Universidad de Los Andes

Bogotá, Colombia

Facultad de Ingeniería de Sistemas y Computación

Proyecto No. 1 Seguridad en Cloud

Desarrollo y despliegue de una aplicación en la nube

MSIN4215

Autores:

Elkin Cuello 202215037

Julián Contreras 202211884

Juan Felipe Lancheros Carrillo 202211004

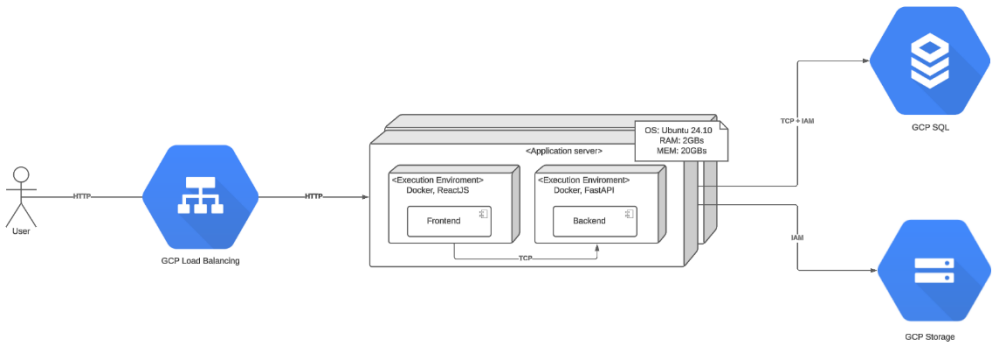
Febrero 2025

Repositorio del Frontend: https://github.com/Seguridad-en-cloud-202510/Proyecto1-front

Repositorio del Backend: https://github.com/Seguridad-en-cloud-202510/Proyecto1-back

Video explicativo: https://youtu.be/_XmGlFzJR3E

1) Decisiones de diseño y de seguridad de la aplicación y del despliegue
    a) Diagrama de arquitectura de la solución desplegada en GCP



        i) Dado que se requiere un modelo escalable, una solución aceptable es el despliegue de una granja de máquinas virtuales donde cada una maneja en contenedores individuales los componentes frontend y backend de la aplicación. Estos servidores replicados se manejan a través del servicio de balanceador de carga, el cual se enfoca en que cuando aumente el número de conexiones a la aplicación un único servidor no sea quien asuma toda la carga. De igual manera, está encargado de constantemente verificar que los servidores a los cuales está conectado estén en funcionamiento.
        ii) Una alternativa con mayor grado de escalabilidad es desplegar cada contenedor en máquinas virtuales diferentes con el objetivo de que se agregue un balanceador de carga. De esta manera, el balanceador de carga inicial estaría orientado a proporcionar escalabilidad al grupo de servidores que repliquen el front y el nuevo balanceador de carga estaría orientado a proporcionar escalabilidad al grupo de servidores que repliquen el back. En consecuencia, esta solución garantiza un mayor índice de disponibilidad, aunque consumiría una mayor cantidad de recursos y, por ende, es más costosa.

    b) Implementación de la Autenticación

i) **Autenticación de Usuarios (Backend):** Para la autenticación, se usa **JWT (JSON Web Token)**, donde los tokens son firmados con una **SECRET_KEY** utilizando el algoritmo **HS256**. Esto asegura la integridad del token y permite la autenticación segura de los usuarios sin almacenar sus credenciales en el cliente.
La SECRET_KEY se generó con un password manager que corre en local (Bitwarden) usando 128 caracteres aleatorios que incluyen alfanuméricos, mayúsculas, minúsculas y caracteres especiales.
Todo el proceso de creación y verificación de los JWT se hace en el archivo ***security.py*** del Backend.

ii) **Autenticación de Usuarios (Frontend):** El proceso de autenticación se basa en un "access token" proporcionado por el backend, que el frontend utiliza para verificar la sesión y otorgar o denegar el acceso. Cuando el usuario inicia sesión, dicho token se almacena en el local storage del navegador, lo que permite conservar la autenticación incluso al recargar la página. Después, cada vez que el usuario interactúa con la aplicación (por ejemplo, al hacer clic en un botón o navegar a otra sección), se verifica la presencia y la vigencia del token. En caso de que el token no exista o haya expirado, el usuario es redirigido automáticamente a la página de inicio de sesión para evitar accesos no autorizados.

c) **Interfaz de Usuario**: La interfaz está diseñada para ser lo más intuitiva posible para el usuario. Al ingresar a la página, se presenta una landing page en la que el usuario puede iniciar sesión o ver los blogs publicados. Si el usuario decide no iniciar sesión, podrá únicamente visualizar y filtrar los blogs. Al hacerlo, se habilita un panel lateral mediante el botón de filtrado, que permite buscar publicaciones por el nombre del blog o sus etiquetas.

   En caso de que el usuario opte por iniciar sesión, se activan funciones adicionales: puede crear nuevos blogs y administrar los que le pertenecen (editarlos, eliminarlos, publicarlos o despublicarlos). Para optimizar la lectura y carga de contenido, la página implementa un componente de paginación en la parte inferior. Esto ayuda a gestionar grandes volúmenes de datos, evitando largos listados de publicaciones en una sola vista y brindando una experiencia de usuario más fluida y ordenada.

Otro aspecto importante es el uso del editor WYSIWYG de ReactQuill para gestionar el contenido de los blogs. Gracias a esta herramienta, los usuarios pueden redactar y dar formato a sus publicaciones.

d) **Lógica-Base de** Datos: Para la interacción entre el Backend en Python FastAPI con la base de datos hay varias librerías disponibles. Se optó por utilizar "***asyncpg***". Esta librería ofrece dos características en particular que fueron determinantes al escoger una:

   i) **Protección contra inyección SQL:** Esta librería no toma lo que el usuario ingresa directamente como parte de la sentencia SQL. En su lugar, utiliza un sistema de parámetros, por lo que, aunque se pasen parámetros potencialmente peligrosos, estos se interpretarán como texto plano y no como código ejecutable.

   ii) **Concurrencia:** En experiencias reales en la industria, parte del equipo ha trabajado previamente con librerías que conectan python a una base de datos postgresql como "psycopg2". La mayoría de estas generan un enorme cuello de botella al ejecutar un gran volumen de queries, por lo que una solución concurrente mejora mucho la latencia y escalabilidad de nuestra aplicación.

e) Otras medidas de seguridad

   i) **Almacenamiento de contraseñas:** Las contraseñas no se almacenan en texto claro dentro de la base de datos. En su lugar, se almacena un hash generado en el Backend con la librería "**CryptContext"** utilizando el algoritmo "**bcrypt**". Este proceso garantiza que las contraseñas no sean recuperables en texto claro. Así se ven las contraseñas de los usuarios:

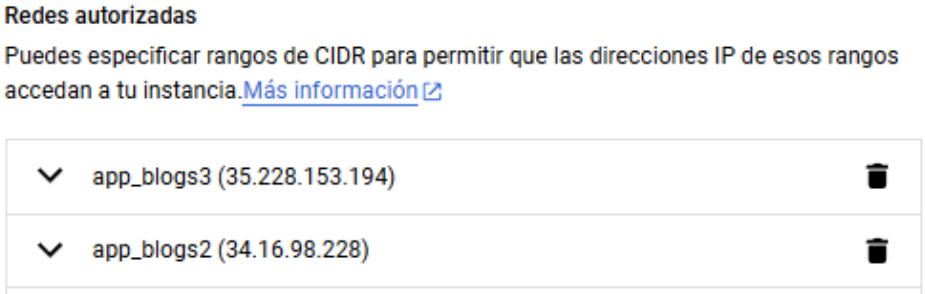
```
db_app_blogs=# select * from usuario;
 id_usuario |     nombre     |          email          |                          contrasenia
------------+----------------+-------------------------+--------------------------------------------------------------
          9 | string         | user@example.com        | $2b$12$xuVc6WJ3OcetPs3wNqimL.3OBsXLNDEGPf0RoK4cVVFu9XKm1ho12
         10 | julian         | julian@julian.com       | $2b$12$XxkG3SUHkm2MNwEuNp54A.CeTwMzu8xqlNMJB.xf/BL6U3./1sgFm
         11 | usuario_prueba | usuario_prueba@gmail.com | $2b$12$i7u6JLD8elhr.YAYg/oG8eK6nWXaZeT9h/UUxZTo2QtAWT.Q4Hc92
```

   ii) **Protección contra cross-site scripting:** La aplicación utiliza la librería DOMPurify para eliminar o "sanitizar" el contenido HTML que los usuarios ingresan en las entradas de los blogs antes de guardarlas. Esto resulta fundamental, ya que las publicaciones se renderizan directamente como HTML al momento de mostrarse, lo que representa un riesgo significativo si el usuario incluye código malicioso. Por ejemplo, un atacante podría inyectar scripts que se ejecuten en el navegador, poniendo en peligro la información y la seguridad de los

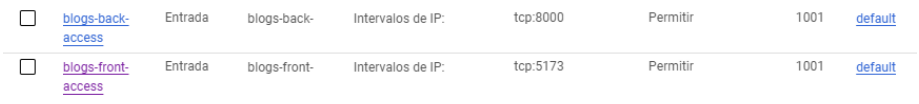
demás usuarios. DOMPurify se encarga de filtrar y remover las etiquetas o atributos peligrosos, minimizando así la posibilidad de que se ejecute código malicioso.

iii) **Restricción de edición, eliminación y publicación**: Se definió que toda operación diferente a **GET** requeriría de autenticación por medio de **JWT.** No es posible editar, eliminar o crear publicaciones sin haber iniciado sesión previamente.
La única operación **POST** que se puede ejecutar sin necesidad de un JWT es crear un usuario nuevo o iniciar sesión.

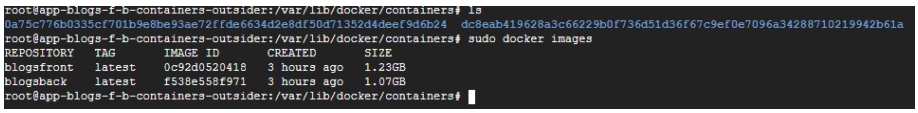
iv) Redes autorizadas para la conexión a Cloud SQL: Es una capa adicional de seguridad enfocada en garantizar que las únicas instancias que se podrían conectar con el servicio son las explícitamente definidas:



v) Reglas de Firewall: Usando esta herramienta, se habilitan únicamente los puertos de las instancias de cómputo en los que funciona el servicio (los que no están definidos se bloquean por defecto), como se detalla en la siguiente imagen:



2) Documentación del proyecto
   a) Configuración de la infraestructura
      i) Uso de contenedores Docker y Docker Compose: Como se mencionó el primer punto, los componentes de frontend y backend fueron desplegados individualmente en contenedores Docker dentro de una misma máquina.



Cada imagen asociada a cada contenedor fue generada a partir del repositorio específico donde se almacena el código principal de la aplicación para cada componente. En otras palabras, en cada máquina virtual se manejaba un directorio con dos carpetas: una con el repositorio clonado del backend de la aplicación y otra con el repositorio del frontend de la aplicación. Dentro de cada carpeta, se generaron las imágenes asociadas a cada componente individual con base a sus "dependencias". Los archivos Dockerfile usados fueron los siguientes:

(1) Backend

```
  GNU nano 8.1
FROM python:3.12.7

COPY . .

RUN pip install -r requirements.txt

CMD uvicorn main:app --host 0.0.0.0 --port 8000
```

(2) Frontend

```
  GNU nano 8.1
FROM node:latest

WORKDIR /

COPY . .

RUN npm install

RUN npm install react-bootstrap

CMD npm run dev
```

Ejecutar un contenedor por cada imagen generada dentro de la misma máquina virtual fue "suficiente" para que la aplicación completa funcionara. Por lo anterior, no fue requerido usar Docker Compose.

ii) "Aprovisionamiento" de instancias de Compute Engine: Para minimizar el costo de despliegue mientras se accede a un desempeño aceptable de la aplicación, las dos instancias de cómputo fueron generadas con las siguientes características de máquina:

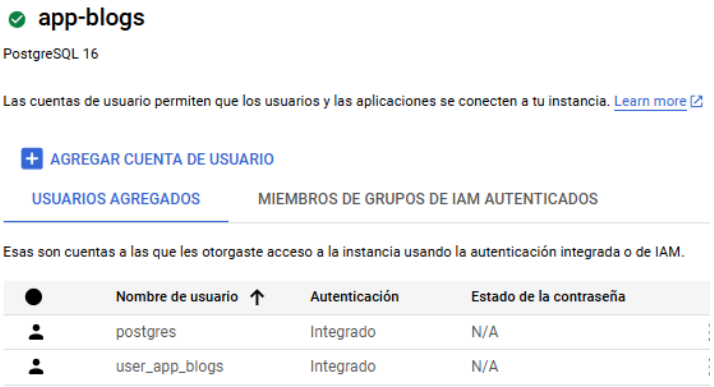
Tipo de máquina	e2-small (2 vCPUs, 2 GB Memory)
Plataforma de CPU	AMD Rome
Plataforma de CPU mínima	Ninguna
Arquitectura	x86-64

iii) Integración de la aplicación con los siguientes servicios:

(1) Cloud SQL: La aplicación por defecto usaba almacenamiento por PostgreSQL 16 de manera local. En la nube, se mantuvo dicha asociación, lo cual se expresa en los siguientes puntos:

(a) Base de datos

(b) Usuario de acceso (ignorar “postgres”)



(c) Conexión con el backend (ver archivo database.py)

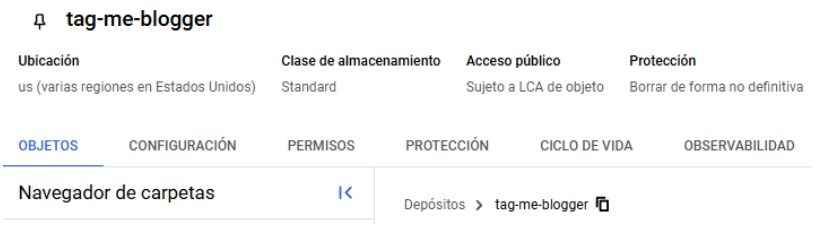
```
DATABASE_URL = "postgresql://user_app_blogs:password1234@34.60.85.3:5432/db_app_blogs"

async def connect_to_db():
    return await asyncpg.create_pool(DATABASE_URL)

async def close_db_connection(pool):
    await pool.close()
```

(d) Las consultas SQL fueron realizadas directamente por consola ejecutando el comando “psql --host=34.60.85.3 --port=5432 --username=user_app_blogs --password --dbname=db_app_blogs”.

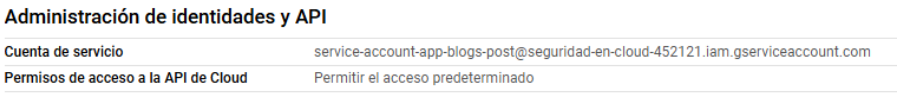
(2) Cloud Storage: Generación del servicio:



No obstante, no usamos Cloud Storage, ya que las imágenes requeridas para la aplicación se almacenan directamente en la base de datos de Cloud SQL. Estas imágenes se guardan como

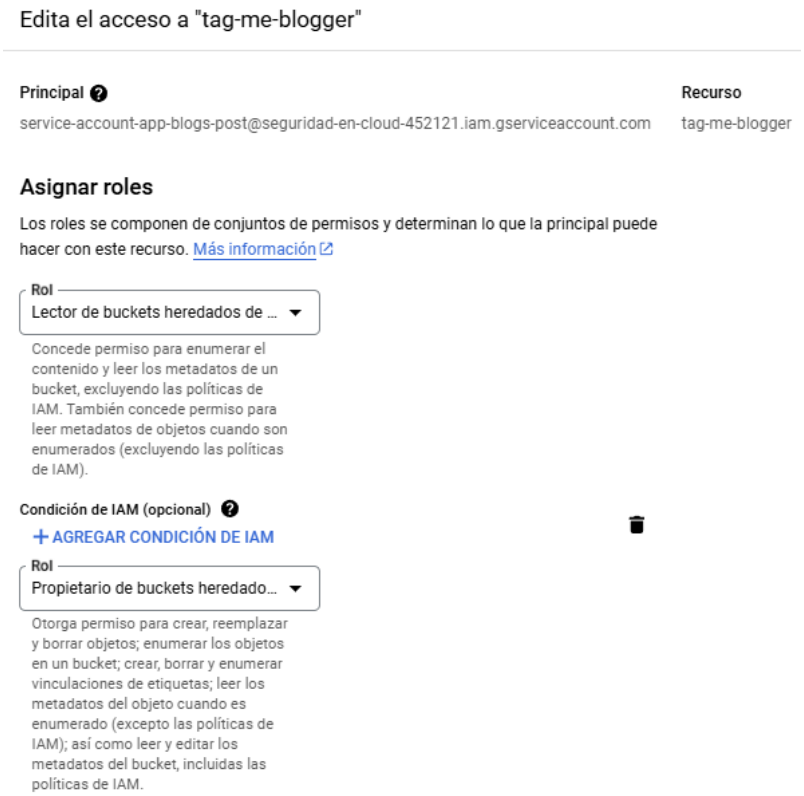
cadenas de texto en formato base64, lo que permite gestionarlas sin necesidad de un servicio de almacenamiento adicional. Al estar incluidas en la base de datos, se simplifica la arquitectura de la aplicación y se eliminan los costos y la complejidad habituales de un servicio en la nube para guardar archivos. Además, el volumen de imágenes esperado resulta manejable a través de este método, asegurando un equilibrio adecuado entre simplicidad y eficiencia.

iv) Acceso seguro a través de IAM y cuentas de servicio: Las instancias de cómputo fueron asociadas con una cuenta de servicio específica con el objetivo de que el acceso a los servicios de almacenamiento fuera totalmente exclusivo para dichos recursos. En la siguiente imagen, se muestra su configuración en las máquinas virtuales:

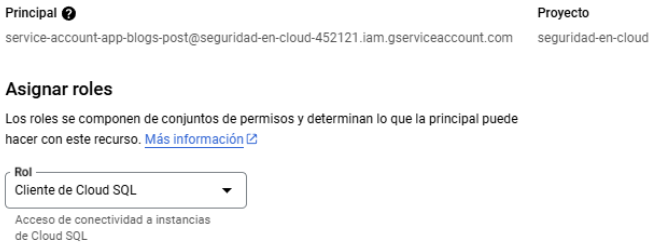


Esto se aplicó en los siguientes servicios:

(1) Cloud Storage



(2) Cloud SQL



b) API

i) Implementación de la API REST

(1) Para la API se utilizó FastAPI de python. La implementación sigue una estructura modular que facilita la mantenibilidad y escalabilidad del sistema.

(2) Para garantizar una correcta separación de responsabilidades, la API se organizó en los siguientes módulos principales:

(a) **models.py**: Define los esquemas de datos mediante Pydantic, asegurando validación estricta de los datos de entrada y salida. Incluye modelos para usuarios, publicaciones, etiquetas y calificaciones.

(b) **crud.py**: Contiene la lógica de acceso a la base de datos utilizando asyncpg, implementando operaciones CRUD (Create, Read, Update, Delete) de forma asincrónica.

(c) **main.py**: Configura la aplicación principal de FastAPI, define los endpoints y gestiona la interacción entre los modelos y la capa de datos. Además, implementa la protección de rutas con JWT, asegurando que solo los usuarios autenticados puedan acceder a ciertas funcionalidades.

(3) Gestión de Publicaciones y Etiquetas:

(a) Se implementaron endpoints para la creación, actualización, eliminación y consulta de publicaciones.

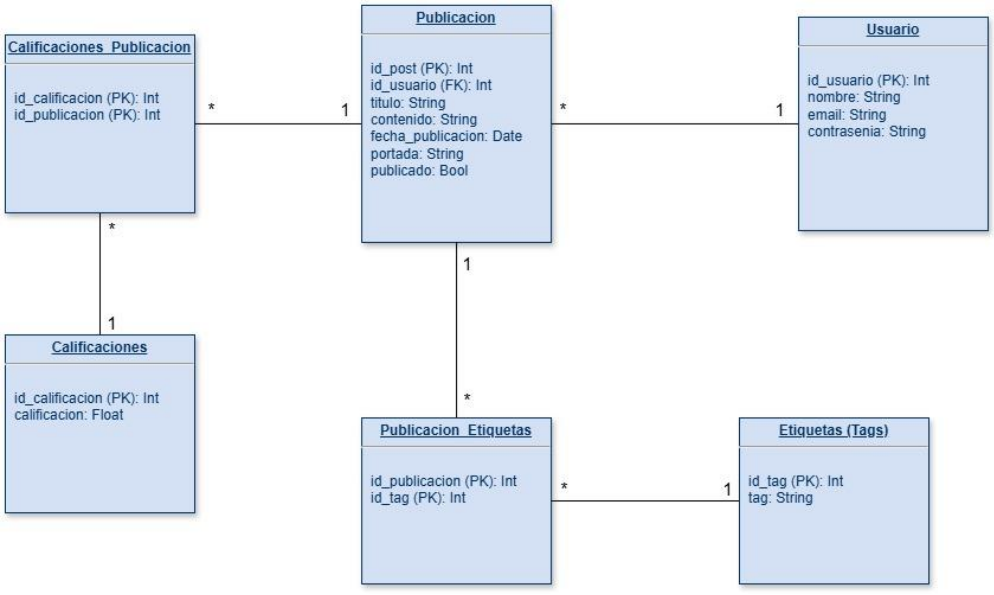
(b) Se incorporó un sistema de etiquetas que permite asignar múltiples categorías a cada publicación y filtrar contenido por etiquetas.

(4) Paginación:

(a) Se aplicó paginación en las consultas de publicaciones desde el Backend para manejar grandes volúmenes de datos de manera eficiente.

ii) Integración con Backend (Consumo de API): La integración entre la aplicación en React y el backend en FastAPI se logró mediante el consumo de una API REST utilizando axios para realizar solicitudes HTTP. FastAPI se encargó de gestionar los endpoints, procesando peticiones GET, POST, PUT y DELETE, y enviando respuestas en formato JSON. En el frontend, se implementaron funciones asincrónicas junto con hooks de React como useState y useEffect para manejar el estado y las actualizaciones de datos en tiempo real. Para garantizar una comunicación fluida entre ambos componentes, se configuró CORS en FastAPI, permitiendo que el frontend pudiera acceder a los recursos del backend sin restricciones de seguridad. Además, se utilizaron validaciones de datos con pydantic en FastAPI para garantizar la integridad de la información enviada y recibida. Esta arquitectura permitió una integración eficiente, escalable y fácil de mantener.

c) Esquema de la base de datos



i) Integración con la Base de Datos: La API REST se integra con una base de datos **PostgreSQL** utilizando ***asyncpg***, un cliente asincrónico que permite manejar conexiones y consultas de manera eficiente. La estructura de la base de datos está diseñada para soportar la gestión de usuarios, publicaciones, etiquetas y calificaciones, asegurando

relaciones adecuadas entre las entidades mediante claves foráneas. Para la conexión, se configura un pool de conexiones en el archivo ***database.py***, lo que optimiza el acceso concurrente y la escalabilidad de la aplicación. Además, todas las consultas están parametrizadas para prevenir inyecciones SQL y garantizar la seguridad de los datos.

3) Instrucciones de despliegue y uso
    a) Asumiendo que las imágenes de los dos contenedores a generar se encuentran de manera local, se debe ejecutar los siguientes comandos:
        i) Para el frontend: sudo docker run -d -p 5173:5173 --name <nombre_contenedor_del_front_a_generar> <nombre_imagen_front>
        ii) Para el backend: sudo docker run -d -p 5173:5173 --name <nombre_contenedor_del_back_a_generar> <nombre imagen_front>

    Con lo anterior, se deberían ejecutar sin problema la aplicación, la cual se encontrará desplegada en los puertos 5173 para el frontend y 8000/docs para el backend de la IP pública de la máquina virtual. El valor de la IP pública fue modificado previamente en el código de los repositorios para generar las conexiones. Sin embargo, nos dimos cuenta que a la hora de desplegar el frontend dentro de un contenedor, usando un navegador la aplicación nunca terminaba de cargar. Nunca salió un error, solo se quedaba cargando. El backend corre sin problema. En cambio, si corremos el frontend sin un contenedor desde el directorio donde se encuentre el repositorio clonado como lo siguiente:
        iii) “npm install react-bootstrap" y luego “npm run dev”, la aplicación funciona con normalidad.

    El anterior error podría ser una de las causas por las que el balanceador de carga detectada los grupos de instancias como “unhealthy”.

    Nota: El comando para correr el backend sin contendor es: “uvicorn main:app --host 0.0.0.0 --port 8000”.

4) HTTPS: No fue implementado porque desde la opción proporcionada por GCP se requiere un dominio, tal y como se mencionó en clase.