Despliegue De Ambientes

En la documentación anterior vimos cómo darle host a un proyecto sencillo, este tenía únicamente el scaffolding de un proyecto y un "hola mundo" formateado en json.

Ahora veremos cómo desplegar un ambiente entero, backend, frontend y base de datos, antes de comenzar vamos a hablar de qué tecnologías estaremos usando:

Backend: FastAPI
Frontend: Vue.js
Base De Datos: PostgreSql

> **Primeros pasos a nuevas tecnologías**
>
> Si no estás familiarizado con estas herramientas podrías revisar la documentación de cada una para entender un poco más del ambiente de trabajo, pues en este documento veremos algunos conceptos que pueden ser medio-avanzado.

Zona de enlaces a documentación externa.

***Backend: https://fastapi.tiangolo.com/tutorial/
Frontend: https://vuejs.org/guide/introduction.html
Base De Datos: https://www.postgresql.org/docs/

Nuestros primeros pasos

A partir de este punto supondremos que contamos con los conocimientos previos requeridos para generar un proyecto completo en FastAPI, este documento no está solamente orientado a un tipo de proyecto, si no busca ser la base para que puedas generar tus propios desarrollos, y usar esta documentación cómo un escalón para conocer cómo podrías desplegar tu ambiente en un entorno productivo.

¿Qué es lo que veremos?

Vamos a ver la importancia de ***Manejadores de carga, volumenes***, ya que en la documentación anterior no vimos esto, debido a que en ese caso específico no era necesario, pero ahora vamos a manejar conceptos cómo:

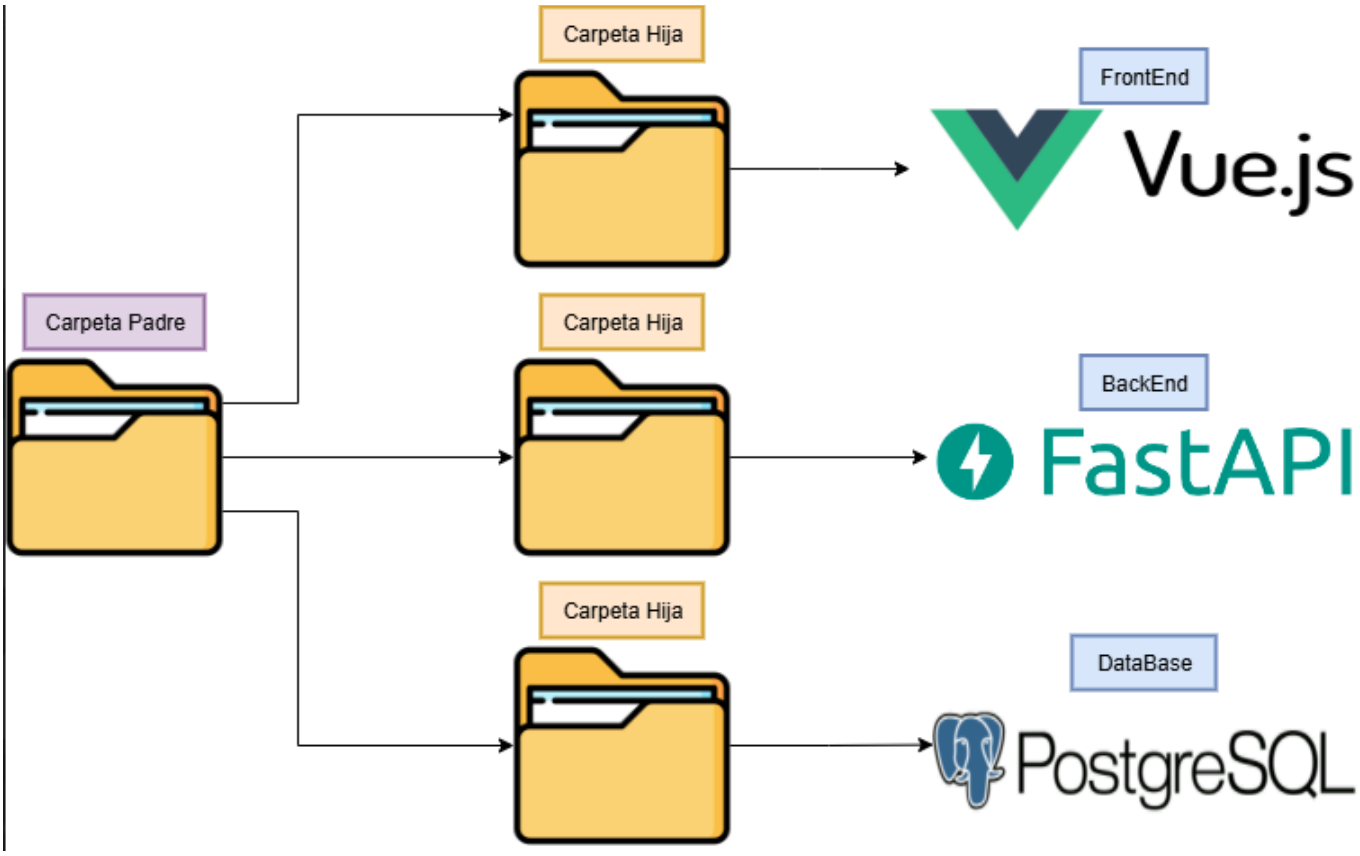
1. Persistencia de datos
2. Contenidos elásticos
3. Principios de trabajo colaborativo en entornos productivos
4. Github Actions

Si no conocemos todos estos conceptos, no te preocupes, paso a paso vamos a ir viendo qué, cómo y cuándo los vamos a aplicar, por eso te invitamos a revisar con detenimiento cada sección antes de empezar a seguir cada paso que detallemos.

Principios De Construcción De Ambiente

Uno de los primeros pasos para subir un ambiente completo que nos permita tener una modularidad completa es entender cómo funciona este mismo.

Supongamos que tenemos una carpeta principal, esta tiene dentro 3 carpetas y cada una está orientada a contener un subproyecto, por ejemplo:



Aunque es importante que especifiquemos que la imagen de la base de datos no necesariamente tendría que ir en nuestra carpeta padre, podemos incluirla en nuestro docker-compose, pero podemos usar esa carpeta para estar almacenando los backups de nuestra base de datos y permitir un histórico más claro de lo que estamos haciendo o está sucediendo en la aplicación.

> **Aclaraciones**
>
> Aunque vemos que todo nuestro ambiente vive dentro de un repositorio raíz es importante recalcar que cada subcarpeta representa un contenedor independiente, el cuál será construido y ejecutado de manera aislada en nuestro docker-compose, por lo que cada una tendrá su propia imagen

---

¿Qué es docker-compose?

`docker-compose` es una herramienta que nos permite definir y ejecutar múltiples contenedores Docker desde un solo archivo, usualmente llamado `docker-compose.yml`.

Este archivo actúa como una especie de **pipeline de construcción y despliegue**, permitiéndonos:

- Declarar múltiples servicios (backend, frontend, db, etc.)
- Configurar redes, volúmenes, variables de entorno y dependencias entre contenedores
- Automatizar el arranque completo de un ambiente con un simple `docker-compose up`

Si queremos conocer más acerca de cómo funciona podemos darle un vistazo a la documentación oficial de docker: https://docs.docker.com/compose/

> **¿Aún con dudas?**
>
> Es normal si no terminamos de entender este concepto a la primera, pero si prestamos atención a la arquitectura base de la comunicación entre usuarios - aplicación web - servidor - base de datos podemos notar cómo es que nuestra configuración cobra vida.

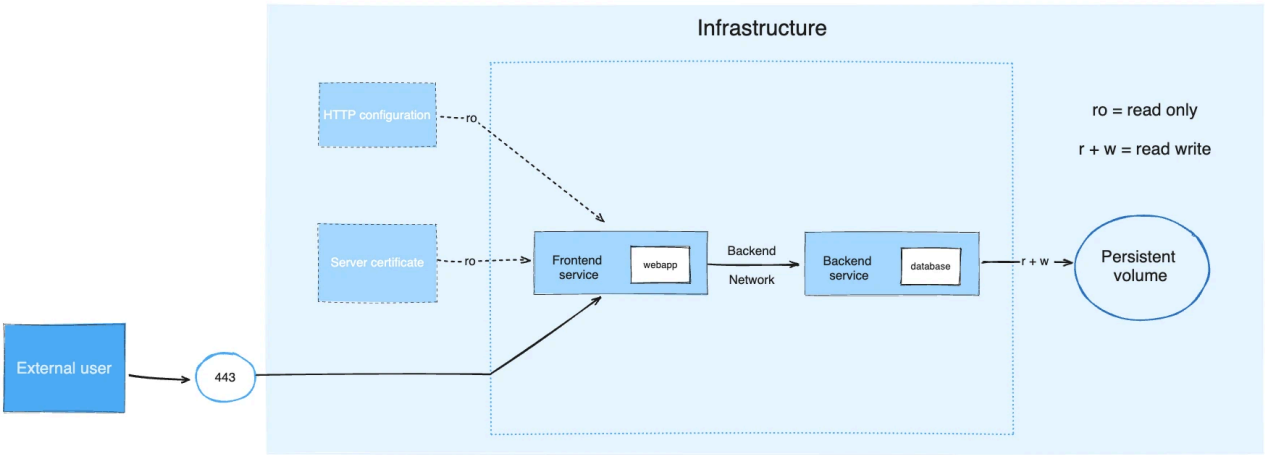


Imagen recuperada de la documentación de docker

Y el mismo ejemplo que nos deja docker:

```
services:
  frontend:
    image: example/webapp
    ports:
      - "443:8043"
    networks:
      - front-tier
      - back-tier
    configs:
```

```
      - httpd-config
    secrets:
      - server-certificate

  backend:
    image: example/database
    volumes:
      - db-data:/etc/data
    networks:
      - back-tier

volumes:
  db-data:
    driver: flocker
    driver_opts:
      size: "10GiB"

configs:
  httpd-config:
    external: true

secrets:
  server-certificate:
    external: true

networks:
  # The presence of these objects is sufficient to define them
  front-tier: {}
  back-tier: {}
```

Aquí nosotros estamos declarando un ambiente compuesto por tres servicios independientes (`frontend` , `backend` y `db`), cada uno basado en una imagen diferente.
Además, se utilizan volúmenes para la persistencia, redes privadas para separar tráfico interno, y configuración externa como certificados o archivos de configuración HTTP.

¿Qué es un volumen?

Hemos estado hablando de los volumenes como si fueran una herramienta mágica, pero qué son realmente?, son mecanismos que nos permiten persistir datos generados y utilizados por contenedores, son muy útiles en caso de que un servicio llegue a fallar, nosotros podríamos tener un punto al cuál poder hacer un rollback.

Cómo pudimos ver en la imagen ilustrativa anterior cada volumen puede contar con permisos asociados que sean específicos, si queremos que la aplicación solo pueda leerlos o si pueda interactuar de manera directa con ellos, por lo que cuándo nosotros lo configuremos dentro de nuestro docker-compose podemos hacer:

1. Indicar dónde se van a guardar
2. Indicar el tipo de volumen
3. Indicar los permisos asociados

¿Cuáles son los tipos de volumenes?

Existe más de un tipo de volumen y estos tienen diferentes usos:

Tipo	Descripción
Anónimos	Se crean sin nombre específico y se eliminan automáticamente.
Nombrados	Tienen un nombre identificable y sobreviven entre reinicios y recreaciones.
Montajes bind	Vinculan una carpeta del host directamente dentro del contenedor.

Montando nuestro docker-compose

Ya que hemos visto los ejemplos básicos y toda la teoría detrás de un ambiente y lo que implica su creación, ahora podremos empezar a montar nuestro propio docker-compose.

Docker

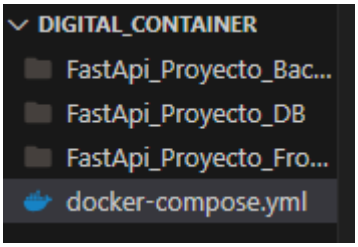
Es importante que nosotros estemos seguros de contar con docker-compose, si estamos en windows y tenemos docker desktop es bastante probable que ya venga con compose, por lo que podremos lanzar el comando `docker compose version`. En cambio si estamos en linux deberemos instalarlo con:

```
sudo apt update
sudo apt install docker-compose-plugin -y
```

```
docker compose version
```

Paso 1: Crear el .yml

Vamos a ir a la ruta principal de nuestra carpeta padre y haremos `touch docker-compose.yml`



En caso de que estemos trabajando desde windows podemos crear de manera manual nuestro archivo.

Paso 2: Configurar nuestro .yml

Si tenemos correctamente creada nuestra estructura de carpetas podemos definir las rutas específicas en nuestro archivo, para conocer si configuramos de manera correcta esta parte haremos un ls:

```
PS C:\Users\x\Desktop\Digital_Container> ls


    Directorio: C:\Users\x\Desktop\Digital_Container


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----     09/07/2025  09:04 p. m.                FastApi_Proyecto_BackEnd
d-----     18/07/2025  10:07 a. m.                FastApi_Proyecto_DB
d-----     18/07/2025  10:08 a. m.                FastApi_Proyecto_FrontEnd
-a----     18/07/2025  11:36 a. m.              0 docker-compose.yml
```

Veremos que tenemos creadas las carpetas:

1. FastApi_Proyecto_BackEnd
2. FastApi_Proyecto_DB
3. FastApi_Proyecto_FrontEnd

Por lo que la configuración de nuestro documento debería ser algo parecida a:

```
version: '3.8'

services:
  backend:
    build: ./FastApi_Proyecto_BackEnd
    ports:
      - "8000:8000"
    environment:
      - DATABASE_URL=postgresql://user:password@db:5432/appdb
    depends_on:
      - db

  frontend:
    build: ./FastApi_Proyecto_FrontEnd
    ports:
      - "3000:80"

  db:
    image: postgres:14
    restart: always
    environment:
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=password
      - POSTGRES_DB=appdb
    volumes:
      - db_data:/var/lib/postgresql/data

volumes:
  db_data:
```

Seguridad de los datos

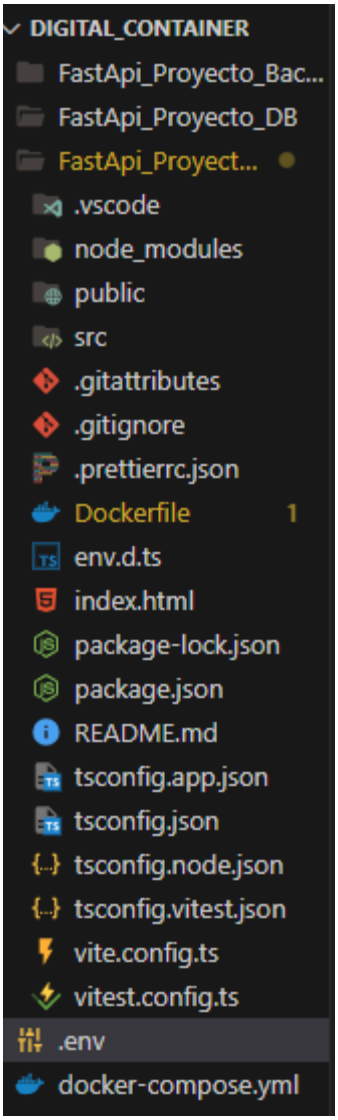
Normalmente debemos hacer una configuración .env para guardar nuestras credenciales, evitando la exposición de estas mismas en repositorios públicos, por lo que deberemos crear un .env y luego re configurar nuestro docker-file.

Paso 3: Configurando nuestro .env

En nuestra carpeta raíz a la misma altura que tenemos nuestro docker-compose vamos a crear un archivo de la siguiente manera:

```
touch .env
```

O en caso de que estemos en Windows lo crearíamos como un archivo .env, una vez que lo tenemos creado usaremos una estructura base.



La estructura base que vamos a estar manejando para este ejemplo será la siguiente:

```
POSTGRES_USER=admin
POSTGRES_PASSWORD=supersecret123
POSTGRES_DB=digital_container
```

Consideraciones

Estas variables se usarán para inicializar la base de datos solo **la primera vez** que el contenedor se cree.
Si ya existe el volumen `db_data`, PostgreSQL **no recreará la base de datos ni cambiará la contraseña** a menos que borres ese volumen manualmente.

Paso 4: Configurando nuestro PostgreSql

En el paso anterior pasamos nuestras credenciales a variables de entorno, por lo que en lugar de pasar en seco estas mismas ahora podemos hacer uso de:

```
version: '3.8'

services:
  backend:
    build: ./FastApi_Proyecto_BackEnd
    ports:
      - "8000:8000"
    env_file:
      - .env
    environment:
      -
DATABASE_URL=postgresql://${POSTGRES_USER}:${POSTGRES_PASSWORD}@db:5432/${POST
GRES_DB}
    depends_on:
      - db

  frontend:
    build: ./FastApi_Proyecto_FrontEnd
    ports:
      - "3000:80"

  db:
    image: postgres:14
    restart: always
    env_file:
      - .env
    volumes:
      - db_data:/var/lib/postgresql/data
```

```
volumes:
  db_data:
```

¿Cómo funciona la imagen de Postgres?

Cuando nosotros levantamos el contenedor con Docker, se basa en la imagen oficial `postgres:14`, pero lo que realmente se ejecuta es un **contenedor** que corre PostgreSQL.

Este contenedor guarda los datos de la base de datos en un **volumen externo** (`db_data`), el cual se encuentra **fuera del contenedor**, directamente en el sistema de archivos del host (ya sea tu máquina local o el servidor).

> Esto significa que aunque apagues o elimines el contenedor, **los datos se mantienen intactos** gracias al volumen persistente.

Por eso, cada vez que volvemos a levantar el servidor con:

```
docker compose up -d
```

- Se puede crear un nuevo contenedor (si el anterior fue eliminado).
- Pero **no se recrea la imagen ni se pierden los datos**, ya que Docker reutiliza el volumen `db_data`.
- Las credenciales declaradas en el archivo `.env` solo serán usadas la **primera vez** que se cree el volumen y PostgreSQL inicialice.

¿Quieres reiniciar todo desde cero?

Entonces deberás eliminar manualmente el volumen con:

```
docker volume rm digital_container_db_data
```

Conectando nuestro backend a PostgreSQL

Antes de hacer cualquier conexión debemos saber exactamente qué estamos conectando?.

Debido a la estructura que tenemos en nuestro proyecto, una carpeta padre que tiene carpetas específicas para back, front y la data de nuestra DB, podemos generar una especie de intranet

dentro de nuestro ambiente, esto nos va a permitir comunicar nuestros servicios para poder hacer un funcionamiento más optimizado.
Una vez que hemos entendido esto, vamos a hablar de para qué sirve el DATABASE_URL.

¿Qué es DATABASE_URL?

Es una variable de entorno cuya información le ayudará al backend a conectarse a nuestra instancia de base de datos, pero no todo es color de rosa, debemos instalar una librería para poder hacer una conexión asíncrona o síncrona según nuestras necesidades.

Por eso es importante que la información de esta URL tenga:

1. Usuario
2. Contraseña
3. Hostname o nombre del servicio (`db` en este caso)
4. Puerto de conexión
5. Nombre de la base de datos.

Por ejemplo

```
postgresql://admin:supersecret123@db:5432/digital_container
```

¿Cómo usamos esa variable dentro de FastAPI?

Paso 1: Instalar las dependencias

Vamos a agregar algunas dependencias a nuestro documento de requerimientos, ubicado en nuestra carpeta de backend.

```
    Directorio: C:\Users\x\Desktop\Digital_Container\FastApi_Proyecto_BackEnd


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        10/07/2025  12:42 p. m.              .venv
d-----        09/07/2025  12:35 p. m.              app
d-----        09/07/2025  12:07 p. m.              data
d-----        09/07/2025  12:08 p. m.              notebooks
d-----        09/07/2025  12:07 p. m.              tests
d-----        30/06/2025  11:30 a. m.              __pycache__
-a----        09/07/2025  12:18 p. m.            665 .gitignore
-a----        17/07/2025  10:31 a. m.            393 Dockerfile
-a----        09/07/2025  09:04 p. m.           4858 README.md
-a----        09/07/2025  09:04 p. m.            633 requirements.txt
```

El requerimiento que vamos a tener es:

```
sqlalchemy
psycopg2-binary
```

> **Consideraciones**
>
> Esta documentación se basa más en desplegar un ambiente en un entorno productivo cómo aws, por lo que usaremos ejemplos prácticos, por lo que no veremos a profundidad el desarrollo de un proyecto en específico, si no, las bases necesarias para hacer que todo se pueda conectar y desplegar

Paso 2: Crear el archivo de conexión db.py

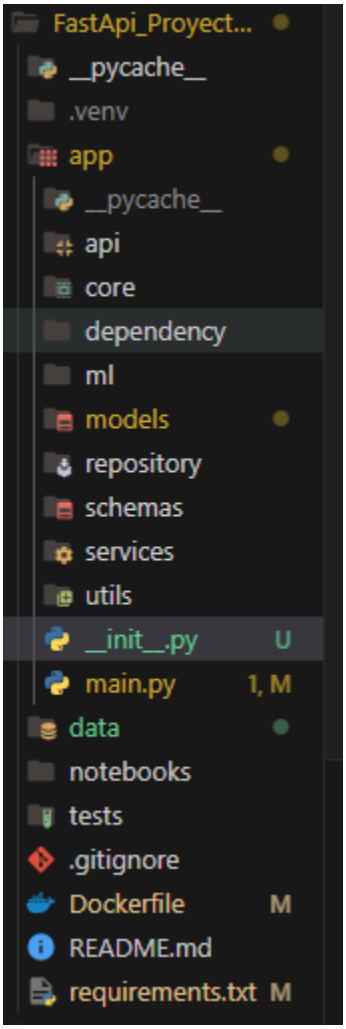
Para que podamos hacer el enlace de nuestra aplicación a una base de datos requerimos tener una serie de archivos dentro de nuestro proyecto, tales cómo:

```
1. db.py
2. __init__.py
```

Siendo que el primero será nuestra conexión a la base de datos y el segundo será el archivo que se encargue de decir que es contamos con un paquete válido para python.

> **Consejo**

Si no contamos de forma automática con un archivo que contenga ese init, podemos crearlo de manera manual y dejarlo vacío



Una vez que tenemos esto cubierto, ahora vamos a crear el db.py con:

```
touch db.py
```

O creándolo desde el editor de código que estemos manejando, este documento va a recuperar nuestra URL que previamente declaramos en nuestro archivo .env, por lo que la estructura que vamos a manejar debería ser algo cómo:

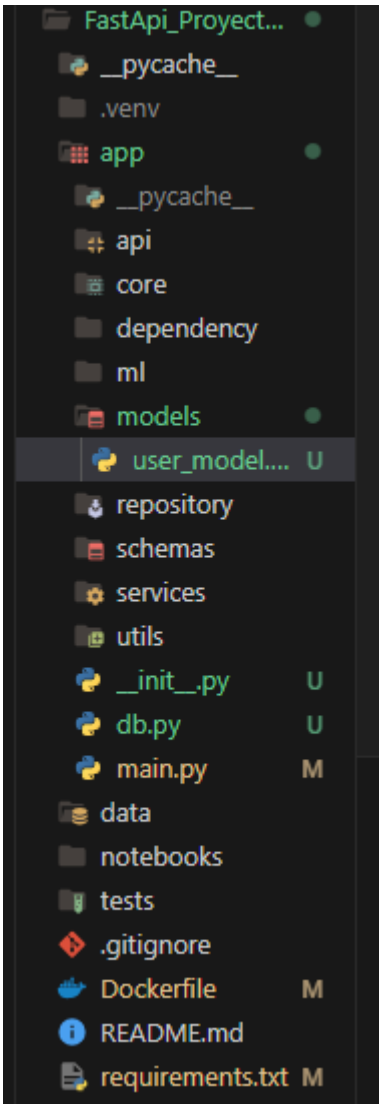
```
import os
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
```

```
DATABASE_URL = os.getenv("DATABASE_URL")
engine = create_engine(DATABASE_URL)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()
```

Ahora que tenemos esto, podemos hacer que al iniciar la aplicación se conecte de forma automática a nuestra base de datos y construya las tablas que nosotros necesitemos, esto gracias a modelos previamente establecidos, y configuraciones mínimas.

Consideraciones

Al final de crear nuestro archivo db.py dentro de la carpeta app, nuestro proyecto debería verse de la siguiente manera:



Paso 3: Conectar main.py con el db.py

Ahora deberemos conectar nuestro archivo main con la configuración de la base de datos, para eso bastará con que nosotros hagamos algo cómo:

```
from fastapi import FastAPI
from . import models
from .db import engine

app = FastAPI()

models.Base.metadata.create_all(bind=engine)

@app.get("/")
def index():
    return {"mensaje": "¡Conexión exitosa con PostgreSQL!"}
```

Con esto estamos importando el .db que es nuestra configuración a la base de datos y con la línea:

```
models.Base.metadata.create_all(bind=engine)
```

Estamos creando todos los modelos que nosotros hayamos específicado.

Verificando nuestra conexión

Ya hemos hecho la conexión a nuestra base de datos, ahora debemos probar que esta funcione de manera correcta, ¿Cómo lo lograremos?, vamos primero a crear un modelo.

Creando nuestro propio modelo

Si prestamos atención a la captura anterior, pudimos ver que dentro de app/models, contamos con un archivo llamado ***user_models.py***, En este creamos una pequeña tabla de usuarios:

```
from sqlalchemy import Column, Integer, String
from app.db import Base

class Usuario(Base):
```

```
    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, index=True)
    email = Column(String, unique=True, index=True)
```

Consideraciones

Aquí estamos utilizando `SQLAlchemy` para definir una tabla llamada `"users"` con sus respectivas columnas. Este modelo será usado para que PostgreSQL cree automáticamente la estructura cuando levantemos la aplicación.

Modificando nuestro main.py

Dado que ahora el modelo no se encuentra directamente en la raíz de la carpeta `app/`, sino dentro de una subcarpeta, haremos un pequeño ajuste para importar correctamente la clase `Base` (la base de todos nuestros modelos SQLAlchemy):

```
from fastapi import FastAPI
from .models.user_model import Base
from .db import engine

app = FastAPI()

Base.metadata.create_all(bind=engine)

@app.get("/")
def index():
    return {"mensaje": "¡Conexión exitosa con PostgreSQL!"}
```

Con esto estamos iterando de manera directa sobre la carpeta modelos y obteniendo como base el user_model.

Entornos productivos

Con esto ya hemos asentado las bases para verificar que todo esté funcionando de manera correcta.

Sin embargo, si realmente queremos visualizar que lo que hicimos está corriendo, es común que implementemos endpoints similares a los *health checks*, que permiten saber si el servidor está respondiendo correctamente.
De esta forma, se **limita el acceso** al servidor a casos específicos, evitando conexiones innecesarias solo para comprobar el estado de la base de datos o del backend. Además, esto nos permite tener un log en tiempo real del estado del proyecto sin necesidad de entrar en configuraciones adicionales, terminales o lugares que puedan dejar vulnerable nuestra aplicación.

Con esto en mente, vamos a desarrollar un endpoint que nos permita verificar si la base de datos creó correctamente nuestro modelo.

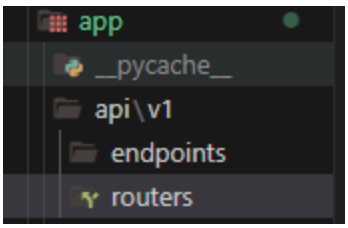
Creando mi endpoint

> **NOTA**
>
> Como se especificó anteriormente, esta documentación no se enfoca en desarrollar un proyecto desde cero, sino en explicar lo **mínimo requerido para desplegar un entorno en Docker**.
> Aun así, consideramos importante incluir algunas bases para personas que están aprendiendo desde cero. Por lo tanto, este apartado es completamente opcional de leer.

Normalmente, en proyectos pequeños es común ver que los endpoints se definen directamente en el archivo `main.py`. Sin embargo, esta lógica debería separarse para mantener un proyecto limpio y ordenado.

Por eso, dentro de la carpeta `app/api/v1`, tendremos dos subcarpetas:

- `endpoints/`: Contiene los controladores (handlers) reales.
- `routers/`: Encapsula y registra esos endpoints en rutas agrupadas.



Endpoint - /db-ping

Cómo primer paso dentro de nuestra carpeta `app/api/v1/endpoints` vamos a crear un archivo llamado `dp_ping.py`, este contendrá toda la lógica de nuestro endpoint.

```
from fastapi import APIRouter
from sqlalchemy.exc import OperationalError
from app.db import SessionLocal

router = APIRouter()

@router.get("/db-ping", tags=["health"])
def db_ping():
    try:
        db = SessionLocal()
        db.execute("SELECT 1")
        db.close()
        return {"db_status": "conexión exitosa"}
    except OperationalError:
        return {"db_status": "fallo de conexión"}
```

Una vez que hayamos creado nuestra lógica, vamos a pasarlo al router, el cuál es el encargado de centralizar todos los endpoints que nosotros estemos creando.

Router

```
from fastapi import APIRouter
from app.api.v1.endpoints import db_ping

router = APIRouter()
router.include_router(db_ping.router)
```

Main.py

Una vez que tengamos todos nuestros modelos creados, que hayamos declarado todos nuestros endpoints y hayamos centralizado el enrutamiento de estos, bastará con agregar las siguientes líneas a nuestro archivo main:

```
from app.api.v1.routers.router import router

app.include_router(router)
```

De este modo nuestro Main debería verse cómo la siguiente imagen de referencia:

Y si nosotros accedemos a nuestra ruta `http://localhost:8000/db-ping` deberíamos ver un mensaje de conexión exitosa

---

Empaquetado de nuestra aplicación

Ahora que hemos configurado nuestra aplicación, hemos conectado la base de datos y hemos creado nuestros primeros EndPoints, lo que sigue es encender nuestro primer contenedor que será el Backend.

> **CONSIDERACIONES**
>
> Aunque esta parte la hemos estado manejando a mano, más adelante veremos cómo levantar los servicios de manera simultanea, para automatizar todo este proceso.

Si todo ha salido de manera correcta tendremos que lanzar el comando:

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

Este comando está usando el compose que creamos para levantar el servicio especificado en el apartado de backend.

Cómo vemos este apartado tiene el `depends_on` que dice que nuestro servicio depende de que otro se haya creado, por lo que primero se estaría creando nuestra base de datos, sin embargo, esto no garantiza que se termine de configurar antes de disparar nuestro backend, por lo que bajo ciertos casos, es probable que pese a que se cree la base de datos, el backend se apague de forma automática, por no poder conectarse en ese momento, pero eso no quiere decir que tengamos algo mal configurado, si no que no se ejecutó el orden específico.

¿Cómo lo solucionamos?

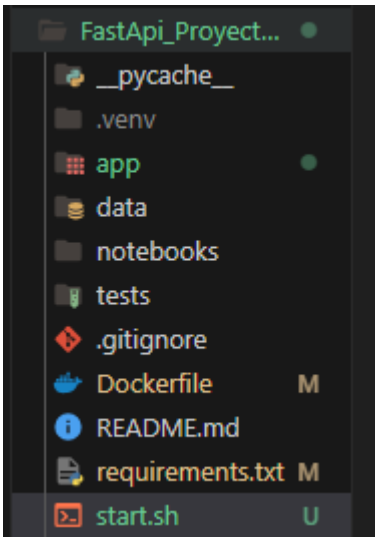
Bastará con que nosotros creemos un archivo de referencia para un inicio seguro, este es similar a configuraciones que a veces manejamos cuándo usamos Nginx o servicios para levantar nuestros entornos, para eso en nuestra carpeta raíz dónde tenemos el backend, haremos:

```
touch start.sh
```

O lo crearemos dentro de nuestro editor de código, una vez que tengamos este archivo en una dirección parecida a:

`Directorio: C:\Users\x\Desktop\Digital_Container\FastApi_Proyecto_BackEnd`

Al mismo nivel dónde tenemos nuestro Dockerfile y requirements.txt.



Aquí vamos a agregar una configuración parecida a la siguiente:

```
echo "Esperando a PostgreSQL (db:5432)..."
while ! nc -z db 5432; do
  sleep 1
done

echo "PostgreSQL está listo, iniciando FastAPI..."
uvicorn app.main:app --host 0.0.0.0 --port 8000
```

Con esta parte estamos diciendo que mientras el puerto 5432 no esté preparado para recibir conexiones, no corramos la instancia de nuestro backend.
Para terminar la configuración vamos a editar nuevamente nuestro Dockerfile para cambiar nuestro CMD a:

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

En este caso nuestro Dockerfile debería verse de la siguiente manera:

```
FROM ubuntu:22.04

ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update && \
    apt-get install -y python3 python3-pip python3-venv tesseract-ocr libgl1
netcat && \
    apt-get clean
```

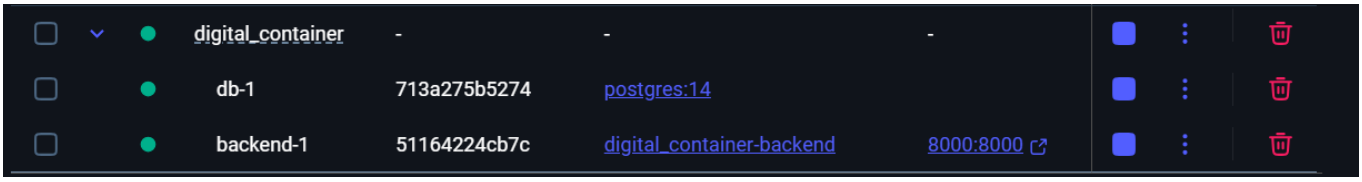
```
WORKDIR /app

COPY . /app

RUN pip3 install --upgrade pip
RUN pip3 install --no-cache-dir -r requirements.txt

COPY start.sh /start.sh
RUN chmod +x /start.sh

EXPOSE 8000

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

Ahora al final deberemos volver a correr el comando:

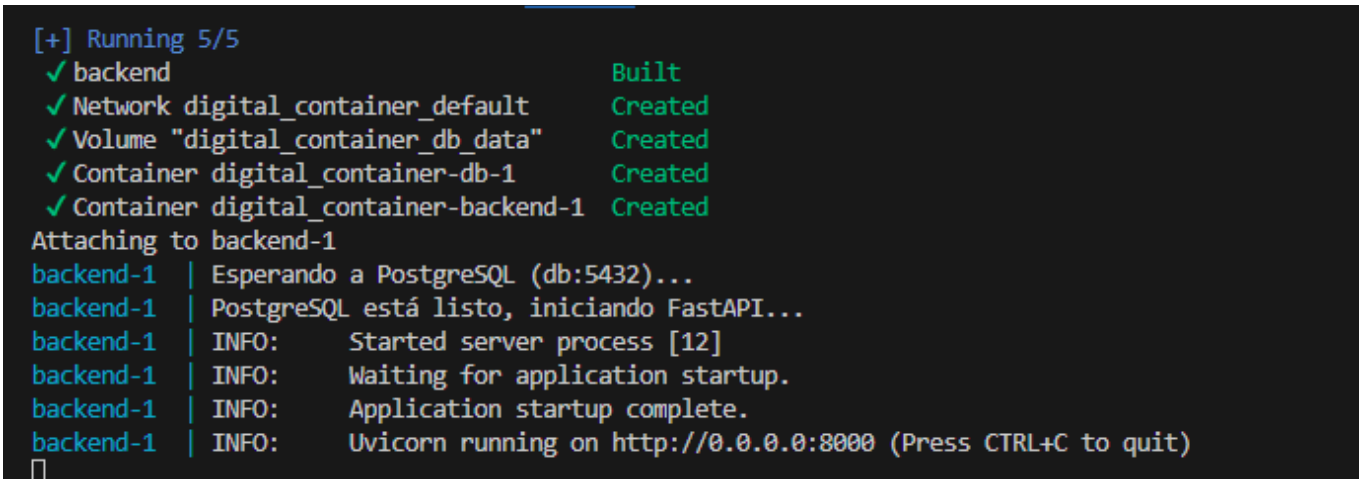
```
docker compose down -v
docker compose up backend --build
```

Al finalizar, si hemos configurado todo de manera correcta veremos en nuestro Docker Desktop que se creó nuestro contenedor con:

1. DB (PostgreSQL)
2. Backend (FastAPI)



Y en nuestra terminal veremos:



HOST

Aunque veamos que está corriendo en 0.0.0.0, no quiere decir que tenemos que entrar a esa ruta, podemos acceder desde localhost o cualquier otro host, siempre y cuándo mantengamos el prefijo del puerto 8000

---

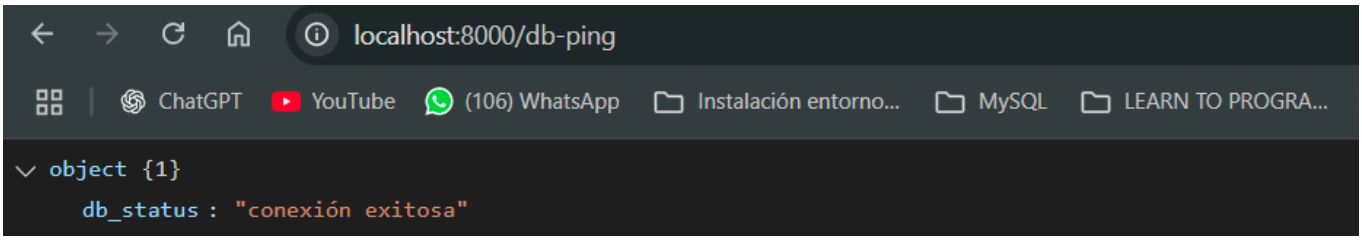
db-ping

Algunas versiones de sqlalchemy no permiten directamente pasar el texto sin envolverlo primero, por lo que si nosotros al acceder a la ruta `localhost:8000/db-ping` vemos error de servidor podemos verificar nuestro endpoint `db_ping-py`:

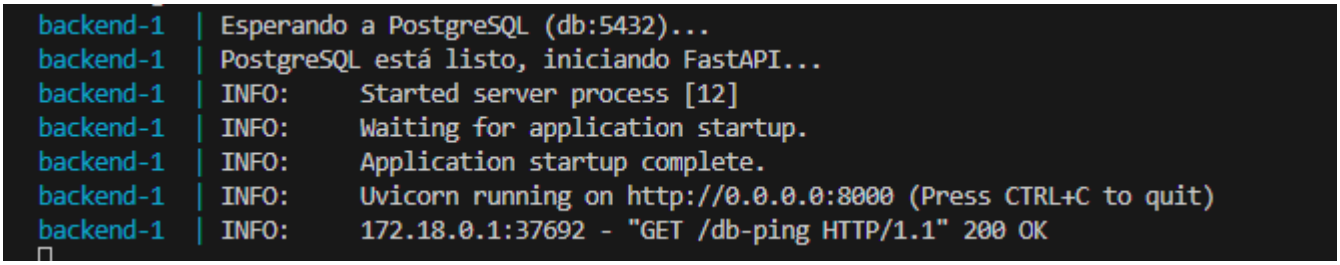
```
from fastapi import APIRouter
from sqlalchemy.exc import OperationalError
from sqlalchemy import text
from app.db import SessionLocal

router = APIRouter()

@router.get("/db-ping", tags=["health"])
def db_ping():
    try:
        db = SessionLocal()
        db.execute(text("SELECT 1"))
        db.close()
        return {"db_status": "conexión exitosa"}
```

```
    except OperationalError:
        return {"db_status": "fallo de conexión"}
```

Con esto deberíamos ser capaces de ver el ping de nuestra base de datos, en el navegador veríamos:



Y en nuestra consola:



Empaquetando el Frontend

Ya que nuestro Back y la Base de Datos están conectados de manera correcta, ahora veremos las configuraciones básicas para poder enlazar nuestro front a estos, de manera que podamos lanzar un solo comando y esto ya esté desplegando nuestros 3 servicios, podemos verlo como el punto de inicio antes de empezar a ver cómo podríamos automatizarlo y subirlo a nuestro dominio haciendo uso de AWS.

Tenemos que entender que la parte fundamental de comunicación vía HTTP dentro del contenedor se basa en las imágenes que creemos dentro de nuestro contenedor, al ser un proyecto semi-desacoplado contamos con separación de responsabilidades, por lo que necesitamos.

1. Interpretador.
2. Manejador de rutas estáticas.

A qué nos referimos con esto?, es que el `Interpretador` será nuestro archivo encargado de pasar las peticiones HTTP de un punto a otro, aquí podemos fijar nuestro "vite.config.ts", de forma nativa vite tiene un interpretador, pero en casos de producción este ya no nos es de

utilidad, por lo cuál tenemos que pasarle una manera de intérprete haciendo uso de un proxy, pero por si solo no puede ser el intérprete, por lo cuál vamos a pasar al `Manejador de rutas estáticas`, es aquí dónde decimos que parte será estática y cuál debe ser dinámica.

En nuestro front vamos a crear un archivo `nginx.conf` este deberá contener una estructura similar a la siguiente.

```
server {

  listen 80;
  server_name localhost;
  root /usr/share/nginx/html;
  index index.html;

  location / {
    try_files $uri $uri/ /index.html;
  }



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

}
```

Lo que estamos haciendo es que en nuestro servidor servimos las vistas de manera estática, como archivos que se precargan en un index.html, para que el usuario interactué con ellas desde el navegador y cuándo se introduzca una ruta cómo por ejemplo /api, se dispare un proxy que pase esa parte a nuestro backend.

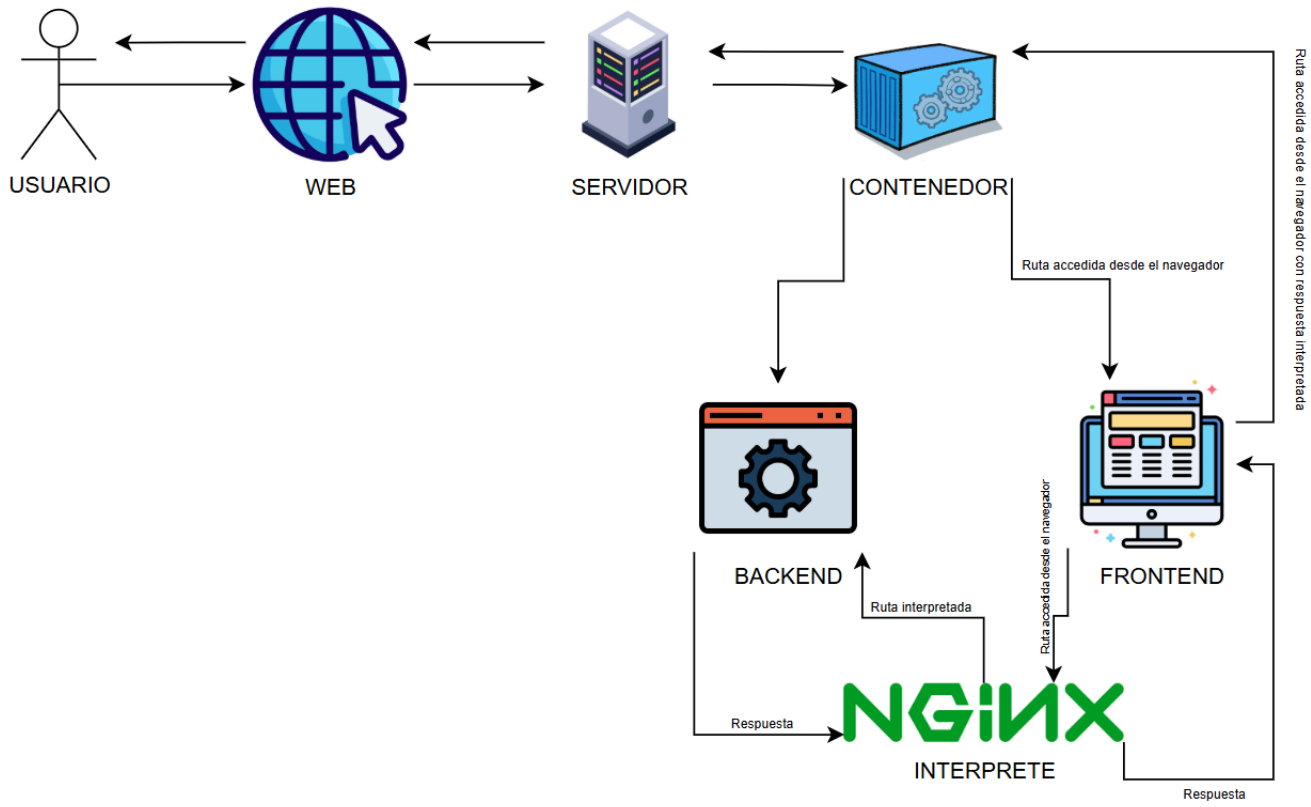
Ahora nuestro intérprete `vite.config.ts` tendrá una estructura similar a:

```
import { fileURLToPath, URL } from 'node:url'
import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'
import vueDevTools from 'vite-plugin-vue-devtools'
```

```
// https://vitejs.dev/config/
export default defineConfig({
  plugins: [
    vue(),
    vueDevTools(),
  ],
  resolve: {
    alias: {
      '@': fileURLToPath(new URL('./src', import.meta.url)),
    },
  },
  server: {
    proxy: {
      '/api': {
        target: 'http://localhost:8000',
        changeOrigin: true,
        rewrite: path => path.replace(/^\/api/, ''),
      },
    },
  },
})
```

Lo que hace es declarar esa ruta /api, como un target al backend, reemplazando la ruta del front por la que nos dirige al back y con ayuda de nuestro servidor terminamos de apuntar al back para cerrar el ciclo del proxy.

Podemos verlo cómo:



Si ya tenemos todo esto configurado de manera correcta, solamente deberemos eliminar la imagen previa que creamos que solo contiene el back, con el comando:

```
docker compose down -v
```

Construimos ahora todo el proyecto entero

```
docker compose buid --no-cache
```

Y finalmente levantamos todo junto:

```
docker compose up
```

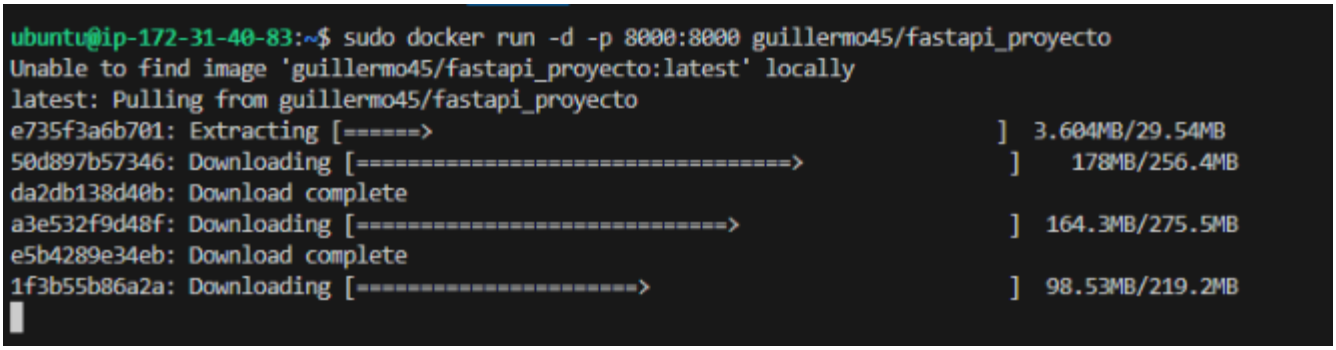
Con eso veremos lo siguiente en consola:

```
db-1       | 2025-07-23 16:59:23.549 UTC [1] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
db-1       | 2025-07-23 16:59:23.556 UTC [64] LOG:  database system was shut down at 2025-07-23 16:59:23 UTC
db-1       | 2025-07-23 16:59:23.563 UTC [1] LOG:  database system is ready to accept connections
backend-1  | PostgreSQL está listo, iniciando FastAPI...
backend-1  | INFO:     Started server process [12]
backend-1  | INFO:     Waiting for application startup.
backend-1  | INFO:     Application startup complete.
backend-1  | INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
frontend-1 | 172.18.0.1 - - [23/Jul/2025:16:59:32 +0000] "GET /db-ping HTTP/1.1" 200 428 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Ch
frontend-1 | 172.18.0.1 - - [23/Jul/2025:16:59:32 +0000] "GET /assets/index-DQ1kE7kI.js HTTP/1.1" 200 86206 "http://localhost:3000/db-ping" "Mozilla/5.0 (Windows NT 10.0; Win64;
0 Safari/537.36" "-"
frontend-1 | 172.18.0.1 - - [23/Jul/2025:16:59:32 +0000] "GET /assets/index-CEpdF7g3.css HTTP/1.1" 200 3123 "http://localhost:3000/db-ping" "Mozilla/5.0 (Windows NT 10.0; Win64;
0 Safari/537.36" "-"
```

Eso quiere decir que levantamos los 3 servicios de manera correcta y ahora cada que hagamos docker compose up estaremos desplegando todo junto de manera local.

Migrando a AWS

Ahora que hemos visto de manera local que nuestro proyecto funciona es momento de ver cómo podemos desplegarlo en AWS, en la documentación anterior desplegamos solamente el back cuándo corrimos desde la SSH nuestro proyecto



Ahora debemos actualizar la imagen que tenemos en DockerHub para que esta sea la del proyecto completo y no solamente la del Back.

¿Cómo manejamos nuestros entornos?

Podemos seguir haciendo uso de la separación de responsabilidades para mantener una mejor legibilidad de proyecto y que nuestros colaboradores puedan hacer cambios en las partes necesarias sin la necesidad de tener que estar recargando constantemente nuestra imagen general, por lo que podemos crear en DockerHub 2 repositorios.

1. backend_proyecto
2. frontend_proyecto

Para eso en nuestra terminal local hacemos:

```
docker build -t usuario/backend_proyecto ./FastApi_Proyecto_BackEnd
docker push usuario/backend_proyecto
```

```
PS C:\Users\x\Desktop\Digital_Container> docker push guillermo45/backend_proyecto
Using default tag: latest
The push refers to repository [docker.io/guillermo45/backend_proyecto]
98de21f3a999: Pushed
9c9d07396520: Pushed
106dd723fa93: Pushed
6a28d56a2a10: Pushed
8bd52e0207b5: Pushed
86b2831e5ceb: Pushed
3cc982388b71: Mounted from library/ubuntu
latest: digest: sha256:a51acbbb7006d20dafa1e5bbd9c80f3148146ae97de8ce99a1a8e59745a944f5 size: 1999
```

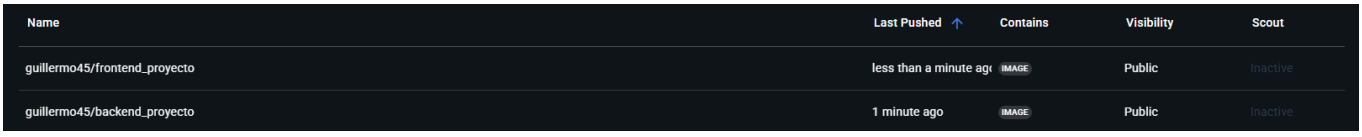
Y para el front:

```
docker build -t usuario/frontend_proyecto ./FastApi_Proyecto_FrontEnd
docker push usuario/frontend_proyecto
```

```
PS C:\Users\x\Desktop\Digital_Container> docker push guillermo45/frontend_proyecto
Using default tag: latest
The push refers to repository [docker.io/guillermo45/frontend_proyecto]
c34fa5f1b681: Pushed
bf0f8fc3603a: Pushed
ed869fc5c50f: Pushed
365f05739a0b: Mounted from library/nginx
09a78e1c8457: Mounted from library/nginx
aabf53a58c5a: Mounted from library/nginx
39b5112a4779: Mounted from library/nginx
243fb2529a82: Mounted from library/nginx
3c0990ff3868: Mounted from library/nginx
daf8a96d7e43: Mounted from library/nginx
7003d23cc217: Mounted from library/nginx
latest: digest: sha256:6c2a95d46d69b233a024da81408e2be96edf1d4584f4a2230acb544cf07485fa size: 2612
```

> **Usuario**
>
> En el apartado de usario, debemos colocar nuestro usuario de DockerHub, pero los comandos siguen funcionando igual, siempre y cuándo nuestras carpetas se llamen de la misma manera, pero si tu proyecto fue nombrado de otra manera, bastará con cambiar los nombres, lo único que permanece inmutable es `docker build -t` y `docker push`.

Ahora al entrar a nuestro Dockerhub veremos que tenemos creadas dos imágenes.

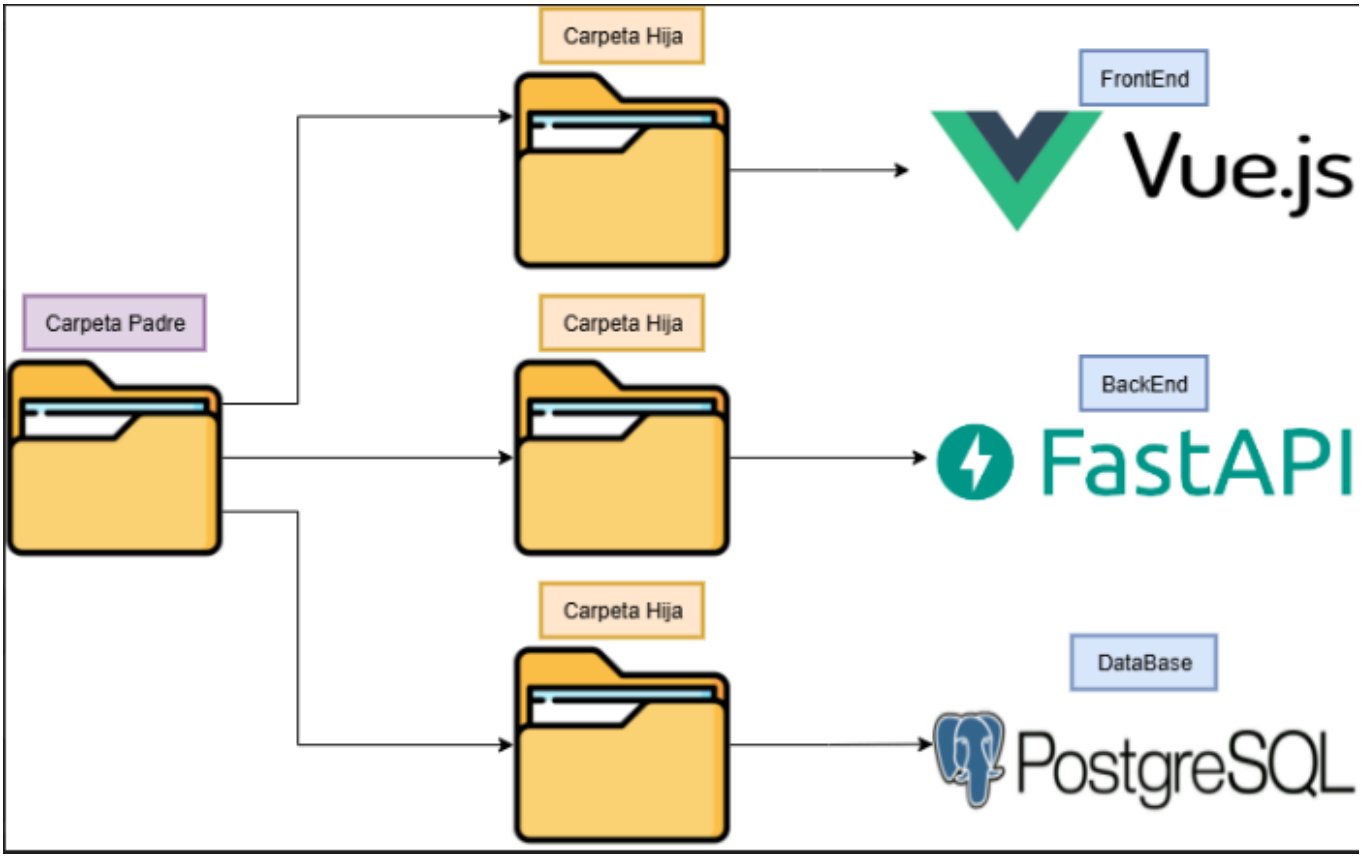


Enlazando nuestros entornos

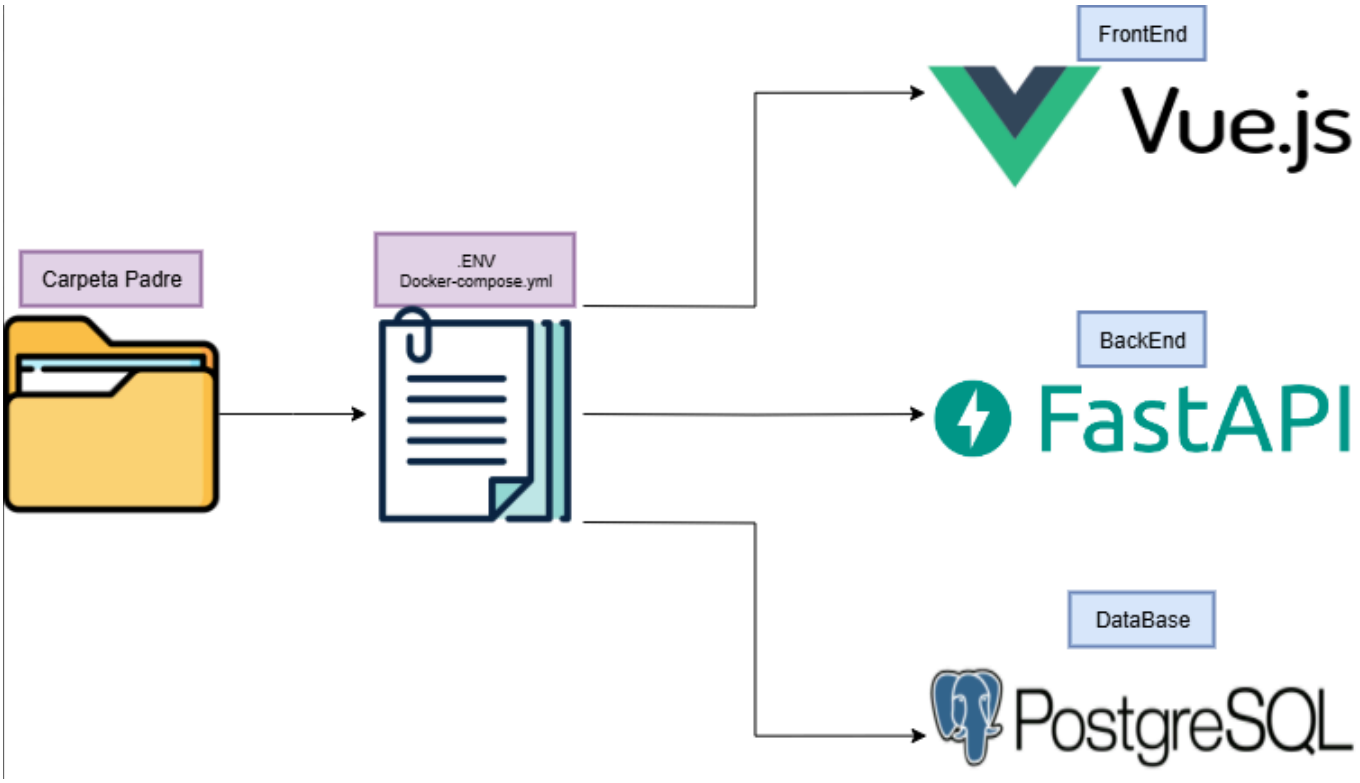
Si podemos ver nuestras dos imágenes en nuestro perfil de DockerHub, ahora debemos modificar nuestra carpeta padre local, los pasos anteriores fueron una forma de prueba para saber que funciona tanto el proyecto de back, como de front, en nuestro local era semi-desacoplada, por que nuestro servidor era la máquina local, pero cuándo estemos en `EQUPOS DE TRABAJO`, no siempre nos van a liberar el permiso para ver todo el proyecto, si no que nos dejarán ver solamente la parte del proyecto que tengamos que trabajar.

Con esto en mente, sabemos que nuestras imágenes ya están a salvo en DockerHub, ya podemos eliminar del local las carpetas de back y front, esto con la finalidad de que nuestra carpeta padre solo contenga el .env y el docker-compose.yml.

Esto haría que nuestro flujo anterior:



Ahora se simplifique de la siguiente manera:



Ya que el documento `docker-compose.yml` en lugar de hacer referencia a carpetas hijas dentro de la carpeta padre, haría referencia a las imágenes que tenemos en DockerHub, por lo que nuestro nuevo compose, se vería de la siguiente manera:

```
version: '3.8'

services:
  backend:
    image: guillermo45/backend_proyecto:latest
    container_name: backend_app
    ports:
      - "8000:8000"
    env_file:
      - .env
    environment:
      -
DATABASE_URL=postgresql://${POSTGRES_USER}:${POSTGRES_PASSWORD}@db:5432/${POST
GRES_DB}
    depends_on:
      - db
    restart: always

  frontend:
```

```
    image: guillermo45/frontend_proyecto:latest
    container_name: frontend_app
    ports:
      - "3000:80"
    restart: always

  db:
    image: postgres:14
    container_name: postgres_db
    restart: always
    env_file:
      - .env
    volumes:
      - db_data:/var/lib/postgresql/data

volumes:
  db_data:
```

Por lo que solamente quedaría `SUBIR` esto a nuestro servidor para que nuestro contenedor padre solamente sirva de orquestador, pero este proceso ya no lo haríamos en Dockerhub, si no en nuestra terminal local, para copiar la carpeta dentro de la terminal SSH de EC2, para eso haremos:

```
scp -i C:\directory\acceso-servidor.pem -r C:\directory\acceso-servidor.pem
ubuntu@mi-ip-publica:~
```

Con esto ya hemos copiado nuestra carpeta de la máquina local a nuestro servidor de Ubuntu y al conectarnos por ssh con:

```
ssh -i ssh -i C:\directory\acceso-servidor.pem ubuntu@mi-ip-publica
```

Podremos hacer un `ls` y visualizar nuestra carpeta:

```
ubuntu@ip-172-31-40-83:~$ ls
Digital_Container
ubuntu@ip-172-31-40-83:~$ cd Digital_Container/
ubuntu@ip-172-31-40-83:~/Digital_Container$ ls
docker-compose.yml
```

En caso de que estemos en Windows, esto no copiará nuestro .env, pero podemos hacer:

```
scp -i  C:\directory\acceso-servidor.pem  C:\directory\acceso-servidor.pem.env
ubuntu@mi-ip-publica:~/Digital_Container/
```

Una vez que tengamos esto en nuestro servidor ahora debemos estar seguros que tengamos instalado nuestro plugin para docker-compose, por lo que haremos:

```
sudo apt update

sudo apt install ca-certificates curl gnupg -y

sudo install -m 0755 -d /etc/apt/keyrings

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -
o /etc/apt/keyrings/docker.gpg

echo \
  "deb [arch=$(dpkg --print-architecture) signed-
by=/etc/apt/keyrings/docker.gpg] \
  https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt update

sudo apt install docker-compose-plugin -y
```

Explicación:

Aquí estaremos viendo de manera más detallada que es lo que hace cada comando y por qué los utilizamos:

`sudo apt update`

Actualiza la lista de paquetes disponibles en el sistema para asegurar que se obtengan las versiones más recientes.

`sudo apt install ca-certificates curl gnupg -y`

Instala herramientas esenciales:

- `ca-certificates`: para manejar certificados HTTPS.
- `curl`: para hacer peticiones HTTP y descargar archivos.
- `gnupg`: para verificar firmas GPG y seguridad del repositorio.

`sudo install -m 0755 -d /etc/apt/keyrings`

Crea un directorio especial (`/etc/apt/keyrings`) con los permisos apropiados donde se almacenarán las claves públicas de repositorios seguros (como el de Docker).

`curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg`

- **Descarga la clave GPG oficial de Docker** desde el sitio de Docker.
- **Convierte (dearma)** esa clave a un formato binario (`.gpg`) que APT pueda usar.
- **Guarda la clave** en el directorio `/etc/apt/keyrings` .

Esto permite que el sistema **confíe en el repositorio oficial de Docker**.

`echo ... | sudo tee /etc/apt/sources.list.d/docker.list`

Este comando **registra el repositorio oficial de Docker** en tu sistema Ubuntu.

- Usa la arquitectura correcta (`amd64` , `arm64` , etc.).
- Se firma con la clave GPG que agregaste antes.
- Se basa en tu versión de Ubuntu (por ejemplo: `noble` , `jammy` , etc.).

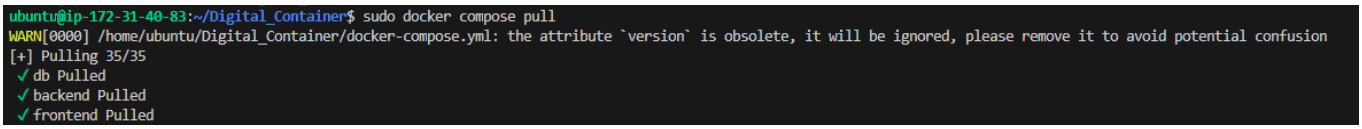
Esto permite que puedas instalar paquetes **directamente desde Docker**.

Información recuperada de: https://gist.github.com/adamltyson/4f502bb326d69e845a92a934e23642e6?permalink_comment_id=4962620

Corriendo nuestro proyecto

Si todo ha salido bien ahora podremos hacer un:

```
sudo docker compose pull
```



Esto nos va a traer nuestra imagen de BD, Front y Back

Para finalmente lanzar un

```
sudo docker compose up -d
```

Ahora con esta configuración seremos capaces de ver nuestro proyecto corriendo en nuestro dominio real.