Universidad de los Andes

Departamento de Ingeniería de Sistemas y Computación

MSIN-4215: SEGURIDAD EN CLOUD

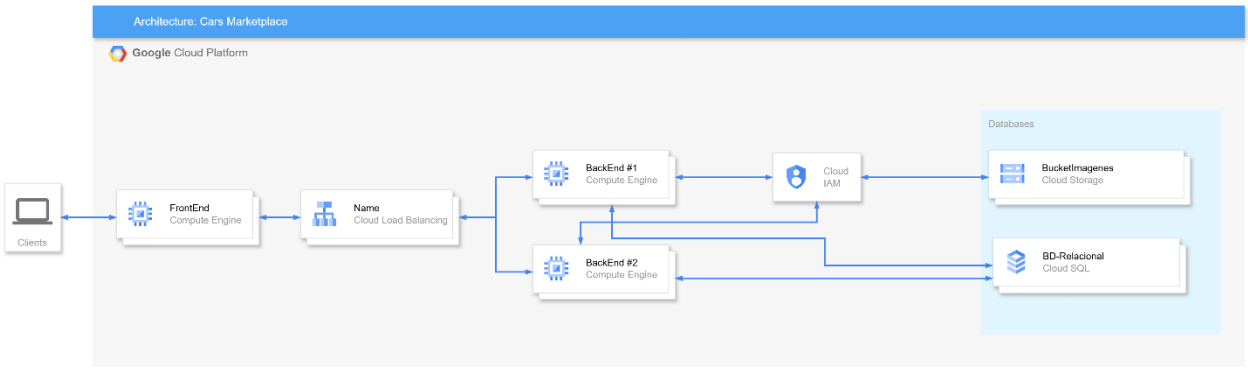
Proyecto #1

Desarrollo y despliegue de una aplicación en la Nube

Nathalia Quiroga Alfaro –
Diego Alejandro González Vargas –
Germán Alberto Rojas Cetina –

2 de marzo de 2025

Configuración de la Infraestructura



En nuestro proyecto de seguridad en Cloud, como se puede ver en el diagrama de arquitectura, se utilizaron múltiples servicios del proveedor de nube Google Cloud Platform (GCP) para garantizar que se cumplieran con cada uno de los requerimientos planteados en el enunciado del proyecto.

De esta forma, se hace necesario explicar cada uno de estos elementos, descirbiendo sus características y la forma en que se encuentran configurados:

Google Cloud SQL:

Sobre este servicio se montaron las bases de datos relacionales que gestionan la información de usuarios, reseñas y publicaciones. Para su funcionamiento, se aseguro su seguridad a partir de la implementación de restricciones de red que permitiesen exclusivamente las conexiones con las entidades contenedoras del backend de la aplicación.
Adicionamente, es importante mencionar que se trabajó con una base de datros Postgres. Lo que garantiza una estructura de precios un poco más económica. Las características de esta base de datos se exponen a continuación:

Resumen

Edición de Cloud SQL	Enterprise
Región	us-central1 (Iowa)
Versión de la base de datos	PostgreSQL 16
CPU virtuales	4 CPU virtual(es)
RAM	16 GB
Caché de datos	Inhabilitada
Almacenamiento	100 GB
Conexiones	IP pública
Copia de seguridad	Automatizada
Disponibilidad	Zona única
Recuperación de un momento determinado	Habilitada

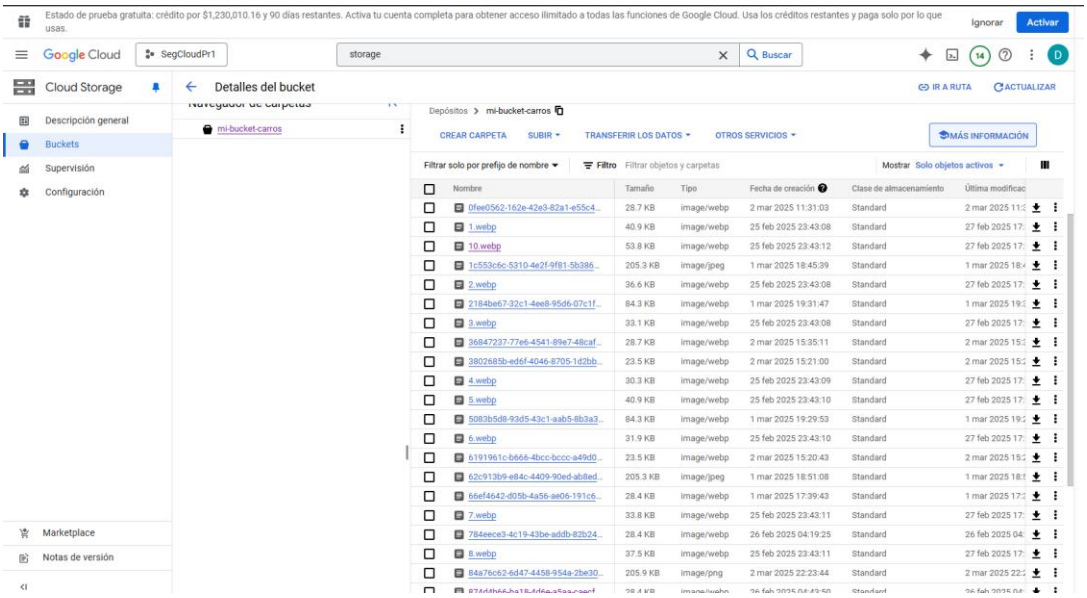
Capacidad de procesamiento de la red (MB/s)	1,000 de 1,000
IOPS	Lectura: 3,000 de 15,000 Escritura: 3,000 de 15,000
Capacidad de procesamiento del disco (MB/s)	Lectura: 48.0 de 240.0 Escritura: 48.0 de 240.0

Así mismo, la estructura de costos es la que se presenta a continuación:

Más información

Elemento	Costo por hora (estimado)
4 CPU virtuales (USD0.041 por CPU virtual por hora)	USD0.17
16 GiB de RAM (USD0.007 por GiB por hora)	USD0.11
100 GiB de SSD (USD0.17 por GiB al mes)	USD0.02
Total sin descuentos por uso	**USD0.30**

Google Storage



Almacenamiento de archivos para fotos de los carros, asegurando su disponibilidad y redundancia a través de Cloud Storage. Este servicio permite la carga, recuperación y administración de imágenes de manera eficiente, facilitando el acceso desde el backend y frontend de la aplicación. Además, se implementó una política de control de acceso mediante IAM para restringir el acceso solo a las cuentas de servicio autorizadas, garantizando la seguridad de los datos almacenados. Es importante mencionar que, para mantener una buena estructura de costos y para mejorar el desempeño de la aplicación, se utilizó, al igual que en todo el proyecto la zona central e1 de Estados Unidos.

Balanceador de carga

Se utilizó un balanceador de carga para la implementación de escalamiento horizontal de las peticiones del backend, tal y como lo sugería el enunciado. De esta forma, se tiene una forma de equilibrar las peticiones de los usuarios en más de una instancia, este balanceador de carga funciona con una verificación de estado que realiza para saber el nivel de saturación de las instancia de back. Esta verificación de estado no es otra cosa que un endpoint de salud. Este endpoint, si llega a recibir 2 peticiones seguidas no satisfactorias de alguna de las máquinas, la declara inutilizable.

Ruta
/health/

Protocolo
HTTP

Puerto
5000

Protocolo de proxy
NINGUNO

Registros
Habilitado

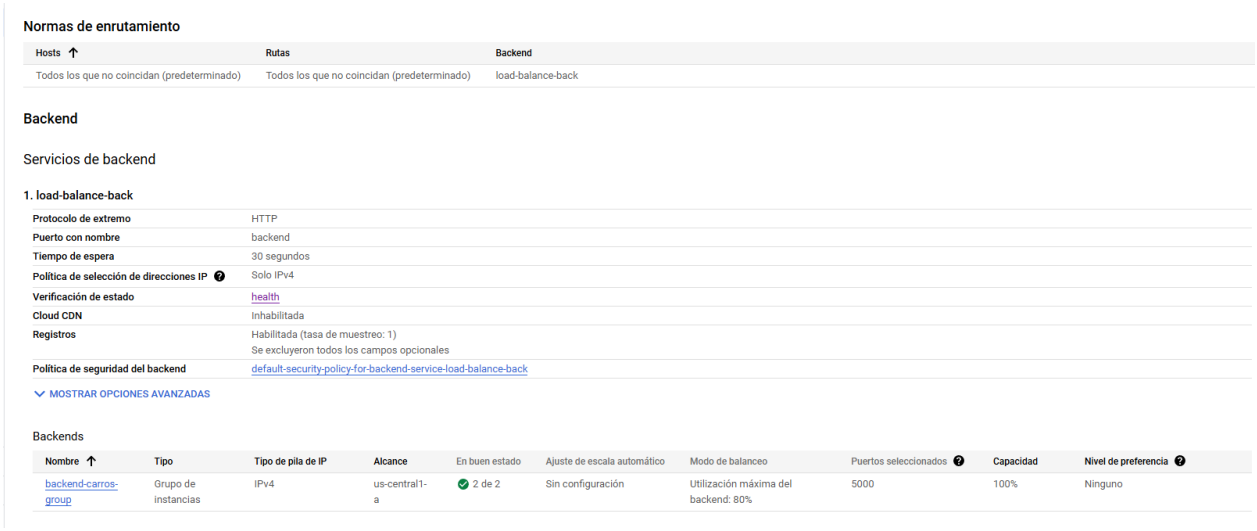
Intervalo
45 segundos

Tiempo de espera
5 segundos

Umbral de buen estado
2 resultados correctos consecutivos

Umbral de mal estado
2 errores consecutivos

A continuación, se evidencia el correcto funcionamiento del balanceador



Normas de enrutamiento

Hosts	Rutas	Backend
Todos los que no coincidan (predeterminado)	Todos los que no coincidan (predeterminado)	load-balance-back

Backend

Servicios de backend

1. load-balance-back

Protocolo de extremo	HTTP
Puerto con nombre	backend
Tiempo de espera	30 segundos
Política de selección de direcciones IP	Solo IPv4
Verificación de estado	health
Cloud CDN	Inhabilitada
Registros	Habilitada (tasa de muestreo: 1) Se excluyeron todos los campos opcionales
Política de seguridad del backend	default-security-policy-for-backend-service-load-balance-back

MOSTRAR OPCIONES AVANZADAS

Backends

Nombre	Tipo	Tipo de pila de IP	Alcance	En buen estado	Ajuste de escala automático	Modo de balanceo	Puertos seleccionados	Capacidad	Nivel de preferencia
backend-carros-group	Grupo de instancias	IPv4	us-central1-a	2 de 2	Sin configuración	Utilización máxima del backend: 80%	5000	100%	Ninguno

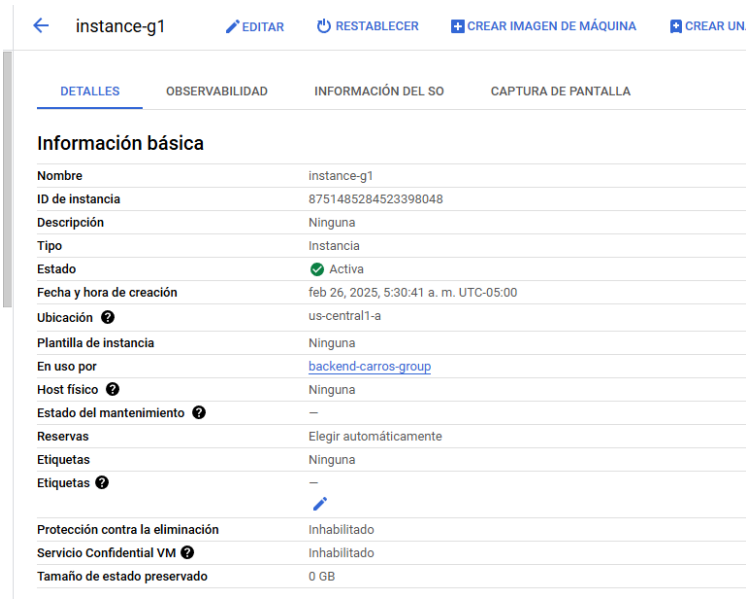
Google Cloud Engine:

Back End:

Como se mencionó anteriormente, se trata de un grupo de instancias de 2 instancias iguales las cuales reciben las peticiones del front. Al igual que en el caso del resto de los elementos, se encuentran implementadas en la primera zona central de Estados Unidos, y con las siguientes características:

- **Tipo de máquina:** e2-small (2 vCPUs, 2 GB RAM)
- **Plataforma de CPU:** AMD Rome
- **Sistema operativo:** Debian 12 (Bookworm)
- **Almacenamiento:** 10 GB SSD
- **Costo:** 14 dolares mensuales aproximadamente(con uso ininterrumpido)

A continuación, se muestra las maquinas corriendo y sus características:



FrontEnd: Para el front end se utilizó una única máquina en la cual se estableció todo el código de node js, con componentes, paginas y demás elementos necesarios para su correcto funcionamiento, respecto de la máquina, cuenta con las siguientes características:

- **Ubicación:** us-central1-c
- **Tipo de máquina:** e2-small (2 vCPUs, 2 GB RAM)
- **Plataforma de CPU:** Intel Broadwell
- **Sistema operativo:** Debian 12 (Bookworm)
- **Almacenamiento:** 10 GB SSD
- **Costo:** 14 dólares mensuales aproximadamente(con uso ininterrumpido)

Configuración de la API

El backend del proyecto **SegCloud** está estructurado dentro del directorio backend/src, donde se organizan todos los archivos necesarios para su funcionamiento. En la raíz de este directorio se encuentra app.py, que probablemente actúa como el punto de entrada de la aplicación. Este archivo es el encargado de inicializar la aplicación, configurar las rutas y establecer la conexión con la base de datos. Junto a él, se encuentran archivos clave como requirements.txt, que lista las dependencias necesarias para ejecutar el backend, y docker-compose.yaml junto con dockerfile, que permiten la configuración y despliegue del backend mediante Docker. Además, existe un archivo .env en el directorio backend, que muy posiblemente almacena variables de entorno como credenciales y configuraciones de base de datos.

Name	Last commit message	Last commit date
..		
database	Implement connection pooling for PostgreSQL database	2 weeks ago
models	Edit Option(b+f)	7 hours ago
routes	Edit Option(b+f)	7 hours ago
utils	UpdateRoutes	4 days ago
app.py	HealthForBalancing	4 days ago
docker-compose.yaml	Add env_file configuration to docker-compose for environment variables	2 weeks ago
dockerfile	Remove config.py creation from Dockerfile	2 weeks ago
requirements.txt	more updates before load balancing	5 days ago

Dentro de la carpeta backend/src/models/ se encuentran los modelos que definen la estructura de los datos en la base de datos. Estos modelos están organizados en dos subdirectorios principales. Por un lado, models/ contiene archivos como CarroModel.py, ReviewModel.py y UsuarioModel.py, que parecen ser los encargados de gestionar la interacción con la base de datos.

```python
@classmethod
def get_carro(cls, id):
    """ Obtiene un carro por su ID. """
    if not isinstance(id, int) or id <= 0:
        return {"error": "ID inválido"}, 400

    try:
        connection = get_connection()
        with connection.cursor() as cursor:
            cursor.execute("SELECT id, usuario_id, location, model, price, year, km, image_url FROM public.carros WHERE id=%s", (id,))
            carro = cursor.fetchone()

            if carro:
                return Carro(*carro).to_JSON(), 200
            return {"message": "Carro no encontrado"}, 404
    except Exception as e:
        print(f"Error en get_carro: {e}")
        return {"error": "Error al obtener el carro"}, 500
    finally:
        release_connection(connection)

@classmethod
def add_carro(cls, carro):
    """ Agrega un nuevo carro a la base de datos. """
    try:
        connection = get_connection()
        with connection.cursor() as cursor:
            if not isinstance(carro.price, (int, float)) or carro.price <= 0:
                return {"error": "El precio debe ser un número positivo"}, 400
            if not isinstance(carro.year, int) or carro.year < 1900 or carro.year > 2050:
                return {"error": "Año inválido"}, 400

            # Sanitizar los campos de texto
            carro.model = bleach.clean(carro.model)
            carro.location = bleach.clean(carro.location)
```

Por otro lado, models/entities/ alberga archivos como Carro.py, Review.py y Usuario.py, que representan las entidades del sistema. La separación entre modelos y entidades expone que los primeros funcionan como una capa de abstracción que permite manipular los datos dentro del backend, mientras que las entidades representan la estructura de los datos en la base de datos.

```python
class Review:

    def __init__(self, id=None, usuario_id=None, carro_id=None, rating=None, comment=None, review_date=None):
        self.id = id
        self.usuario_id = usuario_id
        self.carro_id = carro_id
        self.rating = rating
        self.comment = comment
        self.review_date = review_date

    def to_JSON(self):
        return {
            'id': self.id,
            'usuario_id': self.usuario_id,
            'carro_id': self.carro_id,
            'rating': self.rating,
            'comment': self.comment,
            'review_date': str(self.review_date) if self.review_date else None
        }
```

El manejo de rutas se encuentra en backend/src/routes/, donde se organizan los diferentes endpoints de la API. Aquí se encuentran archivos como Carro.py, Review.py y Usuario.py, que definen las rutas específicas para gestionar cada una de estas entidades. Esto sugiere que el backend sigue una estructura modular donde cada entidad cuenta con sus propias rutas y lógica de negocio.

```python
from flask import Blueprint, request, jsonify
from models.ReviewModel import ReviewModel
from models.entities.Review import Review
from utils.jwt_manager import require_jwt

review_bp = Blueprint('review_bp', __name__)

@review_bp.route('/', methods=['GET'])
def get_reviews():
    return jsonify(ReviewModel.get_reviews())

@review_bp.route('/<int:id>', methods=['GET'])
def get_review(id):
    return jsonify(ReviewModel.get_review(id))

@review_bp.route('/add', methods=['POST'])
@require_jwt
def add_review(user_id):
    data = request.json
    nuevo_review = Review(**data)
    return jsonify(ReviewModel.add_review(nuevo_review, user_id))

@review_bp.route('/update/<int:id>', methods=['PUT'])
@require_jwt
def update_review(id, user_id):
    data = request.json
    review_actualizado = Review(id=id, **data)
    return jsonify(ReviewModel.update_review(review_actualizado))

@review_bp.route('/delete/<int:id>', methods=['DELETE'])
@require_jwt
def delete_review(id, user_id):
    return jsonify(ReviewModel.delete_review(id))
```

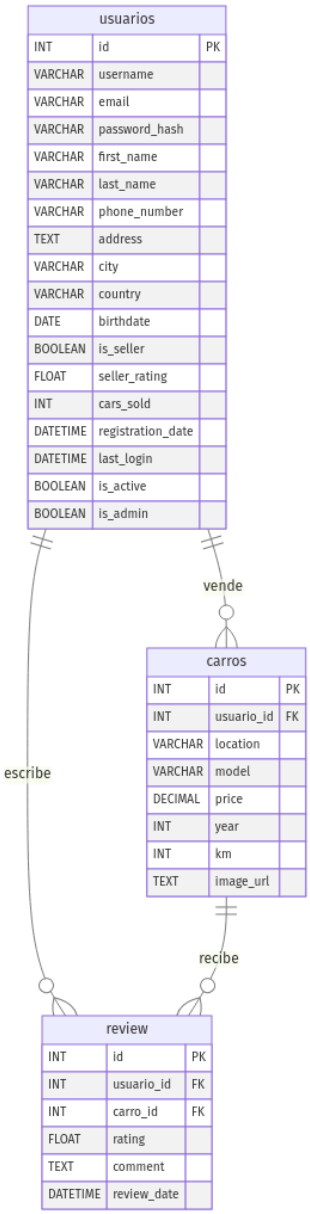
La base de datos se configura en backend/src/database/, donde el archivo db.py maneja la conexión y configuración de la base de datos. Este archivo es esencial para garantizar la persistencia de los datos y la integración del backend con el sistema de almacenamiento.

El backend también cuenta con un conjunto de utilidades dentro de backend/src/utils/. En esta

carpeta se encuentran archivos como gcs_helper.py, que maneja interacciones con Google Cloud Storage, jwt_manager.py, que gestiona la autenticación y la seguridad mediante tokens JWT, y security.py, que podría contiene funciones relacionadas con la seguridad, como el cifrado de contraseñas o validaciones de acceso.

Esquema de la base de datos

Los esquemas de las tablas bases de datos se presentan a continuación:



Decisiones de Seguridad

- **Acceso a la Base de Datos:** Solo permitido desde las IPs de las instancias backend.
- **Autenticación y Autorización:** Uso de **JWT tokens** para inicios de sesión.
    - Garantiza que los usuarios solo puedan editar y eliminar sus propias publicaciones.
    - Asigna automáticamente usuarios a nuevas publicaciones y reseñas.
- **Gestón de Permisos:**
    - Se utilizó **IAM** para crear cuentas de servicio para el manejo de Storage.
    - No se pudo implementar IAM en Cloud SQL, quedó solo en Storage.

Instrucciones de despliegue:

BackEnd:

Para desplegar el backend del proyecto **SegCloud** en un entorno Linux utilizando contenedores de Docker, es necesario seguir una serie de pasos que aseguren una instalación y ejecución eficiente. En primer lugar, se debe verificar que Docker y Docker Compose están correctamente instalados en el sistema. Para ello, se pueden ejecutar los comandos docker --version y docker-compose --version, los cuales deberían devolver información sobre las versiones instaladas. Si Docker no está instalado, se puede instalar ejecutando sudo apt update && sudo apt install docker.io -y en distribuciones basadas en Debian, como la utilizada en las maquinas.

Una vez instalado Docker, se recomienda agregar el usuario actual al grupo de Docker para poder ejecutar comandos sin necesidad de sudo. Esto se puede hacer con sudo usermod -aG docker $USER, tras lo cual se debe cerrar la sesión y volver a iniciarla para que los cambios surtan efecto. Luego, se debe clonar el repositorio del proyecto si aún no se tiene en el sistema, lo cual se hace con git clone https://github.com/Nathaliaq16/SegCloud.git && cd SegCloud/backend/src/. Es importante asegurarse de estar en el directorio correcto antes de proceder con el despliegue.

Antes de levantar los contenedores, se debe configurar el archivo .env con las variables de entorno necesarias para la ejecución del backend. Si el archivo no existe, se puede crear manualmente con nano .env y añadir las variables necesarias, como credenciales de base de datos, claves de API y configuraciones específicas del entorno. Em este caso, es importante tener el archivo json para el acceso con la cuenta de servicio para SQL Storage.

Con todo listo, se procede a construir y ejecutar los contenedores con el comando docker-compose up --build -d. Esto compilará la imagen del backend y ejecutará todos los servicios definidos en docker-compose.yaml, asegurando que el backend funcione correctamente dentro de los contenedores. Para verificar que los contenedores están en ejecución, se puede usar docker ps, lo que mostrará que el contenedor corre en el puerto 5000.

Finalmente, es importante mencionar que no hay que descargar ningún paquete de requirements, ya que los archivos de Docker ya lo hacen por defecto.

FrontEnd:

Se encuentran en el README de la carpeta correspondiente, pero se enuncian nuevamente a continuación: para desplegar un backend desarrollado en Node.js de manera tradicional, primero se debe asegurar que Node.js y npm están instalados en el sistema, lo cual se puede verificar ejecutando node -v y npm -v. Luego, se debe navegar hasta el directorio del proyecto con cd y ejecutar npm install para instalar todas las dependencias especificadas en package.json. Una vez completada la instalación, el frontend se puede iniciar con npm start si el script de inicio está definido en package.json,