

# Proyecto de

## Containerization And Container Orchestration

### (Django REST + Angular)

#### 1. Repositorio del proyecto

- ✓ [https://github.com/ajahuanca/containerizacion\\_y\\_orquestacion](https://github.com/ajahuanca/containerizacion_y_orquestacion)
- ✓ [https://github.com/ajahuanca/containerizacion\\_y\\_orquestacion.git](https://github.com/ajahuanca/containerizacion_y_orquestacion.git)

Este repositorio contiene toda la configuración, código fuente y documentación técnica necesaria para construir, ejecutar y desplegar la aplicación completa, desde el entorno local hasta un clúster Kubernetes.

#### 2. Descripción general

Este proyecto implementa una aplicación web completa para el registro y gestión de usuarios, desarrollada como parte de un ejercicio práctico de containerización y orquestación.

La aplicación está diseñada para demostrar la integración entre Django REST Framework (DRF) y Angular v18, complementada con una base de datos PostgreSQL y un entorno completamente dockerizado preparado para despliegue tanto local como en clústeres Docker Swarm y Kubernetes (KIND).

#### 3. Propósito del proyecto

El objetivo principal es aplicar los conocimientos adquiridos sobre contenedores, redes, volúmenes, servicios, orquestación y despliegue en clústeres, mediante la construcción de una aplicación modular, escalable y portable.

Para este fin práctico, el sistema implementa los modelos de:

- **Usuario**, con funcionalidades básicas de registro, listado, actualización y eliminación.

- **Tipo de Documento (o Documento)**, que permite asociar tipos o categorías de documento a cada usuario.

Gestión de Usuarios y Documentos

Usuarios Documentos

Nuevo Documento

Tipo

Lista de Documentos

ID	Tipo	Estado		
4	libreta	Activo	<input checked="" type="checkbox"/> Editar	<input type="checkbox"/> Eliminar
3	Otro	Activo	<input checked="" type="checkbox"/> Editar	<input type="checkbox"/> Eliminar
2	Visa	Activo	<input checked="" type="checkbox"/> Editar	<input type="checkbox"/> Eliminar
1	Carnet	Activo	<input checked="" type="checkbox"/> Editar	<input type="checkbox"/> Eliminar

*Figura 1. Formulario de Tipo de Documentos*

Gestión de Usuarios y Documentos

Usuarios Documentos

Nuevo Usuario

Numero de CI

Nombre completo

correo electronico

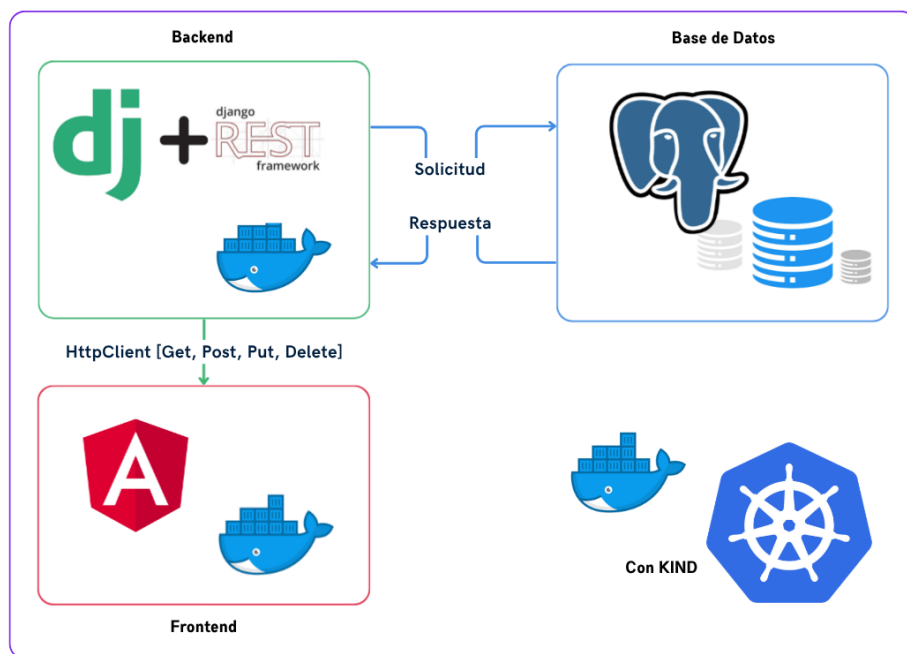
Lista de Usuarios

ID	Numero CI	Documento	Nombre Completo	Email	Estado		
2	1131313	Visa	pepito luis	luis@mail.com	Activo	<input checked="" type="checkbox"/> Editar	<input type="checkbox"/> Eliminar
1	1214646	Carnet	juan perez	juan@mail.com	Activo	<input checked="" type="checkbox"/> Editar	<input type="checkbox"/> Eliminar

*Figura 2. Formulario de Usuarios*

Estos modelos exponen más de 6 endpoints REST funcionales, abarcando operaciones CRUD completas (Create, Read, Update, Delete) y demostrando el flujo completo de interacción entre frontend y backend.

#### 4. Arquitectura base



*Figura 3. Diagrama de la arquitectura monolítica*

La solución está compuesta por los siguientes módulos principales:

Módulo	Descripción
<b>Backend (Django REST Framework + drf-spectacular)</b>	API REST desarrollada en Python/Django. Incluye endpoints para gestión de usuarios, documentos y autenticación básica. Documentación automática generada con Swagger y Redoc.
<b>Frontend (Angular v18 + Bootstrap + Bootstrap Icons)</b>	Interfaz moderna y responsiva. Contiene vistas en pestañas (tabs) para Usuarios y Documentos, formularios con validaciones y consumo de la API mediante HttpClient.
<b>Base de Datos (PostgreSQL)</b>	Motor de base de datos relacional que almacena la información de usuarios y documentos.

<b>Adminer para gestión BD</b>	Herramienta ligera de administración de base de datos incluida solo para entornos de prueba y desarrollo.
<b>Infraestructura (Docker Compose, Swarm y Kubernetes)</b>	Permite la ejecución y orquestación de todos los servicios en contenedores, con configuración de redes, volúmenes, variables de entorno y balanceadores de carga.

## 5. Componentes Clave y Funcionalidad

El proyecto ofrece:

- API REST estructurada en endpoints de Usuario y Documento (más de 6 endpoints disponibles).
- Documentación interactiva de API con drf-spectacular (/api/docs/).
- Frontend Angular con pestañas (Tabs) para la gestión independiente de cada módulo.
- Estilos y componentes visuales basados en Bootstrap 5 y Bootstrap Icons.
- Despliegue completo mediante Docker Compose (local), Docker Swarm (replicado) y Kubernetes KIND (clúster).

**API Registro de usuario**
1.0.0
OAS 3.0

[/api/schema/](#)  
 API REST con Django DRF y drf-spectacular

**Tipo de documento**
^

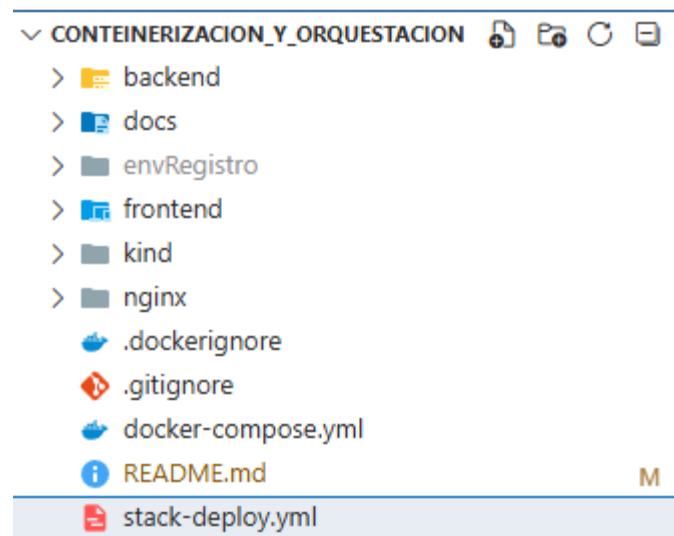
GET	/api/usuario/documentos/	CRUD de Documentos	🔒	▼
POST	/api/usuario/documentos/	CRUD de Documentos	🔒	▼
GET	/api/usuario/documentos/{id}/	CRUD de Documentos	🔒	▼
PUT	/api/usuario/documentos/{id}/	CRUD de Documentos	🔒	▼
PATCH	/api/usuario/documentos/{id}/	CRUD de Documentos	🔒	▼
DELETE	/api/usuario/documentos/{id}/	CRUD de Documentos	🔒	▼

*Figura 4. EndPoints para Tipo de Documento*

Usuarios			^
GET	/api/usuario/usuarios/	CRUD de Usuarios	🔒 ▼
POST	/api/usuario/usuarios/	CRUD de Usuarios	🔒 ▼
GET	/api/usuario/usuarios/{id}/	CRUD de Usuarios	🔒 ▼
PUT	/api/usuario/usuarios/{id}/	CRUD de Usuarios	📄 🔒 ▼
PATCH	/api/usuario/usuarios/{id}/	CRUD de Usuarios	🔒 ▼
DELETE	/api/usuario/usuarios/{id}/	CRUD de Usuarios	🔒 ▼

*Figura 5. EndPoints para Usuarios*

## 6. Estructura base del proyecto



*Figura 6. Estructura base del proyecto*

### Requisitos previos

- Docker ( $\geq 20$ )
- docker-compose (v2)
- docker swarm (para Swarm)
- kind (para cluster Kubernetes local)
- kubectl
- MetalLB y ingress-nginx para KIND

## 7. Pasos a seguir para dockerizar

### a. Desarrollo local (Docker Compose) Containerización

Copiar .env con variables sensibles para el Backend

```
POSTGRES_DB=appdb
POSTGRES_USER=appuser
POSTGRES_PASSWORD=apppassword
POSTGRES_HOST=db
POSTGRES_PORT=5432
DJANGO_SECRET_KEY=dev-secret
DEBUG=1
```

Levantar docker compose

***docker-compose up --build***

```
pws@contenerizacion_y_orquestacion: ~$ docker-compose up --build
time="2025-10-25T13:12:56-04:00" level=warning msg="D:\\MAESTRIAS\\ING. SOFTWARE AVANZADO\\9. CONTAINERIZATION AND CONTAINER
bsolete, it will be ignored, please remove it to avoid potential confusion"
[+] Building 91.4s (40/40) FINISHED
=> [internal] load local bake definitions 0.0s
=> reading from stdin 1.45kB 0.0s
=> [frontend internal] load build definition from Dockerfile 0.0s
=> transferring dockerfile: 452B 0.0s
=> [backend internal] load build definition from Dockerfile 0.1s
=> transferring dockerfile: 926B 0.0s
=> [frontend internal] load metadata for docker.io/library/node:25-alpine3.21 1.9s
=> [frontend internal] load metadata for docker.io/library/nginx:alpine 1.8s
=> [backend internal] load metadata for docker.io/library/python:3.12-alpine 1.8s
=> [auth] library/node:pull token for registry-1.docker.io 0.0s
=> [auth] library/python:pull token for registry-1.docker.io 0.0s
=> [auth] library/nginx:pull token for registry-1.docker.io 0.0s
=> [frontend internal] load .dockerignore 0.0s
=> transferring context: 242B 0.0s
=> [backend internal] load build context 0.0s
=> transferring context: 26.6kB 1.7s
=> [backend build 1/6] FROM docker.io/library/python:3.12-alpine@sha256:d82291d418d5c47f267708393e40599ae836f226 0.0s
=> resolve docker.io/library/python:3.12-alpine@sha256:d82291d418d5c47f267708393e40599ae836f2260b0519dd38670e 0.0s
=> [frontend build 1/6] FROM docker.io/library/node:25-alpine3.21@sha256:54a2c8c7113949ec9b177738aaa7188529b73e2 0.2s
=> resolve docker.io/library/node:25-alpine3.21@sha256:54a2c8c7113949ec9b177738aaa7188529b73e2cbcf1d572e62bbe 0.2s
=> [frontend internal] load build context 80.3s
=> transferring context: 219.49MB 80.3s
=> [frontend stage-1 1/3] FROM docker.io/library/nginx:alpine@sha256:61e01287e546aac28a3f56839c136b31f590273f3b4 0.2s
=> resolve docker.io/library/nginx:alpine@sha256:61e01287e546aac28a3f56839c136b31f590273f3b41187a36f46f6a03bb 0.2s
=> CACHED [backend stage-1 2/10] RUN apk add --no-cache libpq 0.0s
=> CACHED [backend stage-1 3/10] WORKDIR /app 0.0s
=> CACHED [backend build 2/6] RUN apk add --no-cache gcc musl-dev libffi-dev postgresql-dev build-base 0.0s
=> CACHED [backend build 3/6] WORKDIR /app 0.0s
=> CACHED [backend build 4/6] COPY backend/requirements.txt 0.0s
=> CACHED [backend build 5/6] RUN pip install --upgrade pip 0.0s
=> CACHED [backend build 6/6] RUN pip wheel --wheel-dir=/wheels -r requirements.txt 0.0s
=> CACHED [backend stage-1 4/10] COPY --from=build /wheels /wheels 0.0s
=> CACHED [backend stage-1 5/10] COPY backend/requirements.txt 0.0s
=> CACHED [backend stage-1 6/10] RUN pip install --no-index --find-links=/wheels -r requirements.txt 0.0s
=> [backend stage-1 7/10] COPY --chown=appuser:appgroup --chmod=755 backend/entrypoint.sh /entrypoint.sh 0.1s
=> [backend stage-1 8/10] COPY backend/. 0.1s
=> [backend stage-1 9/10] RUN addgroup -S appgroup && adduser -S appuser -G appgroup 0.4s
=> [backend stage-1 10/10] RUN chown -R appuser:appgroup /app 0.5s
=> [backend] exporting to image 0.6s
=> exporting layers 0.3s
=> exporting manifest sha256:b811020fe40fa15dedfb7f729bda4d9dc9d793c5a33a6e183cca8fc4cf80f377 0.0s
=> exporting config sha256:e45e280cfel2888b42f07a17efae0dcae8ff5ac399404660668c91f6831d8ded9 0.0s
=> exporting attestation manifest sha256:63a202d32dd1ba2bfeale00470b7b9d80bcd3dcfd193c938576bc70a48b1d2fc 0.0s
=> exporting manifest list sha256:7c91d51aae758177d8ade79be2960e259d31ee5a9431e42078534b4d3e6b8121 0.0s
=> naming to docker.io/library/registro-backend:1.0.0 0.0s
=> unpacking to docker.io/library/registro-backend:1.0.0 0.1s
=> [backend] resolving provenance for metadata file 0.0s
=> CACHED [frontend build 2/6] WORKDIR /app 0.0s
=> CACHED [frontend build 3/6] COPY frontend/package*.json 0.0s
=> CACHED [frontend build 4/6] RUN npm ci --legacy-peer-deps --force 0.0s
=> [frontend build 5/6] COPY frontend/. 4.1s
=> [frontend build 6/6] RUN npm run build -- --output-path=dist/frontend/ --configuration=production 4.2s
=> CACHED [frontend stage-1 2/3] COPY --from=build /app/dist/frontend/browser/ /usr/share/nginx/html 0.0s
=> CACHED [frontend stage-1 3/3] COPY nginx/default.conf /etc/nginx/conf.d/default.conf 0.0s
=> [frontend] exporting to image 0.1s
=> exporting layers 0.0s
=> exporting manifest sha256:66c17794fd3eadf204c24ba5263c0b340bf486bcb6c156215d7c4b91912f234 0.0s
=> exporting config sha256:cc344d34b1f5b12890260e5a19bb8db222101eb3d27d906ac64aac583a8424e0 0.0s
=> exporting attestation manifest sha256:5c4dc730ffa6be10c5376fd87f741b71a6592f28dc3cb4f1f12bf2f0ba977e5d 0.0s
=> exporting manifest list sha256:6a4d4029e1fc32872c07925c5e14a90feab2b3be5689cbb69191ad1052d5b14 0.0s
```

```
⇒ ⇒ exporting attestation manifest sha256:5c4dc730ffa6be10c5376fd87f741b71a6592f28dc3cb4f1f12bf2f0ba977e5d 0.0s
⇒ ⇒ exporting manifest list sha256:6a4d4029e1fc32872c07925c51e14a90feab2b3be5689cbb69191ad1052d5b14 0.0s
⇒ ⇒ naming to docker.io/library/registro-frontend:1.0.0 0.0s
⇒ ⇒ unpacking to docker.io/library/registro-frontend:1.0.0 0.0s
⇒ [frontend] resolving provenance for metadata file 0.0s
[+] Running 7/7
  ✔ registro-backend:1.0.0 Built 0.0s
  ✔ registro-frontend:1.0.0 Built 0.0s
  ✔ Network containerizacion_y_orquestacion_registro_network Created 0.0s
  ✔ Container registro_db Recreated 0.6s
  ✔ Container registro_adminer Recreated 0.5s
  ✔ Container registro_backend Recreated 0.2s
  ✔ Container registro_frontend Recreated 0.2s
Attaching to registro_adminer, registro_backend, registro_db, registro_frontend
registro_db | PostgreSQL Database directory appears to contain a database; Skipping initialization
registro_db | 2025-10-25 17:14:29.489 UTC [1] LOG: starting PostgreSQL 15.14 (Debian 15.14-1.pgdg13+1) on x86_64-pc-linux-gnu,
registro_db | 2025-10-25 17:14:29.489 UTC [1] LOG: listening on IPv4 address "0.0.0.0", port 5432
registro_db | 2025-10-25 17:14:29.489 UTC [1] LOG: listening on IPv6 address "::", port 5432
registro_db | 2025-10-25 17:14:29.499 UTC [1] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
registro_db | 2025-10-25 17:14:29.509 UTC [29] LOG: database system was shut down at 2025-10-25 17:14:28 UTC
registro_db | 2025-10-25 17:14:29.518 UTC [1] LOG: database system is ready to accept connections
registro_adminer | [Sat Oct 25 17:14:29 2025] PHP 8.4.14 Development Server (http://[::]:8080) started
registro_backend | [2025-10-25 17:14:29 +0000] [1] [INFO] Starting unicorn 23.0.0
registro_backend | [2025-10-25 17:14:29 +0000] [1] [INFO] Listening at: http://0.0.0.0:8000 (1)
registro_backend | [2025-10-25 17:14:29 +0000] [1] [INFO] Using worker: sync
registro_backend | [2025-10-25 17:14:29 +0000] [7] [INFO] Booting worker with pid: 7
registro_frontend | /docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
registro_frontend | /docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
registro_frontend | /docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
registro_frontend | 10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
registro_frontend | 10-listen-on-ipv6-by-default.sh: info: /etc/nginx/conf.d/default.conf differs from the packaged version
registro_frontend | /docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
registro_frontend | /docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
registro_frontend | /docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
registro_frontend | /docker-entrypoint.sh: Configuration complete; ready for start up
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: using the "epoll" event method
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: nginx/1.29.2
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: built by gcc 14.2.0 (Alpine 14.2.0)
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: OS: Linux 6.6.87.2-microsoft-standard-WSL2
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: start worker processes
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: start worker process 29
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: start worker process 30
registro_backend | [2025-10-25 17:14:30 +0000] [8] [INFO] Booting worker with pid: 6
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: start worker process 32
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: start worker process 33
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: start worker process 34
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: start worker process 35
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: start worker process 36
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: start worker process 37
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: start worker process 38
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: start worker process 39
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: start worker process 40
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: start worker process 41
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: start worker process 42
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: start worker process 43
registro_frontend | 2025/10/25 17:14:30 [notice] 1#1: start worker process 44
registro_backend | [2025-10-25 17:14:30 +0000] [9] [INFO] Booting worker with pid: 9
```

Figura 7. Ejecución de docker compose up --build

Acceder a las rutas para verificar las aplicaciones

- ✓ Frontend: <http://localhost:8082>
- ✓ Backend: <http://localhost:8000/api/docs/>
- ✓ Adminer: <http://localhost:8081>

## Vistas en Navegador – Adminer

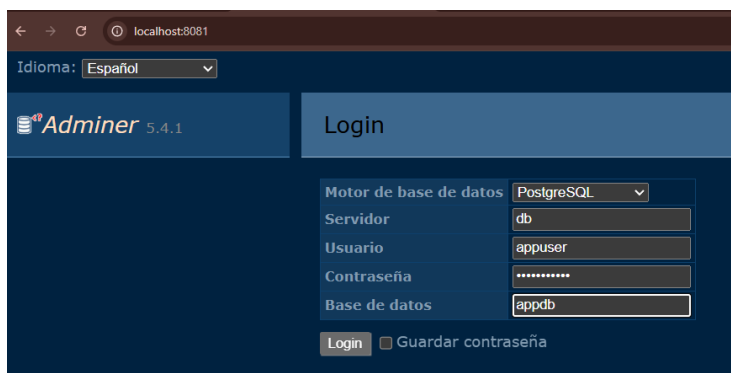


Figura 8. Login de Adminer



Figura 9. Tablas del backend de django

Al tratarse de django si bien se trabaja con los modelos de Usuario (**usuario\_usuario**) y Documento (**usuario\_documento**), se han generado de manera automática, las tablas de configuración por defecto de django.

Además, como se puede observar en la figura 9, se evidencia que se ejecuto de manera correcta el **entryptpoint** de migración de modelos a la base de datos.

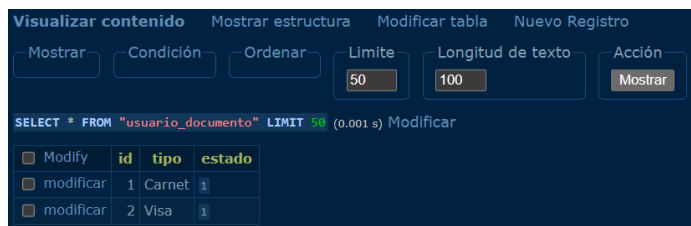
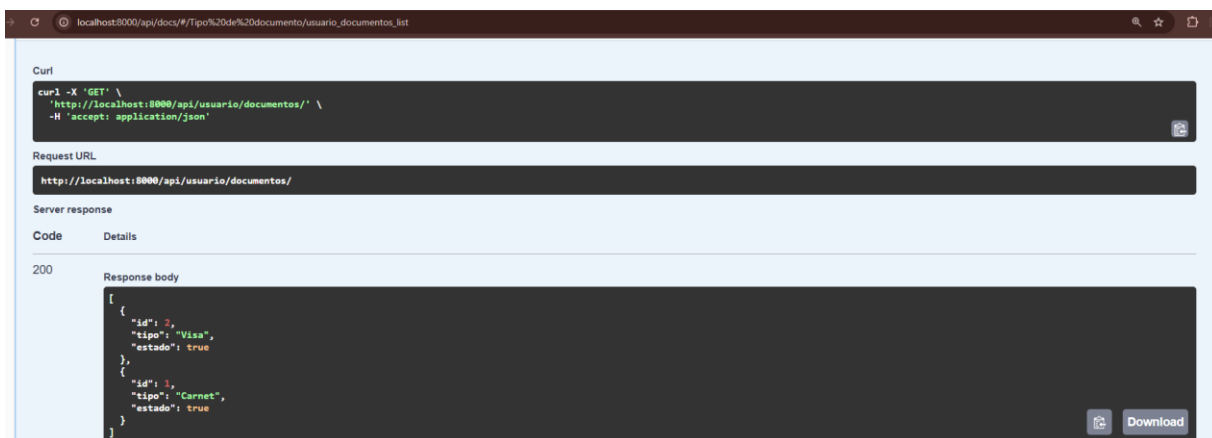


Figura 10. Consulta de datos en la tabla usuario\_documento con 2 registros



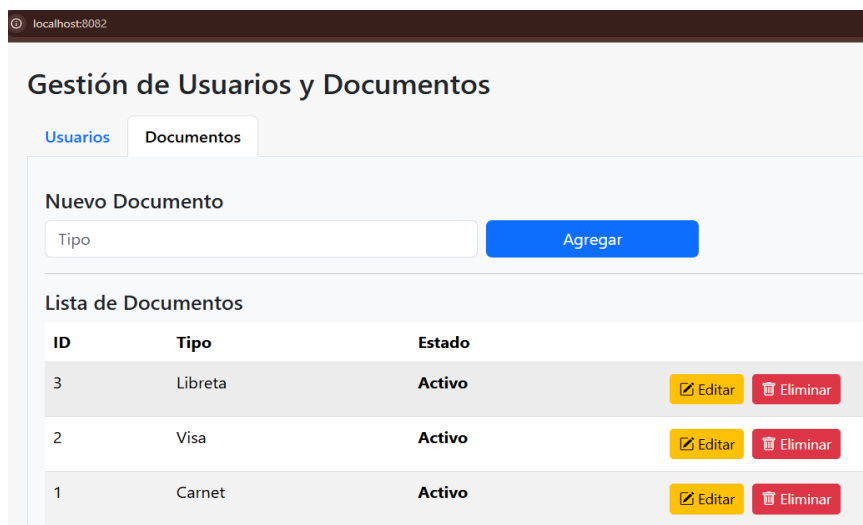
## Vistas en Navegador – Backend



*Figura 11. Consulta de datos desde el backend*

En la figura 11, se puede observar que se realiza la consulta al endpoint <http://localhost:8000/api/usuario/documentos/> y en respuesta se tiene 2 registros al igual que se observa en el Adminer, con esto se evidencia que existe una conexión entre el backend y la base de datos de Adminer y por supuesto sus contenedores.

## Vistas en Navegador – Frontend



*Figura 12. Visualización de formulario Documentos y Usuarios*

Como se puede observar en la figura 12, el frontend esta corriendo en <http://localhost:8082> y además, ya realizó la petición a través de GET para listar todos los documentos. Con esto se evidencia que existe comunicación con el backend y sus contenedores, cabe recalcar que además en el Dockerfile del frontend se esta enviando al contenedor la configuración de **Nginx**.

A continuación, se detalla el contenido de los Dockerfile del backend y frontend.

## Dockerfile del backend

```
# Etapa de build
# =====
FROM python:3.12-alpine AS build

RUN apk add --no-cache gcc musl-dev libffi-dev postgresql-dev build-base

WORKDIR /app

COPY backend/requirements.txt .

RUN pip install --upgrade pip
RUN pip wheel --wheel-dir=/wheels -r requirements.txt

# Etapa final
# =====
FROM python:3.12-alpine

RUN apk add --no-cache libpq

WORKDIR /app

COPY --from=build /wheels /wheels
COPY backend/requirements.txt .

RUN pip install --no-index --find-links=/wheels -r requirements.txt

COPY --chown=appuser:appgroup --chmod=755 backend/entrypoint.sh /entrypoint.sh

COPY backend/ ./

RUN addgroup -S appgroup && adduser -S appuser -G appgroup
RUN chown -R appuser:appgroup /app
USER appuser

ENV PYTHONUNBUFFERED=1
ENV DJANGO_SETTINGS_MODULE=registro.settings

EXPOSE 8000

CMD ["/entrypoint.sh"]
```

## EntryPoint para migrar las tablas a la base de datos

```
#!/bin/sh

# EntryPoint para migración la base de datos limpia
# Espera a que la base de datos esté lista
echo "Esperando a que PostgreSQL esté listo..."
while ! nc -z "$POSTGRES_HOST" "$POSTGRES_PORT"; do
    sleep 1
done

echo "Aplicando migraciones..."
python manage.py migrate

# Ejecuta Gunicorn
exec gunicorn registro.wsgi:application --bind 0.0.0.0:8000 --workers 3
```

## Dockerfile del Frontend

```
FROM node:25-alpine3.21 AS build
WORKDIR /app
COPY frontend/package*.json ./
RUN npm ci --legacy-peer-deps --force
COPY frontend/ ./

RUN npm run build -- --output-path=dist/frontend/ --configuration=production

FROM nginx:alpine
COPY --from=build /app/dist/frontend/browser/ /usr/share/nginx/html

COPY nginx/default.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

## Configuración de Nginx

```
server {
    listen 80;
    server_name _;
    root /usr/share/nginx/html;
    index index.html;
    location / {
        try_files $uri $uri/ /index.html;
    }

    location /api/ {
        proxy_pass http://backend:8000/api/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

## Archivo docker-compose.yml

```
version: "3.9"

services:
  # BASE DE DATOS POSTGRES
  db:
    image: postgres:15
    container_name: registro_db
    restart: always
    environment:
      POSTGRES_DB: ${POSTGRES_DB}
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - registro_network
    ports:
      - "5432:5432"
```

#### **# ADMINER (para DB development)**

```
adminer:
  image: adminer:latest
  container_name: registro_adminer
  restart: always
  depends_on:
    - db
  networks:
    - registro_network
  ports:
    - "8081:8080"
  environment:
    ADMINER_DEFAULT_SERVER: db
```

#### **# BACKEND DJANGO**

```
backend:
  image: registro-backend:1.0.0
  build:
    context: .
    dockerfile: backend/Dockerfile
  container_name: registro_backend
  depends_on:
    - db
  environment:
    DJANGO_SETTINGS_MODULE: registro.settings
    DATABASE_URL: postgresql://${POSTGRES_USER}:${POSTGRES_PASSWORD}@db:5432/${POSTGRES_DB}
    SECRET_KEY: ${DJANGO_SECRET_KEY}
  volumes:
    - ./backend:/app
  networks:
    - registro_network
  ports:
    - "8000:8000"
  command: gunicorn registro.wsgi:application --bind 0.0.0.0:8000 --workers 3
```

#### **# FRONTEND ANGULAR + NGINX**

```
frontend:
  image: registro-frontend:1.0.0
  build:
    context: .
    dockerfile: frontend/Dockerfile
  container_name: registro_frontend
  depends_on:
    - backend
  networks:
    - registro_network
  ports:
    - "8082:80"
```

#### **# REDES PERSONALIZADAS**

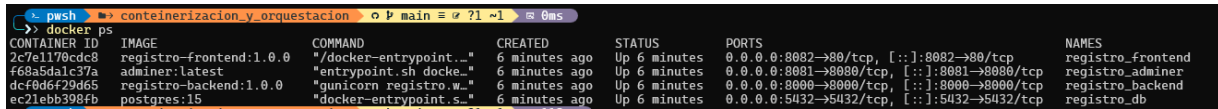
```
networks:
  registro_network:
    driver: bridge
```

#### **# VOLÚMENES PERSISTENTES**

```
volumes:
  postgres_data:
```

## Imágenes creadas

Ejecutar: `docker ps`



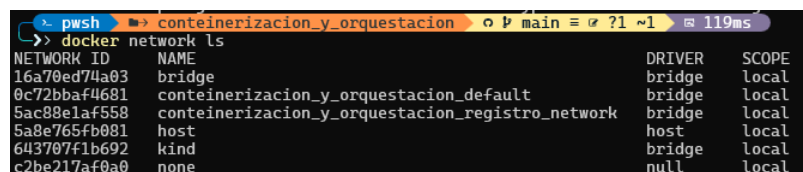
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
2c7e1170cdc8	registro-frontend:1.0.0	"/docker-entrypoint..."	6 minutes ago	Up 6 minutes	0.0.0.0:8082->80/tcp, [::]:8082->80/tcp	registro_frontend
f68a5dalc37a	adminer:latest	"entrypoint.sh docke..."	6 minutes ago	Up 6 minutes	0.0.0.0:8081->8080/tcp, [::]:8081->8080/tcp	registro_adminer
dcf9d6f29d65	registro-backend:1.0.0	"unicorn registro.w..."	6 minutes ago	Up 6 minutes	0.0.0.0:8080->8080/tcp, [::]:8080->8080/tcp	registro_backend
ec21ebb398fb	postgres:15	"docker-entrypoint.s..."	6 minutes ago	Up 6 minutes	0.0.0.0:5432->5432/tcp, [::]:5432->5432/tcp	registro_db

Figura 13. Imágenes creadas

## Redes creadas

Ejecutar: `docker network ls`

Ejecutar: `docker network inspect containerizacion_y_orquestacion_registro_network`



NETWORK ID	NAME	DRIVER	SCOPE
16a70ed74a03	bridge	bridge	local
0c72bbaf4681	containerizacion_y_orquestacion_default	bridge	local
5ac88e1af558	containerizacion_y_orquestacion_registro_network	bridge	local
5a8e765fb081	host	host	local
643707f1b692	kind	bridge	local
c2be217af0a0	none	null	local

Figura 14. Redes personalizadas creadas



```
{
  "Name": "containerizacion_y_orquestacion_registro_network",
  "Id": "5ac88e1af55885e0dc6a3d08ddaa7ef7448774b645592ded4c0fd7794bc77f40",
  "Created": "2025-10-25T17:14:27.931412544Z",
  "Scope": "local",
  "Driver": "bridge",
  "EnableIPv4": true,
  "EnableIPv6": false,
  "IPAM": {
    "Driver": "default",
    "Options": null,
    "Config": [
      {
        "Subnet": "172.20.0.0/16",
        "Gateway": "172.20.0.1"
      }
    ]
  },
  "Internal": false,
  "Attachable": false,
  "Ingress": false,
  "ConfigFrom": {
    "Network": ""
  },
  "ConfigOnly": false,
  "Containers": {
    "2c7e1170cdc8b8824ba9880d7a4e87a6bd9fa1720a3d79f83e717fadc8b7e851": {
      "Name": "registro_frontend",
      "EndpointID": "6dd24f3bd9e15b6c601ef3f1f48853a93c5cd01d7ce6ec84afcb063df22a5a07",
      "MacAddress": "9a:06:9f:95:d7:f0",
      "IPv4Address": "172.20.0.5/16",
      "IPv6Address": ""
    },
    "dcf9d6f29d658aa9386165c2b1ccd7d07b926d3b1eebf6c2e6a8d35fcea9385f": {
      "Name": "registro_backend",
      "EndpointID": "715c029837d27c2e6ac0de4b6463b1a3bf7c284d1e5b519d949f56e6d734f1d",
      "MacAddress": "a2:4c:d5:43:7a:6d",
      "IPv4Address": "172.20.0.4/16",
      "IPv6Address": ""
    },
    "ec21ebb398fb4b5a14eed149dd3903068f31c76bd8f8d8e54893820d37c904f9": {
      "Name": "registro_db",
      "EndpointID": "0ccb44261f985bdf264154e86b88057bdd8e04157d7747d49a591be01738e30d",
      "MacAddress": "56:ff:4b:db:35:5e",
      "IPv4Address": "172.20.0.2/16",
      "IPv6Address": ""
    },
    "f68a5dalc37a1b7f9e6e551e2b80088aee4a4489a7c08f4dcb25f21fa7eb2a4c": {
      "Name": "registro_adminer",
      "EndpointID": "a516a9408a2769dd6d4ac91d8a57b49dd866d723a36b9dd830a96f635e102f95",
      "MacAddress": "42:eb:bc:36:28:c3",
      "IPv4Address": "172.20.0.3/16",
      "IPv6Address": ""
    }
  }
}
```

```

"Options": {
  "com.docker.network.enable_ipv4": "true",
  "com.docker.network.enable_ipv6": "false"
},
"Labels": {
  "com.docker.compose.config-hash": "1a3deb9e3cf8a2cc53d098f26d62849aa6e608046f8dafbf5965256544397e4",
  "com.docker.compose.network": "registro_network",
  "com.docker.compose.project": "containerizacion_y_orquestacion",
  "com.docker.compose.version": "2.40.2"
}
}
]

```

Figura 15. Inspección de la red

## Volúmenes creados

Ejecutar: `docker volume ls`

Ejecutar: `docker volume inspect containerizacion_y_orquestacion_postgres_data`

```

pwhsh ➤ containerizacion_y_orquestacion o p main 72 ~1 8ms
➤ docker volume ls
DRIVER      VOLUME NAME
local      5db5d3c554220015d3b833b77f9a4a4304ca1ab056b194da637fc6713fe5ca8d
local      511753148ee7a925532e0092b294582e9ff3ba99d90994d09fb957a67fb41b94
local      containerizacion_y_orquestacion_postgres_data

```

Figura 16. Volumen creado

```

pwhsh ➤ containerizacion_y_orquestacion o p main 72 ~1 70ms
➤ docker volume inspect containerizacion_y_orquestacion_postgres_data
[
  {
    "CreatedAt": "2025-10-25T07:02:49Z",
    "Driver": "local",
    "Labels": {
      "com.docker.compose.config-hash": "b3acd5027960ce6717df48e27ff0e0717a0223318848e890b79ffee63bc7f027",
      "com.docker.compose.project": "containerizacion_y_orquestacion",
      "com.docker.compose.version": "2.40.2",
      "com.docker.compose.volume": "postgres_data"
    },
    "Mountpoint": "/var/lib/docker/volumes/containerizacion_y_orquestacion_postgres_data/_data",
    "Name": "containerizacion_y_orquestacion_postgres_data",
    "Options": null,
    "Scope": "local"
  }
]

```

Figura 17. Detalle del Volumen

Para el cumplimiento del punto 2 de containerización se ha cumplido con:

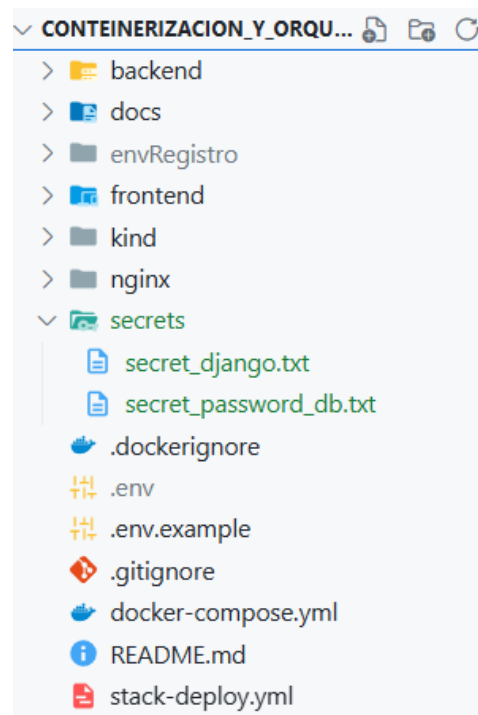
- ✓ Cada servicio debe contar con su Dockerfile, el frontend y el backend cuentan con el archivo solicitado.
- ✓ Las imágenes se han construido con etiquetas versionadas (tags) tanto para el frontend y backend, por lo que se cumple con lo solicitado.
- ✓ Se ha usado imágenes ligeras de tipo alpine la **NodeJs** (`node:25-alpine3.21`), para construir el frontend, y para el backend de igual forma **Python** (`python:3.12-alpine`), además de configurar el archivo **.dockerignore** y tampoco se expuso las credenciales para la plataforma.

Para el cumplimiento del punto 3, se ha creado el archivo **docker-compose.yml** en la raíz del proyecto con las siguientes características:

- ✓ Tiene la configuración para levantar todo el proyecto.
- ✓ Cuenta con la configuración de volúmenes persistentes.
- ✓ Cuenta con la configuración de redes personalizadas.
- ✓ El frontend ha sido expuesto en el puerto 8082:80.
- ✓ Cuenta con variables de entorno para la base de datos y el backend.
- ✓ Se ha incluido un entrypoint para la migración de las tablas a la base de datos.

## b. Docker Swarm

Se cuenta con la siguiente estructura del proyecto para trabajar con docker swarm, como se muestra en la figura 18.



*Figura 18. Nueva estructura del proyecto*

A continuación, se muestra el contenido del archivo stack-deploy.yml

```
version: "3.9"

services:
  # BASE DE DATOS POSTGRES
  db:
    image: postgres:15
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure
    environment:
```

```

    POSTGRES_DB: registro_db
    POSTGRES_USER: registro_user
    POSTGRES_PASSWORD_FILE: /run/secrets/secret_password_db
volumes:
  - pg_data:/var/lib/postgresql/data
secrets:
  - secret_password_db
networks:
  - registro_network
ports:
  - "5432:5432"

# ADMINER
adminer:
  image: adminer:latest
  deploy:
    replicas: 1
    restart_policy:
      condition: on-failure
  environment:
    ADMINER_DEFAULT_SERVER: db
  networks:
    - registro_network
  ports:
    - "8081:8080"
  depends_on:
    - db

# BACKEND DJANGO
backend:
  image: registro-backend:1.0.0
  deploy:
    replicas: 3
    restart_policy:
      condition: on-failure
  environment:
    DJANGO_SETTINGS_MODULE: registro.settings
    DATABASE_URL: postgresql://registro_user:db:5432/registro_db
    SECRET_KEY_FILE: /run/secrets/secret_django
  volumes:
    - backend_data:/app
  secrets:
    - secret_django
    - secret_password_db
  networks:
    - registro_network
  ports:
    - "8000:8000"
  depends_on:
    - db
  command: gunicorn registro.wsgi:application --bind 0.0.0.0:8000 --workers 3

# FRONTEND ANGULAR + NGINX
frontend:
  image: registro-frontend:1.0.0
  deploy:
    replicas: 3
    restart_policy:
      condition: on-failure
  networks:
    - registro_net
  ports:
    - "8082:80"
  configs:
    - source: nginx_default
      target: /etc/nginx/conf.d/default.conf
  depends_on:
    - backend

# REDES Y VOLÚMENES
networks:

```



```

registro_network:
  driver: overlay

volumes:
  pg_data:
  backend_data:

secrets:
  secret_password_db:
    external: true
  secret_django:
    external: true

configs:
  nginx_default:
    external: true

```

Creamos los secrets en docker con la siguiente línea de comando:

**Ejecutar:**

**`echo "django_secret_key" | docker secret create secret_django -`**  
**`echo "apppassword" | docker secret create secret_password_db -`**  
**`docker secret ls`**

The first screenshot shows the execution of three Docker commands to create secrets and list them. The second screenshot shows the execution of two Docker commands to create a configuration and list it.

```

> pwsh ➤ containerizacion_y_orquestacion o p main ≡ 71 ~1 72ms
>> echo "django_secret_key" | docker secret create secret_django -
i502u4ey2crvwaz5aivcwnfsk
> pwsh ➤ containerizacion_y_orquestacion o p main ≡ 71 ~1 157ms
>> echo "apppassword" | docker secret create secret_password_db -
zfja72hdb8mnc8d453vv0dw3b
> pwsh ➤ containerizacion_y_orquestacion o p main ≡ 71 ~1 73ms
>> docker secret ls

```

ID	NAME	DRIVER	CREATED	UPDATED
i502u4ey2crvwaz5aivcwnfsk	secret_django		5 minutes ago	5 minutes ago
zfja72hdb8mnc8d453vv0dw3b	secret_password_db		4 minutes ago	4 minutes ago

```

> pwsh ➤ containerizacion_y_orquestacion o p main ≡ 71 ~1 70ms
>> docker config create nginx_default nginx/default.conf
ugiz4laeb62q5v9c2m4ibobyb
> pwsh ➤ containerizacion_y_orquestacion o p main ≡ 71 ~1 76ms
>> docker config ls

```

ID	NAME	CREATED	UPDATED
ugiz4laeb62q5v9c2m4ibobyb	nginx_default	13 seconds ago	13 seconds ago

*Figura 19. Secrets y config creados*

Iniciamos el cluster Swarm, con la siguiente línea de comando:

**Ejecutar: `docker swarm init`**

The screenshot shows the output of the 'docker swarm init' command, indicating that Swarm is initialized and the current node is now a manager. It also provides instructions on how to add a worker or a manager to the swarm.

```

> pwsh ➤ containerizacion_y_orquestacion o p main ≡ 71 ~1 0ms
>> docker swarm init
Swarm initialized: current node (1tw4l59s0790zeieqzpz79vhs) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-0m5hhydunnwc69oubu3i14z772901zkwuaz169vrc91vd6wd44-7rg12p40e0ah8u7gd709k0xdq 192.168.65.3:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

```

*Figura 20. Resultado de inicialización*

Verificación de Swarm si se encuentra activo.

```

Swarm: active
NodeID: 1tw4l59s0790zeieqzpz79vhs
Is Manager: true
ClusterID: rh2kd8v2s5mb3ikjzj24mm3db
Managers: 1
Nodes: 1
Data Path Port: 4789
Orchestration:
Task History Retention Limit: 5
Raft:
Snapshot Interval: 10000
Number of Old Snapshots to Retain: 0
Heartbeat Tick: 1
Election Tick: 10
Dispatcher:
Heartbeat Period: 5 seconds
CA Configuration:
Expiry Duration: 3 months
Force Rotate: 0
AutoLock Managers: false
Root Rotation In Progress: false
Node Address: 192.168.65.3
Manager Addresses:
192.168.65.3:2377

```

Figura 21. Swarm activo

Construimos las imágenes versionadas tanto del backend y del frontend como se muestran en la figura 22 y figura 23.

Ejecutar:

***docker build -t registro-backend:1.0.0 -f backend/Dockerfile .***

***docker build -t registro-frontend:1.0.0 -f frontend/Dockerfile .***

```

$ pwsh ➤ containerizacion_y_orquestacion ➤ V main ➤ ?1 ~1 ➤ 827ms
➤ docker build -t registro-backend:1.0.0 -f backend/Dockerfile .
[+] Building 3.0s (21/21) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 926B
=> [internal] load metadata for docker.io/library/python:3.12-alpine
=> [auth] library/python:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 242B
=> [internal] load build context
=> => transferring context: 5.28kB
=> [build 1/6] FROM docker.io/library/python:3.12-alpine@sha256:d82291d418d5c47f267708393e40599ae836f2260b0519dd38670e9d281657f5
=> => resolve docker.io/library/python:3.12-alpine@sha256:d82291d418d5c47f267708393e40599ae836f2260b0519dd38670e9d281657f5
=> CACHED [stage-1 2/10] RUN apk add --no-cache libpq
=> CACHED [stage-1 3/10] WORKDIR /app
=> CACHED [build 2/6] RUN apk add --no-cache gcc musl-dev libffi-dev postgresql-dev build-base
=> CACHED [build 3/6] WORKDIR /app
=> CACHED [build 4/6] COPY backend/requirements.txt .
=> CACHED [build 5/6] RUN pip install --upgrade pip
=> CACHED [build 6/6] RUN pip wheel --wheel-dir=/wheels -r requirements.txt
=> CACHED [stage-1 4/10] COPY --from=build /wheels /wheels
=> CACHED [stage-1 5/10] COPY backend/requirements.txt .
=> CACHED [stage-1 6/10] RUN pip install --no-index --find-links=/wheels -r requirements.txt
=> CACHED [stage-1 7/10] COPY --chown=appuser:appgroup --chmod=755 backend/entrypoint.sh /entrypoint.sh
=> [stage-1 8/10] COPY backend/ ./
=> [stage-1 9/10] RUN addgroup -S appgroup && adduser -S appuser -G appgroup
=> [stage-1 10/10] RUN chown -R appuser:appgroup /app
=> exporting to image
=> => exporting layers
=> => exporting manifest sha256:d29df9036c8679c96be6a8c0b0ac0d36cc67719d3b3a25af323be2d298816e42
=> => exporting config sha256:cb1bf61f3f9bbde4a274357a3016f5a6725f4d413ef6a11047b1f2f29a8b835f
=> => exporting attestation manifest sha256:ad710d3a0c6f3ef507b06c0b2b76110e83081b2a70970ace1c2de08877a83359
=> => exporting manifest list sha256:eeab08d23b2cb1c085f78cbe15f83493bd1a8f08359acc5967c3be17901f21cf
=> => naming to docker.io/library/registro-backend:1.0.0
=> => unpacking to docker.io/library/registro-backend:1.0.0

```

Figura 22. Construcción de la imagen backend

```
pws@ containerizacion_y_orquestacion ~$ docker build -t registro-frontend:1.0.0 -f frontend/Dockerfile .
[+] Building 80.5s (17/17) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 452B
=> [internal] load metadata for docker.io/library/nginx:alpine
=> [internal] load metadata for docker.io/library/node:25-alpine3.21
=> [auth] library/nginx:pull token for registry-1.docker.io
=> [auth] library/node:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 242B
=> [build 1/6] FROM docker.io/library/node:25-alpine3.21@sha256:54a2c8c7113949ec9b177738aaa7188529b73e2cbcfd1d572e62bbe4c2e7e4df0
=> => resolve docker.io/library/node:25-alpine3.21@sha256:54a2c8c7113949ec9b177738aaa7188529b73e2cbcfd1d572e62bbe4c2e7e4df0
=> [internal] load build context
=> => transferring context: 219.49MB
=> [stage-1 1/3] FROM docker.io/library/nginx:alpine@sha256:61e01287e546aac28a3f56839c136b31f590273f3b41187a36f46f6a03bbfe22
=> => resolve docker.io/library/nginx:alpine@sha256:61e01287e546aac28a3f56839c136b31f590273f3b41187a36f46f6a03bbfe22
=> CACHED [build 2/6] WORKDIR /app
=> CACHED [build 3/6] COPY frontend/package*.json ./
=> CACHED [build 4/6] RUN npm ci --legacy-peer-deps --force
=> CACHED [build 5/6] COPY frontend/ ./
=> CACHED [build 6/6] RUN npm run build -- --output-path=dist/frontend/ --configuration=production
=> CACHED [stage-1 2/3] COPY --from=build /app/dist/frontend/browser/ /usr/share/nginx/html
=> CACHED [stage-1 3/3] COPY nginx/default.conf /etc/nginx/conf.d/default.conf
=> exporting to image
=> => exporting layers
=> => exporting manifest sha256:41946053c48336aa78746fc9d0d0c1434b9e5c75a50d0deff089e5d72e44fc77
=> => exporting config sha256:a4e1ac83d61fa01f2292543540e08c05e135a7ee669ad4e9eca8120ea8c48ddf
=> => exporting attestation manifest sha256:df74a29ef9c5f78be825100be97bbfc3153fb8d0a92abac8fd6b55bc2b95e6f5
=> => exporting manifest list sha256:cc4ecac3640df8d239e2656ec775f22557590d28b93f1aab013cc5d1ab816e4a
=> => naming to docker.io/library/registro-frontend:1.0.0
=> => unpacking to docker.io/library/registro-frontend:1.0.0
```

Figura 23. Construcción de la imagen frontend

Verificamos las imágenes con el comando de: **docker images**, como se muestra en la figura 23.

```
pws@ containerizacion_y_orquestacion ~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
registro-backend    1.0.0              eeab08d23b2c       11 minutes ago     213MB
registro-frontend   1.0.0              cc4ecac3640d       2 days ago         81.3MB
```

Figura 24. Verificación de las imágenes

Ahora procedemos a desplegar el Stack con la siguiente línea de comando:

**docker stack deploy -c stack-deploy.yml registro\_stack**

```
pws@ containerizacion_y_orquestacion ~$ docker stack deploy -c stack-deploy.yml registro_stack
Since --detach=false was not specified, tasks will be created in the background.
In a future release, --detach=false will become the default.
Creating service registro_stack_backend
Creating service registro_stack_frontend
Creating service registro_stack_db
Creating service registro_stack_adminer
```

Figura 25. Stack desplegado

Lista de registros de stack, con el siguiente comando: **docker stack ls**

```
pws@ containerizacion_y_orquestacion ~$ docker stack ls
NAME                SERVICES
registro_stack      4
```

Figura 26. Servicios registrados

Ahora vemos los servicios del stack con la siguiente línea de comando:

## *docker stack services registro\_stack*

```
pwsh ➤ containerizacion_y_orquestacion ➤ P main ≡ 71 ~1 74ms
```

ID	NAME	MODE	REPLICAS	IMAGE	PORTS
81p661f6s5wz	registro_stack_adminer	replicated	1/1	adminer:latest	*:8081→8080/tcp
l8jj1p4lk3fv	registro_stack_backend	replicated	3/3	registro-backend:1.0.0	*:8000→8000/tcp
umr34xpnv84v	registro_stack_db	replicated	1/1	postgres:15	*:5432→5432/tcp
kevrj6p0jlk2	registro_stack_frontend	replicated	3/3	registro-frontend:1.0.0	*:8082→80/tcp

*Figura 27. Servicios registrados*

Ver las tareas que están corriendo en el stack, con el siguiente comando:

## *Docker stack ps registro\_stack*

```
pwsh ➤ containerizacion_y_orquestacion ➤ P main ≡ 71 ~1 79ms
```

ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE	ERROR	PORTS
kr63ivbzkml1	registro_stack_adminer.1	adminer:latest	docker-desktop	Running	Running 9 minutes ago		
avijrvtgo31	registro_stack_backend.1	registro-backend:1.0.0	docker-desktop	Running	Running 9 minutes ago		
cozegx1u2i5s	registro_stack_backend.2	registro-backend:1.0.0	docker-desktop	Running	Running 9 minutes ago		
w1m4a033zk9	registro_stack_backend.3	registro-backend:1.0.0	docker-desktop	Running	Running 9 minutes ago		
mjg6o5sy2gv	registro_stack_db.1	postgres:15	docker-desktop	Running	Running 9 minutes ago		
jhc052c6p1i	registro_stack_frontend.1	registro-frontend:1.0.0	docker-desktop	Running	Running 9 minutes ago		
xoiewzbnq4fy	registro_stack_frontend.2	registro-frontend:1.0.0	docker-desktop	Running	Running 9 minutes ago		
rjrjj4j4fwb8	registro_stack_frontend.3	registro-frontend:1.0.0	docker-desktop	Running	Running 9 minutes ago		

*Figura 28. Tareas del stack*

Ver los logs en tiempo real del stack de registro backend, con la siguiente línea de comando: **docker service logs registro\_stack\_backend -f**

```
pwsh ➤ containerizacion_y_orquestacion ➤ P main ≡ 71 ~1 109ms
```

registro_stack_backend.3.w1m4a033zk9@docker-desktop	[2025-10-28 05:00:13 +0000] [1] [INFO] Starting gunicorn 23.0.0
registro_stack_backend.3.w1m4a033zk9@docker-desktop	[2025-10-28 05:00:13 +0000] [1] [INFO] Listening at: http://0.0.0.0:8000 (1)
registro_stack_backend.3.w1m4a033zk9@docker-desktop	[2025-10-28 05:00:13 +0000] [1] [INFO] Using worker: sync
registro_stack_backend.3.w1m4a033zk9@docker-desktop	[2025-10-28 05:00:13 +0000] [7] [INFO] Booting worker with pid: 7
registro_stack_backend.3.w1m4a033zk9@docker-desktop	[2025-10-28 05:00:14 +0000] [8] [INFO] Booting worker with pid: 8
registro_stack_backend.3.w1m4a033zk9@docker-desktop	[2025-10-28 05:00:14 +0000] [9] [INFO] Booting worker with pid: 9
registro_stack_backend.2.cozegx1u2i5s@docker-desktop	[2025-10-28 05:00:13 +0000] [1] [INFO] Starting gunicorn 23.0.0
registro_stack_backend.2.cozegx1u2i5s@docker-desktop	[2025-10-28 05:00:13 +0000] [1] [INFO] Listening at: http://0.0.0.0:8000 (1)
registro_stack_backend.2.cozegx1u2i5s@docker-desktop	[2025-10-28 05:00:13 +0000] [1] [INFO] Using worker: sync
registro_stack_backend.2.cozegx1u2i5s@docker-desktop	[2025-10-28 05:00:13 +0000] [7] [INFO] Booting worker with pid: 7
registro_stack_backend.2.cozegx1u2i5s@docker-desktop	[2025-10-28 05:00:14 +0000] [8] [INFO] Booting worker with pid: 8
registro_stack_backend.2.cozegx1u2i5s@docker-desktop	[2025-10-28 05:00:14 +0000] [9] [INFO] Booting worker with pid: 9
registro_stack_backend.1.avijrvtgo31@docker-desktop	[2025-10-28 05:00:13 +0000] [1] [INFO] Starting gunicorn 23.0.0
registro_stack_backend.1.avijrvtgo31@docker-desktop	[2025-10-28 05:00:13 +0000] [1] [INFO] Listening at: http://0.0.0.0:8000 (1)
registro_stack_backend.1.avijrvtgo31@docker-desktop	[2025-10-28 05:00:13 +0000] [1] [INFO] Using worker: sync
registro_stack_backend.1.avijrvtgo31@docker-desktop	[2025-10-28 05:00:13 +0000] [7] [INFO] Booting worker with pid: 7
registro_stack_backend.1.avijrvtgo31@docker-desktop	[2025-10-28 05:00:13 +0000] [8] [INFO] Booting worker with pid: 8
registro_stack_backend.1.avijrvtgo31@docker-desktop	[2025-10-28 05:00:14 +0000] [9] [INFO] Booting worker with pid: 9

*Figura 29. Logs de registro stack backend*

```
pwsh ➤ containerizacion_y_orquestacion ➤ P main ≡ 71 ~1 3m 5s 904ms
```

registro_stack_backend scaled to 4	
overall progress: 4 out of 4 tasks	
1/4: running	[=====→]
2/4: running	[=====→]
3/4: running	[=====→]
4/4: running	[=====→]
verify: Service registro_stack_backend converged	

*Figura 30. Scalando el backend*

```

pws@containerizacion_y_orquestacion: ~$ docker service ls
ID                NAME                MODE                REPLICAS    IMAGE                PORTS
81p661f6s5wz     registro_stack_adminer replicated          1/1          adminer:latest       *:8081->8080/tcp
l8jjlp4lk3fv     registro_stack_backend replicated          4/4          registro-backend:1.0.0 *:8000->8000/tcp
umr34xpnv84v     registro_stack_db    replicated          1/1          postgres:15          *:5432->5432/tcp
kevrj6p0j1k2     registro_stack_frontend replicated          3/3          registro-frontend:1.0.0 *:8082->80/tcp

```

Figura 31. Lista de servicios y replicas

```

pws@containerizacion_y_orquestacion: ~$ docker service ps registro_stack_backend
ID                NAME                IMAGE                NODE                DESIRED STATE    CURRENT STATE    ERROR    PORTS
avijrvtgfo31     registro_stack_backend.1 registro-backend:1.0.0 docker-desktop    Running           Running 23 minutes ago
cozegxlu2i5s     registro_stack_backend.2 registro-backend:1.0.0 docker-desktop    Running           Running 23 minutes ago
w1m4a0332kh9     registro_stack_backend.3 registro-backend:1.0.0 docker-desktop    Running           Running 23 minutes ago
rl9xvvz1d4m7     registro_stack_backend.4 registro-backend:1.0.0 docker-desktop    Running           Running 7 minutes ago

```

Figura 32. Replicas del servicio stack del backend, cantidad 4

Para el cumplimiento de este punto, despliegue de docker swarm se realizó:

- ✓ Conversión de docker-compose.yml a stack-deploy.yml
- ✓ Se tomó el archivo docker-compose.yml como base y se adaptó a las reglas de Docker Swarm, generando un nuevo archivo llamado stack-deploy.yml.
- ✓ En esta conversión se añadieron:
  - Sección deploy: en cada servicio, con configuración de réplicas y políticas de actualización.
  - Uso de configs y secrets externos, gestionados por Swarm.
  - Versionamiento de imágenes (registro-backend:1.0.0, registro-frontend:1.0.0).
  - Redes y volúmenes personalizados, mantenidos desde Compose.

Esto permitió que el mismo proyecto se ejecute como un stack distribuido y escalable, en lugar de simples contenedores aislados.

### c. Kubernetes (KIND, conforme la lista designada)

Para el cumplimiento de esta sección se han creado los siguientes archivos:

Archivo	Función
<b>replicaset.yaml</b>	ReplicaSet base para gestionar réplicas del backend.
<b>deployment-frontend.yaml</b>	Controlador de despliegue (Deployment) para versiones y actualizaciones del backend y frontend.

<b>deployment-backend.yaml</b>	
<b>service-frontend.yaml</b> <b>service-backend.yaml</b>	Servicios internos para exponer backend y frontend dentro del cluster.
<b>loadbalancer.yaml</b>	LoadBalancer (o Ingress) para exponer el frontend y backend al exterior del cluster kind.
<b>secret.yaml</b>	Secret para almacenar variables sensibles (clave Django, contraseña DB).

Para esta sección no se cuenta con Kind en el equipo, sin embargo se instaló con **chocolatey** para su implementación, como se muestra en la siguiente línea de comando:

```
PS C:\WINDOWS\system32> choco -v
2.5.1
PS C:\WINDOWS\system32> choco install kind -y
Chocolatey v2.5.1
Installing the following packages:
kind
By installing, you accept licenses for the packages.
Downloading package from source 'https://community.chocolatey.org/api/v2/'
Progress: Downloading chocolatey-dotnetfx.extension 1.0.1... 100%

chocolatey-dotnetfx.extension v1.0.1
chocolatey-dotnetfx.extension package files install completed. Performing other installation steps.
  Installed/updated chocolatey-dotnetfx extensions.
  The install of chocolatey-dotnetfx.extension was successful.
  Deployed to 'C:\ProgramData\chocolatey\extensions\chocolatey-dotnetfx'
Downloading package from source 'https://community.chocolatey.org/api/v2/'
Progress: Downloading KB2919442 1.0.20160915... 100%

KB2919442 v1.0.20160915 [Approved]
KB2919442 package files install completed. Performing other installation steps.
Skipping installation because this hotfix only applies to Windows 8.1 and Windows Server 2012 R2.
  The install of KB2919442 was successful.
  Software install location not explicitly set, it could be in package or
  default install location of installer.
Downloading package from source 'https://community.chocolatey.org/api/v2/'
Progress: Downloading KB2919355 1.0.20160915... 100%

KB2919355 v1.0.20160915 [Approved]
KB2919355 package files install completed. Performing other installation steps.
Skipping installation because this hotfix only applies to Windows 8.1 and Windows Server 2012 R2.
  The install of KB2919355 was successful.
  Software install location not explicitly set, it could be in package or
  default install location of installer.
Downloading package from source 'https://community.chocolatey.org/api/v2/'
Progress: Downloading dotnetfx 4.8.0.20220524... 100%

dotnetfx v4.8.0.20220524 [Approved]
dotnetfx package files install completed. Performing other installation steps.
Microsoft .NET Framework 4.8 or later is already installed.
  The install of dotnetfx was successful.
  Software install location not explicitly set, it could be in package or
  default install location of installer.
Downloading package from source 'https://community.chocolatey.org/api/v2/'
Progress: Downloading docker-desktop 4.49.0... 100%
```



```

Installed:
- chocolatey-dotnetfx.extension v1.0.1
- docker-desktop v4.49.0
- dotnetfx v4.8.0.20220524
- KB2919355 v1.0.20160915
- KB2919442 v1.0.20160915
- kind v0.30.0
PS C:\WINDOWS\system32> kind version
kind v0.30.0 go1.24.6 windows/amd64
PS C:\WINDOWS\system32>

```

*Figura 34. Instalación de KIND vía chocolatey*

Se cuenta en el directorio de Kind/kind-config.yaml, que contiene la siguiente configuración.

```

kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
containerdConfigPatches:
- |-
  [plugins."io.containerd.grpc.v1.cri".registry.mirrors."localhost:5000"]
    endpoint = ["http://registry:5000"]
nodes:
- role: control-plane
- role: worker
- role: worker

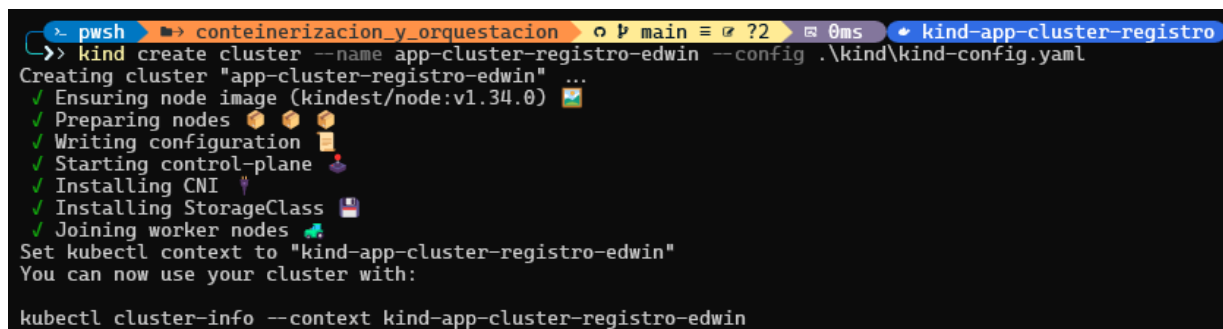
```

A continuación, creamos el cluster con la siguiente línea de comandos:

**Ejecutar:**

***kind create cluster --name app-cluster-registro-edwin***

creamos este cluster con la configuración de archivo kind-config.yaml



```

> pwsh -> containerizacion_y_orquestacion -> main -> ? 0ms -> kind-app-cluster-registro
>> kind create cluster --name app-cluster-registro-edwin --config .\kind\kind-config.yaml
Creating cluster "app-cluster-registro-edwin" ...
✓ Ensuring node image (kindest/node:v1.34.0)
✓ Preparing nodes
✓ Writing configuration
✓ Starting control-plane
✓ Installing CNI
✓ Installing StorageClass
✓ Joining worker nodes
Set kubectl context to "kind-app-cluster-registro-edwin"
You can now use your cluster with:

kubectl cluster-info --context kind-app-cluster-registro-edwin

```

*Figura 35. Creación de Cluster con KIND*

Ahora verificamos que el cluster quedo activo, con la siguiente línea de comandos

***kubectl cluster-info --context kind-app-cluster-registro-edwin***

***kubectl get nodes***

```

> pwsh ➤ conteinerizacion_y_orquestacion o P main ≡ ?2 0ms kind-app-clu
>> kubectl cluster-info --context kind-app-cluster-registro-edwin
Kubernetes control plane is running at https://127.0.0.1:59155
CoreDNS is running at https://127.0.0.1:59155/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

> pwsh ➤ conteinerizacion_y_orquestacion o P main ≡ ?2 261ms
>> kubectl get nodes
NAME                                STATUS    ROLES    AGE    VERSION
app-cluster-registro-edwin-control-plane  Ready    control-plane  8m14s  v1.34.0
app-cluster-registro-edwin-worker        Ready    <none>        8m2s   v1.34.0
app-cluster-registro-edwin-worker2       Ready    <none>        8m2s   v1.34.0

```

Figura 36. Verificación de status

Podemos observar en la figura 36 podemos observar que el cluster creado en su status se encuentra en modo **Ready**, lo que indica que esta funcionando de manera correcta.

Ahora verificamos que el servicio de Kubernetes este usando el contexto correcto con el cluster creado.

Ejecutamos la siguiente línea de comando: `kubectl config current-context`

```

> pwsh ➤ conteinerizacion_y_orquestacion
>> kubectl config current-context
kind-app-cluster-registro-edwin

```

Figura 37. Contexto actual de kubectl

Antes de versionar se ha etiquetado (tag) las imágenes del frontend y backend con el identificador de 1.0.0 para ambas imágenes.

**`docker tag registro-backend:1.0.0 localhost:5000/registro-backend:1.0.0`**

**`docker tag registro-frontend:1.0.0 localhost:5000/registro-frontend:1.0.0`**

```

> pwsh ➤ conteinerizacion_y_orquestacion o P main ≡ ?2 0ms
>> docker tag registro-backend:1.0.0 localhost:5000/registro-backend:1.0.0
> pwsh ➤ conteinerizacion_y_orquestacion o P main ≡ ?2 107ms
>> docker tag registro-frontend:1.0.0 localhost:5000/registro-frontend:1.0.0
> pwsh ➤ conteinerizacion_y_orquestacion o P main ≡ ?2 99ms
>> docker images | findstr registro
registro-backend          1.0.0          eeab08d23b2c   35 hours ago     213MB
localhost:5000/registro-backend  1.0.0          eeab08d23b2c   35 hours ago     213MB
registro-frontend         1.0.0          cc4ecac3640d    4 days ago       81.3MB
localhost:5000/registro-frontend  1.0.0          cc4ecac3640d    4 days ago       81.3MB

```

Figura 38. Etiquetado (tags) para las imágenes

En el caso de que si no está corriendo el **registry** creado lo podemos volver a habilitar con la siguiente línea de comandos.

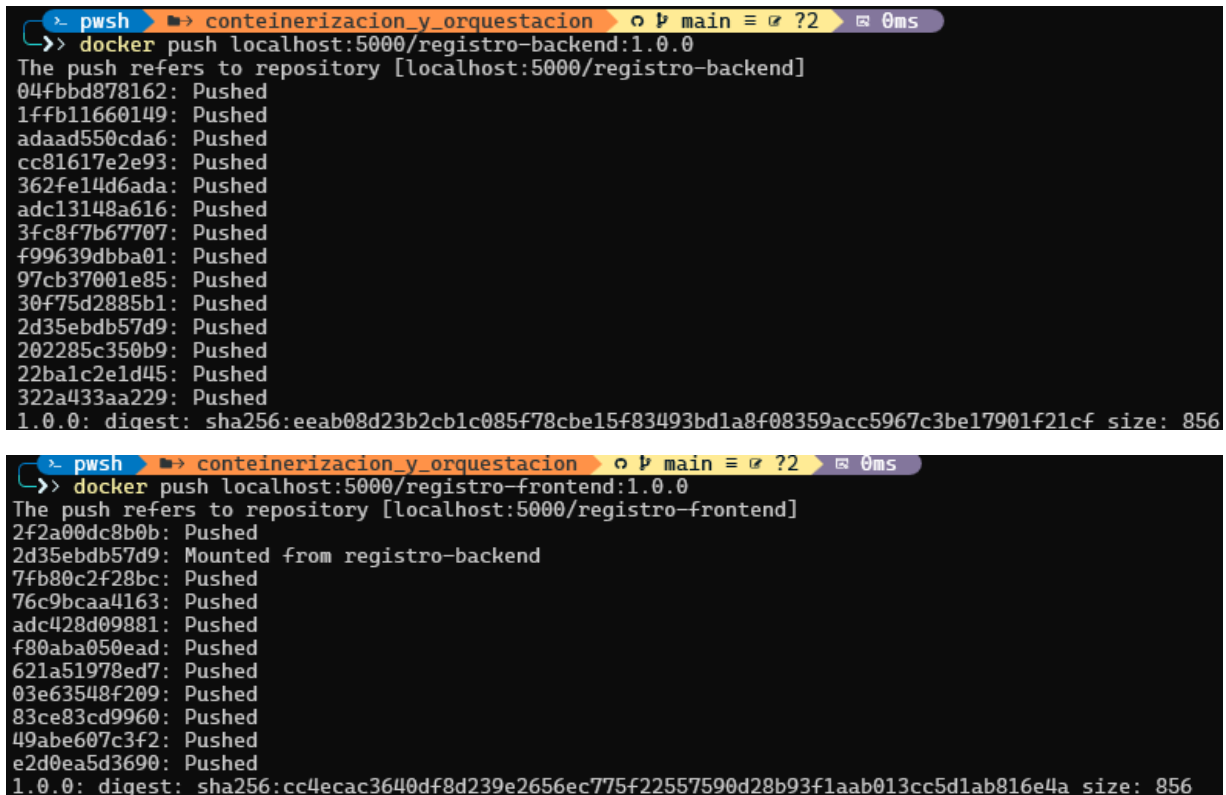
**`docker run -d -p 5000:5000 --name registry registry:2`**



Ahora hacemos el push de las imágenes del backend y frontend con la siguiente línea de comandos hacia el registry local:

***docker push localhost:5000/registro-backend:1.0.0***

***docker push localhost:5000/registro-frontend:1.0.0***



```
pws@contenerizacion_y_orquestacion:~/main$ docker push localhost:5000/registro-backend:1.0.0
The push refers to repository [localhost:5000/registro-backend]
04fbbd878162: Pushed
1ffb11660149: Pushed
adaad550cda6: Pushed
cc81617e2e93: Pushed
362fe14d6ada: Pushed
adc13148a616: Pushed
3fc8f7b67707: Pushed
f99639dbba01: Pushed
97cb37001e85: Pushed
30f75d2885b1: Pushed
2d35ebdb57d9: Pushed
202285c350b9: Pushed
22ba1c2e1d45: Pushed
322a433aa229: Pushed
1.0.0: digest: sha256:eeab08d23b2cb1c085f78cbe15f83493bd1a8f08359acc5967c3be17901f21cf size: 856

pws@contenerizacion_y_orquestacion:~/main$ docker push localhost:5000/registro-frontend:1.0.0
The push refers to repository [localhost:5000/registro-frontend]
2f2a00dc8b0b: Pushed
2d35ebdb57d9: Mounted from registro-backend
7ffb80c2f28bc: Pushed
76c9bcaa4163: Pushed
adc428d09881: Pushed
f80aba050ead: Pushed
621a51978ed7: Pushed
03e63548f209: Pushed
83ce83cd9960: Pushed
49abe607c3f2: Pushed
e2d0ea5d3690: Pushed
1.0.0: digest: sha256:cc4ecac3640df8d239e2656ec775f22557590d28b93f1aab013cc5d1ab816e4a size: 856
```

*Figura 39. Push al registry local backend y frontend*

Para crear una nueva versión ya sea del front o backend, reconstruimos las imágenes con la siguiente identificación que corresponda.

***docker build -t registro-backend:2.0.0 -f backend/Dockerfile .***

***docker build -t registro-frontend:2.0.0 -f frontend/Dockerfile .***

```
➤ pwsh ➤ containerizacion_y_orquestacion ➤ main ➤ 72 ➤ 0ms ➤ kind-app-cluster-registro
➤ docker build -t registro-backend:2.0.0 -f backend/Dockerfile .
[+] Building 3.6s (20/20) FINISHED
➤ [internal] load build definition from Dockerfile
➤ ➤ transferring dockerfile: 926B
➤ [internal] load metadata for docker.io/library/python:3.12-alpine
➤ [internal] load .dockerignore
➤ ➤ transferring context: 242B
➤ [internal] load build context
➤ ➤ transferring context: 12.14kB
➤ [build 1/6] FROM docker.io/library/python:3.12-alpine@sha256:d82291d418d5c47f267708393e40599ae836f2260b0519dd38670e9d281657f5
➤ ➤ resolve docker.io/library/python:3.12-alpine@sha256:d82291d418d5c47f267708393e40599ae836f2260b0519dd38670e9d281657f5
➤ ➤ CACHED [stage-1 2/10] RUN apk add --no-cache libpq
➤ ➤ CACHED [stage-1 3/10] WORKDIR /app
➤ ➤ CACHED [build 2/6] RUN apk add --no-cache gcc musl-dev libffi-dev postgresql-dev build-base
➤ ➤ CACHED [build 3/6] WORKDIR /app
➤ ➤ CACHED [build 4/6] COPY backend/requirements.txt .
➤ ➤ CACHED [build 5/6] RUN pip install --upgrade pip
➤ ➤ CACHED [build 6/6] RUN pip wheel --wheel-dir=/wheels -r requirements.txt
➤ ➤ CACHED [stage-1 4/10] COPY --from=build /wheels /wheels
➤ ➤ CACHED [stage-1 5/10] COPY backend/requirements.txt .
➤ ➤ CACHED [stage-1 6/10] RUN pip install --no-index --find-links=/wheels -r requirements.txt
➤ ➤ CACHED [stage-1 7/10] COPY --chown=appuser:appgroup --chmod=755 backend/entrypoint.sh /entrypoint.sh
➤ ➤ [stage-1 8/10] COPY backend/ ./
➤ ➤ [stage-1 9/10] RUN addgroup -S appgroup && adduser -S appuser -G appgroup
➤ ➤ [stage-1 10/10] RUN chown -R appuser:appgroup /app
➤ ➤ exporting to image
➤ ➤ exporting layers
➤ ➤ exporting manifest sha256:f463f37dbc8d8453f5f778da70f1f848457d481d5d1b9d472f75b8c694b4e2a4
➤ ➤ exporting config sha256:5aea8b6a4f73bdbl10753b8565356155d8e9b55093879c868356deb695bf398d
➤ ➤ exporting attestation manifest sha256:ed4a5bd678881754dd478d451144fd341d01c2e5c610924d398deb5d3dadcb1b
➤ ➤ exporting manifest list sha256:f36338a02e5c6ee071d82240f648324f25496b726c738643fcd603248ed0d6ec
➤ ➤ naming to docker.io/library/registro-backend:2.0.0
➤ ➤ unpacking to docker.io/library/registro-backend:2.0.0
```

```
➤ pwsh ➤ containerizacion_y_orquestacion ➤ main ➤ 72 ➤ 0ms ➤ kind-app-cluster-registro
➤ docker build -t registro-frontend:2.0.0 -f frontend/Dockerfile .
[+] Building 89.4s (17/17) FINISHED
➤ [internal] load build definition from Dockerfile
➤ ➤ transferring dockerfile: 452B
➤ [internal] load metadata for docker.io/library/node:25-alpine3.21
➤ [internal] load metadata for docker.io/library/nginx:alpine
➤ [auth] library/node:pull token for registry-1.docker.io
➤ [auth] library/nginx:pull token for registry-1.docker.io
➤ [internal] load .dockerignore
➤ ➤ transferring context: 242B
➤ [build 1/6] FROM docker.io/library/node:25-alpine3.21@sha256:54a2c8c7113949ec9b17773aaa7188529b73e2cbcf1d572e62bbe4c2e7e4df0
➤ ➤ resolve docker.io/library/node:25-alpine3.21@sha256:54a2c8c7113949ec9b17773aaa7188529b73e2cbcf1d572e62bbe4c2e7e4df0
➤ [internal] load build context
➤ ➤ transferring context: 219.49MB
➤ [stage-1 1/3] FROM docker.io/library/nginx:alpine@sha256:9dacca6709f2215cc3094f641c5b6662f7791e66a57ed034e806a7c48d51c18f
➤ ➤ resolve docker.io/library/nginx:alpine@sha256:9dacca6709f2215cc3094f641c5b6662f7791e66a57ed034e806a7c48d51c18f
➤ ➤ sha256:bdabb0d442710d667f4fd871b5fd215cc2a430a95b102bc508bf045b8e60999b 16.97MB / 16.97MB
➤ ➤ sha256:ff8a36d5502a57c3fc8eeff48e578ab433a03b1dd528992ba0d966ddf853309a 1.40kB / 1.40kB
➤ ➤ sha256:d9a55dab595458833096b28b35199099bea5eb3c68c10e99f175b12c97198d 1.21kB / 1.21kB
➤ ➤ sha256:d4f13d6ebdc834bccc63178455d406c4d25e2c2d38d2c1ab79ee5494b18e5624 403B / 403B
➤ ➤ sha256:3eaba6cd10a374d9ed629c26d76a5258e20ddfa09fcef511c98aa620dcf3fae4 955B / 955B
➤ ➤ sha256:194fa24e147df0010e146240d3b4bd25d04180c523dc717e4645b269991483e3 628B / 628B
➤ ➤ sha256:8f6a6833e95d43ac524f1f9c5e7c1316c1f3b8e7ae5ba3db4e54b0c5b910e80a 1.84MB / 1.84MB
➤ ➤ extracting sha256:8f6a6833e95d43ac524f1f9c5e7c1316c1f3b8e7ae5ba3db4e54b0c5b910e80a
➤ ➤ extracting sha256:194fa24e147df0010e146240d3b4bd25d04180c523dc717e4645b269991483e3
➤ ➤ extracting sha256:3eaba6cd10a374d9ed629c26d76a5258e20ddfa09fcef511c98aa620dcf3fae4
➤ ➤ extracting sha256:d4f13d6ebdc834bccc63178455d406c4d25e2c2d38d2c1ab79ee5494b18e5624
➤ ➤ extracting sha256:d9a55dab595458833096b28b35199099bea5eb3c68c10e99f175b12c97198d
➤ ➤ extracting sha256:ff8a36d5502a57c3fc8eeff48e578ab433a03b1dd528992ba0d966ddf853309a
➤ ➤ extracting sha256:bdabb0d442710d667f4fd871b5fd215cc2a430a95b192bc508bf945b8e60999b
➤ ➤ CACHED [build 2/6] WORKDIR /app
➤ ➤ CACHED [build 3/6] COPY frontend/package*.json ./
➤ ➤ CACHED [build 4/6] RUN npm ci --legacy-peer-deps --force
➤ ➤ CACHED [build 5/6] COPY frontend/ ./
➤ ➤ CACHED [build 6/6] RUN npm run build -- --output-path=dist/frontend/ --configuration=production
➤ ➤ [stage-1 2/3] COPY --from=build /app/dist/frontend/browser/ /usr/share/nginx/html
➤ ➤ [stage-1 3/3] COPY nginx/default.conf /etc/nginx/conf.d/default.conf
➤ ➤ exporting to image
➤ ➤ exporting layers
➤ ➤ exporting manifest sha256:8bb58377a3b8e72c0d28a84b48fce4dee1cb5542ac44f2c3be5293aab6b0632
➤ ➤ exporting config sha256:67116f27b877056a10035365b54141e65f60d8c7534108d049947b608ea858e5
➤ ➤ exporting attestation manifest sha256:9ad783d535e532048044cef4b5a743f6302ec9c87cfff023480c8fa4c106ac1e1
➤ ➤ exporting manifest list sha256:9ccbf013b90caa618672e8f269c3369a20fdecccd044d696ce7173155ea9b27
➤ ➤ naming to docker.io/library/registro-frontend:2.0.0
➤ ➤ unpacking to docker.io/library/registro-frontend:2.0.0
```

Figura 40. Nueva versión de las imágenes.

Ahora como en el paso anterior procedemos a etiquetar (tag) ambas versiones para subirlas al registro local, tanto el backend como el frontend

**`docker tag registro-backend:2.0.0 localhost:5000/registro-backend:2.0.0`**

**`docker tag registro-frontend:2.0.0 localhost:5000/registro-frontend:2.0.0`**

**`docker push localhost:5000/registro-backend:2.0.0`**

**`docker push localhost:5000/registro-frontend:2.0.0`**

```
> pwsh ➔ containerizacion_y_orquestacion > P main ≡ ?2 0ms
>> docker tag registro-backend:2.0.0 localhost:5000/registro-backend:2.0.0
> pwsh ➔ containerizacion_y_orquestacion > P main ≡ ?2 93ms
>> docker push localhost:5000/registro-backend:2.0.0
The push refers to repository [localhost:5000/registro-backend]
8148365982f7: Pushed
2d35ebdb57d9: Layer already exists
362fe14d6ada: Layer already exists
cc81617e2e93: Layer already exists
f99639dbba01: Layer already exists
97cb37001e85: Layer already exists
121657b7a077: Pushed
322a433aa229: Layer already exists
8fb00b1f653e: Pushed
3fc8f7b67707: Layer already exists
30f75d2885b1: Layer already exists
adaad550cda6: Layer already exists
22ba1c2e1d45: Layer already exists
5efd3c210112: Pushed
2.0.0: digest: sha256:f36338a02e5c6ee071d82240f648324f25496b726c738643fcd603248ed0d6ec size: 856

> pwsh ➔ containerizacion_y_orquestacion > P main ≡ ?2 0ms
>> docker tag registro-frontend:2.0.0 localhost:5000/registro-frontend:2.0.0
> pwsh ➔ containerizacion_y_orquestacion > P main ≡ ?2 93ms
>> docker push localhost:5000/registro-frontend:2.0.0
The push refers to repository [localhost:5000/registro-frontend]
d9a55dab5954: Pushed
8f6a6833e95d: Pushed
194fa24e147d: Pushed
3eaba6cd10a3: Pushed
bdabb0d44271: Pushed
df413d6ebdc8: Pushed
0ea687ed17ab: Pushed
d323bbe3a56b: Pushed
ff8a36d5502a: Pushed
2d35ebdb57d9: Layer already exists
9c41dc03ab1d: Pushed
2.0.0: digest: sha256:9ccb013b90caa618672e8f269c3369a20fdeccc0d44d696ece7173155ea9b27 size: 856
```

*Figura 41. Nueva versión de las imágenes*

Verificamos las versiones creadas con la siguiente línea de comando:

**`curl http://localhost:5000/v2/\_catalog`**

```
> pwsh ➔ containerizacion_y_orquestacion > P main ≡ ?2 0ms
>> curl http://localhost:5000/v2/_catalog
{"repositories":["registro-backend","registro-frontend"]}
```

*Figura 42. Versiones creadas*

Ahora podemos además consultar las versiones de cada repositorio, con la siguiente línea de comando:

**`curl http://localhost:5000/v2/registro-backend/tags/list`**

**`curl http://localhost:5000/v2/registro-frontend/tags/list`**

```
pwsh ➜ containerizacion_y_orquestacion > main ≡ ?2 0ms
>> curl http://localhost:5000/v2/registro-backend/tags/list
{"name":"registro-backend","tags":["1.2.0","1.0.0","1.1.0","2.0.0"]}

pwsh ➜ containerizacion_y_orquestacion > main ≡ ?2 0ms
>> curl http://localhost:5000/v2/registro-frontend/tags/list
{"name":"registro-frontend","tags":["1.2.0","1.0.0","1.1.0","2.0.0"]}
```

Figura 42. Versiones de los repositorios de las imágenes

## Despliegue de Manifiestos

Es importante mencionar que antes de aplicar los archivos de manifiesto YAML, KIND no siempre usa el mismo Daemon de Docker, así que confirmamos si las imágenes ya están visibles dentro del nodo local:

***docker exec -it kind-control-plane crictl images | findstr registro***

A continuación, se cuenta los archivos de manifiesto de Deployment para el backend (con 3 réplicas de nodo) y frontend (con 12 réplicas de nodo).

### deployment-backend.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: registro-backend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: registro-backend
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
  template:
    metadata:
      labels:
        app: registro-backend
    spec:
      containers:
        - name: backend
          image: registro-backend:2.0.0
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 8000
          envFrom:
            - secretRef:
                name: registro-secrets
```

## deployment-frontend.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: registro-frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: registro-frontend
  template:
    metadata:
      labels:
        app: registro-frontend
    spec:
      containers:
        - name: frontend
          image: registro-frontend:2.0.0
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
```

Una vez que se cuenta con los archivos de Deployment aplicamos los mismos:

***kubectl apply -f deployment-backend.yaml***

***kubectl apply -f deployment-frontend.yaml***

y verificamos que los **Pods** estén corriendo

***kubectl get deployments***

***kubectl get pods -o wide***

```
pwsh ➤ containerizacion_y_orquestacion o P main = 74 -1 0ms
➤ kubectl apply -f kind/deployment-backend.yaml
deployment.apps/registro-backend created
pwsh ➤ containerizacion_y_orquestacion o P main = 74 -1 227ms
➤ kubectl get deployments
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
registro-backend 0/3      3            0           17s
pwsh ➤ containerizacion_y_orquestacion o P main = 74 -1 136ms
➤ kubectl get pods -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP            NODE                                NOMINATED NODE   READINESS GATES
registro-backend-7fc69bfc04-czfsp    0/1     ImagePullBackOff    0       31s   10.244.1.2    app-cluster-registro-edwin-worker  <none>           <none>
registro-backend-7fc69bfc04-fjfsj    0/1     ImagePullBackOff    0       31s   10.244.1.3    app-cluster-registro-edwin-worker  <none>           <none>
registro-backend-7fc69bfc04-hwc9p    0/1     ImagePullBackOff    0       31s   10.244.2.2    app-cluster-registro-edwin-worker2  <none>           <none>
pwsh ➤ containerizacion_y_orquestacion o P main = 74 -1 124ms
➤ kubectl apply -f kind/deployment-frontend.yaml
deployment.apps/registro-frontend created
pwsh ➤ containerizacion_y_orquestacion o P main = 74 -1 184ms
➤ kubectl get deployments
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
registro-backend 0/3      3            0           3m48s
registro-frontend 0/2      2            0           5s
pwsh ➤ containerizacion_y_orquestacion o P main = 74 -1 109ms
➤ kubectl get pods -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP            NODE                                NOMINATED NODE   READINESS GATES
registro-backend-7fc69bfc04-czfsp    0/1     ImagePullBackOff    0       3m56s   10.244.1.2    app-cluster-registro-edwin-worker  <none>           <none>
registro-backend-7fc69bfc04-fjfsj    0/1     ImagePullBackOff    0       3m56s   10.244.1.3    app-cluster-registro-edwin-worker  <none>           <none>
registro-backend-7fc69bfc04-hwc9p    0/1     ImagePullBackOff    0       3m56s   10.244.2.2    app-cluster-registro-edwin-worker2  <none>           <none>
registro-frontend-84865987c6-26jql   0/1     ErrImagePull        0       13s    10.244.1.4    app-cluster-registro-edwin-worker  <none>           <none>
registro-frontend-84865987c6-b5nwc   0/1     ErrImagePull        0       13s    10.244.2.3    app-cluster-registro-edwin-worker2  <none>           <none>
```

Figura 43. Deployments corriendo con sus PODS

Ahora creamos los service tanto para el backend como para el frontend, a continuación, se detallan los contenidos respectivos de los servicios.

### service-backend.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: registro-backend-service
  labels:
    app: registro-backend
spec:
  type: LoadBalancer
  selector:
    app: registro-backend
  ports:
    - name: http
      port: 8000          # Puerto interno del servicio (el que usarán otros pods)
      targetPort: 8000    # Puerto en el contenedor backend (del Deployment)
```

### service-frontend.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: registro-frontend-service
spec:
  type: LoadBalancer
  selector:
    app: registro-frontend
  ports:
    - name: http
      protocol: TCP
      port: 80          # Puerto interno dentro del cluster
      targetPort: 80    # Puerto del contenedor
      nodePort: 30080
```

Antes de aplicar los manifiestos de tipo service, falta habilitar el **LoadBalancer** y para esto se ha instalado MetalLB en el cluster de KIND.

Se ha expuesto el frontend en el host usando Service + LoadBalancer (con MetalLB), porque KIND no tiene LoadBalancer nativo, pero sí puede simularlo usando MetalLB.

MetalLB permite que type: LoadBalancer funcione en ambientes locales.

**kubectl apply -f <https://raw.githubusercontent.com/metallb/metallb/v0.14.3/config/manifests/metallb-native.yaml>**

```

pws@pws:~/contenedorizacion_y_orquestacion$ kubectl apply -f https://raw.githubusercontent.com/metallb/metallb/v0.14.3/config/manifests/metallb-native.yaml
namespace/metallb-system created
Warning: unrecognized format "int32"
customresourcedefinition.apiextensions.k8s.io/bfdprofiles.metallb.io created
customresourcedefinition.apiextensions.k8s.io/bgppeeradvertisements.metallb.io created
customresourcedefinition.apiextensions.k8s.io/bgppeers.metallb.io created
customresourcedefinition.apiextensions.k8s.io/communities.metallb.io created
customresourcedefinition.apiextensions.k8s.io/ipaddresspools.metallb.io created
customresourcedefinition.apiextensions.k8s.io/l2advertisements.metallb.io created
serviceaccount/controller created
serviceaccount/speaker created
role.rbac.authorization.k8s.io/controller created
role.rbac.authorization.k8s.io/pod-lister created
clusterrole.rbac.authorization.k8s.io/metallb-system:controller created
clusterrole.rbac.authorization.k8s.io/metallb-system:speaker created
rolebinding.rbac.authorization.k8s.io/controller created
rolebinding.rbac.authorization.k8s.io/pod-lister created
clusterrolebinding.rbac.authorization.k8s.io/metallb-system:controller created
clusterrolebinding.rbac.authorization.k8s.io/metallb-system:speaker created
configmap/metallb-excludel2 created
secret/webhook-server-cert created
service/webhook-service created
deployment.apps/controller created
daemonset.apps/speaker created
validatingwebhookconfiguration.admissionregistration.k8s.io/metallb-webhook-configuration created

```

Figura 44. Instalación de **MetallB**

A continuación, podemos observar la cantidad de PODS que están corriendo con metallLB.

***kubectl get pods -n metallb-system***

```

pws@pws:~/contenedorizacion_y_orquestacion$ kubectl get pods -n metallb-system
NAME                                READY   STATUS    RESTARTS   AGE
controller-7cf7f554b6-p4bsj        1/1     Running   0           19m
speaker-4mdbh                       1/1     Running   0           19m
speaker-gdgvf                      1/1     Running   0           19m
speaker-hqk6r                      1/1     Running   0           19m

```

Figura 45. Pods corriendo para metallB

Ahora se procede a configurar el archivo LoadBalancer con el rango de IP:

### Loadbalancer.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: frontend-loadbalancer
spec:
  selector:
    app: registro-frontend
  ports:
    - name: http
      port: 80
      targetPort: 80
  type: LoadBalancer

apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: ip-pool

```



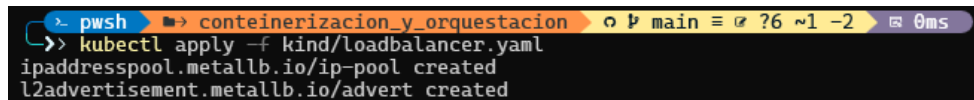
```

  namespace: metallb-system
spec:
  addresses:
    - 192.168.0.200-192.168.0.250    # rango IP de docker network
---
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: advert
  namespace: metallb-system

```

Ahora aplicamos a KIND la configuración de LoadBalancer.

***kubectl apply -f kind/loadbalancer.yaml***



```

➤ pwsh ➤ containerizacion_y_orquestacion ➤ main ➤ 76 ~1 -2 ➤ 0ms
➤ kubectl apply -f kind/loadbalancer.yaml
ipaddresspool.metallb.io/ip-pool created
l2advertisement.metallb.io/advert created

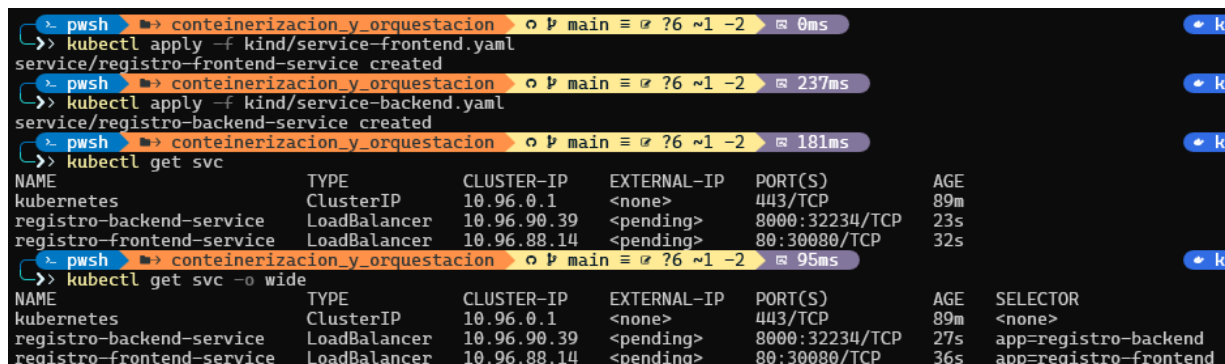
```

*Figura 46. LoadBalancer aplicado*

Ahora aplicamos los servicios del backend y frontend

***kubectl apply -f kind/service-frontend.yaml***

***kubectl apply -f kind/service-backend.yaml***



```

➤ pwsh ➤ containerizacion_y_orquestacion ➤ main ➤ 76 ~1 -2 ➤ 0ms
➤ kubectl apply -f kind/service-frontend.yaml
service/registro-frontend-service created
➤ pwsh ➤ containerizacion_y_orquestacion ➤ main ➤ 76 ~1 -2 ➤ 237ms
➤ kubectl apply -f kind/service-backend.yaml
service/registro-backend-service created
➤ pwsh ➤ containerizacion_y_orquestacion ➤ main ➤ 76 ~1 -2 ➤ 181ms
➤ kubectl get svc

```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	89m
registro-backend-service	LoadBalancer	10.96.90.39	<pending>	8000:32234/TCP	23s
registro-frontend-service	LoadBalancer	10.96.88.14	<pending>	80:30080/TCP	32s

```

➤ pwsh ➤ containerizacion_y_orquestacion ➤ main ➤ 76 ~1 -2 ➤ 95ms
➤ kubectl get svc -o wide

```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE	SELECTOR
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	89m	<none>
registro-backend-service	LoadBalancer	10.96.90.39	<pending>	8000:32234/TCP	27s	app=registro-backend
registro-frontend-service	LoadBalancer	10.96.88.14	<pending>	80:30080/TCP	36s	app=registro-frontend

*Figura 47. Servicios aplicados*

## 8. Buenas prácticas implementadas

- Dockerfiles optimizados en varias etapas.
- Imágenes base ligeras (alpine) por temas de practicidad.
- Variables de entorno y secretos (Docker secrets y Kubernetes Secrets).
- Volúmenes persistentes para Postgres.
- Red personalizada en Docker Compose y Swarm (overlay).
- Versionamiento de imágenes: registro/backend:1.0.0, registro/frontend:1.0.0.