica de negocio, rutas y servicios. La aplicación se apoya en un servicio interno de conexión a MongoDB implementado como un singleton, lo que garantiza una única instancia de cliente en memoria.

En la carpeta `docker/` se agrupan todos los archivos vinculados al proceso de virtualización y orquestación de contenedores, incluyendo:

- `Dockerfile`: define la imagen base del contenedor Flask.
- `docker-compose.yml`: describe la topología de red, la base de datos MongoDB y el contenedor de pruebas.
- `run-all.sh` y `stop-all.sh`: scripts para automatizar el despliegue y finalización del entorno completo.



Esta estructura facilita la portabilidad del sistema y permite ejecutar la aplicación completa con un solo comando, manteniendo aisladas las dependencias y asegurando la reproducibilidad del entorno.

3. Archivo .env

Las variables de entorno definen parámetros reutilizables y sensibles fuera del código fuente:

```
APP_DEBUG=True
APP_HOST=0.0.0.0
APP_PORT=5000
APP_SECRET_KEY=defsupersecret

DB_DEFAULT_HOST=mongo
DB_DEFAULT_PORT=27017
DB_DEFAULT_NAME=mongo_db_default
DB_DEFAULT_USER=mongo_admin
DB_DEFAULT_PASSWORD=password
```

Este enfoque desacopla la configuración del entorno, lo que permite cambiar credenciales o parámetros sin modificar la aplicación.

4. Ejecución

Con todos los archivos configurados, el entorno completo se levanta con un solo comando:

```
docker compose up --build
```

Este comando compila la imagen de la API Flask a partir del `Dockerfile`, crea las redes internas (`public_net` y `backend_net`), inicializa MongoDB con las credenciales especificadas y lanza el contenedor `cliente`.

5. Pruebas

El contenedor `cliente` ejecuta periódicamente una prueba de conexión hacia la API mediante `curl`:

```
curl http://app:5000/healthcheck/mongo 2>&1;
```

(`2>&1` redirige la salida de error estándar (stderr, descriptor 2) a la salida estándar (stdout, descriptor 1))

Salida esperada:

```
{
  "status": 200,
  "message": "MongoDB connection is successful (default alias)",
  "db_name": "mongo_db_default",
  "collections_count": 3
}
```

En caso de fallo de conexión con la base de datos, el cliente mostrará:

```
{
  "status": 503,
  "message": "MongoDB connection FAILED for alias 'default'"
}
```

6. Resultado esperado

La arquitectura desplegada demuestra los siguientes conceptos:

- **Aislamiento por red:** el contenedor de MongoDB no expone puertos al exterior y sólo es accesible internamente desde la API.
- **Orquestación automatizada:** un solo comando (`docker compose up`) inicializa todos los componentes y sus dependencias.
- **Escalabilidad:** la API puede escalar horizontalmente (réplicas) sin modificar la configuración de red.
- **Monitoreo sintético:** el contenedor cliente permite validar la disponibilidad y el estado de la base de datos en tiempo real.

Dificultades y Soluciones

Dificultad	Solución
Error al construir la imagen por falta de dependencias	Se creó el archivo requirements.txt con las librerías necesarias (Flask, pymongo).
Volumen no persistente	Se definió un volumen db_data en el docker-compose.yml.

```
Iniciando pruebas periodicas de API...
Ejecutando healthcheck:
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed

  0     0     0     0     0     0     0     0  --:--:-- --:--:-- --:--:--    0
  0     0     0     0     0     0     0     0  --:--:-- --:--:-- --:--:--    0
  0     0     0     0     0     0     0     0  --:--:--  0:00:01 --:--:--    0
  0     0     0     0     0     0     0     0  --:--:--  0:00:02 --:--:--  0curl: (6) Could not resolve host: app
```

El contenedor `cliente`, el cual consulta a la aplicación por el estado de la conexión, no podía resolver el host del contenedor de esta misma.

Este fallo se daba, debido a que cliente estaba operando en una red default por fuera de la zonificación, ya que no se había definido correctamente su red en el archivo `docker-compose.yml`

Reflexión y Conclusiones Personales

La realización de este proyecto integrador no sólo validó los objetivos planteados, sino que proporcionó una comprensión práctica y profunda de los paradigmas de la **infraestructura moderna** y la **orquestación de servicios**.

Este trabajo permitió contrastar de forma tangible la **eficiencia operativa** de la **contenedorización** frente a la virtualización tradicional. El uso de Docker **simplificó drásticamente el despliegue** y redujo los tiempos de configuración, demostrando la capacidad de crear un **entorno reproducible y portable** con un **mínimo footprint**.

Los puntos técnicos más relevantes y de mayor aprendizaje fueron:

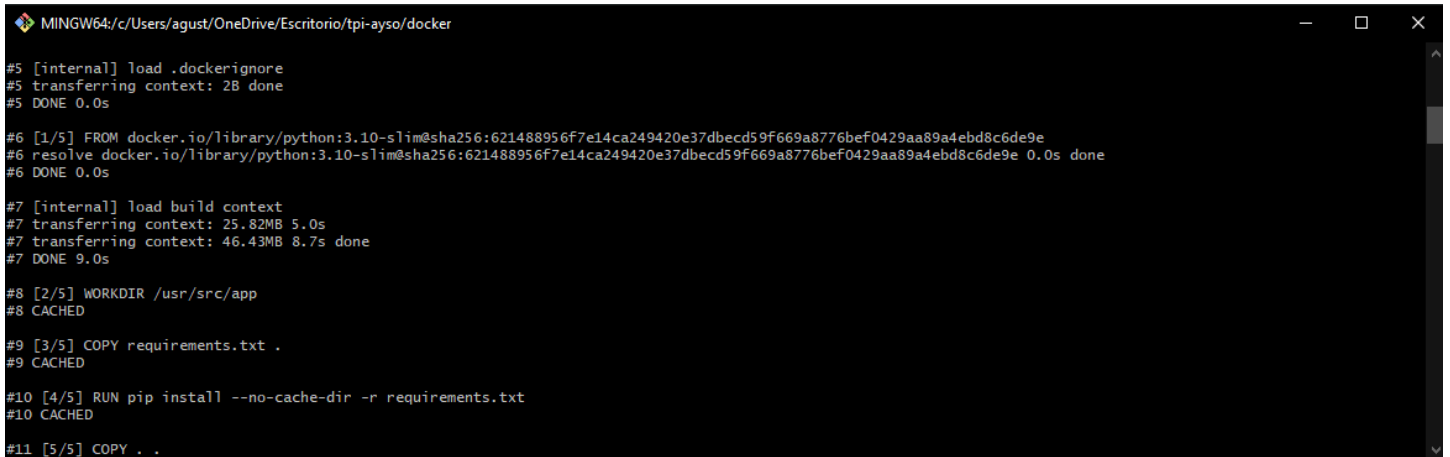
- **Aislamiento y Zonificación:** La implementación de redes bridge definidas por el usuario (`public_net` y `backend_net`) fue fundamental. Logramos con éxito la segregación de la capa de datos, aislando el servicio `mongo` en una red privada y obligando a la API (`app`) a actuar como un gateway **multi-homed**.
- **Descubrimiento de Servicio:** Se confirmó la eficacia de Docker Compose para manejar la comunicación interna. La conexión entre `app` y `mongo` se basó únicamente en el **nombre del servicio** como hostname (`mongodb://mongo:27017`), eliminando la necesidad de gestionar direcciones IP estáticas.
- **Continuidad Operativa:** La inclusión del contenedor `cliente` demostró la importancia del monitoreo sintético para validar no solo la disponibilidad del endpoint de la API, sino también la integridad transaccional de la conexión end-to-end (API > DB).

Bibliografia

- Documentación oficial de Docker: <https://docs.docker.com/>
 - Documentacion Flask: <https://flask.palletsprojects.com/>
 - Documentacion MongoDB: <https://www.mongodb.com/docs/>
-

Anexo

- Google Drive: [TPI-AySO Drive](#)
- GitHub: [TPI-AySO GitHub](#)
- Youtube: [TPI-AySO Youtube](#)



```
MINGW64:/c:/Users/agust/OneDrive/Escritorio/tpi-ayso/docker

#5 [internal] load .dockerignore
#5 transferring context: 28 done
#5 DONE 0.0s

#6 [1/5] FROM docker.io/library/python:3.10-slim@sha256:621488956f7e14ca249420e37dbecd59f669a8776bef0429aa89a4ebd8c6de9e
#6 resolve docker.io/library/python:3.10-slim@sha256:621488956f7e14ca249420e37dbecd59f669a8776bef0429aa89a4ebd8c6de9e 0.0s done
#6 DONE 0.0s

#7 [internal] load build context
#7 transferring context: 25.82MB 5.0s
#7 transferring context: 46.43MB 8.7s done
#7 DONE 9.0s

#8 [2/5] WORKDIR /usr/src/app
#8 CACHED

#9 [3/5] COPY requirements.txt .
#9 CACHED

#10 [4/5] RUN pip install --no-cache-dir -r requirements.txt
#10 CACHED

#11 [5/5] COPY . .
```

Containers [Give feedback](#)

Container CPU usage





















0.87% / 1200% (12 CPUs available)

Container memory usage


262.2MB / 7.53GB

Show charts


Only show running containers


<input type="checkbox"/>	Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
<input type="checkbox"/>	 docker	-	-	-	0.54%	52 seconds ago	   
<input type="checkbox"/>	 mongo	776ed6c314c3	mongo:latest	27017:27017	0.35%	53 seconds ago	   
<input type="checkbox"/>	 app	016a3906c868	docker-app	5000:5000	0.19%	53 seconds ago	   
<input type="checkbox"/>	 cliente	e79f3268cf7a	curlimages/curl:8.6.0		0%	52 seconds ago	   

<



cliente

 e79f3268cf7a

 [curlimages/curl:8.6.0](#)

Logs

Inspect

Bind mounts

Exec

Files

Stats

Iniciando pruebas periodicas de API...

Ejecutando healthcheck:

% Total	% Received	% Xferd	Average Speed	Time	Time	Time	Current
			Dload	Upload	Total	Spent	Left
<pre> "collections_count": 0, "db_name": "mongo_db_default", "message": "MongoDB connection is successful (default alias)", "status": 200 </pre>							