https://github.com/fruizotero/back-calculadora-template

Claro, aquí tienes ejemplos de `Dockerfile` y `docker-compose.yml` con varias alternativas disponibles para configurar contenedores.

1. Ejemplo de `Dockerfile`:

Este `Dockerfile` se utiliza para crear una imagen Docker a partir de una aplicación Node.js.

Dockerfile para una aplicación Node.js:

```
dockerfile
Copiar código
# Usa una imagen base de Node.js
FROM node:18

# Establece el directorio de trabajo en /app
WORKDIR /app

# Copia los archivos package.json y package-lock.json
COPY package*.json ./

# Instala las dependencias de la aplicación
RUN npm install

# Copia todo el código fuente al contenedor
COPY . .

# Expone el puerto en el que la aplicación escuchará
EXPOSE 3000

# Define el comando para ejecutar la aplicación
CMD ["npm", "start"]
```

Alternativas en Dockerfile:

- **Construcción con archivos específicos**: Puedes usar `COPY` o `ADD` para copiar archivos específicos o directorios.
- **Uso de** `ENTRYPOINT`: En lugar de `CMD`, puedes usar `ENTRYPOINT` para definir cómo se inicia la aplicación, lo que no se puede sobrescribir fácilmente en el comando de ejecución.

Ejemplo con `ENTRYPOINT`:

```
dockerfile
Copiar código
# Usa una imagen base de Node.js
FROM node:18

WORKDIR /app
COPY . .

RUN npm install

EXPOSE 3000
```

```
ENTRYPOINT ["npm", "start"]
```

Este enfoque asegura que el contenedor siempre ejecute `npm start` cuando se inicie.

2. Ejemplo de `docker-compose.yml`:

En este ejemplo, vamos a definir un servicio que ejecuta la aplicación Node.js (de nuestro `Dockerfile`) y también configura un servicio de base de datos PostgreSQL.

docker-compose.yml básico con Node.js y PostgreSQL:

```
yaml
Copiar código
version: '3.8'

services:
  # Servicio de la aplicación Node.js
  app:
    build: .
    ports:
      - "3000:3000"  # Mapea el puerto 3000 del contenedor al puerto 3000 de la máquina local
    environment:
      - NODE_ENV=development
    depends_on:
      - db  # Asegura que la base de datos se inicie antes de la app

  # Servicio de PostgreSQL
  db:
    image: postgres:13
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: myapp
    ports:
      - "5432:5432"  # Mapea el puerto de PostgreSQL
    volumes:
      - postgres_data:/var/lib/postgresql/data  # Volumen persistente para la base de datos

volumes:
  postgres_data:  # Define un volumen persistente para los datos de PostgreSQL
```

Alternativas en Docker Compose:

- **Uso de `networks`**: Puedes definir redes personalizadas para la comunicación entre contenedores.

```
yaml
Copiar código
networks:
  custom_network:
    driver: bridge
```

Y asignar a cada servicio:

```
yaml
Copiar código
app:
  networks:
    - custom_network
db:
  networks:
    - custom_network
```

- **Reiniciar contenedores automáticamente**: Usa `restart` para que los contenedores se reinicien si fallan.

```
yaml
Copiar código
app:
  restart: always  # Reinicia el contenedor si se detiene
```

- **Dependencias con** `depends_on`: Esto asegura que el servicio de base de datos se inicie antes que la aplicación.

```
yaml
Copiar código
app:
  depends_on:
    - db
```

3. Ejemplo de Docker Compose con Redis y Node.js (Escalado y Uso de Caché):

En este ejemplo, tenemos un servicio de Node.js que se conecta a un servicio de Redis, además de usar la opción de escalado para el servicio de la aplicación.

```
yaml
Copiar código
version: '3.8'

services:
  app:
    build: .
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=production
    depends_on:
      - redis
    scale:
      replicas: 3  # Escala el servicio para que se ejecute en 3 contenedores
    networks:
      - backend

  redis:
    image: redis:alpine
    ports:
      - "6379:6379"
```

```
    networks:
      - backend

networks:
  backend:
    driver: bridge
```

En este caso, el servicio app (Node.js) usa Redis como caché, y se escalan tres instancias de la aplicación. La opción scale te permite gestionar fácilmente la cantidad de réplicas de un servicio.

4. Docker Compose con Variables de Entorno en un Archivo .env:

Si prefieres manejar las variables de entorno fuera del archivo docker-compose.yml, puedes usar un archivo .env.

.env (archivo de entorno):

```
env
Copiar código
DB_USER=user
DB_PASSWORD=password
DB_NAME=myapp
```

docker-compose.yml con el archivo .env:

```
yaml
Copiar código
version: '3.8'

services:
  app:
    build: .
    ports:
      - "3000:3000"
    environment:
      - DB_USER=${DB_USER}
      - DB_PASSWORD=${DB_PASSWORD}
      - DB_NAME=${DB_NAME}
    depends_on:
      - db

  db:
    image: postgres:13
    environment:
      POSTGRES_USER: ${DB_USER}
      POSTGRES_PASSWORD: ${DB_PASSWORD}
      POSTGRES_DB: ${DB_NAME}
    ports:
      - "5432:5432"
```

Este enfoque te permite manejar variables de entorno sin tener que hardcodearlas directamente en el archivo docker-compose.yml, lo que es útil para mantener configuraciones más seguras.

5. Docker Compose con un Servicio Web y un Backend en Diferentes Contenedores (Node.js y Nginx):

Aquí tienes un ejemplo de cómo se puede configurar un contenedor para un backend (Node.js) y otro contenedor para un frontend con Nginx.

```
yaml
Copiar código
version: '3.8'

services:
  backend:
    build: ./backend
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=production

  frontend:
    image: nginx:alpine
    ports:
      - "80:80"
    volumes:
      - ./frontend:/usr/share/nginx/html:ro
    depends_on:
      - backend
```

En este caso, el servicio de `backend` está en un contenedor con Node.js, mientras que el servicio de `frontend` está configurado con Nginx, sirviendo archivos estáticos.

Resumen:

1. `Dockerfile`: Dependiendo de tus necesidades, puedes usar `CMD` o `ENTRYPOINT`, y definir si usarás `COPY` o `ADD` para tus archivos.
2. `docker-compose.yml`: Usualmente, se usan `depends_on` y redes (`networks`) para controlar la relación entre servicios. Puedes escalar servicios con `scale` y manejar variables con `.env`.
3. **Escalabilidad y Volúmenes**: Docker Compose permite manejar escalabilidad y almacenamiento persistente de manera sencilla con la opción `volumes`.

Estas son algunas formas básicas y avanzadas de configurar contenedores y servicios con Docker. Puedes adaptarlas según tu proyecto y requisitos específicos.

- `docker --version`

- **Ver información sobre Docker (sistema):**

```
bash
Copiar código
docker info
```

2. Comandos para Gestionar Imágenes:

- **Listar imágenes disponibles:**

```
bash
Copiar código
docker images
```

- **Obtener una imagen desde Docker Hub:**

```
bash
Copiar código
docker pull <imagen>
```

Ejemplo:

```
bash
Copiar código
docker pull node:18
```

- **Eliminar una imagen:**

```
bash
Copiar código
docker rmi <imagen_id>
```

- **Construir una imagen a partir de un Dockerfile:**

```
bash
Copiar código
docker build -t <nombre_imagen> <directorio>
```

Ejemplo:

```
bash
Copiar código
docker build -t mi-app .
```

3. Comandos para Gestionar Contenedores:

- **Ver contenedores en ejecución:**

```
bash
Copiar código
docker ps
```

- **Ver todos los contenedores (incluidos los detenidos):**

```
bash
Copiar código
docker ps -a
```

- **Ejecutar un contenedor:**

```
bash
Copiar código
docker run <opciones> <imagen>
```

Ejemplo:

```
bash
Copiar código
docker run -d -p 3000:3000 node:18
```

Opciones comunes:

  - `-d`: Ejecuta el contenedor en segundo plano (modo "detached").
  - `-p`: Mapea puertos entre el contenedor y el host.

- **Acceder a un contenedor en ejecución (interactivo):**

```
bash
Copiar código
docker exec -it <container_id> /bin/bash
```

- **Detener un contenedor:**

```
bash
Copiar código
docker stop <container_id>
```

- **Reiniciar un contenedor:**

```
bash
Copiar código
docker restart <container_id>
```

- **Eliminar un contenedor:**

```
bash
Copiar código
docker rm <container_id>
```

- **Eliminar contenedores detenidos:**

```
bash
Copiar código
docker container prune
```

4. Comandos para Gestionar Redes:

- **Listar redes existentes:**

```
bash
Copiar código
docker network ls
```

- **Crear una nueva red:**

```
bash
Copiar código
docker network create <nombre_red>
```

- **Conectar un contenedor a una red:**

```
bash
Copiar código
docker network connect <nombre_red> <container_id>
```

- **Desconectar un contenedor de una red:**

```
bash
Copiar código
docker network disconnect <nombre_red> <container_id>
```

- **Eliminar una red:**

```
bash
Copiar código
docker network rm <nombre_red>
```

5. Comandos para Gestionar Volúmenes:

- **Listar volúmenes:**

```
bash
Copiar código
docker volume ls
```

- **Crear un volumen:**

```
bash
Copiar código
docker volume create <nombre_volumen>
```

- **Ver detalles de un volumen:**

```
bash
Copiar código
docker volume inspect <nombre_volumen>
```

- **Eliminar un volumen:**

```
bash
Copiar código
docker volume rm <nombre_volumen>
```

- **Eliminar volúmenes no utilizados:**

```
bash
Copiar código
```

```
docker volume prune
```

6. Comandos para Docker Compose:

- **Iniciar servicios definidos en un archivo** `docker-compose.yml`**:**

```
bash
Copiar código
docker-compose up
```

Para ejecutarlo en segundo plano (detached):

```
bash
Copiar código
docker-compose up -d
```

- **Detener los servicios de Docker Compose:**

```
bash
Copiar código
docker-compose down
```

- **Ver los logs de los contenedores en Docker Compose:**

```
bash
Copiar código
docker-compose logs
```

- **Escalar servicios (por ejemplo, 3 réplicas de un servicio):**

```
bash
Copiar código
docker-compose up --scale <servicio>=<numero_de_replicas>
```

Ejemplo:

```
bash
Copiar código
docker-compose up --scale app=3
```

- **Ejecutar un comando en un contenedor de un servicio:**

```
bash
Copiar código
docker-compose exec <servicio> <comando>
```

Ejemplo:

```
bash
Copiar código
docker-compose exec app /bin/bash
```

7. Comandos para Gestionar Logs:

- **Ver los logs de un contenedor:**

```
bash
Copiar código
docker logs <container_id>
```

- **Seguir los logs de un contenedor en tiempo real:**

```
bash
Copiar código
docker logs -f <container_id>
```

8. Comandos de Utilidades y Diagnóstico:

- **Ver la utilización de recursos (CPU, memoria, etc.) de los contenedores:**

```
bash
Copiar código
docker stats
```

- **Ver los eventos de Docker:**

```
bash
Copiar código
docker events
```

- **Ver las métricas del contenedor:**

```
bash
Copiar código
docker top <container_id>
```

9. Comandos Avanzados de Docker:

- **Generar una nueva imagen desde un contenedor existente:**

```
bash
Copiar código
docker commit <container_id> <nombre_imagen>
```

- **Crear un contenedor a partir de una imagen sin ejecutarlo:**

```
bash
Copiar código
docker create <imagen>
```

- **Ver el historial de una imagen:**

```
bash
Copiar código
docker history <imagen>
```

- **Obtener el ID de un contenedor o imagen:**

```
bash
Copiar código
docker inspect <container_id>
```

10. Comandos para Administrar el Docker Daemon:

- **Iniciar el servicio Docker:**

```
bash
Copiar código
sudo systemctl start docker
```

- **Detener el servicio Docker:**

```
bash
Copiar código
sudo systemctl stop docker
```

- **Reiniciar el servicio Docker:**

```
bash
Copiar código
sudo systemctl restart docker
```

- **Ver el estado del servicio Docker:**

```
bash
Copiar código
sudo systemctl status docker
```