Actividad 4: Frontend, Dockerización y Despliegue en la Nube

Alumno: Antonio López

Para la realización de las 4 actividades he creado una aplicación web que permite gestionar los activos de una empresa y sus riesgos asociados. La aplicación está dividida en 4 módulos:

- Frontend: Interfaz de usuario interactiva.
- Backend: Servidor API REST.
- ETL Worker: Servicio en segundo plano para la ingesta automatizada de datos.
- Base de Datos: Sistema gestor de base de datos relacional.
  La he alojado dentro de mi web construida con docker compose. En la web aprovecho para mostrar y aprender sobre desarrollo de software gis devops etc..
  https://tonilogar.com/

La herramienta de la asignatura es:
https://dev-web.tonilogar.com/

Tarea 1: Frontend con Mapa y Filtro Reactivo

El frontend se ha construido utilizando **Svelte** con VITE para manejar la reactividad de los elementos del frontend. El objetivo principal ha sido consumir los datos de la API (Backend) e interactuar con ellos.

1.1 Filtrado Reactivo (Client-Side)

He implementado un sistema de **filtrado en cliente**.
Al arrancar la aplicación (`Dashboard.svelte`), se obtiene el listado completo de activos con una única llamada a la API:

```
let assets: any[] = [];
// Variable reactiva para almacenar el hazard seleccionado
let selectedHazardId: number = 0;

// Filtra dinámicamente sin mutar el array original ni recargar
$: filteredAssets = selectedHazardId === 0
  ? assets
  : assets.filter(a => a.AssetHazardExposure?.some((exp: any) => exp.hazard_id === selectedHazardId || exp.Hazard?.id === selectedHazardId));
```

1.2 El Mapa (MapLibre)

Para la visualización de los activos he utilizado la librería **MapLibre**. El componente del mapa recibe el array `filteredAssets`. Cuando Svelte detecta que el usuario cambia el selector de peligro, la variable `filteredAssets` se actualiza instantáneamente y muestra solo los que cumplen el filtro.

1.3 Variables de Entorno

En este proyecto se usan ficheros `.env` para configurar variables tanto en desarrollo como en producción.

- `.env.development`
- `.env.production`

Cuando se levanta la herramienta con `docker-compose` en local, se utilizan las variables de `.env.development`. Cuando se hace un *push*, se utilizan las variables de `.env.production` alojadas en GitHub Secrets para el despliegue automático.

Ejemplo:

`.env.development` (Entorno Local)

```
# 3. DOCKER COMPOSE ROUTING (Parametrización Traefik Local)
DOMAIN_NAME=localhost
TRAEFIK_HTTP_PORT=8001
SENTINEL_URL=http://sentinel.localhost:8001/
CASSINI_URL=http://cassini.localhost:8001/
DEV_WEB_URL=http://dev-web.localhost:8001/
DEV_WEB_API_URL=http://api.dev-web.localhost:8001

ETL_API_KEY=xxxxxxxxxxxxxxx
```

`.env.production` (Entorno Cloud / VPS)

```
# Entorno de Producción (Hetzner VPS) - CI/CD
# Estas variables serán leídas nativamente por /docker-compose.yml al orquestar Traefik.

DOMAIN_NAME=tonilogar.com
TRAEFIK_HTTP_PORT=80
SENTINEL_URL=https://sentinel.tonilogar.com/
```

```
CASSINI_URL=https://cassini.tonilogar.com/
DEV_WEB_URL=https://dev-web.tonilogar.com/
DEV_WEB_API_URL=https://api.dev-web.tonilogar.com

ETL_API_KEY=xxxxxxxxxxxxxxxxx
```

Tarea 2: Dockerización de DataSphere

Para preparar la aplicación para producción, el frontend no se puede servir simplemente con el servidor de desarrollo (`npm run dev`). Es necesario construir una imagen estática y servirla de forma eficiente.

Para ello, se utiliza un **Dockerfile Multi-Stage** (Multietapa):

```
# Etapa 1: Build (Node.js)
FROM node:18-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
# Se inyecta la URL de la API mediante argumentos de build
ARG VITE_API_URL
ENV VITE_API_URL=$VITE_API_URL
# Se compila el código (HTML, CSS, JS minificado en la carpeta /dist)
RUN npm run build

# Etapa 2: Producción con Nginx
FROM nginx:alpine
# Copiar configuración personalizada para SPA (Single Page Applications)
COPY nginx.conf /etc/nginx/conf.d/default.conf
# Copiar los archivos estáticos generados en la etapa anterior
COPY --from=builder /app/dist /usr/share/nginx/html

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

Ventajas de este enfoque:

1. **Seguridad y Peso:** La imagen final solo contiene Nginx y archivos estáticos. No hay rastro de Node.js, `npm`, ni del código fuente original. Esto hace que la imagen pese unos pocos megabytes.
2. **Eficiencia:** Nginx está optimizado para servir archivos estáticos (`.html`, `.js`, `.css`).

Tarea 3: Despliegue en la Nube y Arquitectura Global

3.1 Proceso de Despliegue en la Nube (Práctica Real con CI/CD)

En lugar de un despliegue manual teórico, el proyecto **DataSphere** se despliega de forma totalmente automatizada en un VPS de **Hetzner** mediante flujos de Integración y Despliegue Continuo (CI/CD) utilizando **GitHub Actions**.

La infraestructura global gestiona el ciclo de vida completo de la aplicación:

1. **Aprovisionamiento como Código (IaC):** La infraestructura del servidor en Hetzner está construida con **Terraform**, parametrizando todas las variables posibles a través del fichero `.env.development` (como el tipo de máquina `TF_VAR_server_type` o el sistema operativo `TF_VAR_image`). Terraform se encarga de aprovisionar la máquina física, instalar Ubuntu, asignar una IP pública fija y blindar el acceso configurando **claves criptográficas SSH**, permitiendo una conexión M2M (Machine to Machine) segura.

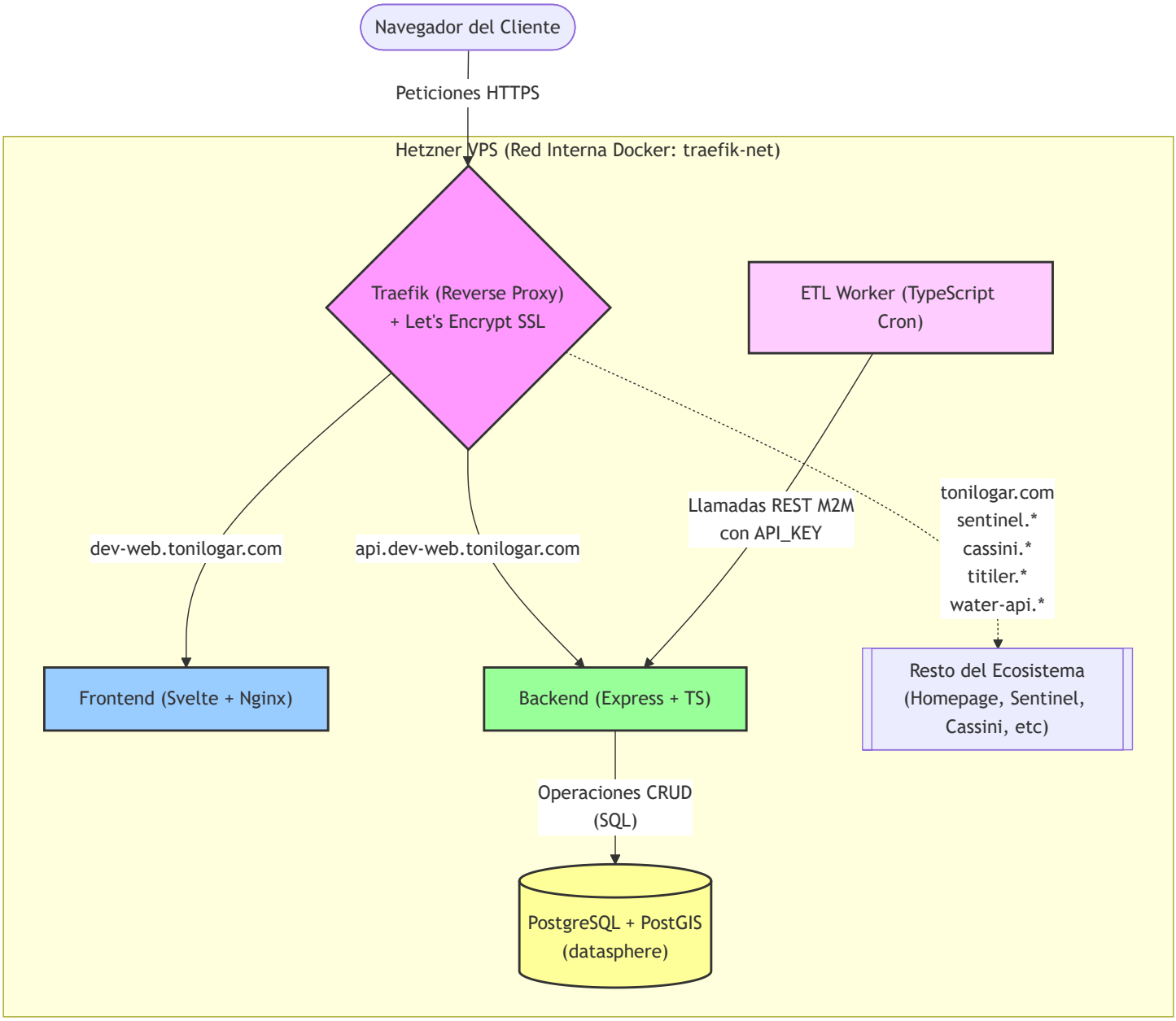
2. **Repositorio y Secretos:** El código fuente está en GitHub. Las variables de producción (`.env.production` como `DEV_WEB_DB_URL`, `ETL_API_KEY`) no se suben al código, sino que están en **GitHub Secrets** (ej. `PROD_ENV_FILE`, `VPS_IP`, `VPS_SSH_KEY`).

3. **Pipeline de CI/CD (GitHub Actions):**

   - A través del fichero de workflow `production-deploy.yml`, se define un proceso que se dispara **automáticamente con cada `push`** hacia las ramas `main` o `main_dev_pro`.
   - Cuando GitHub Actions detecta el push, se conecta vía SSH al servidor de Hetzner usando las claves guardadas en los Secrets.
   - Transfiere el código actualizado descartando ficheros innecesarios según `.gitignore`.
   - Docker se encarga de recompilar el Frontend (con su *multi-stage build*), el Backend y el Worker, reiniciando los servicios de manera controlada sin intervención manual.

4. **Orquestación, Enrutamiento y SSL (Traefik):** Dentro del VPS, **Traefik** actúa como el único punto de entrada público (*Reverse Proxy*). Enrutando las peticiones, Traefik se comunica automáticamente con Let's Encrypt para solicitar, generar y renovar **certificados SSL Wildcard** (ej. `*.tonilogar.com`). Esto garantiza que cualquier subdominio tenga HTTPS de forma totalmente autónoma.

3.2 Diagrama de Arquitectura Global



El ecosistema completo se comunica de la siguiente manera:

1. **Navegador del Cliente**: El usuario accede a la web pública.
   Peticiones seguras (HTTPS)
2. **Traefik (Reverse Proxy)**: Recibe todo el tráfico externo en el VPS, resuelve el cifrado SSL y enruta la petición hacia la red privada de contenedores.
   Red Interna Docker
3. **Backend Express (API REST)**: Contenedor Node.js/TypeScript que procesa las peticiones de los usuarios, gestiona las operaciones CQRS y protege el acceso a la base de datos.
   Consultas SQL (Prisma ORM)
4. **Base de Datos PostgreSQL + PostGIS**: Contenedor maestro que centraliza toda la persistencia de datos (Activos, Peligros, Exposición).
   Llamadas M2M seguras (API Keys)
5. **ETL TypeScript Worker**: Contenedor independiente ejecutando un Cron Job (`node-cron` con TypeScript). Lee datos crudos, los transforma y los inyecta periódicamente llamando al Backend.