Parte Práctica del Segundo Parcial

Jhonatan Cabezas - 70416
Luis Callapa - 68881
Ernesto Juarez - 68763
Diego Ledezma - 68779
Adrian Sánchez - 69546

Facultad de Ingeniería y Arquitectura

Ingeniería de Sistemas Computacionales

Certificación DevOps

10 de Diciembre de 2025

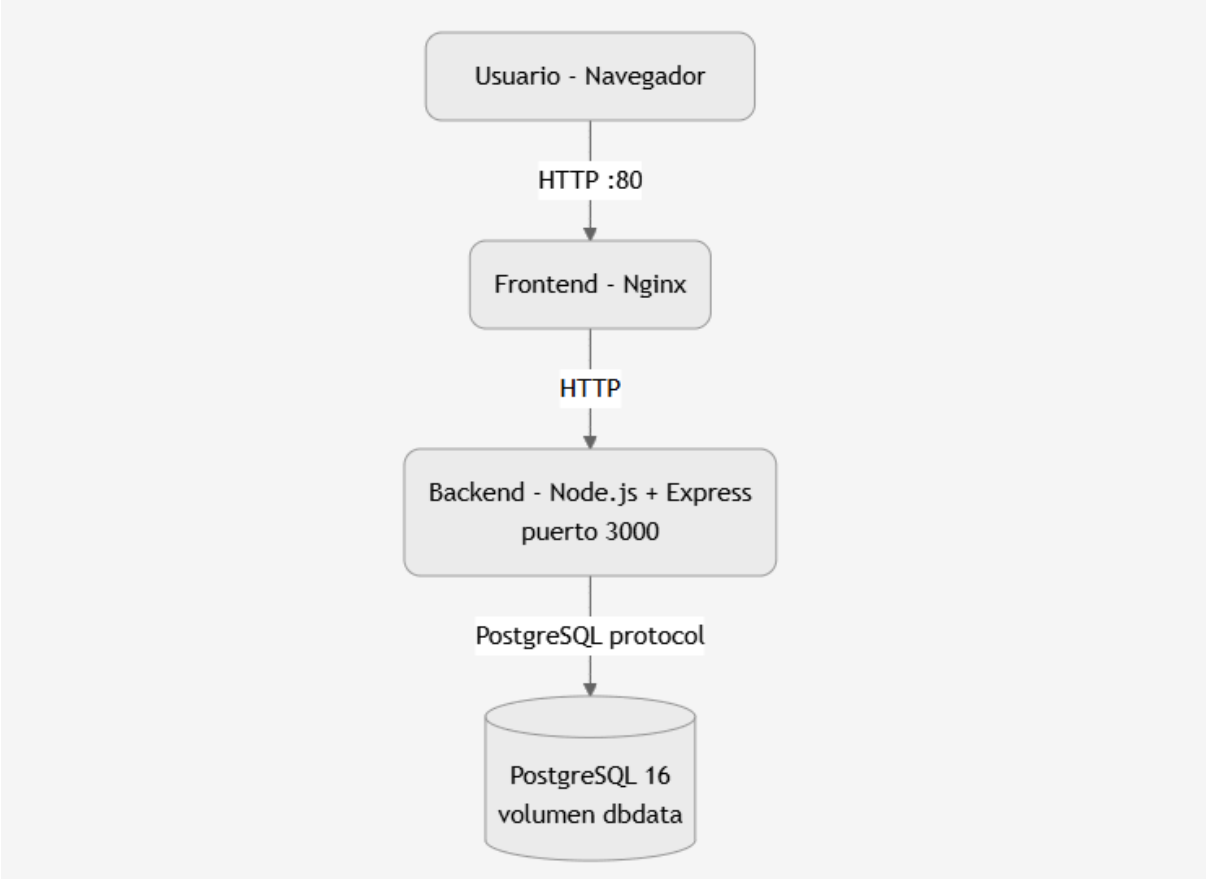
ÍNDICE

2. Requisitos y Alcance

Requisito	Estado	Observaciones
Frontend React + Vite + TypeScript + Tailwind	Implemented	Deploy con Nginx
Backend Node.js + Express + TypeScript	Implemented	Prisma ORM + JWT
Base de datos PostgreSQL 16	Implemented	Contenedor dedicado + volumen persistente
Dockerización completa (3 servicios)	Implemented	Dockerfiles multistage (backend) y multi-stage (frontend → Nginx)
Docker Compose para desarrollo y producción	Implemented	docker-compose.yml en raíz
CI/CD con GitHub Actions	Implemented	3 workflows independientes + 1 global
Despliegue automatizado en AWS EC2	Implemented	appleboy/scp-action + appleboy/ssh-action
Backups automáticos de base de datos	Implemented	backup_db.sh + limpieza de backups antiguos
Logs centralizados	Implemented	carpeta logs/ + rotación manual
Restauración de backup	Implemented	restore_backup.sh probado
Gestión segura de secretos	Implemented	GitHub Secrets + .env creado en runtime en EC2

3. Arquitectura

3.1 Diagrama de arquitectura



3.2 Descripción de Componentes

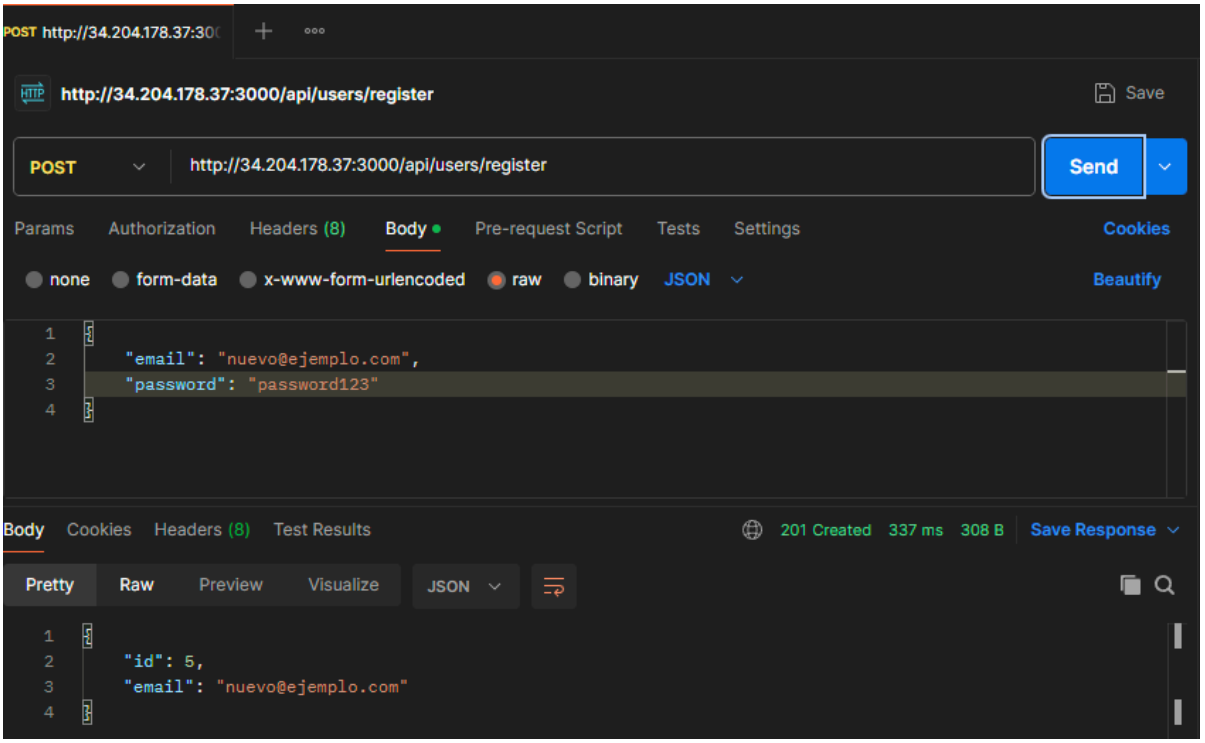
3.2.1 Frontend Container (myapp-frontend)

- **Tecnología:** React 19 + TypeScript + Vite
- **Servidor Web:** Nginx (stable-alpine)
- **Puerto Expuesto:** 80 (contenedor)
- **Función:** Interfaz de usuario para login y gestión de usuarios.
- **Build:** Multi-stage (Node.js para build, Nginx para serving)
- **URL Producción:** `http://34.204.178.37:3000/api/users/`

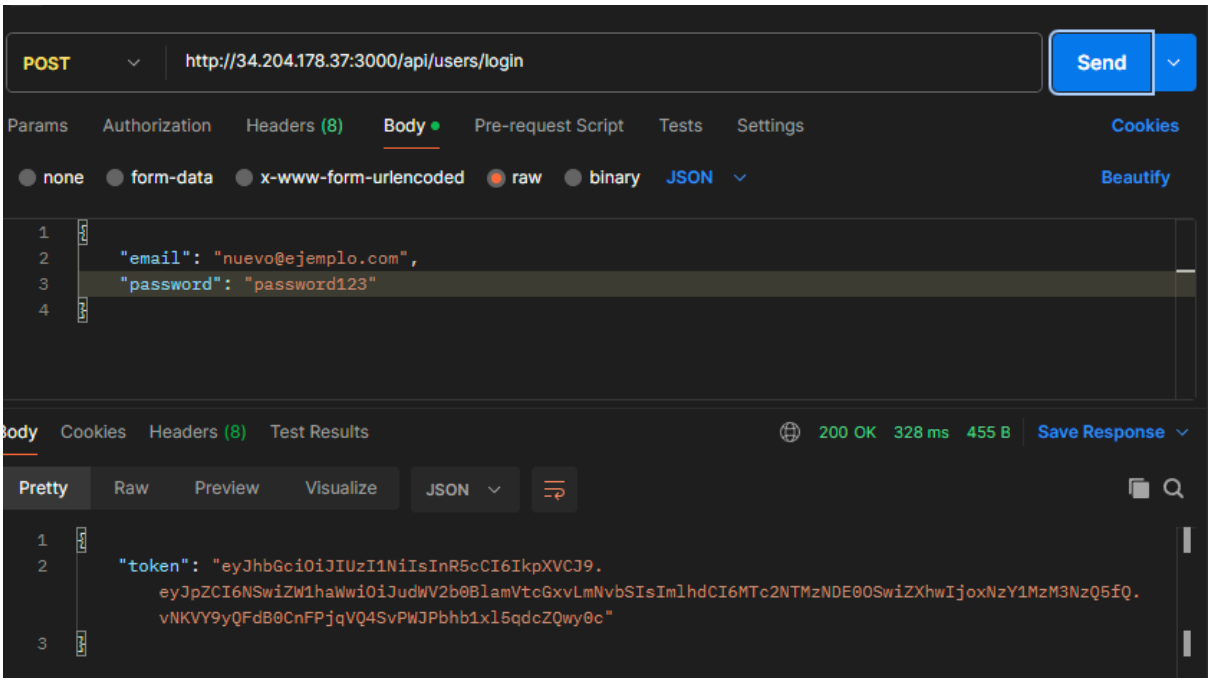
3.2.2 Backend Container (myapp-backend)

- **Tecnología:** Node.js 18 + Express + TypeScript + Prisma
- **Puerto Expuesto:** 3000 (variable desde .env)
- **Función:** API RESTful, autenticación JWT, lógica de negocio

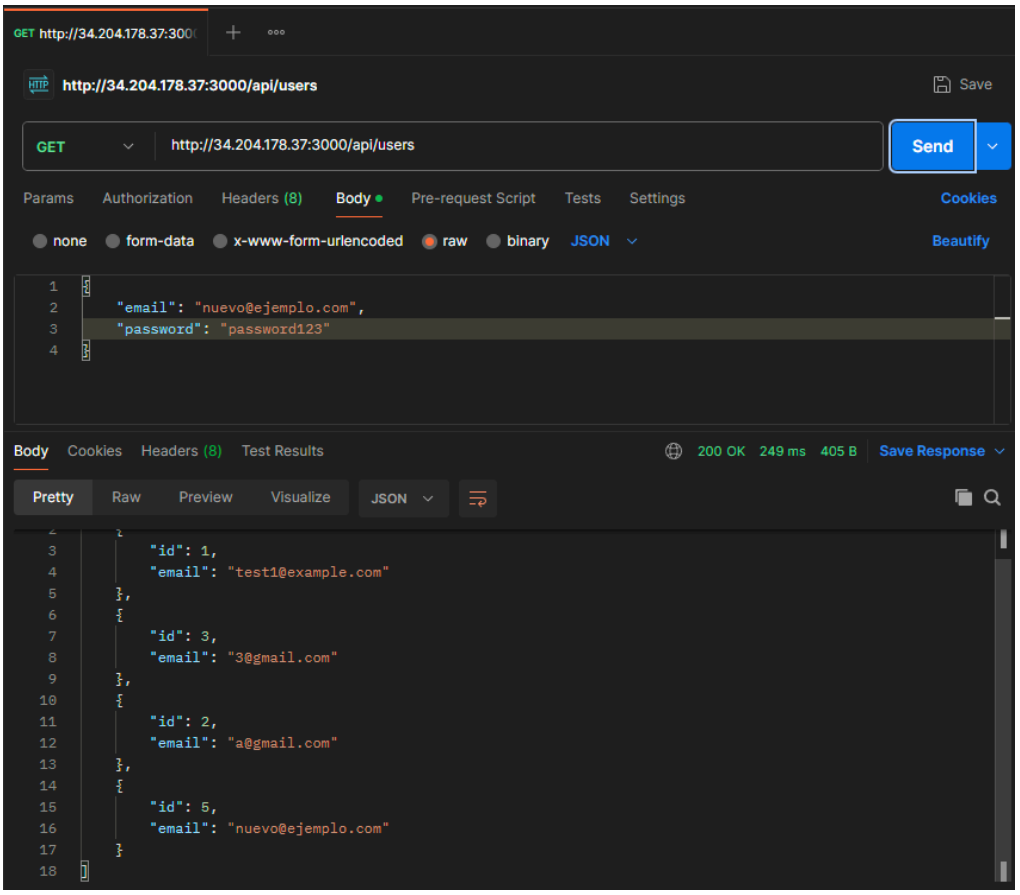
- **Base de Datos:** Conexión a PostgreSQL vía Prisma ORM
- **Endpoints Principales:**
    - POST `/api/users/register` - Registro de usuarios



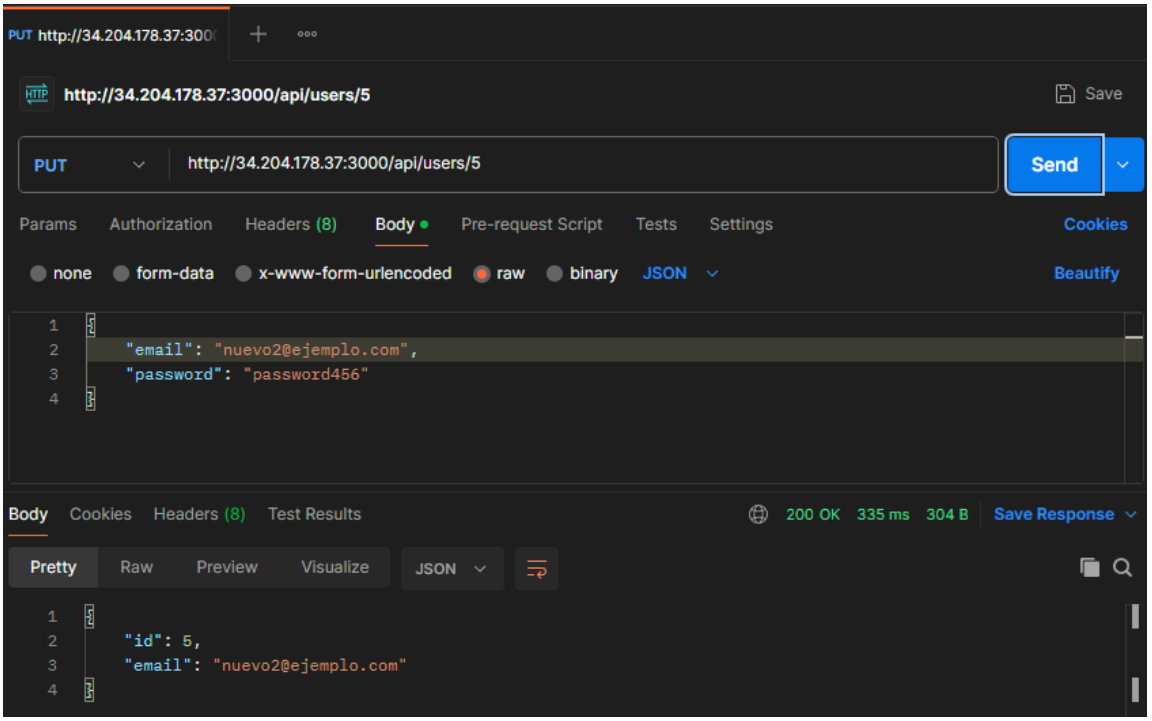
- POST `/api/users/login` - Autenticación



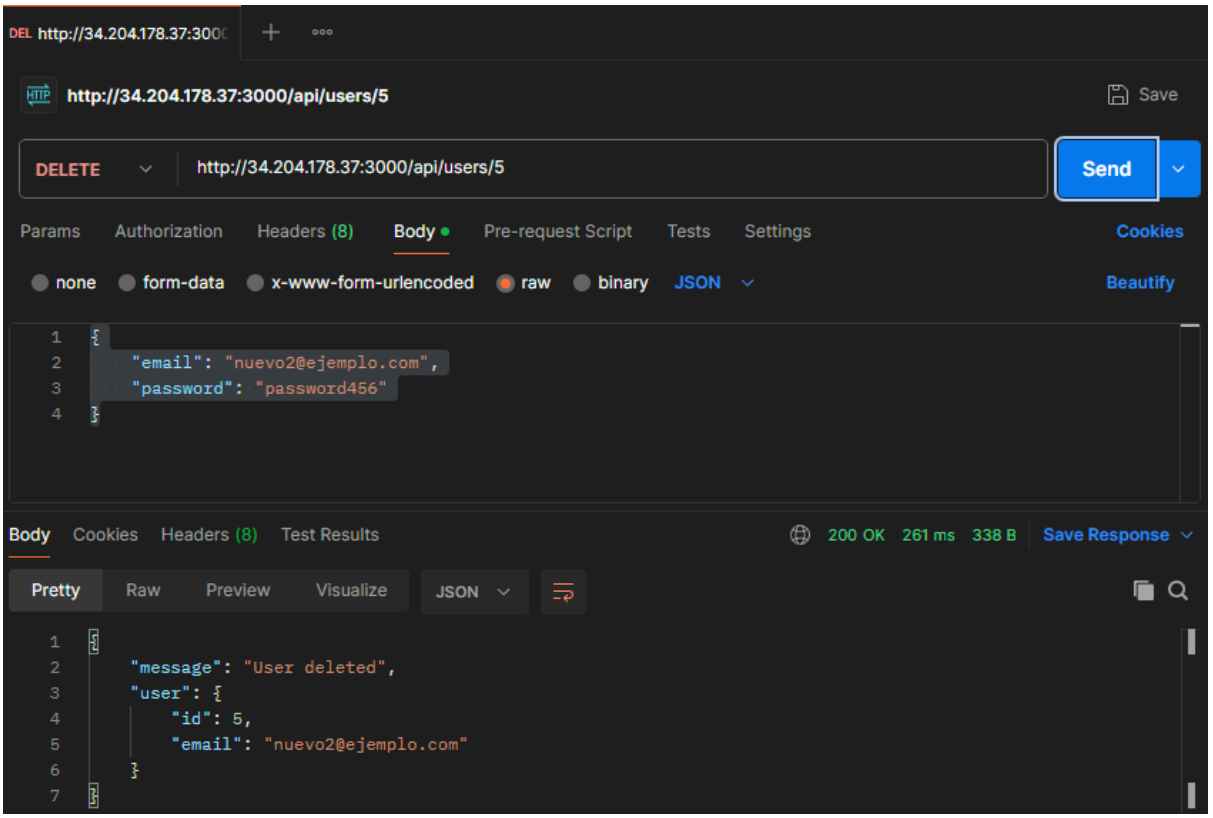
- `GET /api/users` - Lista de usuarios (protegido)



- PUT /api/users/:id - Actualizar datos de usuario



- DELETE /api/users/:id - Actualizar datos de usuario



- **URL Producción:** `http://34.204.178.37:3000/api/users/`

3.2.3 Database Container (myapp-database)

- **Tecnología:** PostgreSQL 16
- **Puerto Expuesto:** 5432
- **Volumen:** `dbdata` montado en `/var/lib/postgresql/data`
- **Persistencia:** Datos sobreviven al reinicio de contenedores
- **Schema:** Tabla `User` (id, email, password, createdAt)
- **IP Interna:** 172.31.76.165

3.2.4 Red y Comunicación

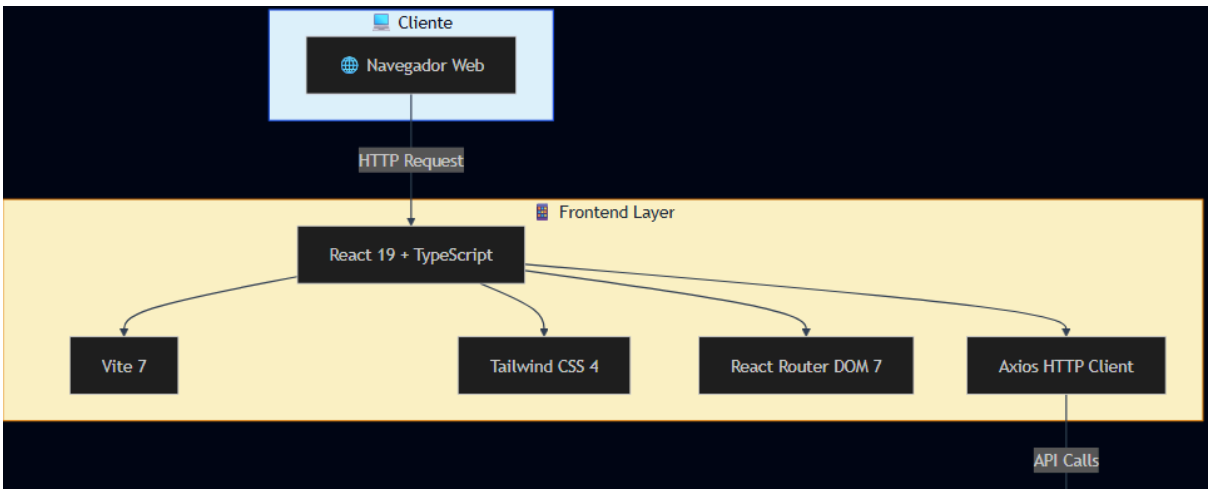
- **Red Docker:** `app-network` (driver bridge)
- **Resolución DNS:** Los servicios se comunican por nombre de servicio
- **Orden de Inicio:**
  1. Database (primero)
  2. Backend (depende de database)
  3. Frontend (depende de backend)

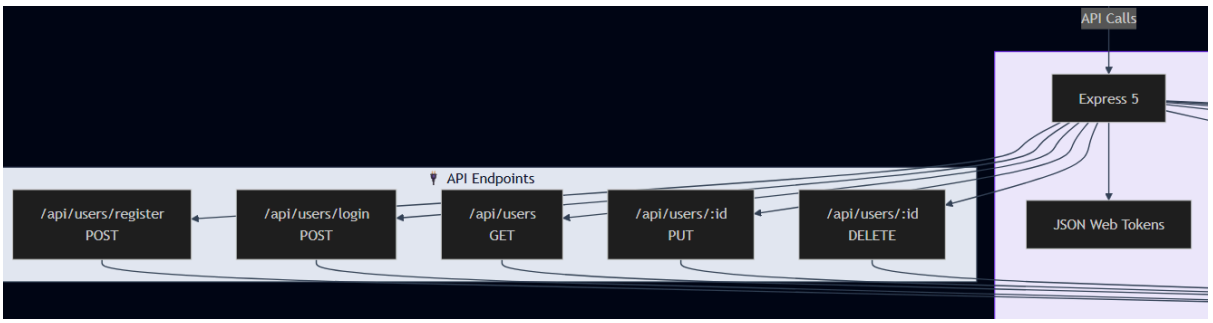
Monorepo con tres servicios dockerizados y comunicados por red bridge interna (app-network).

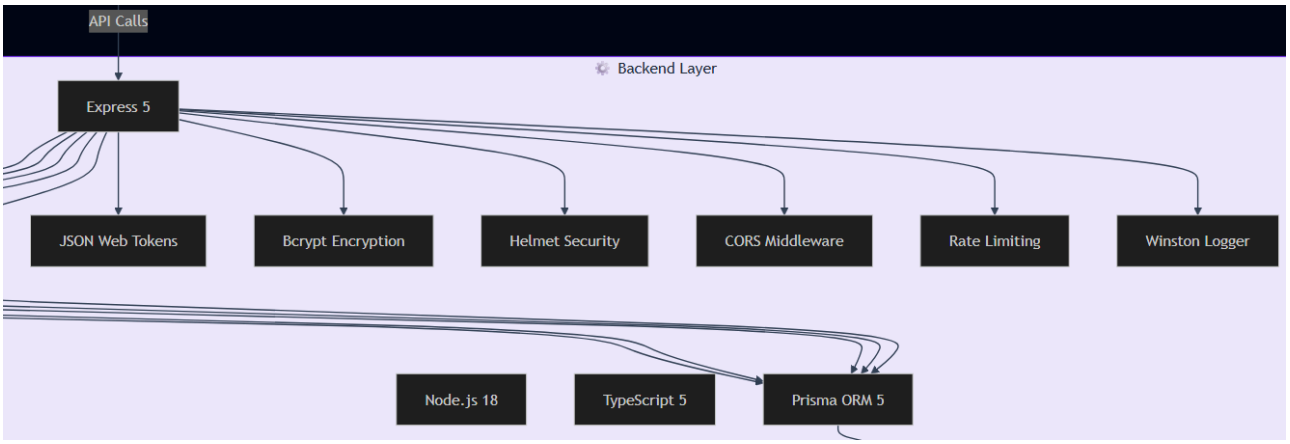
El frontend se sirve estáticamente con Nginx (puerto 80 expuesto al mundo).

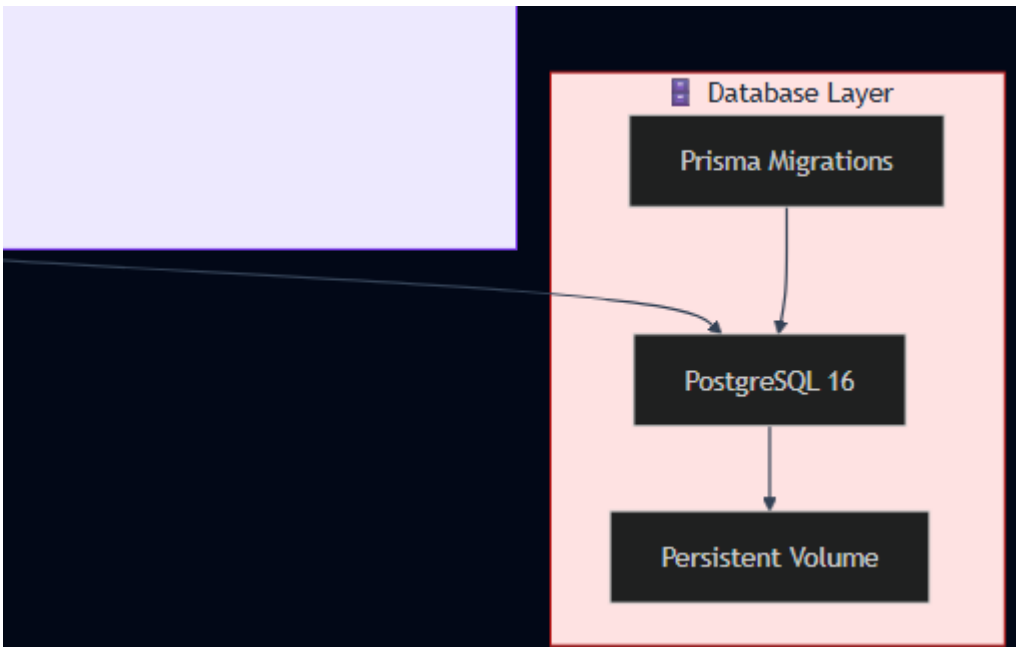
El backend expone la API en el puerto definido por la variable PORT (por defecto 3000).

La base de datos solo es accesible desde la misma red Docker.









4. Dockerfiles (optimizados)

4.1 Backend – Dockerfile (single-stage optimizado para producción)

```
FROM node:18-alpine
RUN apk add --no-cache openssl
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY . .
RUN npx prisma generate
RUN npm run build
EXPOSE ${PORT:-3000}
CMD ["sh", "-c", "npx prisma migrate deploy && node dist/src/server.js"]
```

Este Dockerfile crea una imagen Docker optimizada para producción de una aplicación backend en Node.js (usando TypeScript y Prisma como ORM). Parte de la imagen ligera node:18-alpine, instala openssl (necesario para el engine de Prisma), establece el directorio de trabajo en /app, copia primero solo los archivos package.json para instalar únicamente las dependencias de producción con npm ci --only=production (aprovechando el caché de Docker), luego copia el resto del código, genera el cliente de Prisma, compila la aplicación a JavaScript en la carpeta dist, expone el puerto configurable (por defecto 3000) y, al iniciar el contenedor, aplica las migraciones de la base de datos con prisma migrate deploy antes de lanzar el servidor con node dist/src/server.js. Todo ello en un único stage, resultando en una imagen pequeña, segura y lista para entornos de producción.

4.2 Frontend – Dockerfile (multi-stage)

```
# Build
FROM node:18-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
ARG VITE_REACT_APP_API_URL
ENV VITE_REACT_APP_API_URL=${VITE_REACT_APP_API_URL}
RUN npm run build

# Production
FROM nginx:stable-alpine
COPY --from=build /app/dist /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Este Dockerfile implementa un enfoque **multi-stage** para construir y desplegar una aplicación frontend en React (usando Vite como bundler) de forma optimizada para producción. En la primera etapa ("build"), parte de la imagen `node:18-alpine`, establece el directorio `/app`, copia los `package*.json` e instala todas las dependencias con `npm ci` (incluyendo devDependencies necesarias para el build), luego copia el código fuente completo, permite pasar una variable de entorno `VITE_REACT_APP_API_URL` como argumento de build (para configurar la URL del backend en tiempo de compilación), y ejecuta `npm run build` para generar los archivos estáticos optimizados en la carpeta `/app/dist`. En la segunda etapa ("production"), utiliza una imagen ligera de `nginx:stable-alpine` como servidor web, copia los archivos generados en la etapa anterior a `/usr/share/nginx/html` (directorio por defecto de Nginx para servir contenido estático), expone el puerto 80 y arranca Nginx en modo foreground con `daemon off;`. El resultado es una imagen final extremadamente pequeña y segura, que solo contiene los archivos estáticos servidos por Nginx, ideal para entornos de producción como Kubernetes, Docker Compose o cualquier hosting de contenedores.

4.3 Database – Dockerfile

```
FROM postgres:16-alpine
ENV POSTGRES_USER=${POSTGRES_USER}
ENV POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
ENV POSTGRES_DB=${POSTGRES_DB}
```

Este Dockerfile crea una imagen Docker para una base de datos PostgreSQL optimizada para producción, basada en la imagen oficial ligera `postgres:16-alpine` (versión 16 de PostgreSQL sobre Alpine Linux, lo que la hace pequeña y eficiente).

Define tres variables de entorno esenciales: `POSTGRES_USER` (nombre del usuario superadmin), `POSTGRES_PASSWORD` (contraseña del usuario) y `POSTGRES_DB` (nombre de la base de datos predeterminada que se crea automáticamente al iniciar el contenedor). Estas variables se configuran usando valores pasados desde fuera (generalmente a través de variables de entorno en docker run o docker-compose), lo que permite personalizar la instancia sin reconstruir la imagen. Al iniciar el contenedor, PostgreSQL crea automáticamente el usuario y la base de datos especificados si no existen, y escucha en el puerto 5432 por defecto (expuesto en la imagen base). Es una configuración mínima y segura, ideal para entornos de desarrollo, testing o producción cuando se combina con herramientas como Docker Compose o Kubernetes, evitando credenciales hardcodeadas en la imagen.

5. CI/CD – GitHub Actions (archivos reales del repo)

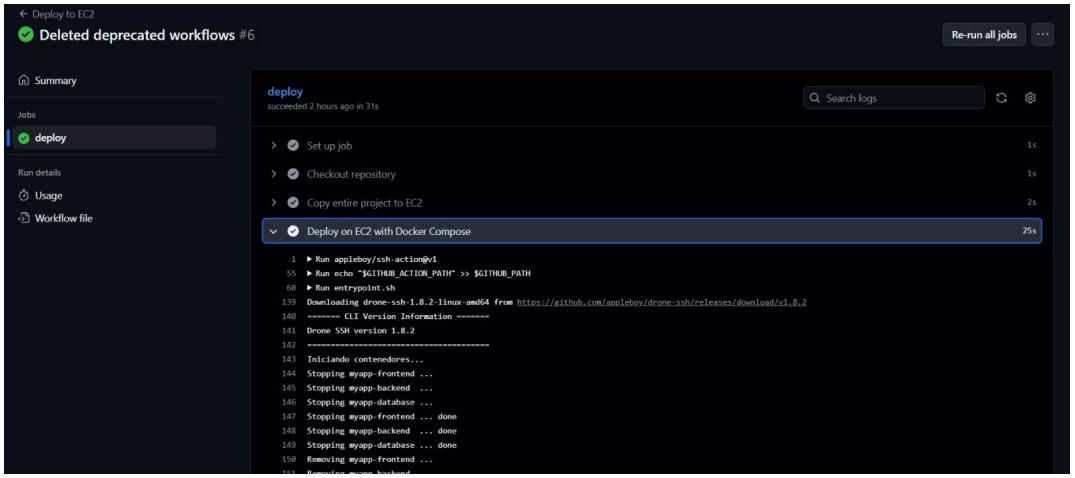
5.1 Workflow global (raíz) – despliegue completo con docker-compose

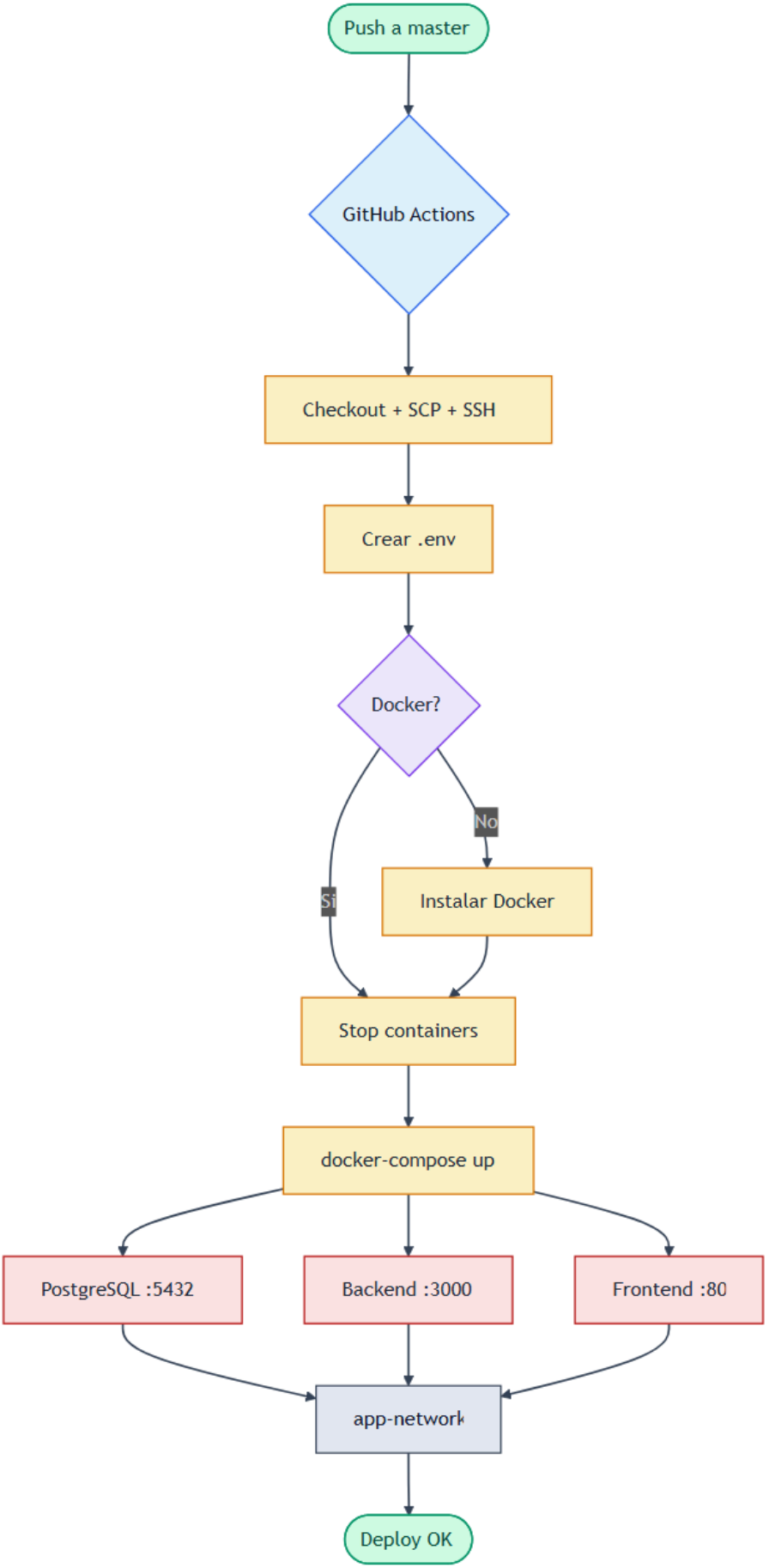
YAML

```yaml
name: Deploy to EC2 (docker-compose)

on:
  push:
    branches: [ master ]
  workflow_dispatch:

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Copy project to EC2
        uses: appleboy/scp-action@v1
        with:
          host: ${{ secrets.EC2_HOST }}
          username: ${{ secrets.EC2_USER }}
          key: ${{ secrets.EC2_KEY }}
          source: "./"
          target: "/home/${{ secrets.EC2_USER }}/app"
          rm: true

      - name: Create .env files & deploy
        uses: appleboy/ssh-action@v1
        with:
          host: ${{ secrets.EC2_HOST }}
```

```
username: ${{ secrets.EC2_USER }}
key: ${{ secrets.EC2_KEY }}
script: |
  cd /home/${{ secrets.EC2_USER }}/app
  cat > .env << EOF
  POSTGRES_USER=${{ secrets.POSTGRES_USER }}
  POSTGRES_PASSWORD=${{ secrets.POSTGRES_PASSWORD }}
  POSTGRES_DB=${{ secrets.POSTGRES_DB }}
  PORT=${{ secrets.PORT }}
  JWT_SECRET=${{ secrets.JWT_SECRET }}
  DATABASE_URL=${{ secrets.DATABASE_URL }}
  VITE_REACT_APP_API_URL=${{ secrets.VITE_REACT_APP_API_URL }}
  EOF
  sudo docker-compose down
  sudo docker-compose up -d --build
```

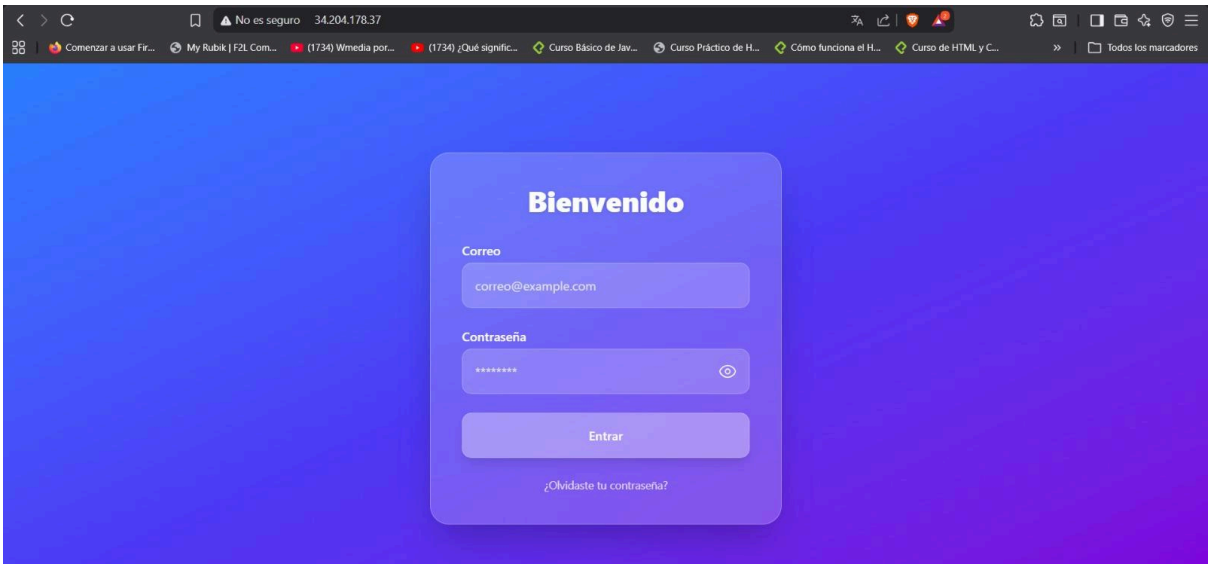
Este archivo YAML define un workflow de GitHub Actions llamado "Deploy to EC2 (docker-compose)" que se ejecuta automáticamente al hacer push a la rama “master” o manualmente mediante workflow_dispatch. El workflow contiene un único job llamado “deploy” que corre en un runner `ubuntu-latest`. Primero, realiza el checkout del código con “actions/checkout@v4”. Luego, copia todo el proyecto al servidor EC2 usando “appleboy/scp-action”, transfiriendo los archivos a “/home/<usuario>/app” (sobrescribiendo si existe gracias a “rm: true”), autenticandose mediante host, usuario y clave SSH almacenados en secrets. Finalmente, ejecuta comandos remotos en el EC2 con “appleboy/ssh-action”: navega al directorio de la app, crea (o sobrescribe) un archivo “.env” con variables de entorno sensibles (credenciales de PostgreSQL, puerto, JWT secret, URL de base de datos y URL de API para el frontend), todas tomadas de secrets de GitHub, y luego detiene los contenedores existentes con “docker-compose down” y los reinicia en modo detached con reconstrucción de imágenes usando `sudo docker-compose up -d --build`.

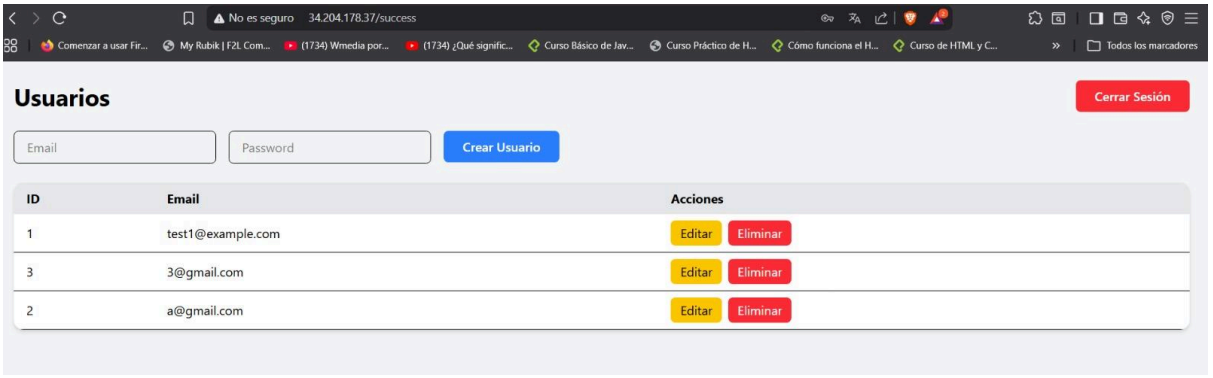



Push a master
GitHub Actions
Checkout + SCP + SSH
Crear .env
Docker?
No
Si
Instalar Docker
Stop containers
docker-compose up
PostgreSQL :5432
Backend :3000
Frontend :80
app-network
Deploy OK

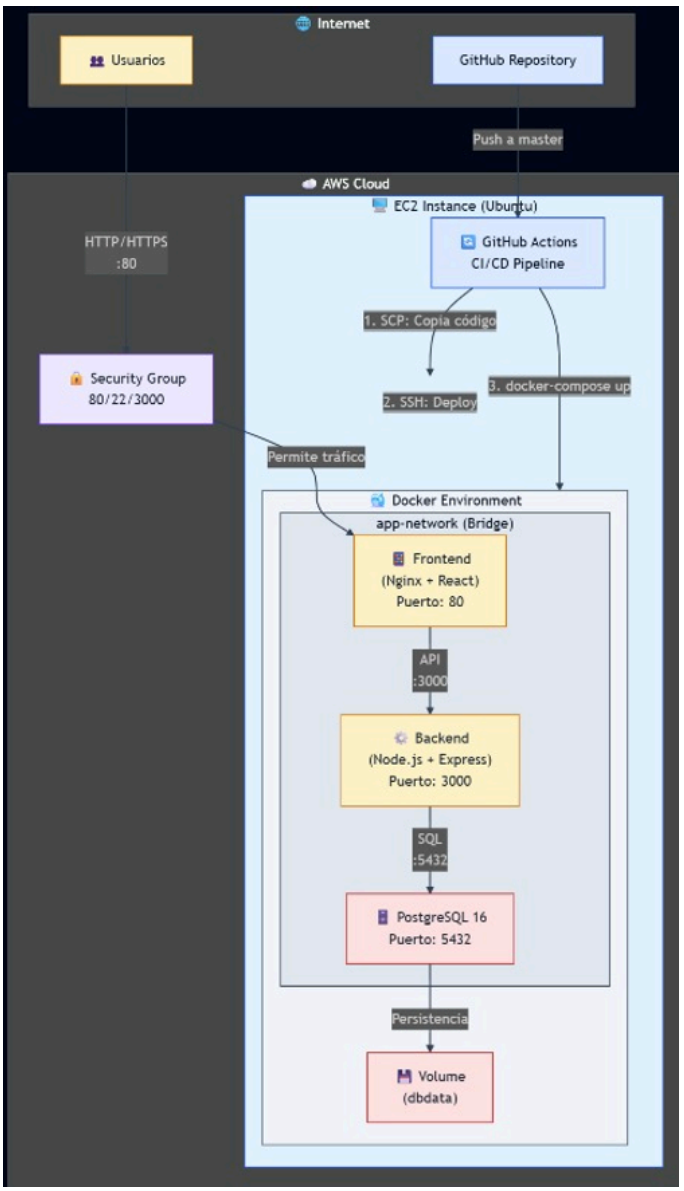
6. Despliegue en EC2 – Pasos realizados

1. Instancia EC2 Amazon Linux 2023 / Ubuntu 22.04 (t3.small)
2. Security Group:
    - 22 (SSH) → IP propia
    - 80 (HTTP) → 0.0.0.0/0 (frontend)
    - 3000 → solo desde VPC (opcional)





3. Instalación única de Docker + Docker Compose en la instancia
4. Clonado del repositorio y primer docker-compose up -d --build manual
5. Desde ese momento todo se despliega automáticamente con el workflow global



7. Seguridad y secretos

7.1 Dónde se almacenan los secretos y donde se inyectan

Secreto	Dónde está guardado	Cómo se inyecta
EC2_HOST, EC2_USER, EC2_KEY	GitHub Secrets	appleboy actions
POSTGRES_USER / PASSWORD	GitHub Secrets	.env creado en runtime en EC2
DATABASE_URL	GitHub Secrets	.env raíz y backend/.env
JWT_SECRET	GitHub Secrets	.env raíz y backend/.env
VITE_REACT_APP_API_URL	GitHub Secrets	.env raíz y frontend/.env

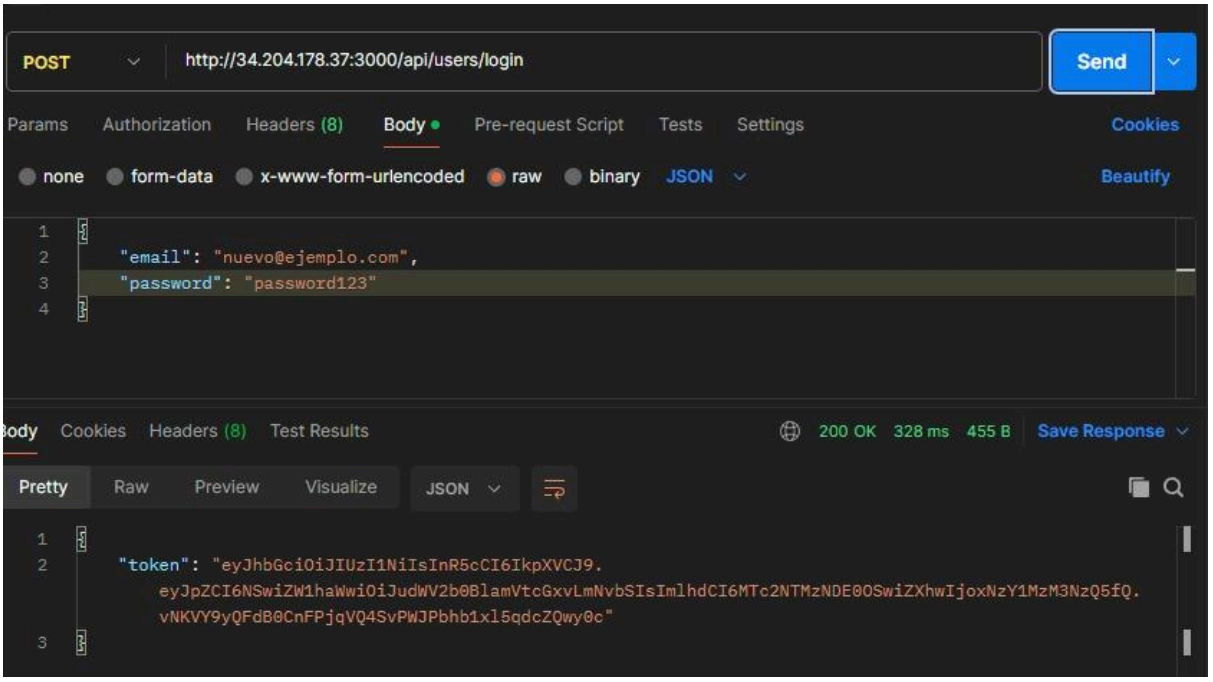
7.2 Recomendaciones futuras

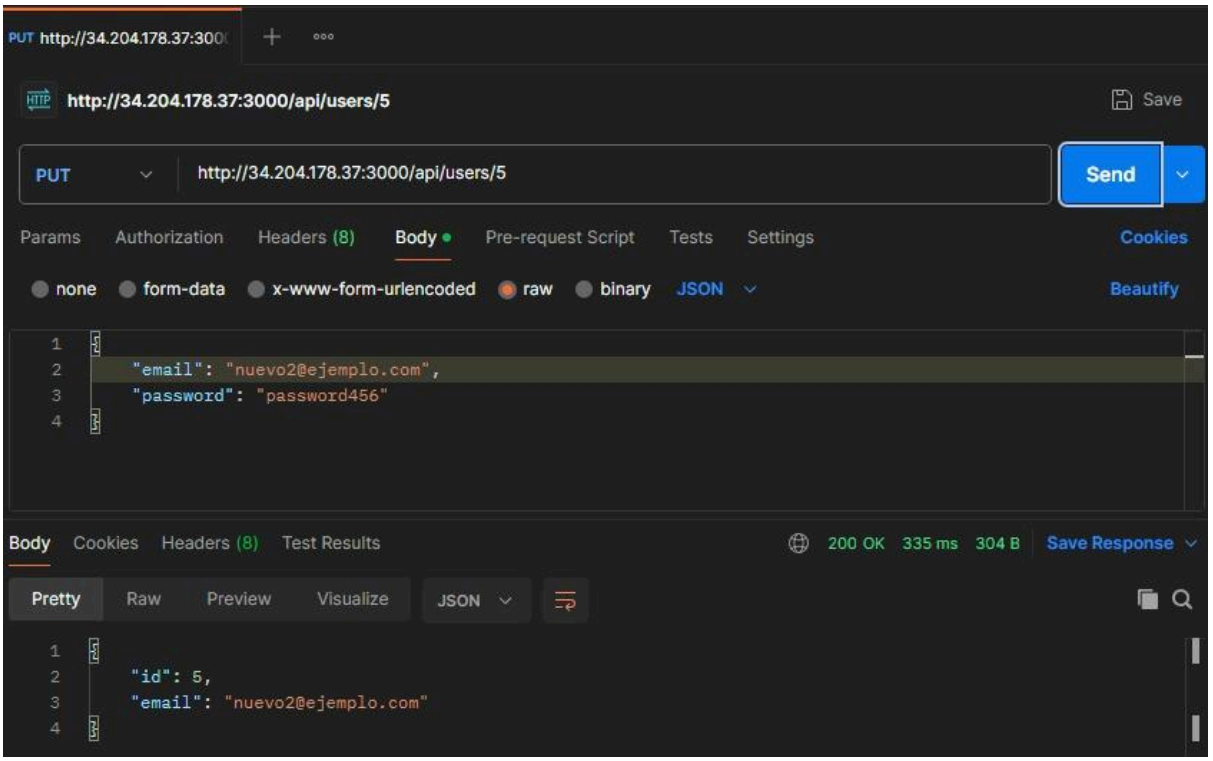
- Implementar HTTPS / TLS en EC2 usando Nginx + Let's Encrypt.

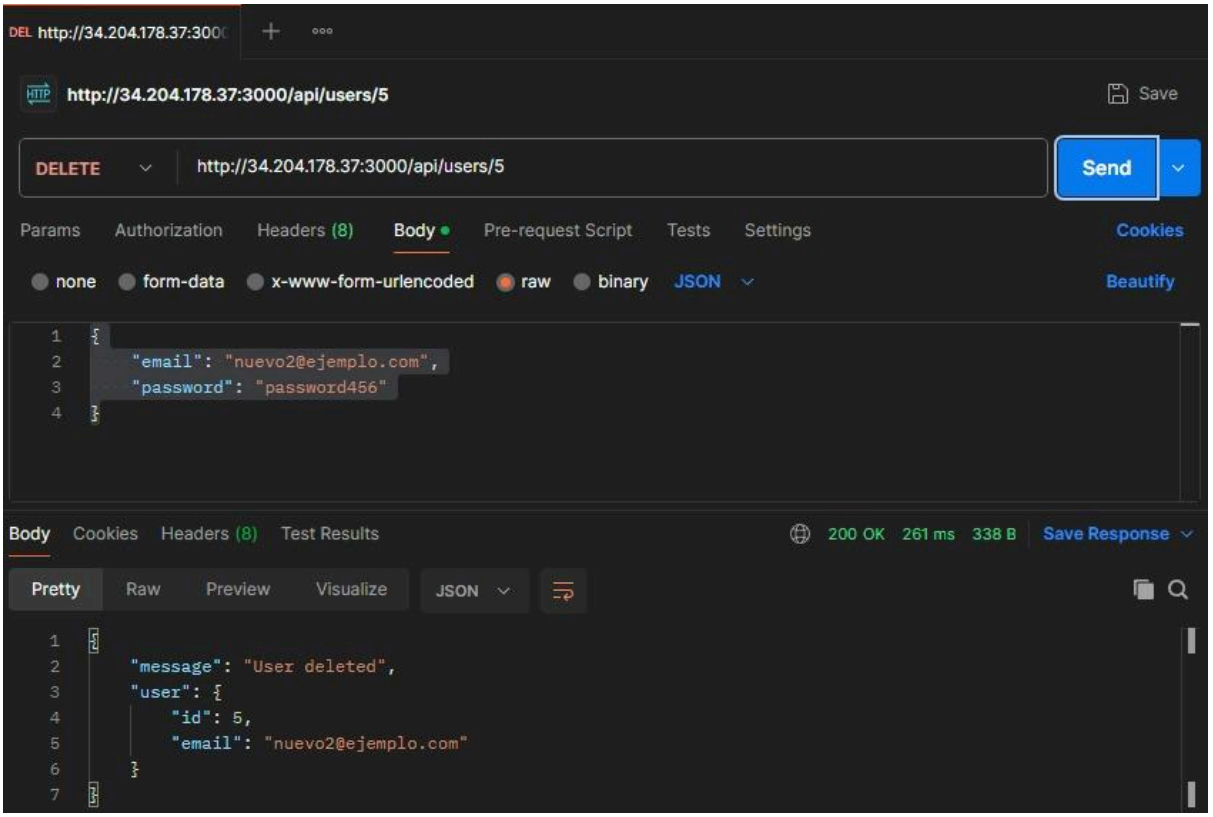
- Rotación periódica de claves SSH.
- Migrar secretos a AWS SSM Parameter Store o AWS Secrets Manager.
- Añadir un reverse proxy seguro (Nginx) con rate limiting contra ataques.
- Implementar autenticación 2FA en GitHub.

8. Pruebas realizadas

- Registro y login manual → OK







- Token JWT guardado en localStorage → OK
- Acceso a /success protegido → OK
- Backup manual ejecutando backup_db.sh → OK (archivo .gz creado)
- Restore manual → OK (datos recuperados)
- Caída del contenedor backend → se reinicia automáticamente (restart: always)

9. Operación y mantenimiento

Backups:backend/backup_db.sh (se puede ejecutar con cron)
Bash
PGPASSWORD="..." pg_dump -U adminuser -h database myappdb > backup.sql
gzip backup.sql
temp.sh: line 1: pg_dump: command not found

- Conserva solo los últimos 7 backups.
- Logs: se escriben en backend/logs/ (puedes añadir winston más adelante)

Restauración:
Bash
gunzip backup.sql.gz

- psql -U adminuser -h database myappdb < backup.sql

- Reinicio automático: política restart: always en docker-compose.yml

10. Conclusiones

- Se logró un sistema full-stack completamente contenedorizado y desplegado automáticamente en AWS EC2 mediante GitHub Actions.
- La arquitectura es limpia, escalable y sigue buenas prácticas (multistage Dockerfiles, secretos fuera del código, backups).
- El uso de Docker Compose simplificó enormemente el despliegues y actualizaciones.
- Pendientes para mejora futura: HTTPS, tests automatizados y uso de ECR + ECS Fargate.