**Implementación De Docker En Tres Servicios (Full-Stack) Usando Docker-Compose Y Publicarlo A Docker-Hub.**

Contenido

[**Implementación de docker y su configuración.** 3](#_Toc167055095)

[Prologo 3](#_Toc167055096)

[Introduccion 5](#_Toc167055097)

[Descripción 5](#_Toc167055098)

[Preparación del Entorno de trabajo: 6](#_Toc167055099)

[1. Estructura del proyecto 9](#_Toc167055100)

[2. Dockerfile para la base de datos 10](#_Toc167055101)

[3.Configuración de la API (Spring Boot) 12](#_Toc167055102)

[4. Configuración de la aplicación (Vue.js) 16](#_Toc167055103)

[5.-Verificación antes de levantar el docker principal. 20](#_Toc167055104)

[Estructura Final 24](#_Toc167055105)

[Subir y Compartir Los repositorios de Docker Hub 25](#_Toc167055106)

[Conclusión 32](#_Toc167055107)

[Referencias 33](#_Toc167055108)

# **Implementación de docker y su configuración.**

## Prologo

“Si tienes una aplicación o servicio y quieres que funcione en diferentes [**sistemas como VPS**](https://www.hostinger.es/tutoriales/que-es-un-vps) o máquinas, sin ningún problema, considera la posibilidad de utilizar contenedores. Una de las plataformas de contenedores más populares es Docker, aunque no todo el mundo sabe qué es y cómo funciona.

En este tutorial, explicaremos qué es Docker, cómo funciona y en qué se diferencia de las máquinas virtuales (VM) y los sistemas, como Kubernetes y Jenkins. También repasamos los pros y los contras de Docker y enumeramos sus casos de uso más populares”.

¿Qué es Docker?

“Docker es un software de código abierto utilizado para desplegar aplicaciones dentro de contenedores virtuales. La contenerización permite que varias aplicaciones funcionen en diferentes entornos complejos. Por ejemplo, Docker permite ejecutar el sistema de gestión de contenidos WordPress en sistemas Windows, Linux y macOS sin ningún problema” (Hostinger.com, s.f.)

Algún otor concepto,

(Docker.com, s.f.) “es **Acelerar la forma de crear, compartir y ejecutar aplicaciones**

Docker ayuda a los desarrolladores a crear, compartir, ejecutar y verificar aplicaciones en cualquier lugar, sin tediosas configuraciones o administración del entorno”

**Conceptos básicos acerca de Docker que debemos conocer:**

1. **Contenedores Docker:** Son entornos aislados y ligeros que contienen todo lo necesario para ejecutar una aplicación, incluidas las dependencias y la configuración. Los contenedores son portátiles y pueden ejecutarse en cualquier máquina que tenga Docker instalado, independientemente de las diferencias en el sistema operativo o el hardware subyacente.
2. **Dockerfile:** Es un archivo de texto que contiene las instrucciones para construir una imagen Docker. Las imágenes Docker son la base de los contenedores, y se crean a partir de los Dockerfiles mediante un proceso de construcción. Los Dockerfiles especifican qué software se incluirá en la imagen, cómo se configurará y cómo se ejecutará la aplicación.
3. **Docker Hub:** Es un registro público de imágenes Docker donde los desarrolladores pueden compartir y descargar imágenes preconstruidas para usar en sus aplicaciones. También se puede utilizar para almacenar imágenes personalizadas y privadas.

Docker Compose, por otro lado, es una herramienta que permite definir y gestionar aplicaciones multi-contenedor en Docker de manera sencilla. Con Docker Compose, puedes utilizar un archivo YAML (usualmente llamado **docker-compose.yml**) para definir la configuración de tus servicios, incluyendo la configuración de redes, volúmenes y variables de entorno. Luego, puedes utilizar el comando **docker-compose** para levantar, gestionar y detener tu aplicación de manera coordinada.

Quieres saber más de docker y sus servicios poder ir ala su página oficial https://www.docker.com/

## Introduccion

En este proyecto, vamos a desplegar una arquitectura de microservicios utilizando Docker y Docker-Compose para orquestar tres componentes esenciales que podría tener 2 o 3 servicios interconectados entre sí, en este caso será un proyecto full stack de una AgendaContactosi; el primer servicio es una base de datos PostgreSQL, una API desarrollada en Spring Boot, y una aplicación frontend desarrollada en Vue.js completamente responsive. Docker nos permite crear entornos aislados y reproducibles para cada uno de estos servicios, facilitando el desarrollo, la prueba y la implementación.

## Descripción

El proyecto, denominado my\_project\_docker, consta de las siguientes carpetas y componentes:

1. db: contiene el archivo backup.sql con las instrucciones para crear las tablas e insertar datos iniciales en la base de datos PostgreSQL.
2. demoApiAgenda: La aplicación backend desarrollada en Spring Boot. Aquí se encuentra y código fuente de la API y el archivo Dockerfile para construir la imagen Docker correspondiente.
3. app-mi-agenda: La aplicación frontend desarrollada en Vue.js. Esta carpeta contiene el código fuente del frontend y el archivo Dockerfile para construir su imagen Docker.

Para comenzar a implementar Docker en este proyecto, seguiremos estos pasos:

## Preparación del Entorno de trabajo:

Para verificar la instalación de Docker y Docker Compose en tu máquina, puedes seguir los pasos a continuación según el sistema operativo que estés usando: Windows 10, macOS, o Linux.

**Windows 10**

1. **Verificar Docker:**
   * Abre PowerShell o el Símbolo del sistema (CMD).
   * Ejecuta el siguiente comando:

docker --version

Deberías ver una salida similar a:

Docker version 20.10.7, build f0df350

1. **Verificar Docker Compose:**
   * En la misma ventana de PowerShell o Símbolo del sistema, ejecuta:

docker-compose --version

Deberías ver una salida similar a:

docker-compose version 1.29.2, build 5becea4c

**macOS**

1. **Verificar Docker:**
   * Abre la Terminal.
   * Ejecuta el siguiente comando:

docker --version

Deberías ver una salida similar a:

Docker version 20.10.7, build f0df350

1. **Verificar Docker Compose:**
   * En la misma ventana de la Terminal, ejecuta:

docker-compose --version

Deberías ver una salida similar a:

docker-compose version 1.29.2, build 5becea4c

**Linux**

1. **Verificar Docker:**
   * Abre una terminal.
   * Ejecuta el siguiente comando:

docker --version

Deberías ver una salida similar a:

Docker version 20.10.7, build f0df350

1. **Verificar Docker Compose:**
   * En la misma terminal, ejecuta:

docker-compose --version

Deberías ver una salida similar a:

docker-compose version 1.29.2, build 5becea4c

**Instalación del Desktop (si no está instalado)**

Si no tienes Docker o Docker Compose instalados, aquí tienes una guía rápida para cada sistema operativo:

**Windows 10**

1. Descarga e instala Docker Desktop para Windows.
2. Durante la instalación, asegúrate de que Docker Compose esté seleccionado.
3. Sigue las instrucciones de instalación y reinicia tu máquina si es necesario.

**macOS**

1. Descarga e instala Docker Desktop para Mac.
2. Durante la instalación, asegúrate de que Docker Compose esté seleccionado.
3. Sigue las instrucciones de instalación.

**Linux**

1. **Instalar Docker:**

Ejecuta los siguientes comandos en la terminal:

* sudo apt-get update
* sudo apt-get install -y docker.io
* sudo systemctl start docker
* sudo systemctl enable docker

1. **Instalar Docker Compose:**

Ejecuta los siguientes comandos en la terminal:

* sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
* sudo chmod +x /usr/local/bin/docker-compose

Después de seguir estos pasos, puedes verificar la instalación de Docker y Docker Compose nuevamente utilizando los comandos mencionados al principio.

Vamos a realizar la configuración con las versiones que tengo instalada en mi computadora. Para la base de datos vamos a utilizar PostgreSQL 14, para el desarrollo de la Api RESRT con Spring utilizaremos Java 17 y los puertos que especificaremos mas adelante. Realizaremos una guía detallada paso a paso para implementar Docker en nuestro proyecto:

## 1. Estructura del proyecto

La estructura de nuestro proyecto es la siguiente:

jose\_@DESKTOP-KKFSQ9M MINGW64 ~/my\_project\_docker/

$

my\_project\_docker/ #directtorio raíz de proyecto docker

|-- README.md

|-- app-mi-agenda/

| |-- Dockerfile #Dokerfile para App Vue.js

| |-- README.md

| |-- dist/ #contruir dist/ para lanzarla a producción

| | |-- assets/

| | |-- index.html

| | `-- vite.svg

| |-- index.html

| |-- node\_modules/

| |-- package-lock.json

| |-- package.json

| |-- public/

| |-- src/

| `-- vite.config.js

|-- db/ ##scripts y backups sql

| |-- backup.sql #Scrip creacion de tablas y insercciones

| `-- init.sql #Scrip crear Ususario y otorgar privilegios

|-- demoApiAgenda/

| |-- Dockerfile #Dockerfile para Api Spring Boot

| |-- HELP.md

| |-- mvnw

| |-- mvnw.cmd

| |-- pom.xml

| |-- src/

| `-- target/

| |-- demoAgenda-0.0.1-SNAPSHOT.jar

|-- docker-compose.yml #Docker-compose para orquestar los 3 servicios

|-- docs/

`-- ...Docker.docx # documentación del Reporte del proyecto(Word)

10 directories, 16 files

Antes de pasar a la configuración , vamos a explicar que en nuestro caso empezaremos con la configuración de nuestro archivo Docker-Compose .yml vacío para irle agregado las configuraciones respectivas de cada servicio que utilizaremos, en este caso primero agregaremos la configuración de la base de datos en el servidor de PostgreSQL.

## 2. Dockerfile para la base de datos

En la carpeta **db**, no necesitamos un **Dockerfile**, pero necesitaremos un archivo de inicialización para PostgreSQL. Vamos a usar una imagen oficial de PostgreSQL en su versión 14 y un volumen para cargar el backup.

Para una configuración más segura y flexible en entorno de trabajo postgres, es recomendable usar variables de entorno para la configuración de nuestra aplicación, especialmente para datos sensibles como las credenciales de la base de datos. Además, es una buena práctica no utilizar el usuario **postgres** para aplicaciones en producción, sino crear un usuario con los privilegios necesarios.

Como en mi caso creare un usuario ‘**usr\_admin’** con password ‘**usr\_admin’** con sus respectivos asignación de privilegios a la base de datos ‘**db\_agenda’,** una vez dejando esto claro avancemos con la configuración. Vamos a manejarlo de una manera que no requiera ejecutar comandos manualmente en la base de datos una vez que el contenedor esté en funcionamiento. Utilizaremos scripts de inicialización de Docker para crear el usuario y la base de datos con los privilegios necesarios.

Estos son los pasos detallados para no perdernos:

**Paso 2.1. Preparar el script de inicialización para PostgreSQL**

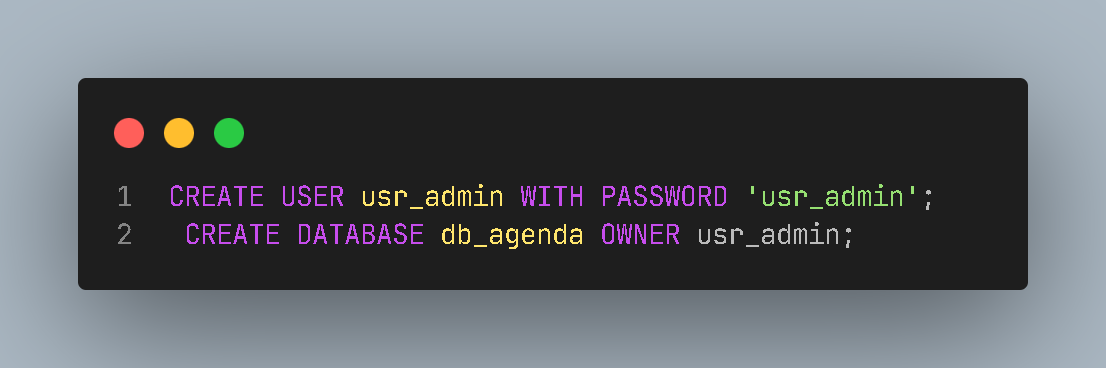
Vamos a crear un script de inicialización SQL adicional para crear el usuario **usr\_admin** y otorgarle los privilegios necesarios.

**Paso 2.2: Crear el script de inicialización**

En la carpeta **db** de nuestro proyecto, crea un archivo llamado **init.sql** con el siguiente contenido:

CREATE USER usr\_admin WITH PASSWORD 'usr\_admin';

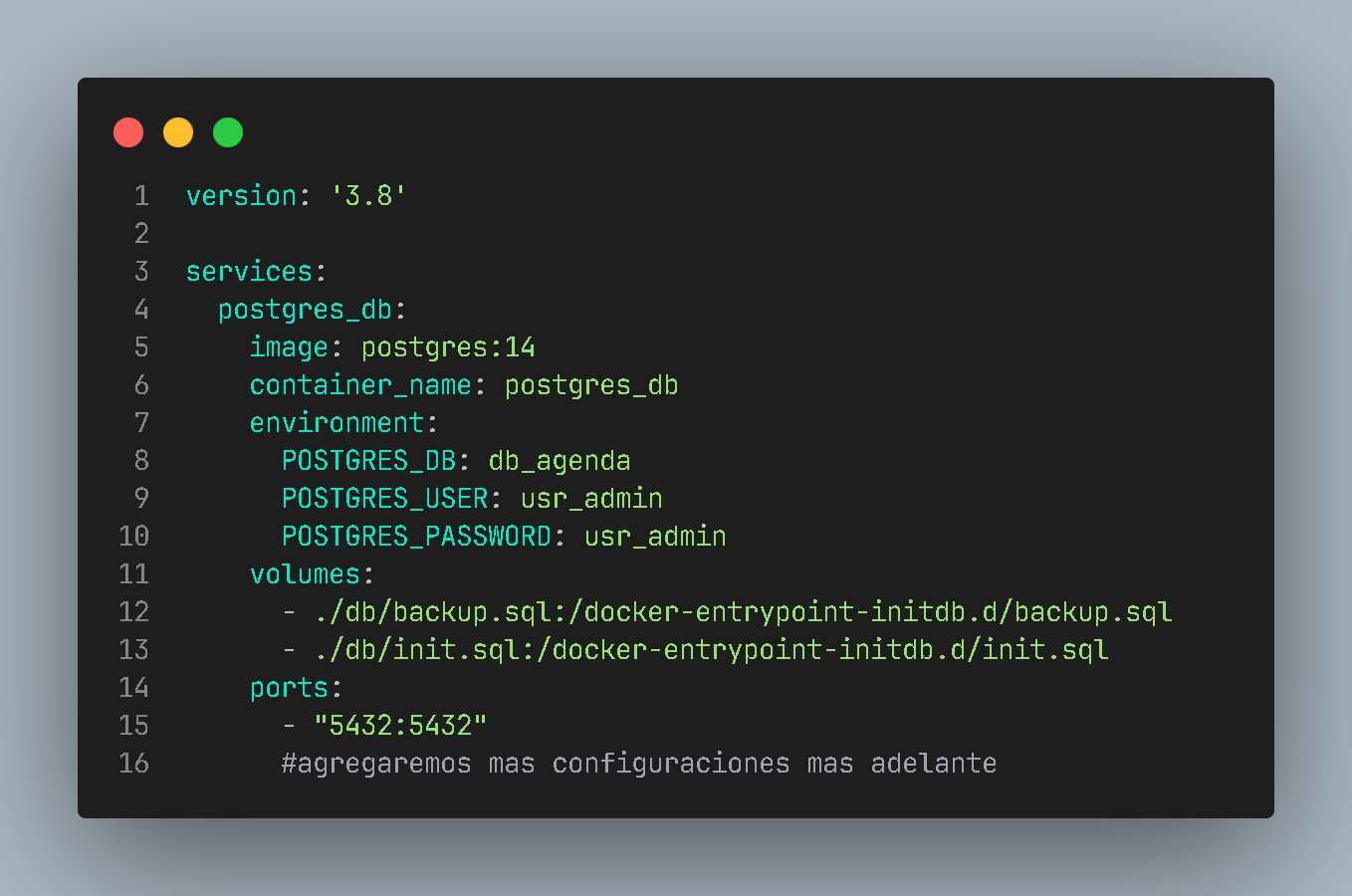
CREATE DATABASE db\_agenda OWNER usr\_admin;

Debería verse así:

**Paso 2.3 Ajustar el archivo docker-compose.yml**

Modificaremos el archivo **docker-compose.yml** para asegurarnos de que PostgreSQL ejecute este script de inicialización cuando el contenedor se levante.

**Paso 2.4: Modificar docker-compose.yml**

****Añade el volumen para el nuevo script de inicialización:

## 3.Configuración de la API (Spring Boot)

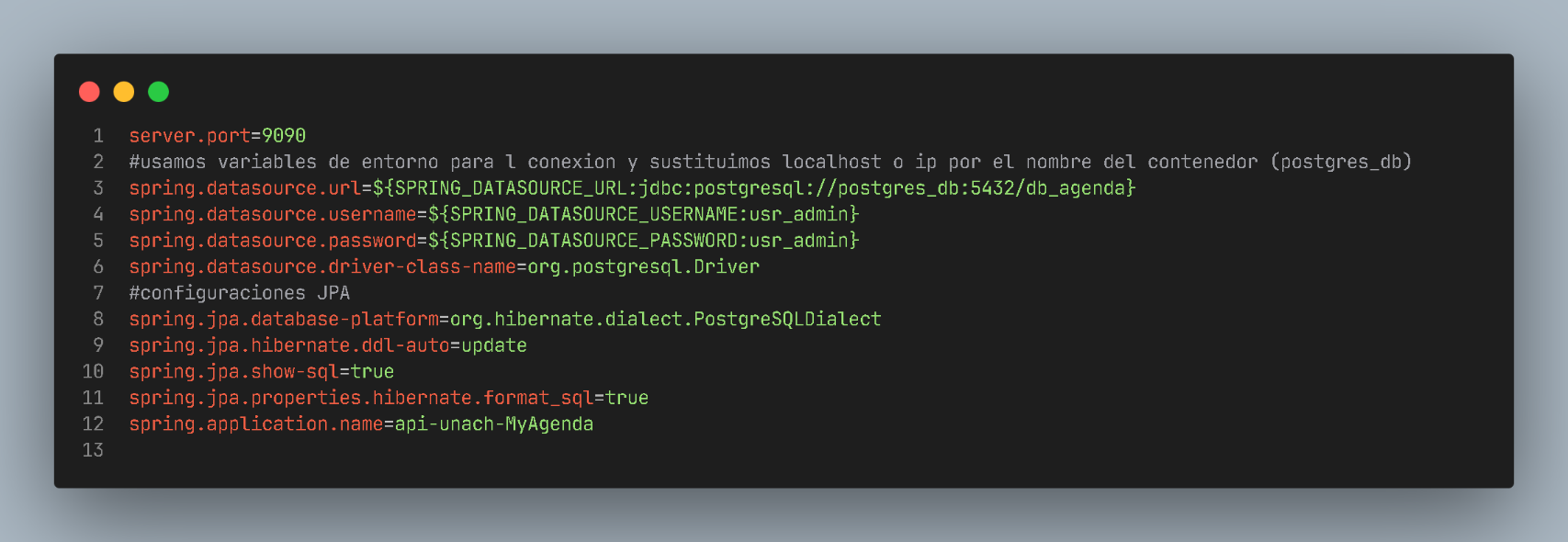
**Paso 3.1 Creación de archivo Dockfile**

Crea un Dockerfile sin extensión en la carpeta demoApiAgenda o donde se encuentre loa carpeta raíz de nuestra aplicación de Spring Boot:

**Dockfile**

**Paso 3.2 Crear el build para tu aplicación Spring Boot**

Primero debemos modificar el archivo ‘**Application.properties’** de nuestro proyecto Spring Boot para establecer las variables de entorno que será establecida por nuestro **Dockfile-Compose.yml** Por lo tanto, recordemos que, para evitar conflictos, no deberíamos usar **localhost** ni **ip** para acceder a la API desde el backend cuando se está usando conexión entre contenedores, en su lugar se usa el nombre del contenedor de la base de datos, en este caso ‘**postgres\_db’**:

**Paso 3.3 Actualización Application.properties**

Primero, compila tu aplicación Spring Boot para generar el archivo JAR que se alojará en la carpeta **target**.

1. **Navega a la carpeta de tu aplicación Spring Boot desde terminal**:

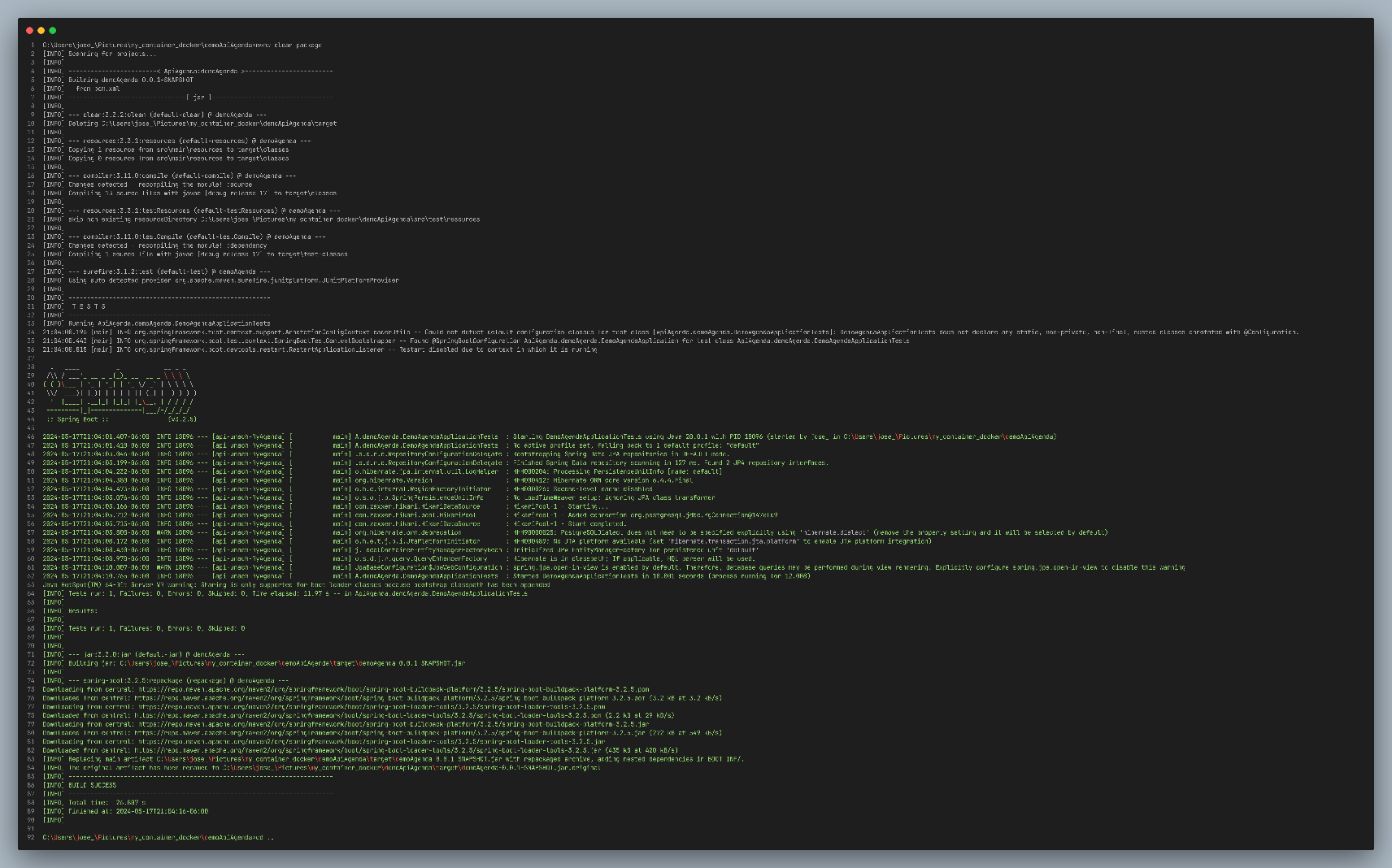
cd path/to/tu\_proyectoApi

en mi caso:

cd my\_project\_docker/demoApiAgenda/

1. **Compila tu aplicación utilizando Maven (o Gradle, dependiendo de la configuración que usemos)**:

mvn clean package

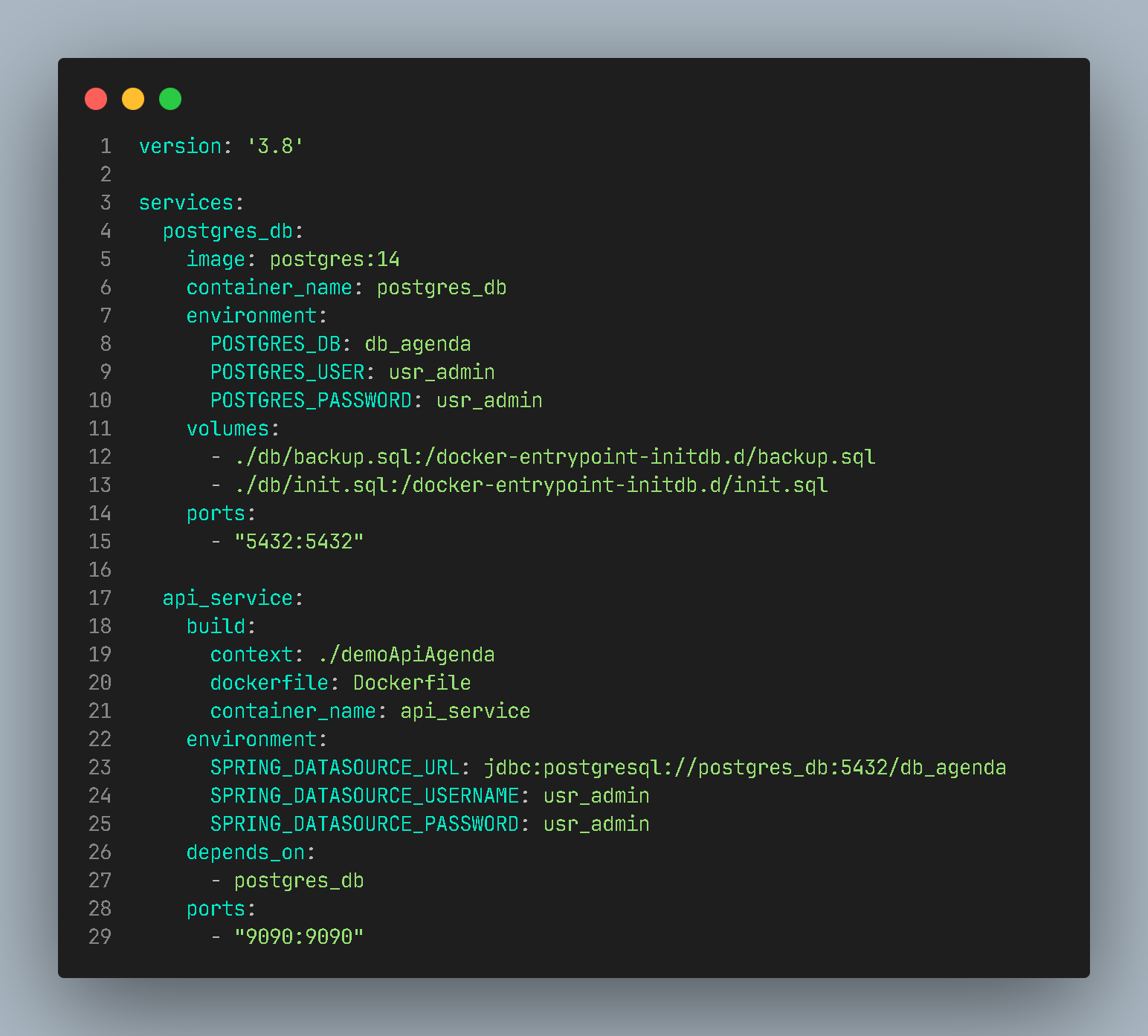
**Ejemplo salida en consola:**

Esto genero un archivo JAR en la carpeta **target**. hay que asegurarse de que el archivo generado sea **demoApiAgenda.jar 0.0.1-SNAPSHOT** o ajusta el **Dockerfile** si es necesario.

Podemos crear el Jar en la carpeta **target.** y decirle a nuestro Dockfile donde se encentra dicho jar y cargarlo a la imagen, pero nos evitaremos de problemas y le diremos a nuestro archivo Compose que lo compile y que lo cargue a nuestra imagen después de haberlo construido.

**Paso 3.4 Actualización del archivo Docker-compose.yml**

Debemos de asegurarnos que las variables de entorno estén incluidas correctamente para la API y además se pondrá un nombre al contenedor de postgres para hacer más flexible y consistente la comunicación entre contenedores y poderlos adminístralos correctamente en este caso para postgres **‘postgres\_db’**  y para la API **‘api\_service’**:

**Docker-compose.yml**

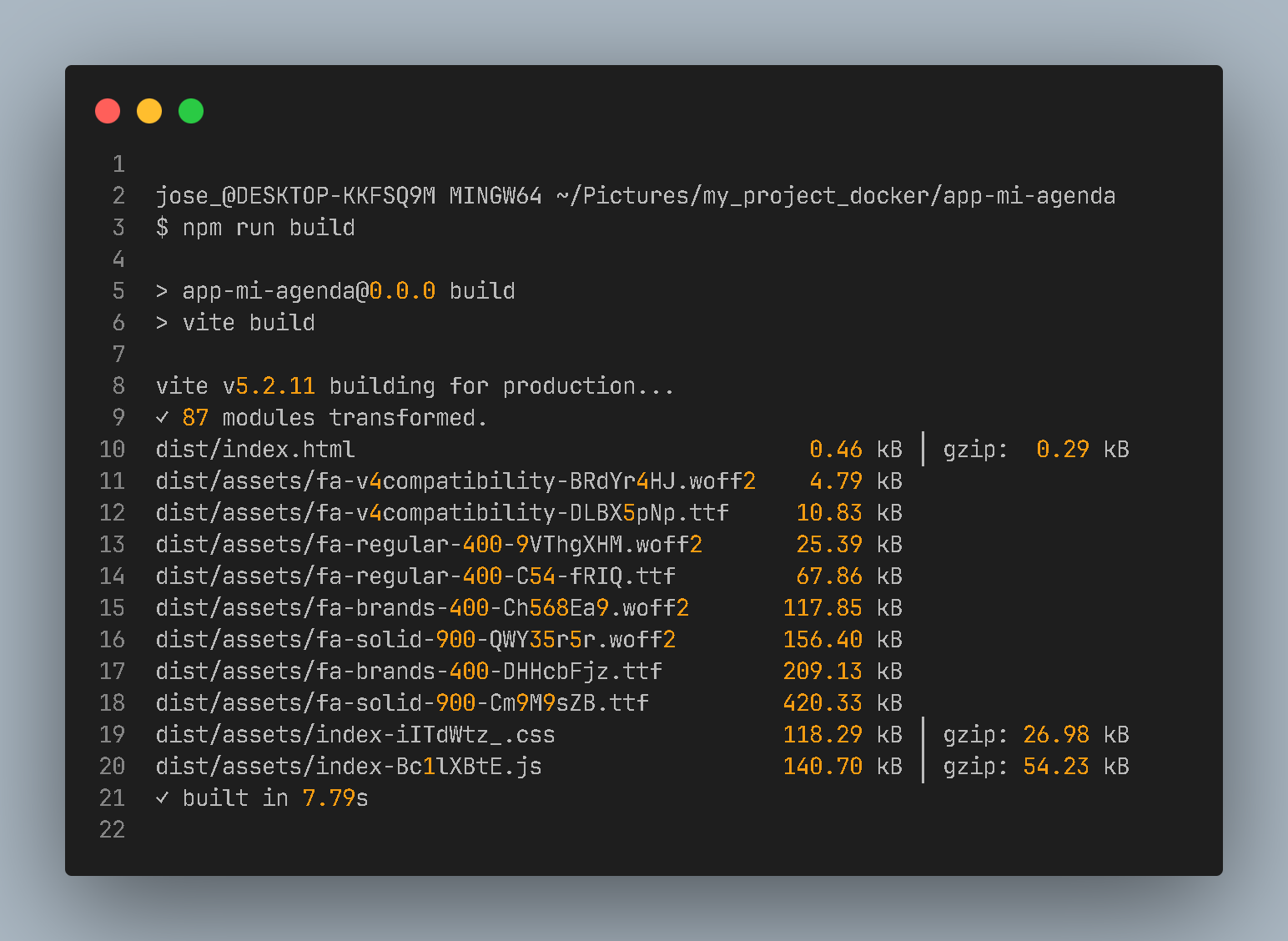
## 4. Configuración de la aplicación (Vue.js)

Necesitamos construir nuestra aplicación Vue.js para generar los archivos estáticos que se alojarán en la carpeta **dist**.

1. **Navega a la carpeta de tu aplicación Vue.js**:
2. **Instala las dependencias**:
3. **Crea el build de producción**:

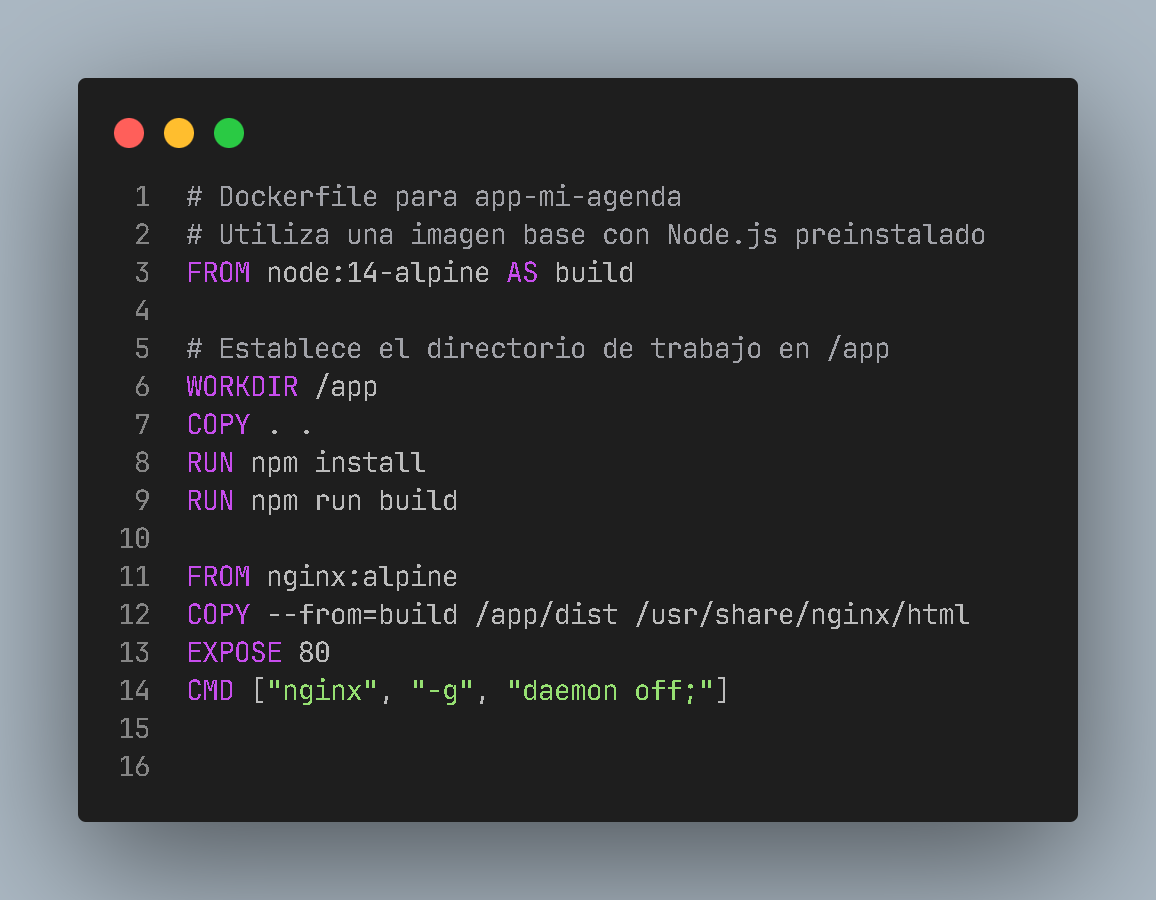
Ejecutar estos comandos:

Esto generará una carpeta **dist** con los archivos estáticos listos para producción.

En la consola debería verse así:

**Pasos finales**

**Paso 4.1 Creación de Dockfile:**

**Dockfile**

Debemos asegurarnos que todos los servicios que realizan solicitudes HTTP al api desde nuestra aplicación frontend deben se configurarse en cada servicio y este funcione sin conflictos al ejecutarse dentro de Docker, es importante no usar **localhost** para conectarse al **API**. En lugar de eso, se debe usar el nombre del servicio definido en **docker-compose.yml**, en este caso es **api\_service.**

Por ejemplo, yo tengo un servicio **categorías\_contactos.js**  se usa el nombre del contenedor al igual que se hizo con la API api\_service en su archivo Application.properties:

Se realizarán los cambios según sean necesarios.

Antes de subir nuestro proyecto a la imagen del contenedor necesitamos crear un archivo ‘**.dockerignore’** en nuestra carpeta raíz de la aplicación **app-mi-agenda** este es opcional , dependiendo de los gustos de cada programador, pero por buena práctica lo haremos; para evitar que ciertos archivos o directorios de nuestro proyecto se incluyan en el contexto de construcción de la imagen Docker.

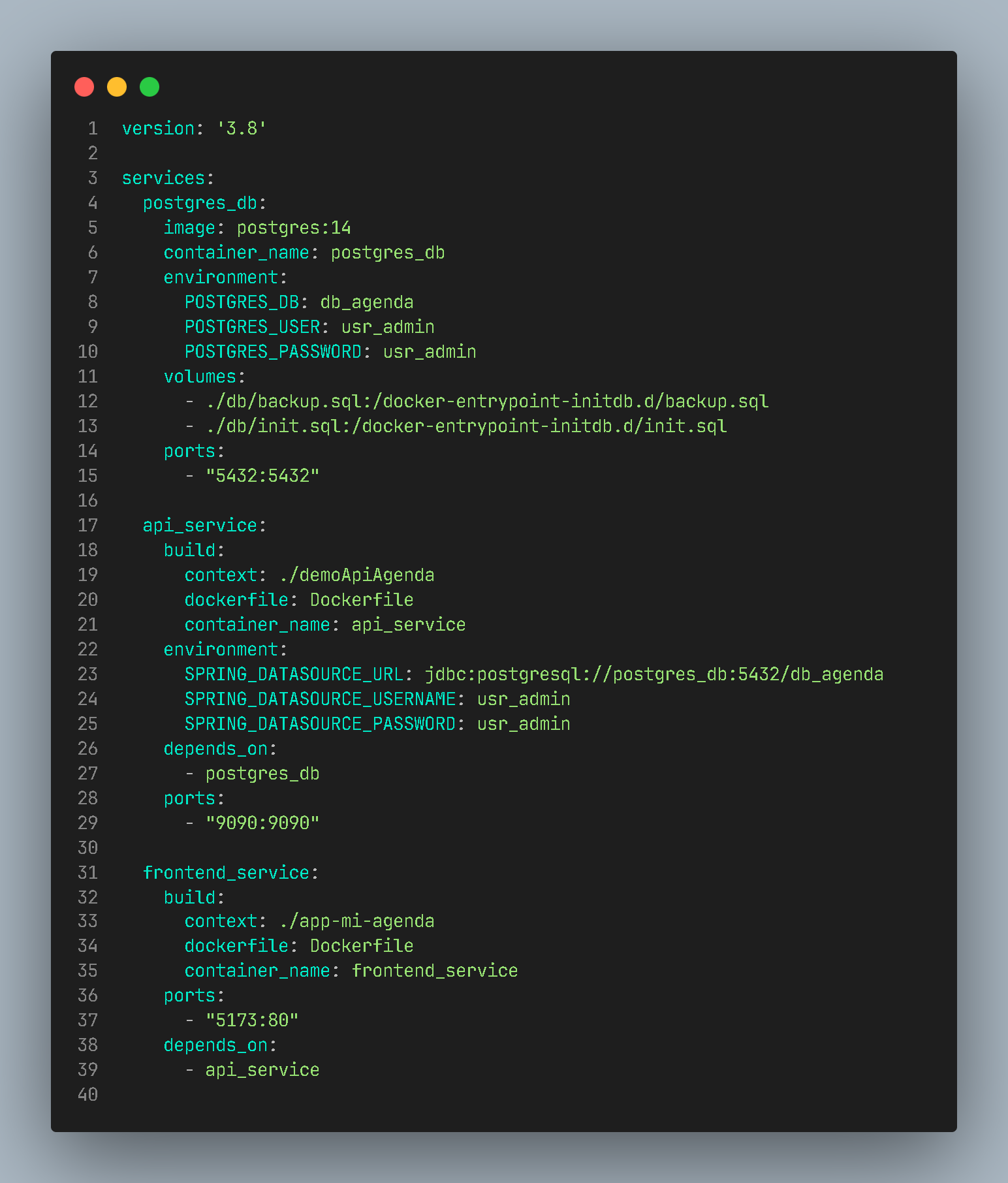
Ejemplo mi archivo .dockerignore de mi aplicación Vue.js:

**.Dockerignore**



En este caso, estamos ignorando el Dockerfile (ya que no es necesario incluirlo en el contexto de construcción), la carpeta **node\_modules** (ya que Docker instalará las dependencias durante la construcción de la imagen), y las carpetas **dist** y **public** que suelen contener archivos generados que no son necesarios para la imagen Docker. Esto puede variar según la estructura y necesidad de cada proyecto que realicemos. Esto lo realice con la importancia de no incluir archivo o directorios innecesarios en la imagen Docker para mantenerla lo más liviana y eficiente posible.

**Paso 4.2 Actualizar Dockfile-compose.yml**

**Docker-compose**

## 5.-Verificación antes de levantar el docker principal.

* **Aplicación Spring Boot**: Verifica que el archivo JAR esté en **demoApiAgenda/target/demoApiAgenda.jar**.
* **Aplicación Vue.js**: Verifica que los archivos de construcción estén en **App-mi-agenda/dist**.
* **Base de datos**: Asegúrate de que **backup.sql** solo contenga la creación de tablas e inserciones, no la creación de la base de datos, también que **init.sql** tenga el script para la creación del usuario **usr\_admin** y ase le asigne privilegios a la base de datos **db\_agenda**.

Siguiendo estos pasos, nuestra aplicación esta lista y configurada para usar variables de entorno, lo que mejora la seguridad y flexibilidad de tu configuración. Además, con Docker Compose, todos los servicios estarán conectados entre sí y configurados para funcionar en los puertos especificados, ahora solo queda levantar el docker principal.

**Como última actualización del archivo docker-compose.yml**

**S**e creo la sección ‘**networks**’¸, esta configuración es crucial para que Docker pueda usar la red personalizada. Permitiendo que los servicios definidos ene l archivo se comuniquen entre sí de manera efectiva.

El archivo ‘**docker-compose.yml** debería verse así:

Vamos a explicar la sección ‘**networks’**:

**1.-Definicion de Red:**

* **‘networks’**: Esta clave define todas las redes que desees utilizar en tu configuración de docker-compose.
* **‘my\_network’**: este es el nombre de la red que se utilizara. Podemos configurarla por un nombre que deseemos , en este caso elegí uno más sencillo.
* **‘driver: bridge’**: El controlador ‘**bridge**

Crea una red de puente estándar que permite que los contenedores se comuniquen entre sí en la misma red. Este es el tipo de red más común para configuraciones de red simple.

**2.-Asignacion de Red a Servicios:**

Cada servicio tiene la clave ’**networks’** donde se especifica a que red debe conectarse.

Esto indica que **‘postgres\_db’**, **’api\_service’**, **‘frontend service’** se conectara a la red **‘my\_network’**. Esto es crucial para que los contendores puedan comunicarse entre sí utilizando los nombres de servicio definidos.

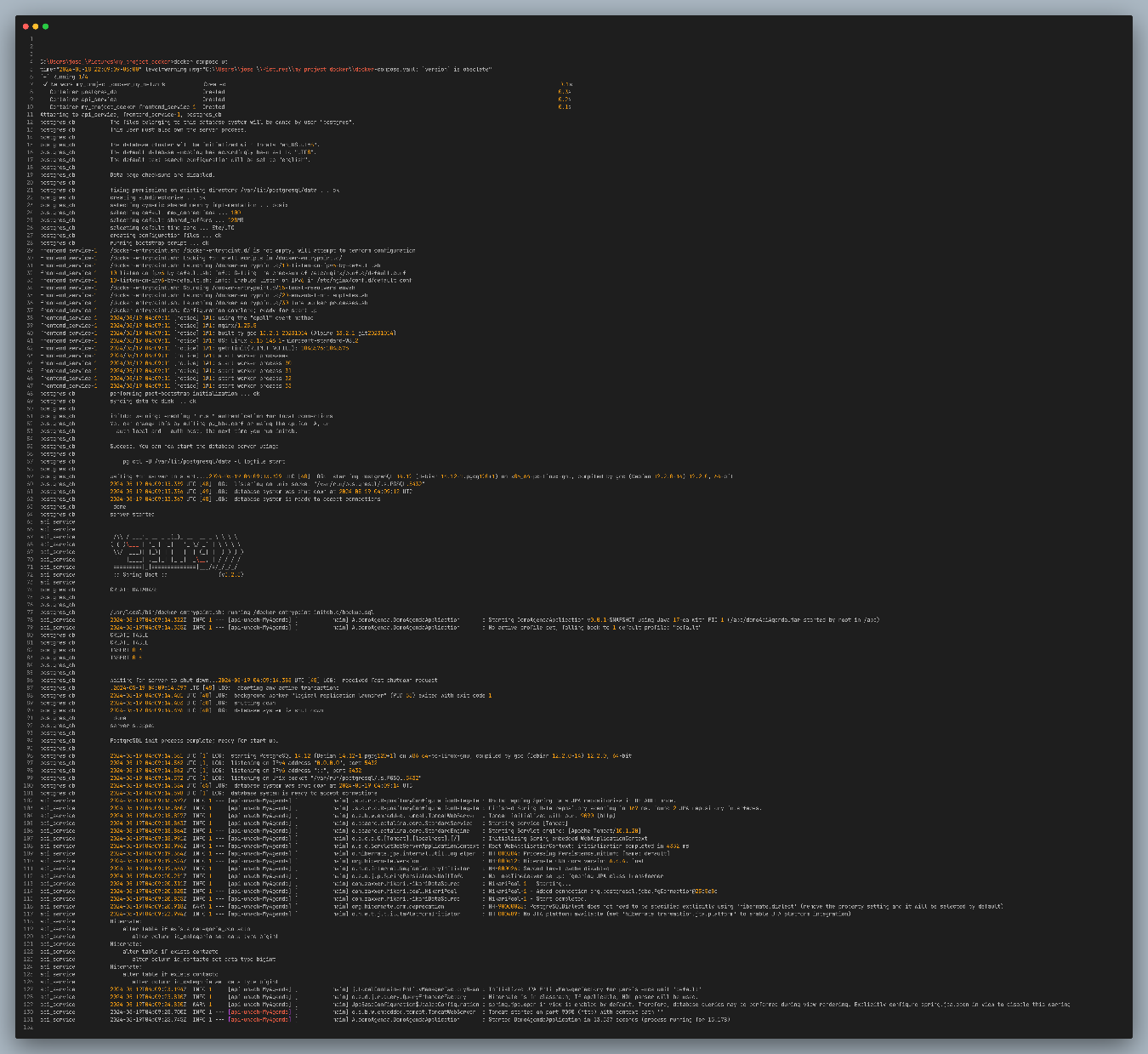
**Comunicación entre Contenedores**

Al usar la red **my\_network**, los servicios pueden referirse entre sí por sus nombres de servicio definidos en el archivo **docker-compose.yml**. Por ejemplo:

* El contenedor **api\_service** puede conectarse a la base de datos PostgreSQL usando **postgres\_db** como el hostname.
* La aplicación frontend puede comunicarse con la API backend usando **api\_service** como el hostname.

**6.- Iniciar los contenedores con Docker Compose**: Desde la raíz del proyecto:

docker-compose up --build

Debería salir algo como esto:

## Estructura Final

**1.- Estructura del proyecto:**

**my\_project\_docker/**

**├── README.md**

**├── app-mi-agenda**

**│ ├── Dockerfile**

**│ ├── README.md**

**│ ├── dist**

**│ ├── index.html**

**│ ├── node\_modules**

**│ ├── package-lock.json**

**│ ├── package.json**

**│ ├── public**

**│ ├── src**

**│ └── vite.config.js**

**├── db**

**│ ├── backup.sql**

**│ └── init.sql**

**├── demoApiAgenda**

**│ ├── Dockerfile**

**│ ├── HELP.md**

**│ ├── mvnw**

**│ ├── mvnw.cmd**

**│ ├── pom.xml**

**│ ├── src**

**│ └── target**

**├── docker-compose.yml**

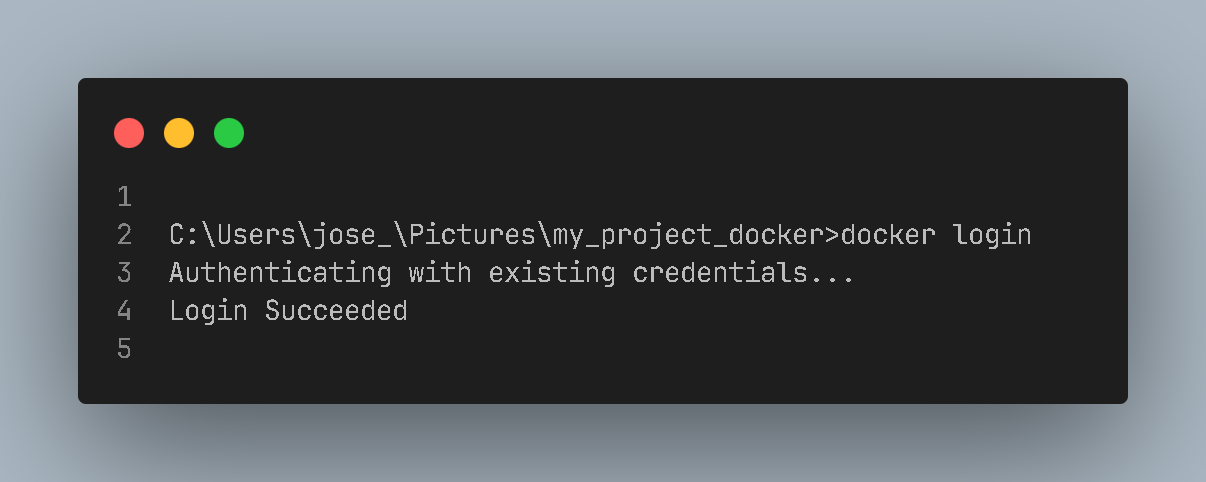
**└── docs**

**├── Instalacion-Docker-&-Implementacion-ServiceFullStack-Agenda-With-Docker.docx**

**└── snap\_code**

## Subir y Compartir Los repositorios de Docker Hub

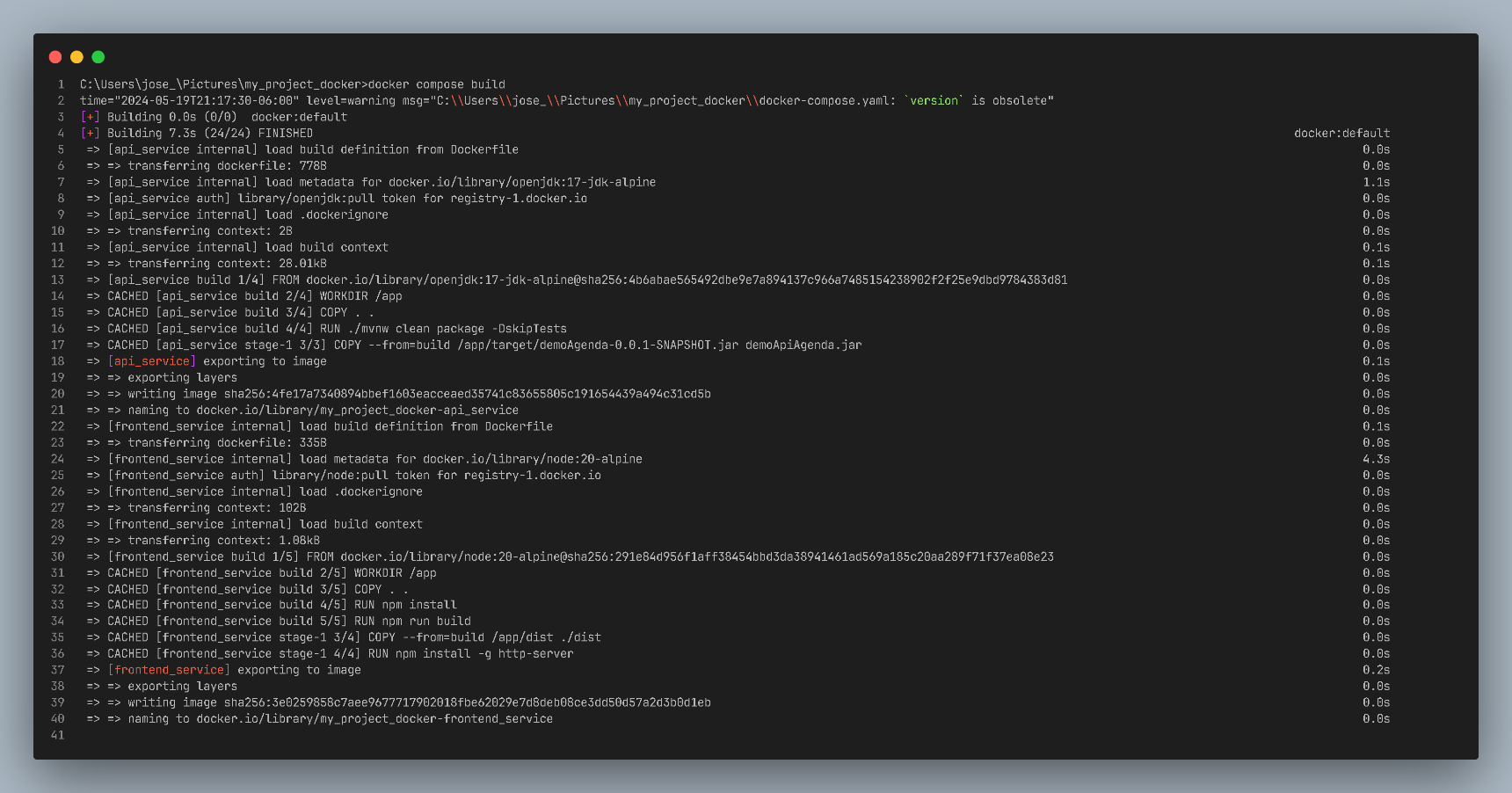
Para compartir tus imágenes de Docker con alguna persona para que pueda calificar y probar los contenedores en su máquina, puedes seguir estos pasos:

Antes de los comenzar debemos logearnos desde nuestra terminal, Ejemplo: En mi caso tengo instalado Docker Desktop e iniciado sesión. Por ende, no me salteo la confirmación de credenciales. En tu caso de no estar logeado en la aplicación Desktop te solicitara las credenciales.

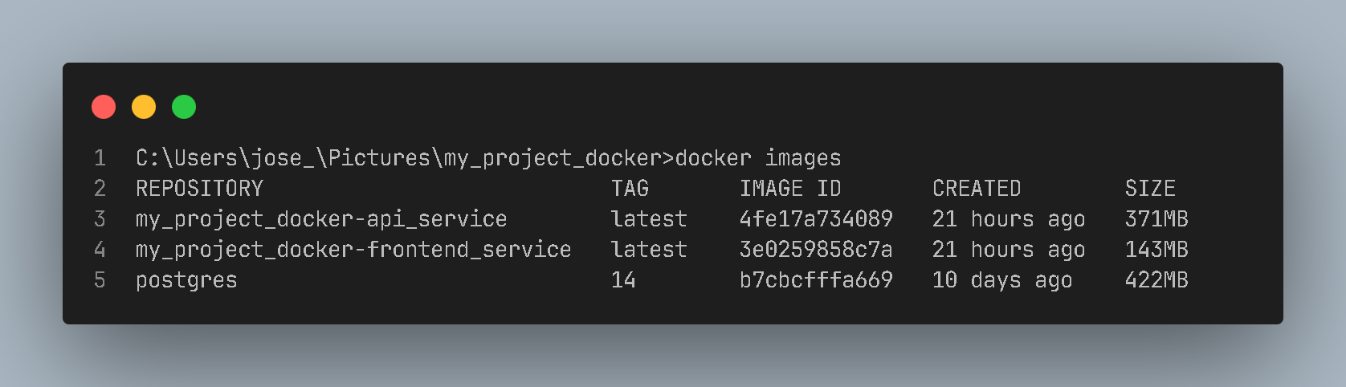
**Paso 1: Subir las Imágenes a Docker Hub**

Como ya hemos discutido, primero necesitas subir tus imágenes a Docker Hub. Aquí está un resumen rápido de los pasos para hacerlo:

1. **Construir las imágenes (si aún no lo has hecho):**

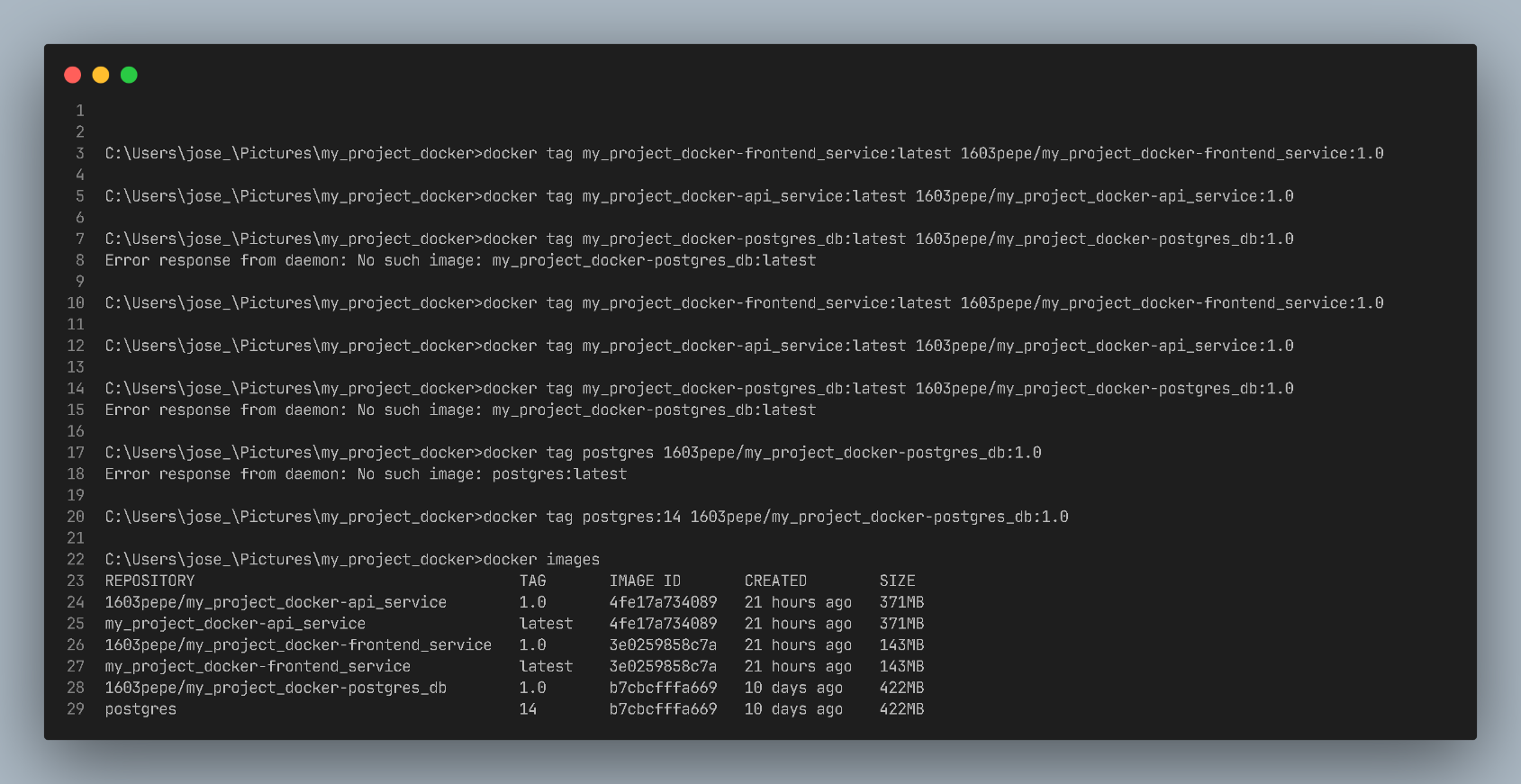
docker-compose build

verificamos que las imágenes se hayan completado exitosamente:



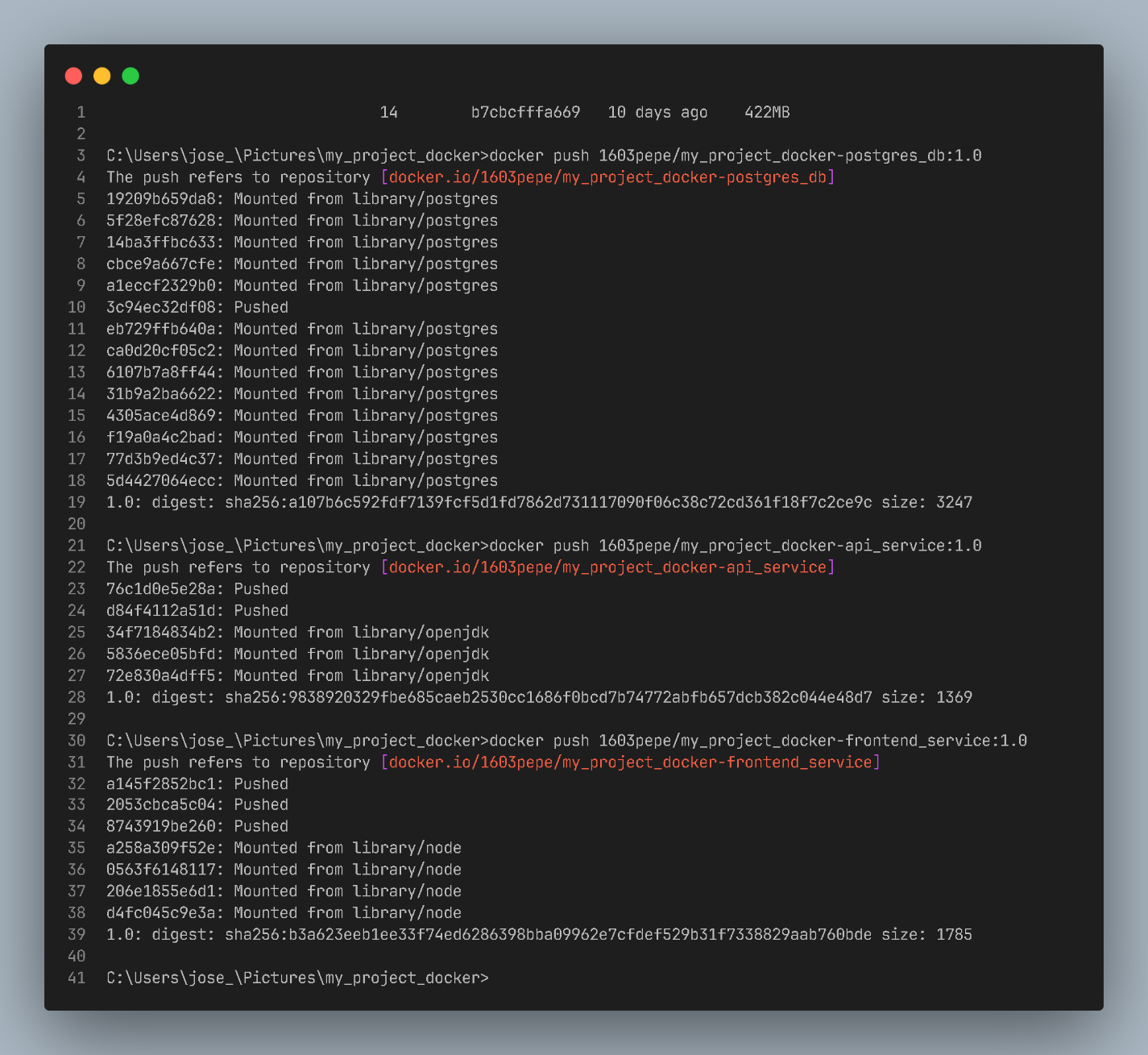
1. **Taguear(asignar tag) las imágenes para Docker Hub:**

* docker tag **my\_project\_docker-frontend\_service:latest** 1603pepe/my\_project\_docker-frontend\_service:1.0
* docker tag **my\_project\_docker-api\_service:latest** 1603pepe/my\_project\_docker-api\_service:1.0
* docker tag **postgres\_db:14** 1603pepe/my\_project\_docker-postgres\_db:1.0

verificamos nuevamente:

1. **Subir las imágenes a Docker Hub:**

docker push your\_dockerhub\_username/my\_project\_docker-images:versión

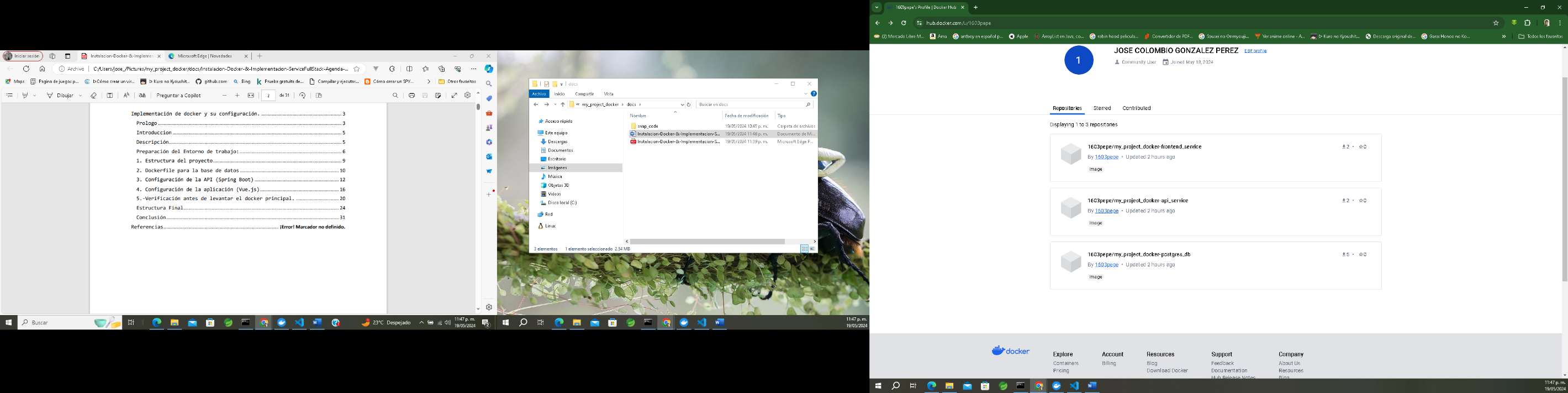
en la terminal debería de aparecer algo como esto:

Siguiendo los las instrucciones del paso 1, debería poderse ver las imágenes a Docker y luego, cualquier persona a Docker-Hub podrá descargar las imágenes en sus propios entornos Docker.

**Paso 2: Compartir los Repositorios de Docker Hub**

Una vez que las imágenes estén en Docker Hub, puedes compartir los enlaces de los repositorios con tu maestro Zacarías. Aquí dejo los links de las imágenes:

* Frontend Service: [**https://hub.docker.com/r/1603pepe/my\_project\_docker-frontend\_service**](https://hub.docker.com/r/1603pepe/my_project_docker-frontend_service)
* API Service: [**https://hub.docker.com/r/1603pepe/my\_project\_docker-api\_service**](https://hub.docker.com/r/1603pepe/my_project_docker-api_service)
* Postgres DB: [**https://hub.docker.com/r/1603pepe/my\_project\_docker-postgres\_db**](https://hub.docker.com/r/1603pepe/my_project_docker-postgres_db)

Ejemplo:

Ahora qué tal si queremos ejecutar los contenedores. ¿Como lo hacemos?, pues agregamos un paso más.

**Paso 3: Instrucciones para Ejecutar los Contenedores**

Proporciona a tu maestro las instrucciones necesarias para ejecutar los contenedores. Aquí hay un ejemplo de lo que podría incluir:

1. **Asegurarse de tener Docker y Docker Compose instalados:**
   * Instalar Docker
   * Instalar Docker Compose
2. **Instrucciones de construcción:**

* Clonar tu repositorio de GitHub

Puedes hacer esto desde tu terminal de comandos con la instrucción:

**Git** [**https://github.com/pepe1603/my\_project\_docker.git**](https://github.com/pepe1603/my_project_docker.git)

Ahora puedes moverte a la carpeta raíz del proyecto

**cd my\_project\_docker**

1. **Instrucciones para ejecutar el proyecto:**

Una vez que te encuentres dentro de la carpeta dentro del directorio raíz ejecuta el comando:

docker-compose up --build

Esto construirá en caso de no estar construidos aun y ejecutará los servicios **postgres\_db**, **api\_service** y **frontend service**.

En la terminal nos aparecerá resultado que nos indican los puertos en los que están corriendo los servicios.

**Acceso a los Servicios**

* Frontend: [**http://localhost:5175**](http://localhost:5175) **(puedes usar el navegador)**
* API: [**http://localhost:9090**](http://localhost:9090) **(puedes usar postman para test)**
* Base de datos: **localhost:5432**

Con todos estos pasos definidos, deberías poder probar los contenedores y los servicios en tu máquina.

## Conclusión

Mediante la configuración adecuada de Dockerfile y **docker-compose.yml**, se facilita el desarrollo, despliegue y administración de aplicaciones complejas que constan de múltiples servicios. Esto no solo asegura un entorno de ejecución consistente, sino que también simplifica el proceso de integración y despliegue continuo cuando queremos implementar más de un servicio y migrar o alojar en sistemas de alojamiento en la nube o servidores de producción sin tener ese problema clásico de ‘ **pero si en mi maquina funcionaba**’, por ende puedes Seguir esta guía, y tendrás configurados tus servicios en Docker de manera que se comuniquen correctamente entre sí usando los nombres de los servicios de Docker en lugar de **localhost**. Esto asegura que tu API se conecte a la base de datos y que tu aplicación frontend pueda comunicarse con la API sin problemas de configuración de red que implementa docker internamente.

# Referencias

Docker.com. (s.f.). *docker.com*. Obtenido de https://www.docker.com/

Hostinger.com. (s.f.). *Hostinger-Tutoriales*. Obtenido de https://www.hostinger.es/tutoriales/que-es-docker

https://www.docker.com/101-tutorial/