

Carlos Medina - 202112046

Pablo Ortega Cadavid - 202021700

Santiago Andrés Gélvez Galvis - 202212387

## 1. Configuración de la Infraestructura

El sistema está compuesto por los siguientes componentes:

- **Frontend:**  
Implementado con React y desplegado en un contenedor Docker que escucha en el puerto **5173**. Los usuarios acceden a la IP pública del frontend desde cualquier ubicación.
- **Backend:**  
Una API desarrollada con FastAPI que se ejecuta en contenedores Docker en instancias de VM. Cada contenedor expone el puerto 8000.
- **Balanceador de Carga:**  
Un balanceador se encarga de distribuir las solicitudes entre dos (o más) instancias del backend. Este componente garantiza que la carga se distribuya de forma eficiente y oculta las IPs privadas de las instancias. Además de permitir escalabilidad horizontal en caso de ser necesario.
- **Base de Datos:**  
Una instancia de CloudSQL con PostgreSQL que se comunica únicamente a través de IPs privadas con el backend, para realizar las operaciones CRUD.

Todos los componentes se encuentran dentro de una VPC, lo que asegura que la comunicación interna se realice de forma privada y segura.

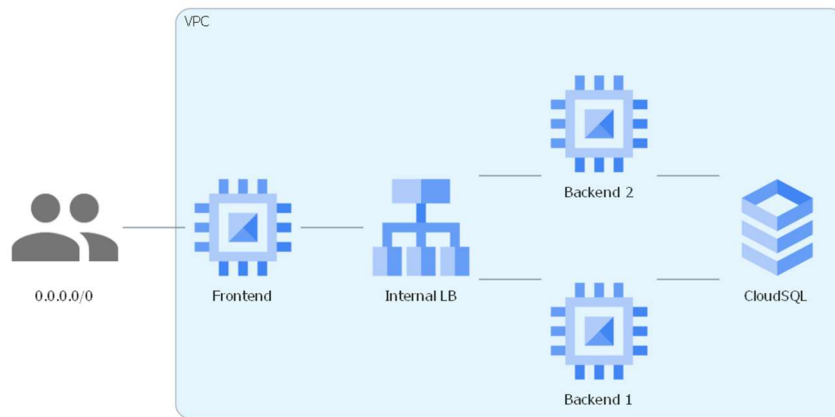


Diagrama de la arquitectura de la solución desplegada en GCP

**Diagrama integral de la infraestructura y flujo de comunicación entre los componentes.**

## 2. Configuración de la API

La API, desarrollada con FastAPI, gestiona usuarios, posts, calificaciones y etiquetas. A continuación, se describen los endpoints principales:

### a) Autenticación

#### **POST /auth/login**

Función: Permite a un usuario iniciar sesión y obtener un token JWT.

Solicitud (JSON): { "email": "usuario@example.com", "password": "password123" }

Respuesta exitosa (200): { "access\_token": "eyJhbGciOiJI...", "token\_type": "bearer" }

Errores: 401 Unauthorized si las credenciales son incorrectas.

#### **POST /auth/register**

Función: Registra un nuevo usuario.

Solicitud (JSON): { "name": "Usuario Ejemplo", "email": "usuario@example.com", "password": "password123" }

Respuesta exitosa (201): Usuario creado exitosamente.

### b) Gestión de Posts

#### **GET /posts**

Función: Recupera la lista de posts existentes.

Autenticación: No requerida.

Respuesta (200):

```
[ { "id": 1, "title": "Primer Post", "content": "Contenido del post", "author":  
"usuario@example.com" } ]
```

## **POST /posts**

Función: Crea un nuevo post.

Autenticación: Requerida (el token JWT se debe enviar en el encabezado "Authorization" como "Bearer <token>").

Solicitud (JSON): { "title": "Nuevo Post", "content": "Contenido del post" }

Respuesta exitosa (201): Se confirma la creación del post.

Errores: 401 Unauthorized si el token es inválido o no se proporciona.

Otros endpoints (por ejemplo, para calificaciones y etiquetas) seguirán una lógica similar: se describe la acción, se indica si se requiere autenticación, y se muestran ejemplos de solicitud y respuesta para casos de éxito y error.

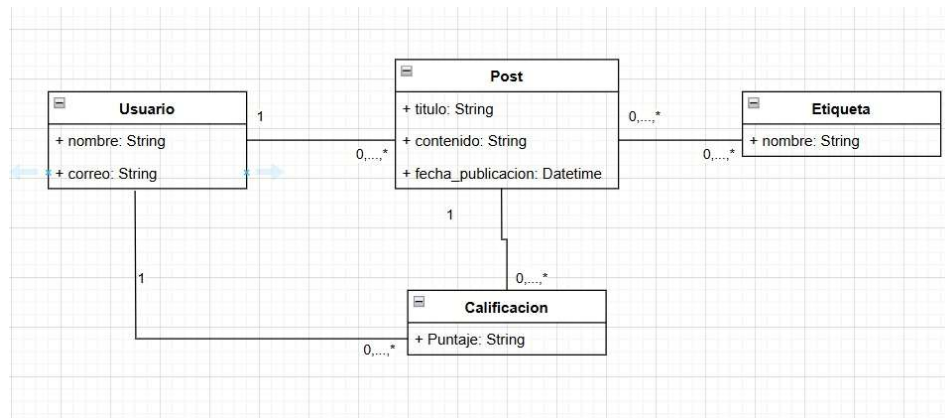
## **3. Esquema de la Base de Datos**

El modelo de datos se organiza de la siguiente forma:

- **Usuarios:**  
Cada usuario tiene nombre, correo y contraseña. Un usuario puede crear posts y calificaciones.
- **Posts:**  
Cada post tiene un autor (usuario) y puede tener uno o más tags asignados.
- **Calificaciones:**  
Se crean por usuarios y se refieren a un post específico.
- **Tags (Etiquetas):**  
Pueden asignarse a posts y no están asociados a un usuario específico.

```
ssh.cloud.google.com/v2/ssh/projects/blogapp-451901/zones/us-east1-d/instances/blogapp-backend1?authuser=0&hl=es_419&projectN...
ssh.cloud.google.com/v2/ssh/projects/blogapp-451901/zones/us-east1-d/instances/blogapp-backend1?authuser=0&hl=es_419&...
SSH en el navegador SUBIR ARCHIVO DESCARGAR ARCHIVO
GNU nano 5.4 Dockerfile
FROM python:3.10
WORKDIR /app
COPY requirements.txt .
RUN pip install --upgrade pip
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 8000
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Se incluye el Dockerfile utilizado para crear la imagen de uno de los contenedores del backend. En él se expone el puerto 8000 y se inicia la aplicación mediante el servidor *uvicorn*.



Esquema del modelo de datos (estructura y relaciones entre entidades).

## 4. Instrucciones de Despliegue y Uso

### a) Despliegue del Backend (FastAPI)

1. Clonar el repositorio:  
git clone <repositorio>  
cd blogApp-main
2. Instalar las dependencias:  
pip install -r requirements.txt
3. Ejecutar la API:  
uvicorn app.main:app --host 0.0.0.0 --port 8000

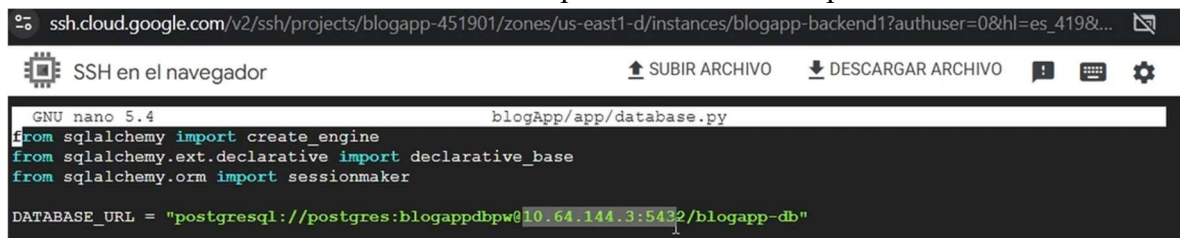
Se recomienda ejecutar la API dentro de un contenedor Docker para facilitar el despliegue y la escalabilidad.

## b) Despliegue del Frontend (React con Vite)

1. Clonar el repositorio:  
`git clone <repositorio>`  
`cd bloggAppFront-main/blog-app-front`
2. Instalar las dependencias:  
`npm install`
3. Iniciar la aplicación:  
`npm run dev`  
La aplicación se ejecutará en el puerto 5173.

## 5. Decisiones de Diseño y Seguridad

- **Infraestructura Segura:**  
El único nodo accesible desde Internet es el frontend, mientras que los demás componentes (balanceador, backend, base de datos) se comunican de forma privada dentro de la VPC.
- **Reglas de Firewall:**
  - Acceso público restringido únicamente al puerto 5173 del frontend.
  - Comunicación entre el balanceador y el backend únicamente por el puerto 8000.
  - (definido en el código) El backend se comunica con la base de datos exclusivamente a través del puerto 5432 de la IP privada del balanceador.



The screenshot shows a terminal window with the title "SSH en el navegador" and a URL bar containing "ssh.cloud.google.com/v2/ssh/projects/blogapp-451901/zones/us-east1-d/instances/blogapp-backend1?authuser=0&hl=es\_419&...". The terminal content shows the file "blogApp/app/database.py" being edited with GNU nano 5.4. The code defines the database URL as "postgresql://postgres:blogappdbpw@10.64.144.3:5432/blogapp-db".

```
GNU nano 5.4 blogApp/app/database.py
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

DATABASE_URL = "postgresql://postgres:blogappdbpw@10.64.144.3:5432/blogapp-db"
```

**Acceso desde los backs a la IP privada del balanceador, puerto 5432**

- (definido en el código) El frontend manda las solicitudes a la ip del balanceador, no a las del back.

```
ssh.cloud.google.com/v2/ssh/projects/blogapp-451901/zones/us-east1-d/instances/blogapp-frontend?authuser=0&hl=es_419&p...
SSH en el navegador SUBIR ARCHIVO DESCARGAR ARCHIVO
GNU nano 5.4 useCreatePost.jsx
import { useState } from 'react';

const useCreatePost = () => {
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);
  const [success, setSuccess] = useState(null);

  const createPost = async (postData) => {
    setLoading(true);
    setError(null);
    setSuccess(null);
    try {
      const storedUser = localStorage.getItem('user');
      const token = storedUser ? JSON.parse(storedUser).token : null;

      const response = await fetch('http://34.23.205.237:8000/posts/create/', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
          ...(token ? { 'Authorization': `Bearer ${token}` } : {})
        }
      });
    } catch (error) {
      setError(error);
    }
  };

  return { createPost, loading, error, success };
};
```

Las funciones fetch usan la IP del balanceador, puerto 8000

## external-lb

Balanceador de cargas de aplicaciones externo regional

### Región

us-east1

### Frontend

Protocolo ↑	IP:Puerto	Nivel de red	Certificado	Política de SSL	Tiempo de espera keepalive de HTTP ?
HTTP	34.23.205.237:8000	Premium	-		600 segundos

- (definido en el código) El backend solo recibe solicitudes de la ip del front (CORS)

```
app = FastAPI()

origins = [
  "http://34.148.86.186:5173",
  "http://10.142.0.5:5173",
]

app.add_middleware(
  CORSMiddleware,
  allow_origins=origins, # Allow these origins
  allow_credentials=True,
  allow_methods=["*"],
  allow_headers=["*"],
)
```

☐ ☒ blogapp-frontend 10.142.0.5 (nic0) 34.148.86.186 (nic0) blogapp-frontend lb-health-check

Rechaza cualquier otra IP intentando hacer requests al backend

- **IAM en CloudSQL:**

Se configuraron roles y políticas de IAM para garantizar que solo las instancias autorizadas del backend puedan acceder a la base de datos.

- **Escalabilidad:**

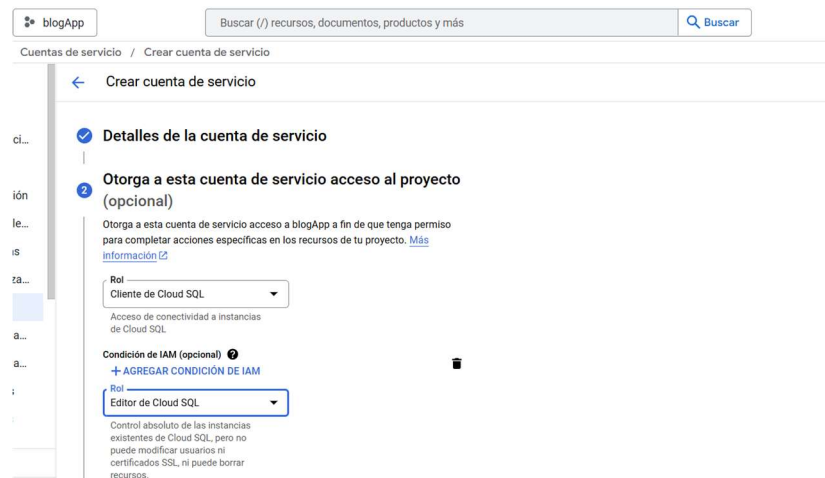
La implementación del balanceador de carga permite la escalabilidad horizontal. En

caso de aumento de demanda, basta con agregar nuevas instancias al grupo de backend y desplegar el contenedor correspondiente en la nueva VM. Adicionalmente con los balanceadores de carga es posible realizar health checks en las instancias para monitorear su funcionamiento.

- **Cuenta de servicio para el backend:** Se creó una cuenta de servicio específica (IAM) para que las instancias del backend accedan a CloudSQL con los permisos mínimos necesarios.

<input type="checkbox"/>	Correo electrónico	Estado	Nombre ↑	Descripción
<input type="checkbox"/>	 <a href="mailto:blogapp-backend-sa@blogapp-451901.iam.gserviceaccount.com">blogapp-backend-sa@blogapp-451901.iam.gserviceaccount.com</a>	 Habilitado	blogapp-backend-sa	Acceder a cloudSQL a través de cuentas de servicio

#### Correo de la cuenta de servicio.



#### Permisos de la cuenta de servicio.

- **Principio de privilegio mínimo:** Se aplican roles con los permisos estrictamente necesarios para cada función dentro del sistema. Como es el caso de las reglas de firewall.

#### 6. Video

<https://youtu.be/CseA9MMoa4Y>