**UNIVERSIDAD PRIVADA BOLIVIANA**
**LA PAZ**
**FACULTAD DE INGENIERÍA Y ARQUITECTURA**

---


# Proyecto Segundo Parcial

# Despliegue con Docker + EC2 + CI/CD

---

**Estudiantes:**

Ivan Iver Poma Maidana - 70419
Wendy Evelyn Cáceres Vasquez - 68874
Andres Raul Sanchez Andrade - 69017

Docente: Rayner Mendieta Villalba

La Paz – Bolivia – 10 de Diciembre de 2025

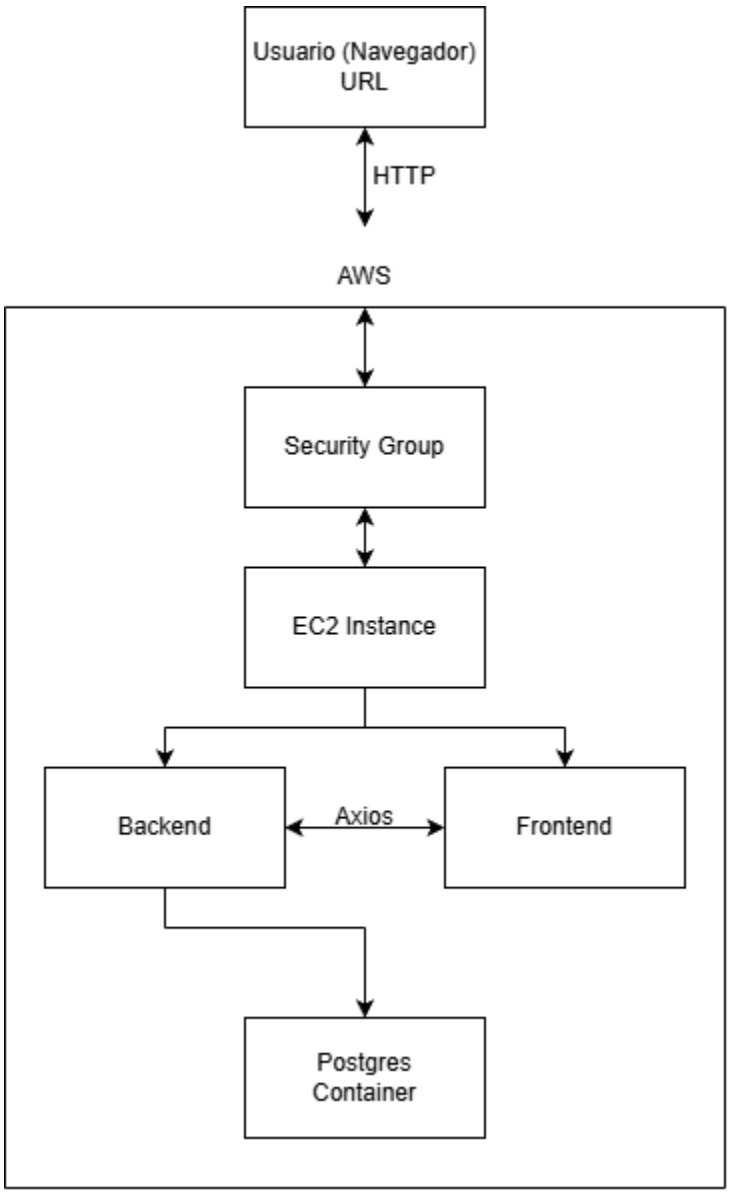
# Índice

# 1. Requisitos y alcance

| Caracteristica | Estado |
|---|---|
| Frontend SPA con CRUD completo | Implementado |
| Backend API REST con lógica de negocio y endpoints para la SPA | Implementado |
| Base de datos relacional con persistencia | Implementado |
| Cada componente en contenedor Docker independiente | Implementado |
| Dockerfiles reproducibles y buenas prácticas | Implementado |
| docker-compose.yml | No implementado |
| Despliegue final de los contenedores en una instancia EC2 | Implementado |
| Pipeline CI/CD en GitHub Action | Implementado |
| Gestión de secretos mediante GitHub Secrets | Implementado |
| Security Groups configurados correctamente | Implementado |

| | |
|---|---|
| README con instrucciones de ejecución local y URL pública de la aplicación | Implementado |
| Monitoreo | Implementado |
| Auto-restart | Implementado |
| Rollback automatizado | No implementado |
| TLS/HTTPS | No implementado |
| Escalado simple | No implementado |

## 2. Arquitectura




El flujo que seguimos es primero, cuando un usuario cualquiera usuario accede desde cualquier navegador a la URL “http://3.14.88.250/app/mapa” .

Al pasar eso el tráfico entra a AWS a través del único Security Group previamente configurado con los puertos autorizados. Ya desde dentro de la Instance EC2 se ejecutan tres contenedores existentes con docker-compose, que serían los de Frontend, Backend y Postgres.

El Frontend consume el Backend mediante peticiones Axios. Mientras que el Backend se conecta al contenedor de PostgreSQL usando la URL interna que se encuentra en el .env.

Ventajas de usar esta arquitectura:

- Todo en una sola instancia EC2 lo cual genera un costo mínimo.
- Base de datos protegida, se tiene protegido el puerto para que no esté expuesto.
- Despliegue automatizado con GitHub Actions + SSH.
- Reinicio automático de contenedores (restart: unless-stopped).

## 3. Dockerfiles

Backend – Dockerfile

```
1   FROM node:18-alpine
2
3   WORKDIR /app
4
5   COPY package*.json ./
6
7   RUN npm ci --omit=dev
8
9   COPY . .
10
11  EXPOSE 3000
12
13  CMD ["npm", "start"]
```

Se logró implementar algunas optimizaciones.

- Primeramente se tiene una imagen base alpine.
- En la línea 7 se tiene “--omit=dev” eso hace que no se instale devDependencies como ser Jest, TypeScript, etc.
- Se tiene capas separadas de dependencias el Docker cachea npm ci cuando solo cambian archivos de código.
- Lo que genera una imagen que es ligera y segura.

Frontend – Dockerfile

```
# Build stage
FROM node:20-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .

ARG VITE_API_URL
ENV VITE_API_URL=$VITE_API_URL

RUN npm run build

# Production stage - Usando Apache (httpd)
FROM httpd:alpine

# Copiar los archivos construidos al directorio de Apache
COPY --from=build /app/dist /usr/local/apache2/htdocs/

# Habilitar mod_rewrite para que funcionen las rutas de React
RUN sed -i '/LoadModule rewrite_module/s/^#//g' /usr/local/apache2/conf/httpd.conf

# Configurar FallbackResource para SPA (La forma más simple en Apache moderno)
# Esto le dice a Apache: "Si no encuentras el archivo, sirve index.html"
RUN echo "FallbackResource /index.html" >> /usr/local/apache2/conf/httpd.conf

EXPOSE 80
```

Se logró implementar algunas optimizaciones:

- Multistage build, es decir que la imagen final NO contiene Node.js ni node_modules.
- Usamos variables de entorno VITE_API_URL inyectada en build time, lo que significa que el frontend sabe dónde está el backend sin hardcodear.
- FallbackResource /index.html se tiene un soporte perfecto para rutas de React Router (SPA)
- Máximo aprovechamiento de caché Docker: si solo cambian componentes lo que se hará sera solo se rebuilda la capa final no todo.

# 4. CI/CD

## Backend CI/CD – .github/workflows/ci-cd.yml

```yaml
name: Backend CI/CD

on:
  push:
    branches: [ main ]
  workflow_dispatch:

jobs:
  deploy:
    name: Build & Deploy
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Login to Docker Hub
        uses: docker/login-action@v3
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Build and Push Docker Image
        uses: docker/build-push-action@v5
        with:
          context: .
          push: true
          tags: ${{ secrets.DOCKER_USERNAME }}/exchange-backend:latest

      - name: Deploy to EC2
        uses: appleboy/ssh-action@master
        with:
          host: ${{ secrets.EC2_HOST }}
          username: ${{ secrets.EC2_USER }}
          key: ${{ secrets.EC2_SSH_KEY }}
          script: |
            docker pull ${{ secrets.DOCKER_USERNAME }}/exchange-backend:latest

            docker stop exchange-backend || true
            docker rm exchange-backend || true

            # Asegurar red
            docker network create exchange-network || true

            # Crear archivo .env con el contenido del secreto BACKEND_ENV
            echo "${{ secrets.BACKEND_ENV }}" > .env

            docker run -d \
              --name exchange-backend \
              --network exchange-network \
              --restart always \
              -p 3000:3000 \
              --env-file .env \
              ${{ secrets.DOCKER_USERNAME }}/exchange-backend:latest

            # Limpiar archivo .env por seguridad
            rm .env

                    # --- VERIFICACIÓN DE DESPLIEGUE ---
            echo "⏳ Esperando 10 segundos para verificar arranque..."
            sleep 10
            echo "📜 Logs de arranque:"
            docker logs exchange-backend
            if [ "$(docker inspect -f '{{.State.Running}}' exchange-backend)" = "true" ]; then
              echo "✅ El backend está corriendo correctamente."
            else
              echo "❌ El contenedor se detuvo inesperadamente."
              exit 1
            fi
```

En nuestro archivo ci-cd.yml del backend tenemos lo siguiente:

- Construye y sube la imagen a Docker Hub (tag latest)
- SSH a EC2 lo que hace un pull de la nueva imagen
- Crea archivo .env temporal con secretos (DATABASE_URL, JWT_SECRET, etc.) usando el secreto BACKEND_ENV
- Ejecuta el contenedor con --restart always y en la red exchange-network
- Borra el .env por seguridad
- Verifica que el contenedor esté corriendo

**Frontend CI/CD – .github/workflows/ci-cd.yml**

```yaml
name: Frontend CI/CD

on:
  push:
    branches: [ main ]
  workflow_dispatch:

jobs:
  deploy:
    name: Build & Deploy
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Setup Node & Test
        uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'
      - run: |
          npm ci
          npm run lint

      - name: Login to Docker Hub
        uses: docker/login-action@v3
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Build and Push Docker Image
        uses: docker/build-push-action@v5
        with:
          context: .
          push: true
          tags: ${{ secrets.DOCKER_USERNAME }}/exchange-frontend:latest
          # Pasamos la variable de entorno como argumento de construcción (build-arg)
          build-args: |
            VITE_API_URL=${{ secrets.VITE_API_URL }}

      - name: Deploy to EC2
        uses: appleboy/ssh-action@master
        with:
          host: ${{ secrets.EC2_HOST }}
          username: ${{ secrets.EC2_USER }}
          key: ${{ secrets.EC2_SSH_KEY }}
          script: |
            # Descargar la nueva imagen
            docker pull ${{ secrets.DOCKER_USERNAME }}/exchange-frontend:latest

            # --- LIMPIEZA DE PUERTO 80 ---
            # Apache se reinicia solo si lo matas con kill. Hay que detener el servicio.
            sudo systemctl stop apache2 || true
            sudo systemctl disable apache2 || true
            # Por si acaso Nginx también está instalado como servicio
            sudo systemctl stop nginx || true
            sudo systemctl disable nginx || true

            # Detener y eliminar el contenedor anterior (si existe)
            docker stop exchange-frontend || true
            docker rm exchange-frontend || true

            # Correr el nuevo contenedor
            docker run -d \
              --name exchange-frontend \
              --restart always \
              -p 80:80 \
              ${{ secrets.DOCKER_USERNAME }}/exchange-frontend:latest
```

En nuestro archivo ci-cd.yml del frontend tenemos lo siguiente:

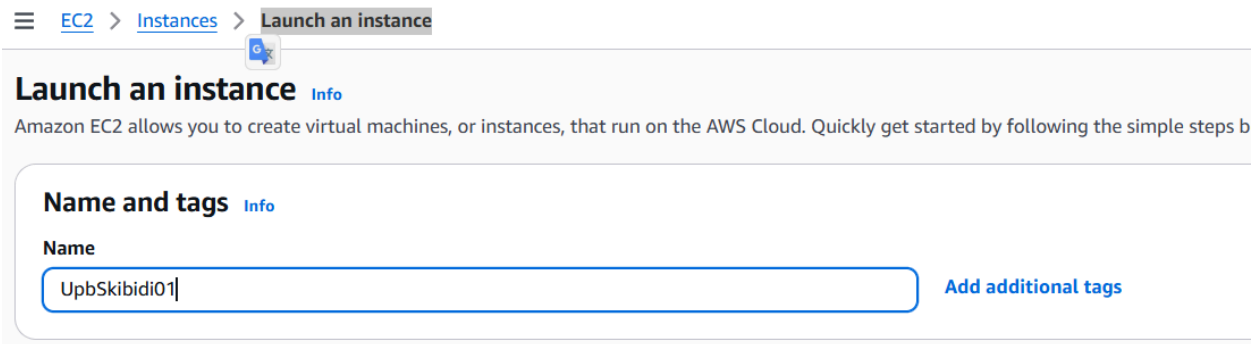
- Ejecuta npm ci y lint
- Inyecta VITE_API_URL como build-arg ya que el frontend sabe dónde está el backend sin hardcodear
- Construye y sube imagen a Docker Hub
- En EC2: detiene servicios web que puedan estar ocupando el puerto 80
- Despliega el nuevo contenedor en el puerto 80 con --restart always

**Secrets utilizados:**

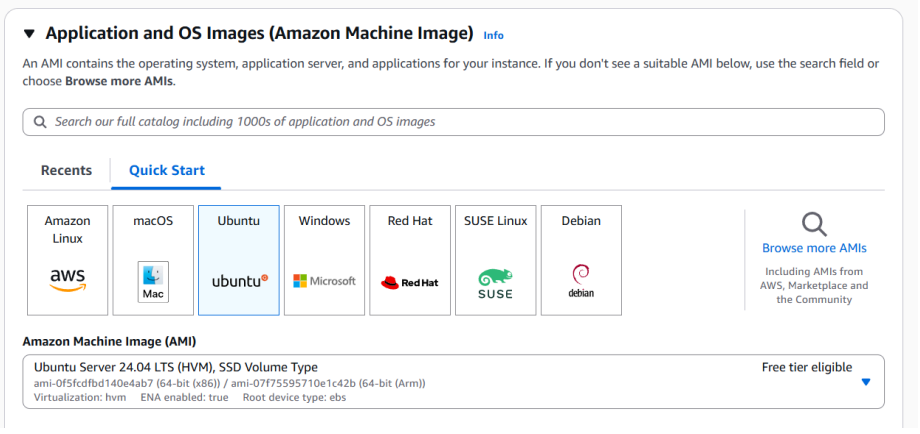
- DOCKER_USERNAME, DOCKER_PASSWORD
- EC2_HOST, EC2_USER, EC2_SSH_KEY
- VITE_API_URL es la URL pública del backend
- BACKEND_ENV tiene el contenido completo del .env del backend

# 5. Despliegue en EC2

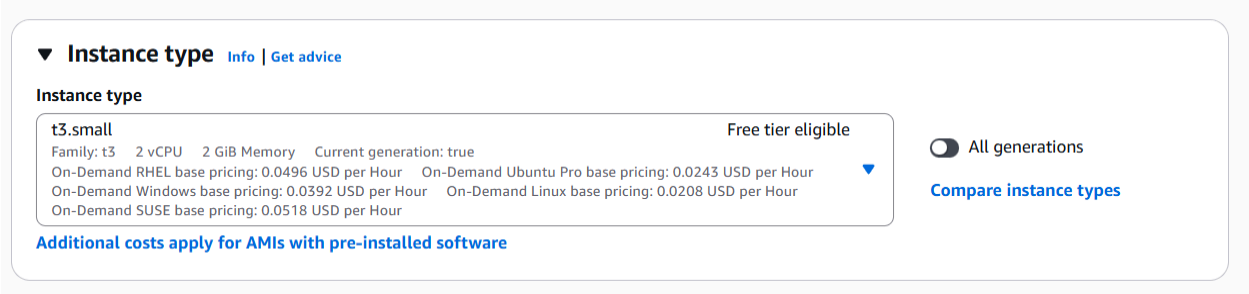
La creación de nuestro EC2 fue primero entrar a nuestra cuenta de AWS, luego nos vamos al apartado de EC2 y ponemos a Launch an instance y al entra ahi debemos colocar un nombre para nuestro EC2, en nuestro caso le pusimos “UpbSkibidi01” pero podemos poner el nombre que mejor nos convenga.



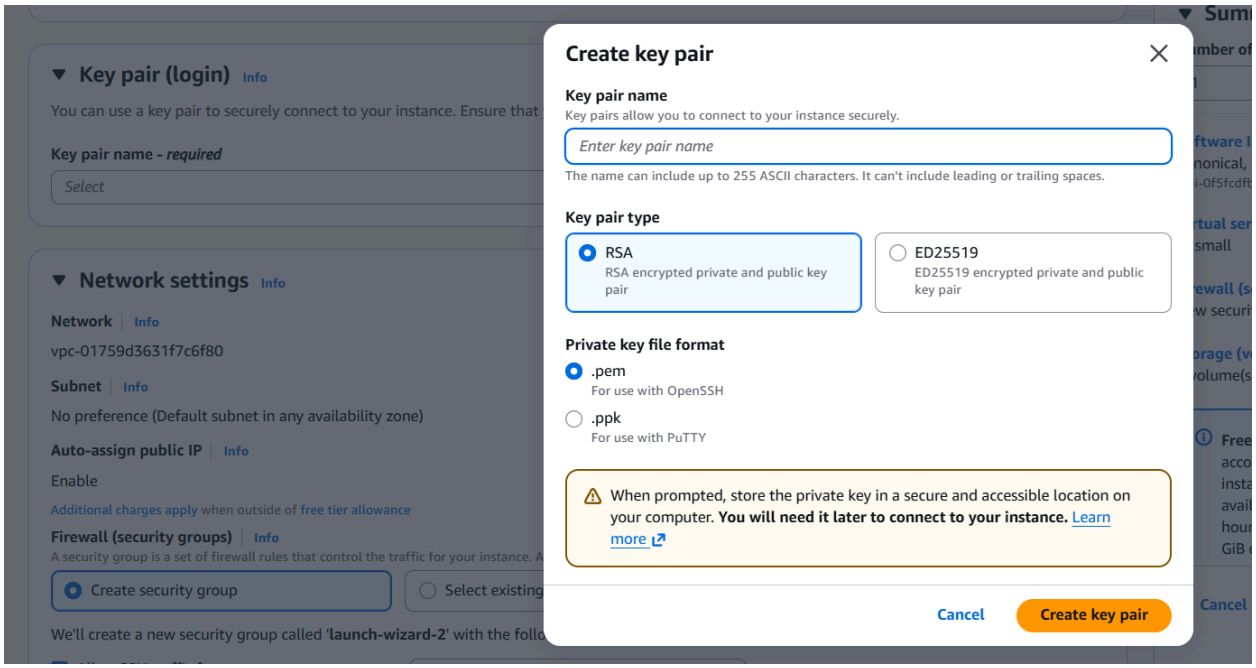

Después seleccionamos la imagen que vamos a usar y en nuestro caso seleccionamos Ubuntu y verificamos que sea la versión Free tier



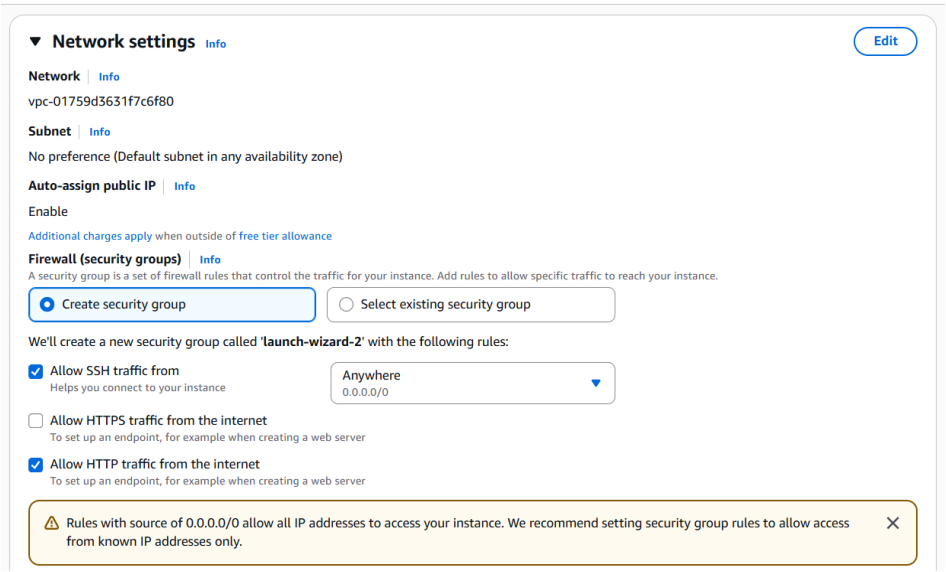
Luego en la instancia le podemos t3.small y vemos que sea Free tier



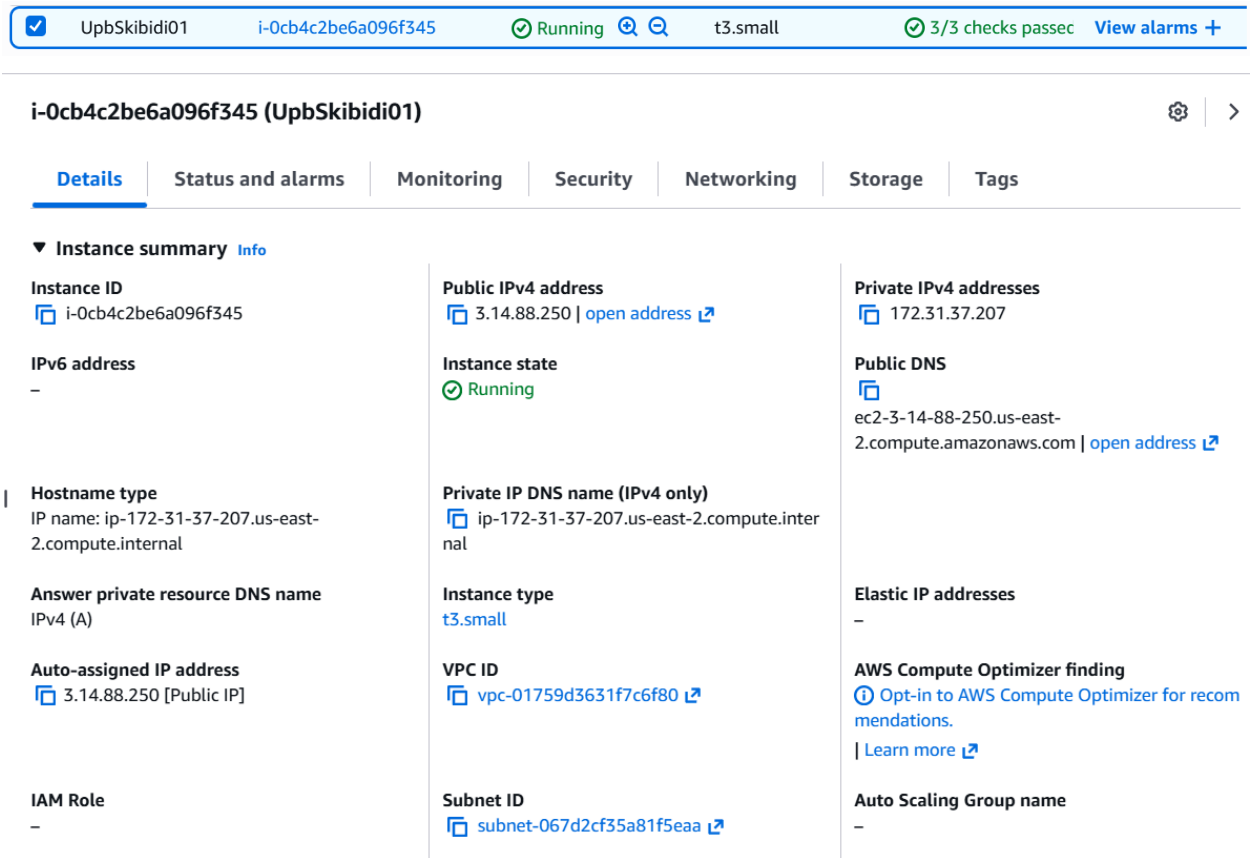

En la Key pair podemos usar alguna ya creada o podemos crear una nueva, pero necesitamos seleccionar que sea .pem para poder usarla luego.




En network settings seleccionamos la opción para acceder con HTTP que es la última



Y ya por ultimo le ponemos Launch Instance para así poder ver nuestra instancia y ver que este corriendo con las características que dijimos.




Podemos agregar a un Security group y poner ahí los puertos que queremos que escuche

| Name | Security group rule ID | Port range | Protocol | Source |
|---|---|---|---|---|
| – | sgr-06b7d58427f2938f5 | 5173 | TCP | 0.0.0.0/0 |
| – | sgr-07571d73d8ad10044 | 80 | TCP | 0.0.0.0/0 |
| – | sgr-0a93b4f20e4e2d102 | 3000 | TCP | 0.0.0.0/0 |
| – | sgr-0371ff1d5c4a61f6f | 22 | TCP | 0.0.0.0/0 |
| – | sgr-0906d34365de4017a | 443 | TCP | 0.0.0.0/0 |
| – | sgr-0037aeaacea290796 | 4000 | TCP | 0.0.0.0/0 |

**Outbound rules**

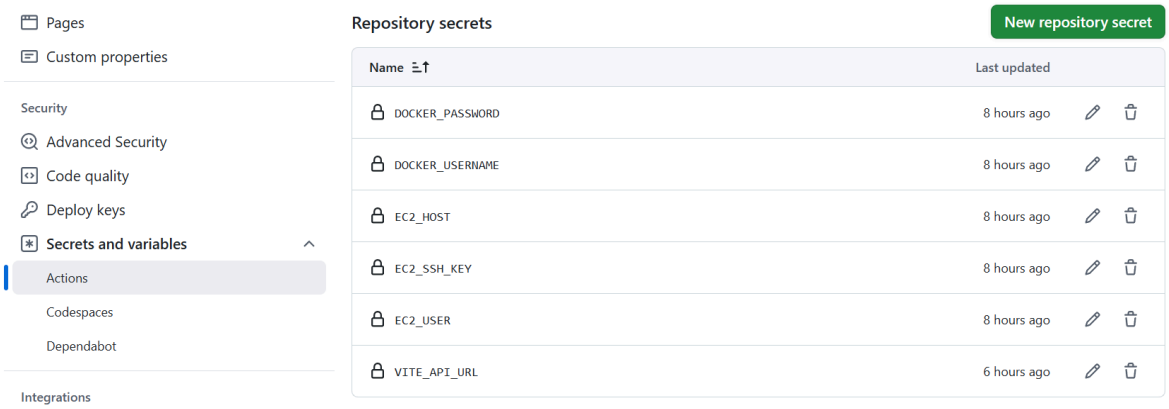
Filter rules

1

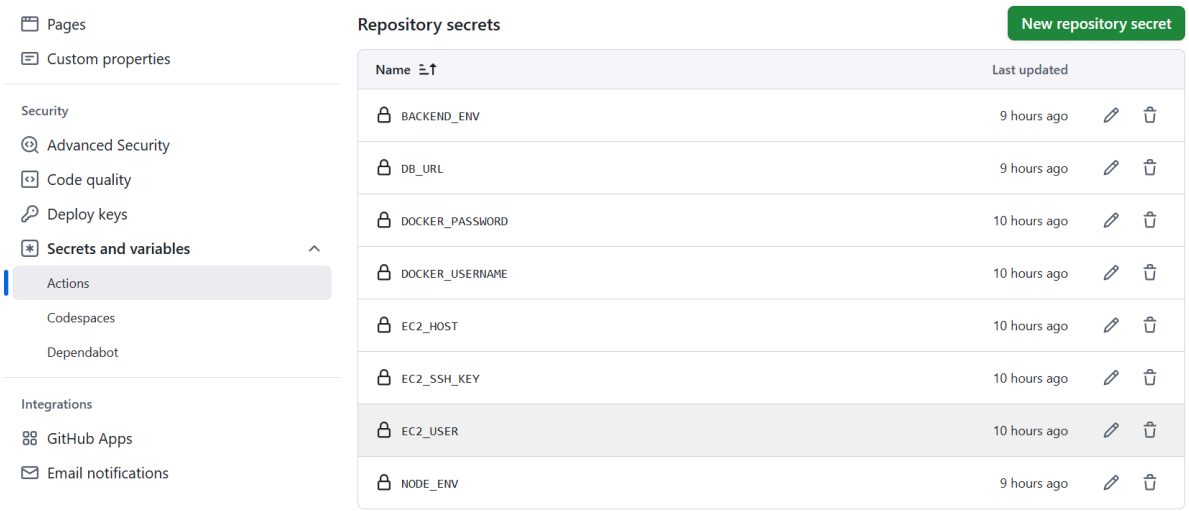
| Name | Security group rule ID | Port range | Protocol | Destination |
|---|---|---|---|---|
| – | sgr-02a0de5ed337be370 | All | All | 0.0.0.0/0 |

# 6. Seguridad y secretos

## Frontend




## Backend



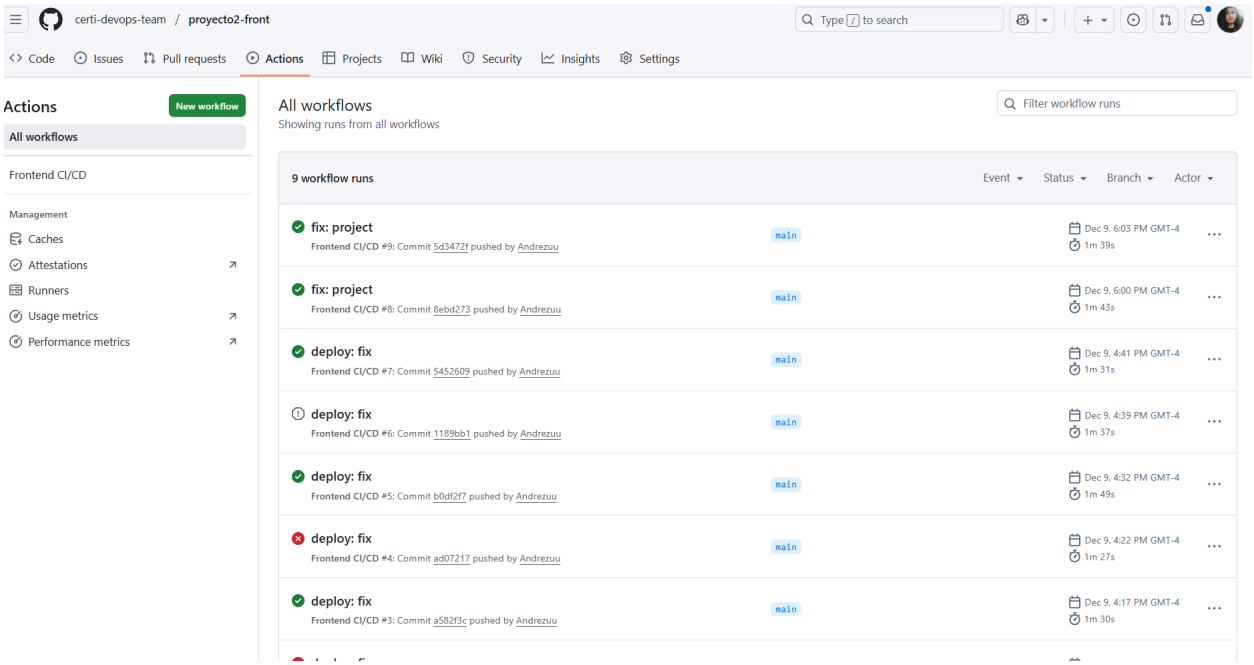

En ambos casos tanto del Frontend como del Backend se crearon secrets para que no esté visible en el código y se tenga mas seguridad, la mayoría de estos se puede encontrar en los archivos Docker en ambos repositorios. En el backend la última secret de `NODE_ENV` se encuentra en el archivo \src\database\connection.js. En el frontend la última secret de `VITE_API_URL` se encuentra en el archivo Dockerfile.

Se recomienda migrar a GitHub Container Registry o Amazon ECR en lugar de Docker Hub para mayor privacidad, implementar rotación automática de claves SSH cada 90 días y adoptar AWS Secrets Manager o Parameter Store para credenciales sensibles como DATABASE_URL y JWT_SECRET. También se sugiere inyectar variables de entorno directamente en el comando docker run en lugar de usar un archivo .env, añadir escaneos de vulnerabilidades con Trivy o Docker Scout en el pipeline, configurar un usuario non-root en los contenedores, activar Docker Content Trust para firmar imágenes, y usar tags con el hash de commit para facilitar rollbacks y garantizar la integridad de las imágenes, elevando así el nivel de seguridad a estándares empresariales.

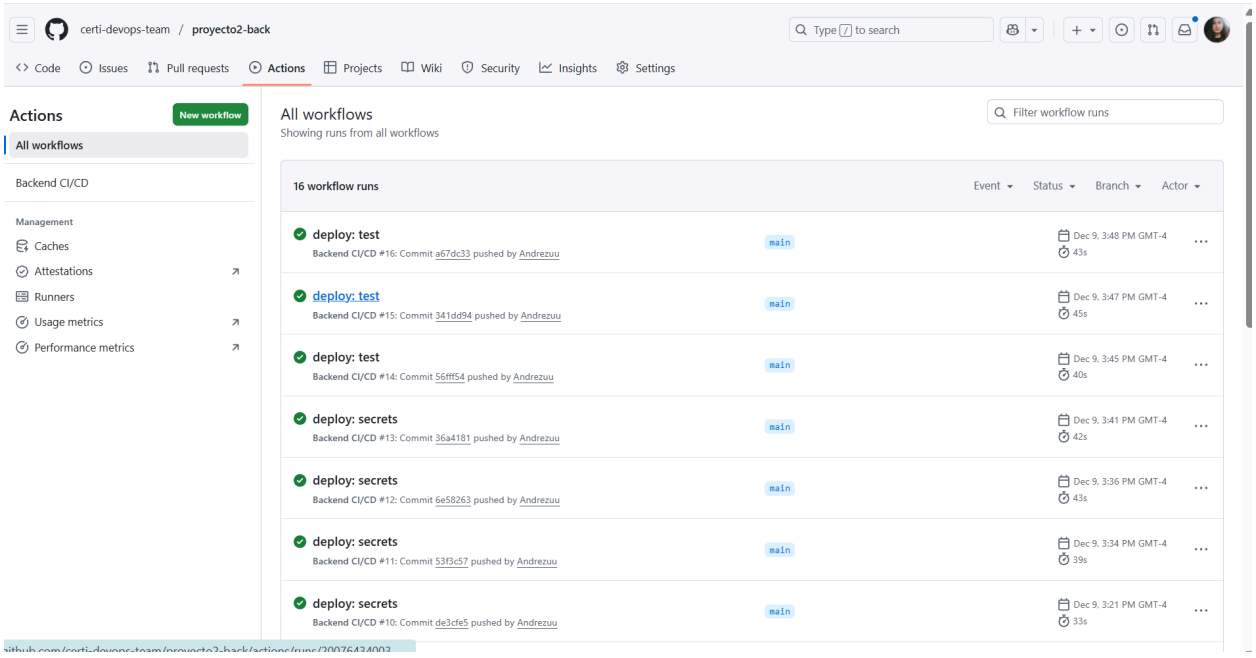
# 7. Pruebas

Pruebas git hub actions

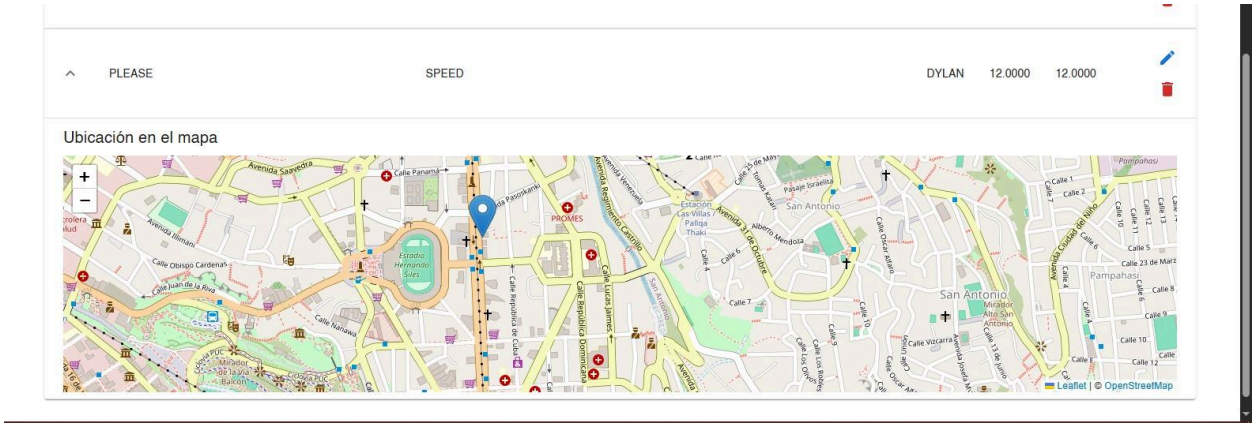
Frontend:

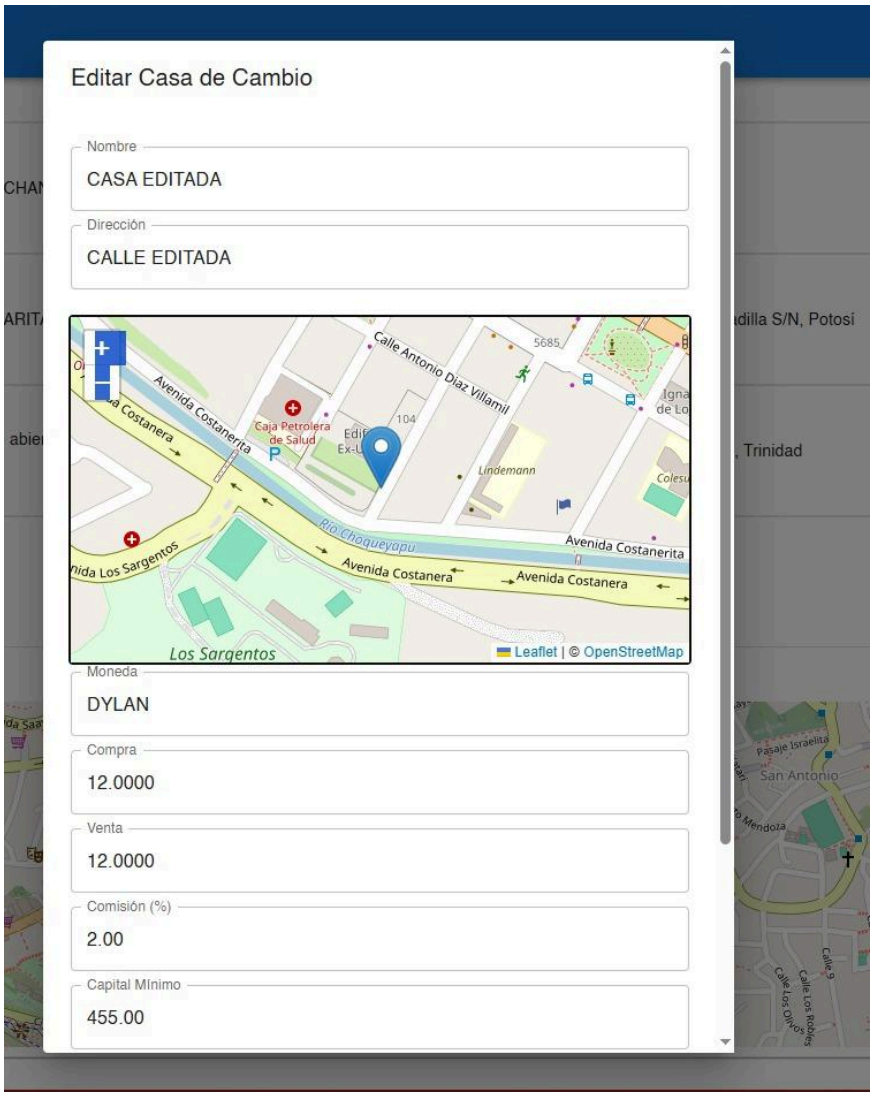


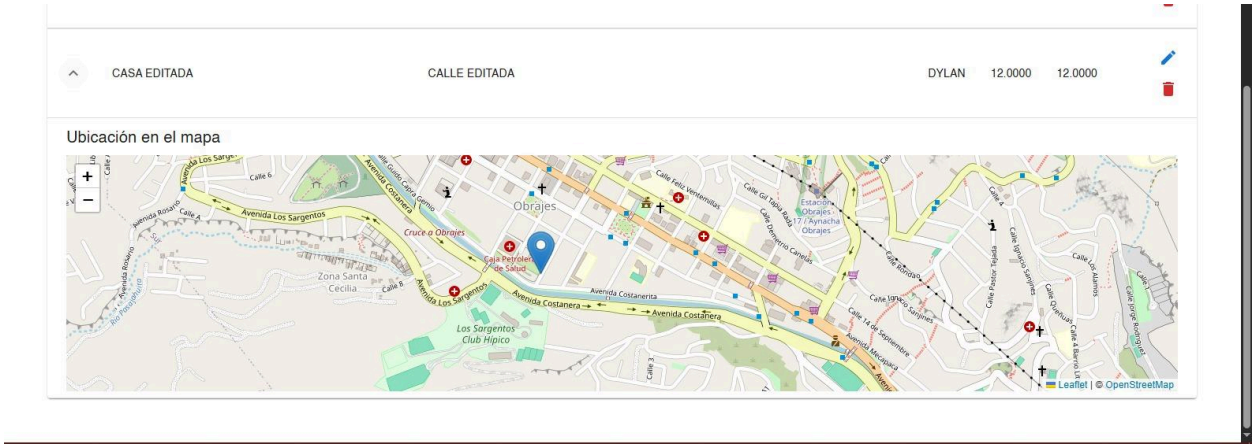

Backend:



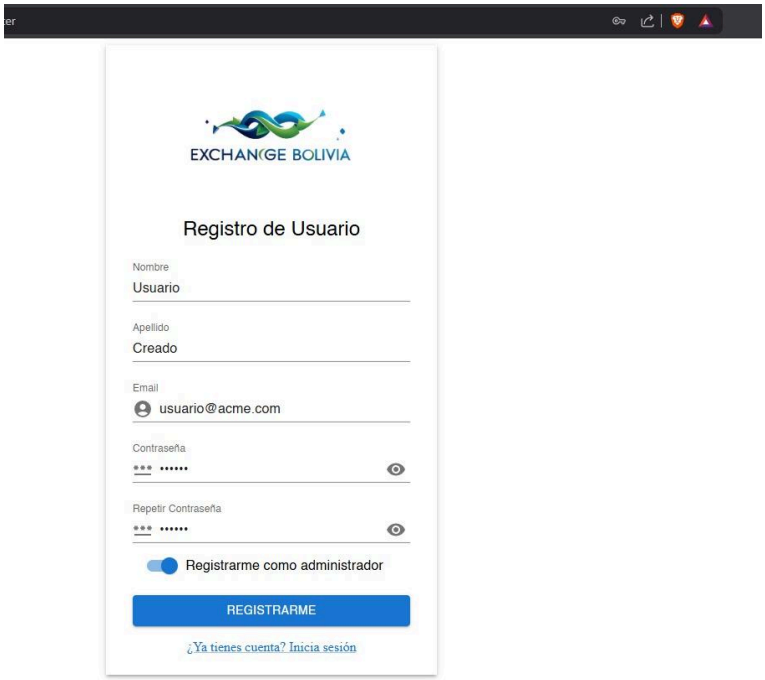
Editar una casa de cambio:

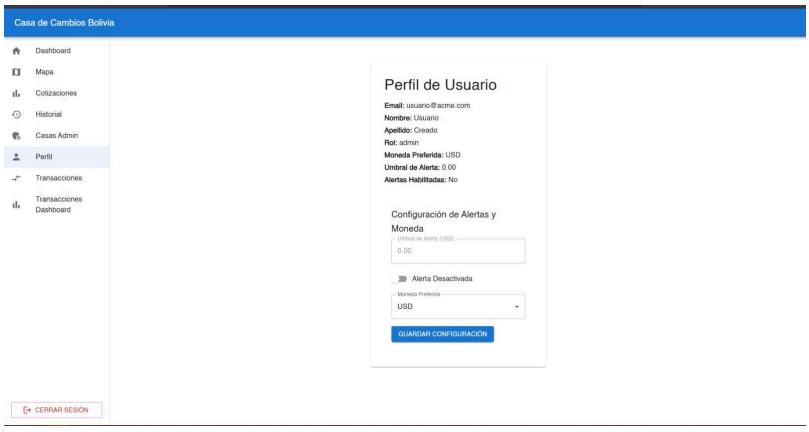





## Creación de un usuario:



## 8. Operación y mantenimiento

**Reinicio automático:** Todos los contenedores (frontend, backend y base de datos) están configurados con la política --restart always (frontend y backend) y unless-stopped (PostgreSQL). En caso de caída del contenedor o reinicio de la instancia EC2, Docker los levanta automáticamente sin intervención manual.

**Logs:** Los logs están disponibles en tiempo real mediante los comandos:

```
docker logs -f exchange-frontend
```

```
docker logs -f exchange-backend
```

```
docker logs -f exchange-postgres
```

## 9. Conclusiones

El proyecto cumplió con la mayoría de los objetivos del segundo parcial: se desarrolló una aplicación full-stack completa para la gestión de casas de cambio en Bolivia, con CRUD completo, autenticación segura y funcionalidades avanzadas (mapa interactivo, gráficos y alertas). Se implementó contenerización profesional con Dockerfiles multistage y buenas prácticas, despliegue en una instancia EC2 de AWS, y un pipeline CI/CD totalmente automatizado mediante GitHub Actions que construye, publica y despliega imágenes Docker en producción con cada push a main. Aprendimos a integrar de forma real todo el ciclo de vida de una aplicación moderna (desarrollo → build → test → deploy → monitoreo), a manejar secretos de forma segura, a optimizar imágenes Docker para producción y a automatizar despliegues sin downtime. Aunque nos quedamos por hacer mejoras como ser TLS/HTTPS, rollback automatizado y escalado horizontal, consideramos que la solución actual es robusta, segura, de bajo costo y completamente operativa en producción, demostrando un dominio sólido de las herramientas DevOps exigidas en el curso.