TEMA 1
MÓDULO PROFESIONAL OPTATIVO

DESPLIEGUE DE APLICACIONES EN CONTENEDORES
DOCKER COMPOSE

Luis Miguel Fuster Garcia

06/01/2025

Parte 1: Implementación Práctica.

1. Estructura del Proyecto

Crea un directorio con la siguiente estructura, el contenido de los ficheros será el del ejemplo visto en la unidad.

```
mi_app_docker/
├── docker-compose.yml    # Archivo de orquestación
├── backend/              # Backend Node.js
│   ├── Dockerfile
│   ├── package.json
│   └── src/
│       └── index.js
└── frontend/             # Frontend React + Nginx
    ├── Dockerfile
    ├── package.json
    ├── public/
    │   └── index.html
    └── src/
        ├── App.js
        └── index.js
```

Para ello instalamos primero DOCKER COMPOSE en nuestro Ubuntu.

Apt update && apt upgrade

Apt install docker-compose-plugin

```
root@propietario-VirtualBox:/home/propietario# apt install docker-compose-plugin
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias... Hecho
Leyendo la información de estado... Hecho
E: No se ha podido localizar el paquete docker-compose-plugin
root@propietario-VirtualBox:/home/propietario#
```

Para corregir el error: Buscamos el paquete en el repositorio oficial

Apt install ca-certificates curl gnupg lsb-release

```
root@propietario-VirtualBox:/home/propietario# apt install ca-certificates curl gnupg lsb-release
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias... Hecho
Leyendo la información de estado... Hecho
lsb-release ya está en su versión más reciente (11.1.0ubuntu4).
fijado lsb-release como instalado manualmente.
```

Seguido añadimos la clave GPG oficial.

Mkdir –p /etc/apt/keyrings (creción del direcctorio)

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

```
root@propietario-VirtualBox:/home/propietario# mkdir -p /etc/apt/keyrings
root@propietario-VirtualBox:/home/propietario# curl -fsSL https://download.docker.com/linux/ubuntu/gpg
| sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
root@propietario-VirtualBox:/home/propietario#
```

Ahora añadimos el repositorio de Docker

```
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
```

```
root@propietario-VirtualBox:/home/propietario# 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
root@propietario-VirtualBox:/home/propietario#
```

Y por ultimo procedemos a actualizar e instalar el paquete:

Apt update

Apt install docker-ce docker-ce-cli containerd.io docker-compose-plugin

```
root@propietario-VirtualBox:/home/propietario# apt install docker-ce docker-ce-cli containerd.io docker
-compose-plugin
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias... Hecho
Leyendo la información de estado... Hecho
```

Verificamos la isntalación con:

Docker compose version

```
root@propietario-VirtualBox:/home/propietario# docker compose version
Docker Compose version v5.0.1
root@propietario-VirtualBox:/home/propietario#
```

Una vez que tenemos todo organizado pasamos a crear la estructura de carpetas y archivos que necesitamso en nustro arbol de trabajo.

Mkdir –p /home/mi_app_docker

Accedemos a el:

Cd /home/mi_app_docker

```
root@propietario-VirtualBox:/home/propietario# mkdir -p /home/mi_app_docker
root@propietario-VirtualBox:/home/propietario# cd /home/mi_app_docker
root@propietario-VirtualBox:/home/mi_app_docker#
```

Y ahora creamos el resto de carpetas y subcarpetas asi como los archivos dentro de las carpetas vacios.

```
root@propietario-VirtualBox:/home/mi_app_docker# mkdir -p backend/src
root@propietario-VirtualBox:/home/mi_app_docker# mkdir -p fronted/public
root@propietario-VirtualBox:/home/mi_app_docker# mkdir -p fronted/src
root@propietario-VirtualBox:/home/mi_app_docker# mkdir -p fronted/build
root@propietario-VirtualBox:/home/mi_app_docker# touch docker-compose.yml
root@propietario-VirtualBox:/home/mi_app_docker# touch backend/dockerfile
root@propietario-VirtualBox:/home/mi_app_docker# touch backend/package.json
root@propietario-VirtualBox:/home/mi_app_docker# touch backend/src/index.js
root@propietario-VirtualBox:/home/mi_app_docker# touch frontend/dockerfile
touch: no se puede efectuar `touch' sobre 'frontend/dockerfile': No existe el archivo o el directorio
root@propietario-VirtualBox:/home/mi_app_docker# touch fronted/dockerfile
root@propietario-VirtualBox:/home/mi_app_docker# touch fronted/package.json
root@propietario-VirtualBox:/home/mi_app_docker# touch fronted/public/index.html
root@propietario-VirtualBox:/home/mi_app_docker# touch fronted/src/app.js
root@propietario-VirtualBox:/home/mi_app_docker# touch fronted/src/intex.js
root@propietario-VirtualBox:/home/mi_app_docker# mv fronted frontend
root@propietario-VirtualBox:/home/mi_app_docker# ls
backend  docker-compose.yml  frontend
root@propietario-VirtualBox:/home/mi_app_docker#
```

Hemos escrito mal el nombre de → fronted en vez de frontend, pero lo modificamos con el comando:

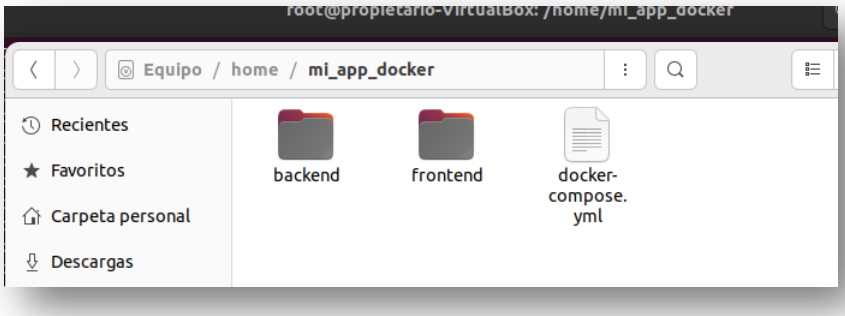
Mv fronted frontend

Los comandos son: para carpetas.

Mkdir –p backend/src
Mkdir –p frontend/public
Mkdir –p frontend/src
Mkdir –p frontend/build

Y para los archivos:

Touch docker-compose.yml
Touch backend/dockerfile
Touch backend/package.json
Touch backaend/src/index.js
Touch frontend/dockerfile
Touch frontend/package.json
Touch frontend/public/index.html
Touch frontend/src/app.js
Touch frontend/src/index.js



2. Configuración de los ficheros de los Servicios

Crea y da contenido a los ficheros necesarios para los tres servicios atendiendo a lo explicado en el ejemplo:

- **Backend (Node.js)**
- **Frontend (React + Nginx)**
- **Base de Datos (PostgreSQL)**

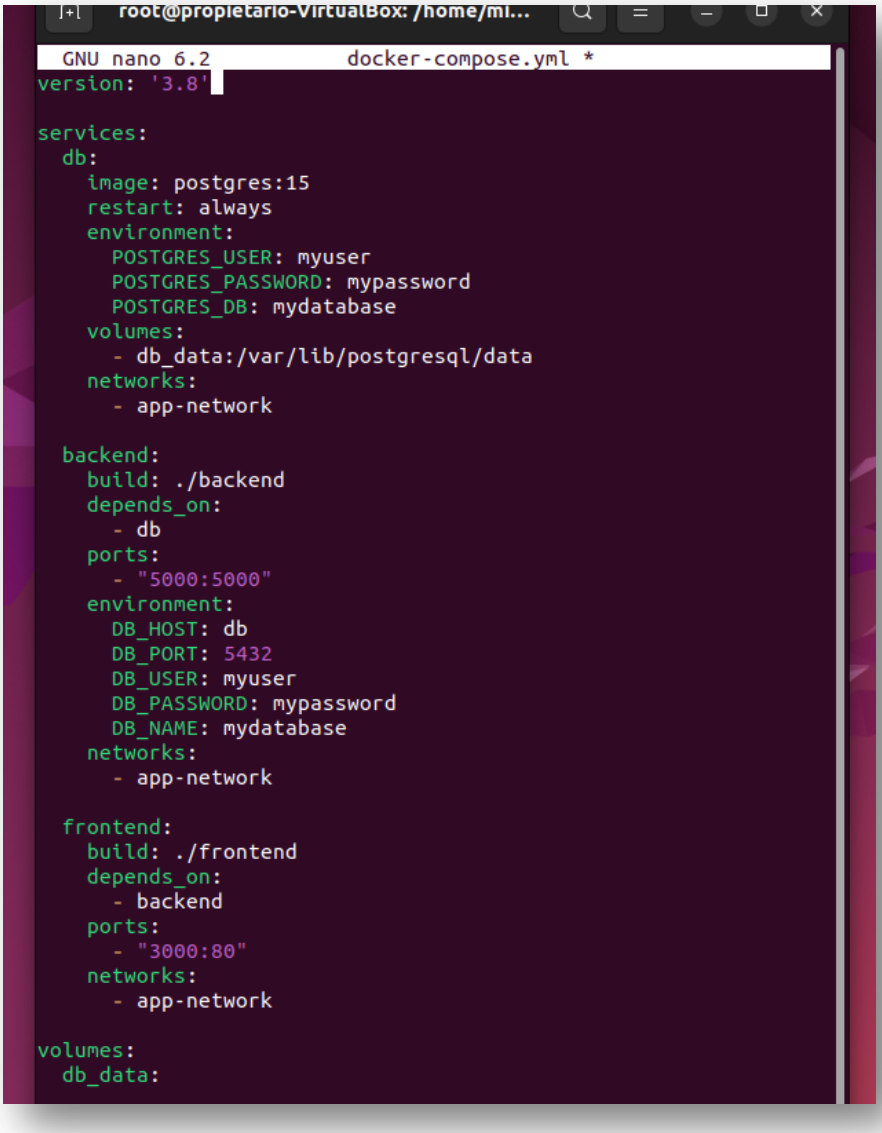
3. Docker Compose

- Usa el archivo `docker-compose.yml` de ejemplo.
- Asegúrate de que:
  - Los servicios estén conectados a una red personalizada (`app-network`).
  - El volumen `db_data` persista los datos de PostgreSQL.
  - Los puertos estén mapeados correctamente (Frontend: `3000:80` , Backend: `5000:5000`).

Una vez creada la instructura del proyecto y las herramientas instaladas nos preparamso para modificar los archivos correspondientes.

El archivo DOCKER-COMPOSE.yml

```yaml
version: '3.8'

services:
  db:
    image: postgres:15
    restart: always
    environment:
      POSTGRES_USER: myuser
      POSTGRES_PASSWORD: mypassword
      POSTGRES_DB: mydatabase
    volumes:
      - db_data:/var/lib/postgresql/data
    networks:
      - app-network

  backend:
    build: ./backend
    depends_on:
      - db
    ports:
      - "5000:5000"
    environment:
      DB_HOST: db
      DB_PORT: 5432
      DB_USER: myuser
      DB_PASSWORD: mypassword
      DB_NAME: mydatabase
    networks:
      - app-network

  frontend:
    build: ./frontend
    depends_on:
      - backend
    ports:
      - "3000:80"
    networks:
      - app-network

volumes:
  db_data:

networks:
  app-network:
```



Explicamos:

Version → Indica la versión del formato de Docker COmpose

Luego vemos:

Servicios → **db:** **backend:** **frontend:**

Que el bloque donde se definen lso contenendores que formaran la aplicación, Cada servicio ejecuta su propio contenedor:

Volumes → **db_data:**

Db_data es el volumen docker como se indica /var/lib/postgresql/data es donde PostgreSQL guardará los datos

Networks → **app-network:**

Conecta una red personalizada que permitirá la comunicación entre contenedores.

Servicio Db: → Image: postgres:15 → Imagen oficial de PostgreSQL v15 desde Docker Hub.
Sino existiese en nuestro equipo lo descargará automaticamente.
➔ Restart: always → En las bases de datos suele ser normal ante una caida se reinicia solo.
➔ Environment: POSTGRES_USER: myuser
POSTGRES_PASSWORD: mypassword
POSTGRESS_DB: mydatabase
Son las variables inidicales que definen el usuario el password y la base de datos al arrancar.
➔ Volumes → db_data:/var/lib/postgresql/data
db_data es un volumen docker
/var/lib/postgresql/data es donde se guardan los datos
➔ Networks → app-network
Conecta con una red personalizada.

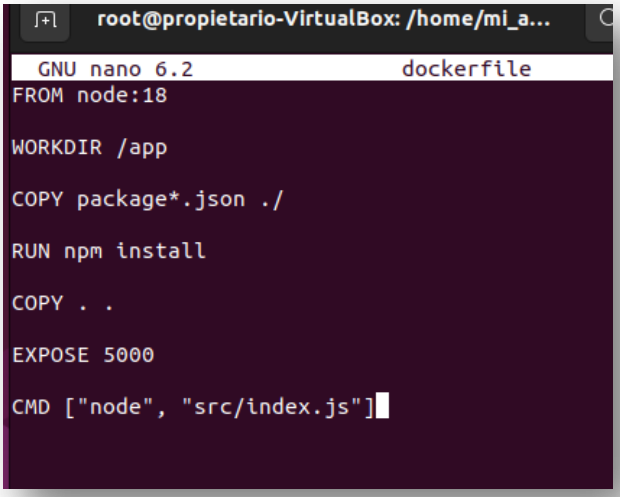
Servicio Backend: → build: ./backend → Docker construye la imagen usando el dockerfile de ./backend
➔ Depends_on: -db → indica la dependenica del servicio con el contenedor db. 1º la db…..
➔ Ports → -"5000:5000" → host:contenedor se permite acceder al backen desde fuera.
➔ Environmet: DB_HOST: db
DB_PORT: 5432
DB_USER: myuser
DB_PASSWORD: mypassword
DB_NAME: mydatabase
Son las variables de backen para conectarse con el servicio db
➔ Networks → app-network
Conecta con la misma red que todos los contenedores.

Servicio Frontend → build: ./frontend → Docker construye la imagen usando el dockerfile de ./frontend
➔ Depends_on: -backend → iniciara despues de backend por dependencia.
➔ Port → -"3000:80" → host:contenedor es decir: http://localhost:3000
➔ Networks → app-network
Conecta con la misma red que todos los contenedores y servicios.

Servicios = contenedores
Image = las imágenes docker (pueden ser docker hub)
Build = Construye desde el dockerfile
Port = acceso entre el host (la maquina) y el contenedor
Volumes = Persistencia de datos con volumenes (teoria de volumens)
Environment = Son variables ususarios db, passwords etc
Depends_on = Orden de arranque
Networks = Es la red creada como comunicación entre contenedores.

El archivo DOCKERFILE de BACKEND

```
FROM node:18

WORKDIR /app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 5000

CMD ["node", "src/index.js"]
```



Explicamos:

FROM node:18

➔ Indica la version de NODE.JS version 18. Se descarga automaticamente sino esta instalado

WORKDIR /app

Define el directorio de trabajo DENTRO del conteneador, apartir de aquí todo se ejecutara desde /app

➔ COPY package*.json ./

Copia package.json y package-lock.json al contenedor

➔ RUN npm install

Instala las dependendias de package.json

➔ COPY . .

Copia todo el contenido del proyecto al contenedor

➔ EXPOSE 5000

Indica que la aplicación escucha el puerto 5000

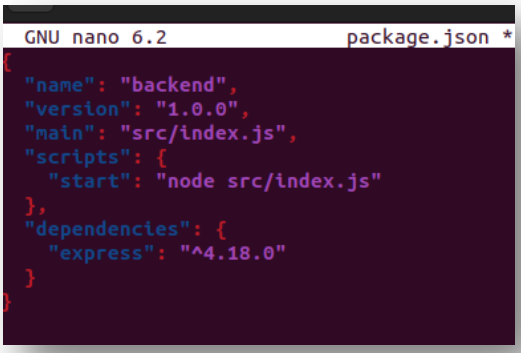
➔ CMD [“node”, “src/index.js”]

Comando que se ejecuta cuando el contenedor arranca, exactamente la alicacion Nodejs

FROM → Imagen Base
WORKDIR → Carpeta de trabajo
COPY → Copiar los archivos al contenedor
RUN → Ejecutar comandos al construir la imagen
EXPOSE → Puerto de la aplicación.
CMD → Comando de arranque del contenedor.

El archivo PACKAGE.json de BACKEND

```
{
  "name": "backend",
  "version": "1.0.0",
  "main": "src/index.js",
  "scripts": {
    "start": "node src/index.js"
  },
  "dependencies": {
    "express": "^4.18.0"
  }
}
```



Explicamos el archvio json

"name": "backend",

➔ Nombre del proyecto, identifica la aplicación dentro del ecosistema Node.js

"version": "1.0.0",

➔ Version

"main": "src/index.js",

➔ Archivo principal de la aplicación

"scripts": { "start": "node src/index.js" }

➔ Define comandos personalizados de npm → npm start ejectura node src/index.js

Ya que en el dockerfile se usa `CMD ["node", "src/index.js"]`

"dependencis": { "express": "4.18.0" }

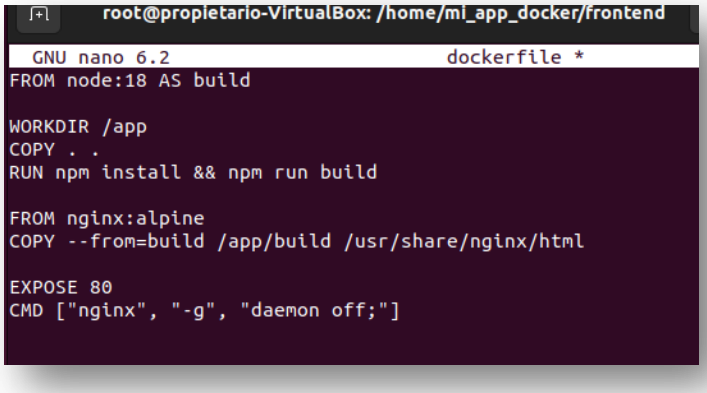
➔ Facilita la creación de APIS

El archivo SRC/INDEX.js de BACKEND

```
const express = require('express');  // Cargamos la librería Express
const app = express();               // Creamos una aplicación Express
const port = 5000;                   // Puerto donde se ejecutará el servidor

// Ruta principal: cuando alguien accede a http://localhost:5000
app.get('/', (req, res) => {
  res.send('¡Backend funcionando!');
});

// Arrancar el servidor y mostrar un mensaje en la consola
app.listen(port, () => {
  console.log(`Servidor backend en http://localhost:${port}`);
});
```

El archivo DOCKERFILE de FRONTEND

```
FROM node:18 AS build

WORKDIR /app
COPY . .
RUN npm install && npm run build

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

EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

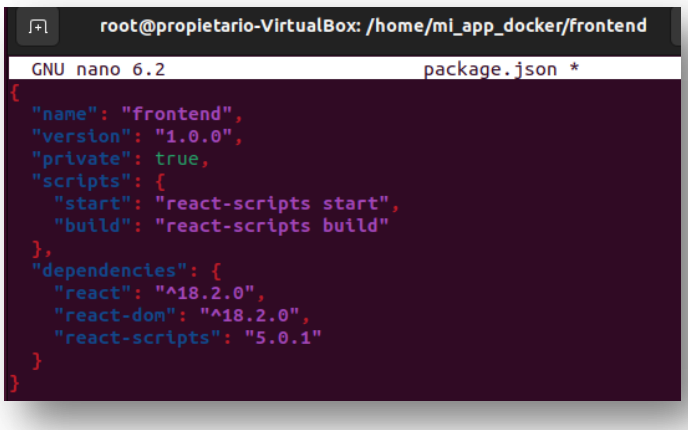


Explicamos:

- ➔ FROM node:18 AS Build
  Indica la version de NODE.JS version 18. AS BUild le pone un nombre para usarlo después.
- ➔ WORKDIR /app
  Define el directorio de trabajo DENTRO del conteneador, apartir de aquí todo se ejecutara desde /app
- ➔ RUN npm install && npm run build
  Instala las dependendias y run build optimza la versión.
- ➔ FROM nginx:alpine
  Usa la imagen de nginx
- ➔ COPY –from=build /app/build /usr/share/nginx/html
  Copia el resultado de build en la carpeta html que es la carpeta por defecto de Nginx
- ➔ EXPOSE 80
  Indica que la aplicación escucha el puerto 80 se define en docker-compose.yml
- ➔ CMD [“nginx”, “-g”, “daemon off;”]
  Arranca en primer plano y necesario para que el contenedor no se detenga

El archivo PACKAGE.json de FRONTEND

```
{
  "name": "frontend",
  "version": "1.0.0",
  "private": true,
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build"
  },
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-scripts": "5.0.1"
  }
}
```



Explicamos el archvio json

“name”: “frontend”,

➔ Nombre del proyecto, identifica la aplicación dentro del ecosistema npm

“private”: true,

➔ Indica que el proyuecto no se puede publicar en npm

“scripts”: { “start”: “react-scripts start”, “build”: “react-scripts build” }

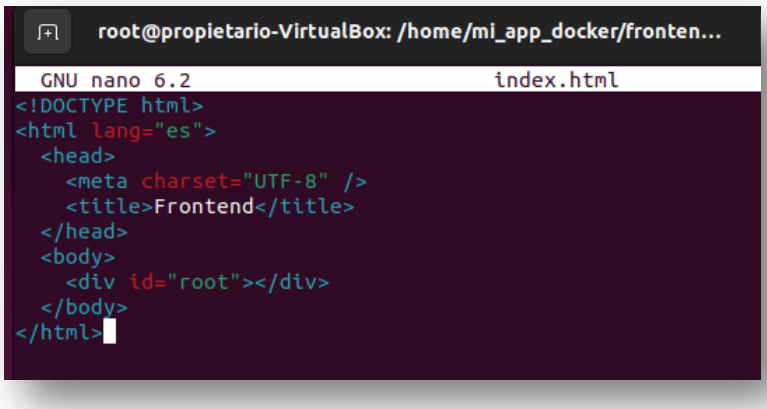
➔ Nmp start Arranca servidor de desarrollo solo local, build optimiza producción

“dependencis”: { “react”: “18.2.0”, “react-dom”: “18.2.0”, react-scrips”: “5.0.1” }

➔ React librería principal → React-dom renderiza el navegador → React-scrips necesaria para start y build.

Archivo /PUBLIC/INDEX.html de FRONTEND

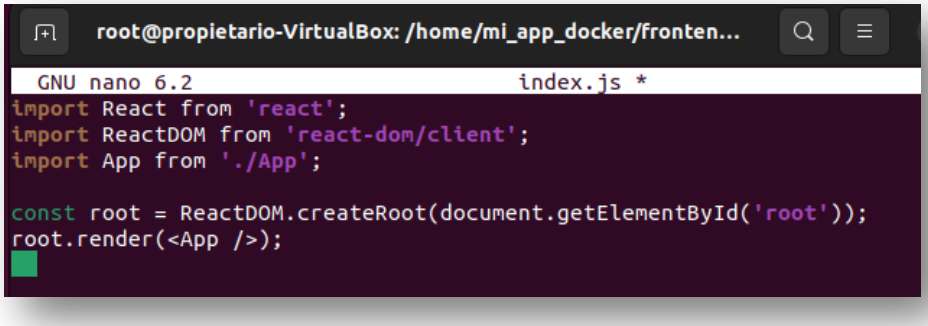
```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8" />
    <title>Frontend</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```



Un archivo HTML basico.

Archivo /SRC/INDEX.JS de FRONTEND

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```



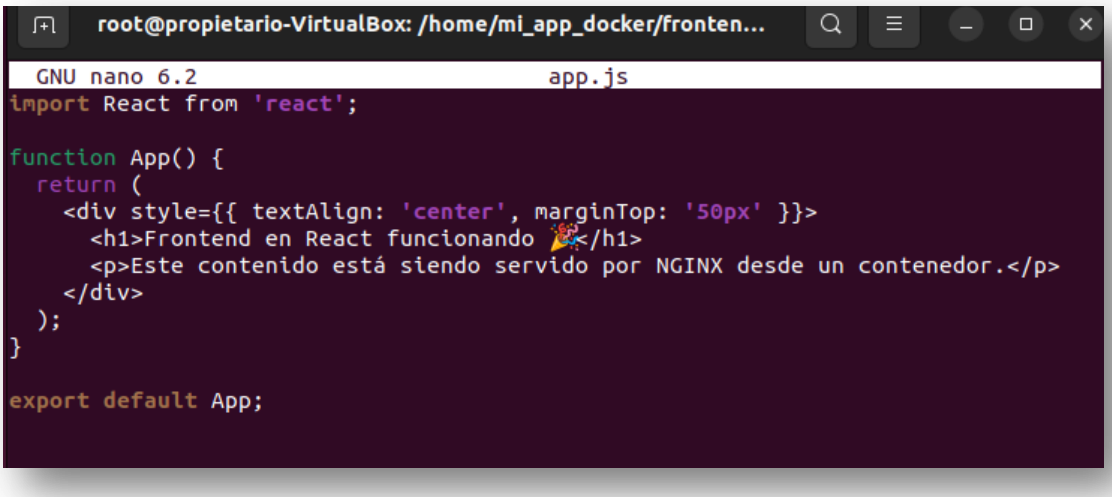
Explicamos:

Las importanciones REACT de la librería principal para crear componentes
REACT-DOM/client permite renderizar REACT en el DOM
APP Componente principal de la aplicación

Const root = busca en el HTML el elemento “id=root” y desde ese punto renderizara.

Archivo /SRC/APP.js de FRONTEND

```
import React from 'react';

function App() {
  return (
    <div style={{ textAlign: 'center', marginTop: '50px' }}>
      <h1>Frontend en React funcionando 🎉</h1>
      <p>Este contenido está siendo servido por NGINX desde un contenedor.</p>
    </div>
  );
}

export default App;
```



Explicación del archivo:

import React from 'react';

Importación

Importa React, necesario para usar JSX.

function App() {

Definición del componente

App es un **componente funcional** de React.

Es el componente raíz de la aplicación.

return (

Devuelve el **JSX**, que describe la interfaz de usuario.

<div style={{ textAlign: 'center', marginTop: '50px' }}>) }

Estilos en línea

Usa **inline styles** en React.

Se definen como un objeto JavaScript:

textAlign en lugar de text-align

Valores como strings

export default App;

Exportación

Exporta el componente para que pueda usarse en index.js.

4. Ejecución y Pruebas

1. Levanta los servicios con:

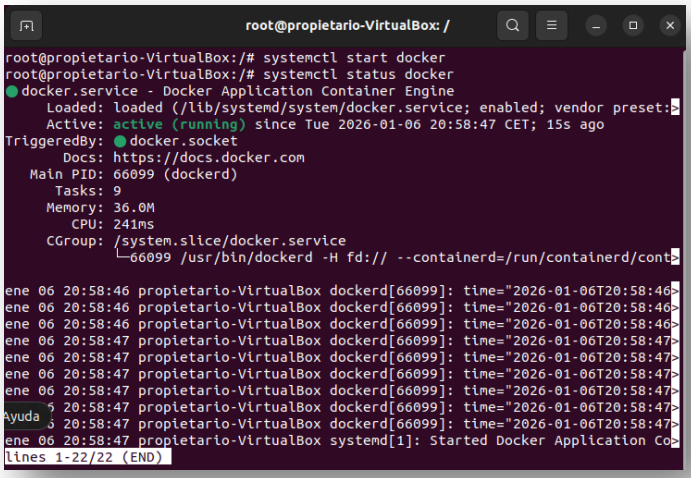
   ```
   docker compose up --build
   ```

2. Verifica que:

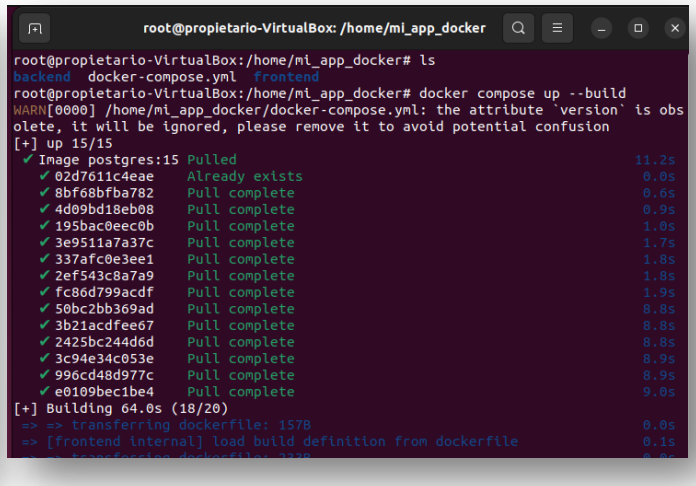
   - El frontend funcione en `http://localhost:3000`.
   - El backend responda en `http://localhost:5000`.
   - Los contenedores estén comunicados (usa `docker exec -it <contenedor> curl http://backend:5000`).

Levantamos el servicio de DOCKER y revisamos su status.

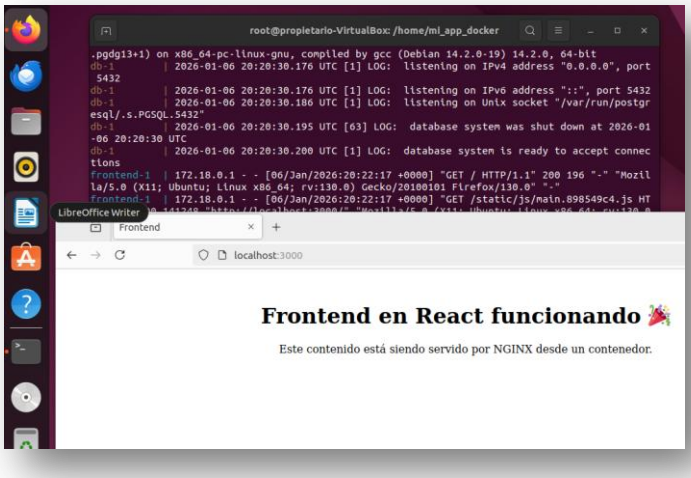
Systemctl start docker
Systemctl status docker



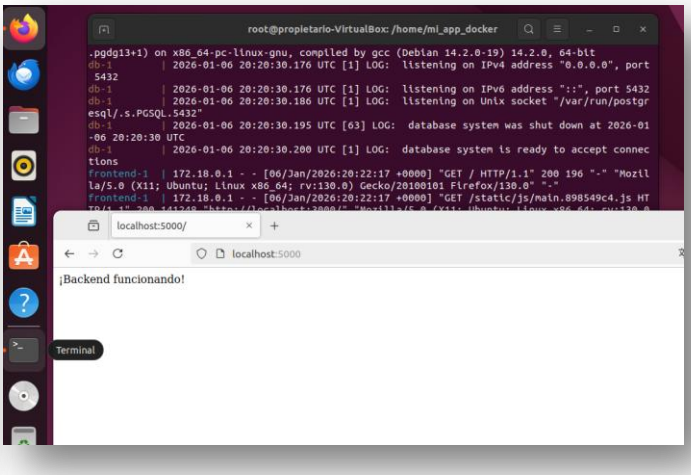
Ejecutamos docker compose up --build
Levantar el servicio docker-compose.yml



Probamos http://localhost:3000



y http://localhost:5000



2. Capturas y Resultados

- Imágenes de la aplicación funcionando (frontend + backend).
- Salida de comandos clave, explicando e interpretando de manera detallada la información ofrecida por los mismos:

```
docker ps
docker network inspect app-network
docker volume inspect mi_app_docker_db_data
```

Docker ps → para ver los servicios activos

```
root@propietario-VirtualBox: /home/mi_app_docker
root@propietario-VirtualBox:/home/mi_app_docker# docker ps
CONTAINER ID   IMAGE                     COMMAND                  CREATED         STATUS         PORTS
                                         NAMES
d8dd732ef83a   mi_app_docker-frontend    "/docker-entrypoint.…"   7 minutes ago   Up 7 minutes   0.0.0.0
:3000->80/tcp, [::]:3000->80/tcp         mi_app_docker-frontend-1
4ae67872673b   mi_app_docker-backend     "docker-entrypoint.s…"   7 minutes ago   Up 7 minutes   0.0.0.0
:5000->5000/tcp, [::]:5000->5000/tcp     mi_app_docker-backend-1
61685431dc0b   postgres:15               "docker-entrypoint.s…"   7 minutes ago   Up 7 minutes   5432/tc
                                         mi_app_docker-db-1
root@propietario-VirtualBox:/home/mi_app_docker#
```

Probamos conectividad. → docker exec –it mi_app_docker-frontend-1 sh (para abrir terminal en frontend)
→ curl http://backend:5000

```
root@propietario-VirtualBox:/home/mi_app_docker# ls
backend  docker-compose.yml  frontend
root@propietario-VirtualBox:/home/mi_app_docker# cd frontend
root@propietario-VirtualBox:/home/mi_app_docker/frontend# docker exec -it mi_app_docker-frontend-1 sh
/ # curl http://backend:5000
¡Backend funcionando!/ #
```

Docker network ls

```
root@propietario-VirtualBox:/home/mi_app_docker# docker network ls
NETWORK ID     NAME                        DRIVER    SCOPE
627e073f38bc   bridge                      bridge    local
614202e0fdfd   host                        host      local
34395f676e9d   mi_app_docker_app-network   bridge    local
c7bab7040f97   none                        null      local
root@propietario-VirtualBox:/home/mi_app_docker#
```

Docker volumen ls

```
root@propietario-VirtualBox:/home/mi_app_docker# docker volume ls
DRIVER    VOLUME NAME
local     contenido_web
local     data_volume
local     mi_app_docker_db_data
root@propietario-VirtualBox:/home/mi_app_docker#
```

docker network inspect mi_app_docker_app-network


```
        "com.docker.compose.version": "5.0.1"
    },
    "Containers": {
        "4ae67872673b0e3637a4e66ea771ba4397d292d4bf662600f237a08f21390c9f": {
            "Name": "mi_app_docker-backend-1",
            "EndpointID": "502468abd584373cc0a5aad9259c8609be99b5134bd419a6773962034bc3b0ac",
            "MacAddress": "da:c5:58:27:9b:15",
            "IPv4Address": "172.18.0.3/16",
            "IPv6Address": ""
        },
        "61685431dc0b349c2a982bb0670089730a93a44cb3f1d41592fc1fa0f5541af0": {
            "Name": "mi_app_docker-db-1",
            "EndpointID": "f32b5e8789ad0c7101edb82cf18d4c65cb4a38b44f61afd5967d99ec8bd1c64b",
            "MacAddress": "32:50:bd:46:20:bf",
            "IPv4Address": "172.18.0.2/16",
            "IPv6Address": ""
        },
        "d8dd732ef83a4fca01ffe550463f1c8fd29f716a36919b3f5ae16c052bb33f47": {
            "Name": "mi_app_docker-frontend-1",
            "EndpointID": "4cc9745e1e95587fbfa9fae687ac241c96cd79c8a83b9f2c334695e2c48f02e9",
            "MacAddress": "a6:85:cc:91:6e:3c",
            "IPv4Address": "172.18.0.4/16",
            "IPv6Address": ""
        }
    },
    "Status": {
        "IPAM": {
            "Subnets": {
                "172.18.0.0/16": {
                    "IPsInUse": 6,
                    "DynamicIPsAvailable": 65530
                }
            }
        }
```


docker volumen inspect mi_app_docker_app_data


```
root@propietario-VirtualBox:/home/mi_app_docker# docker volume inspect mi_app_docker_db_data
[
    {
        "CreatedAt": "2026-01-06T21:20:26+01:00",
        "Driver": "local",
        "Labels": {
            "com.docker.compose.config-hash": "0894cf38d68f3b1d79b5501f41b780df5b3eceba232aaf5dd2c60cb1
7356d483",
            "com.docker.compose.project": "mi_app_docker",
            "com.docker.compose.version": "5.0.1",
            "com.docker.compose.volume": "db_data"
        },
        "Mountpoint": "/var/lib/docker/volumes/mi_app_docker_db_data/_data",
        "Name": "mi_app_docker_db_data",
        "Options": null,
        "Scope": "local"
    }
]
root@propietario-VirtualBox:/home/mi_app_docker#
```

3. Pregunta de Reflexión

1. ¿Qué ocurriría si eliminamos el volumen `db_data` ?
2. ¿Por qué el backend usa `depends_on` con la base de datos? ¿Garantiza que PostgreSQL esté listo?

> **"En Docker, los contenedores son efímeros, pero los volúmenes permiten persistir información entre ejecuciones. Esto es fundamental para aplicaciones reales, ya que los datos (como archivos de usuario, logs o contenido web) no deben perderse al reiniciar o actualizar contenedores."**

El volumen db_data

Si eliminamos el volumen:

La base de datos se reinicia como nueva, vacía.

Todos los datos almacenados previamente se pierden completamente.

El sistema (contenedores) seguiría corriendo, pero la próxima vez que se inicie el contenedor de PostgreSQL se creará una DB limpia.

En otras palabras, el volumen funciona como la “memoria persistente” de la base de datos. Sin él, cada reinicio o recreación del contenedor genera una base de datos nueva y vacía.

En depends_on

depends_on controla el orden de inicio de los contenedores en Docker Compose.

En tu caso, el backend espera a que se inicie primero la base de datos.

Importante: Esto no garantiza que PostgreSQL esté listo para recibir conexiones, solo que Docker lo habrá arrancado antes del backend.

Por eso, en sistemas reales, se usan scripts de espera o lógica de reintentos en el backend para asegurarse de que la base de datos esté disponible antes de intentar conectarse.