

# Entrega 3 - Arquitectura, conclusiones y consideraciones

Javier Alejandro Gómez, Alejandra Guerrero, Diego Alejandro Peña, Juan Ignacio Arbeláez

{ja.gomez1003, a.guerrero10, da.pena20, ji.arbelaez}@uniandes.edu.co

Desarrollo de soluciones cloud - MINE

Universidad de los Andes, Bogotá, Colombia

Fecha de presentación: abril 4 de 2023

## Tabla de contenido

1. Introducción
2. Arquitectura planteada
3. Instrucciones para la ejecución del entorno
4. Consideraciones para lograr la escalabilidad y conclusiones

### 1. Introducción

En el presente documento se expone la arquitectura implementada para generar una aplicación web en Flask, la cual permita la compresión de archivos por medio de peticiones HTTP y utilizando un enfoque de procesamiento asíncrono. También se presentan las consideraciones que se tuvieron en cuenta en el desarrollo de este proyecto, teniendo en cuenta que con esta aplicación se busca a futuro aplicar los conceptos de escalabilidad, una de las características principales de los ambientes en nube.

El código y la documentación de este proyecto se pueden consultar en los siguientes enlaces:

- Repositorio GitHub: <https://github.com/aguerrero10/Desarrollo-cloud-grupal>
- Documentación de Postman: <https://documenter.getpostman.com/view/13843294/2s93CPrY2y>
- Video de sustentación: [Link del video](#)

El proyecto se implementó ejecutando las siguientes actividades que se identificaron en la fase de entendimiento del problema planteado:

1. Comprensión del problema descrito en el enunciado del proyecto
2. Preparación del servidor web: VM en GCP, Python, librerías, paths, entorno virtual, entre otros.
3. Preparación del servidor de base de datos Cloud SQL y del motor PostgreSQL
4. Desarrollo en Flask de los servicios que permiten crear un usuario e iniciar sesión
5. Desarrollo en Flask de los servicios que permiten crear una tarea de compresión, consultarla, eliminarla y descargar su archivo original y comprimido
6. Creación y documentación de los escenarios de prueba en Postman
7. Construcción de la lógica de compresión de un archivo
8. Construcción del proceso asíncrono usando Celery y Redis
9. Desarrollo de la capa de presentación usando HTML, CSS y Javascript
10. Pruebas y revisión del funcionamiento global de la solución
11. Documentación de la arquitectura implementada

## 2. Arquitectura planteada

En este trabajo, se ha desarrollado una arquitectura robusta y escalable para la gestión de datos masivos que utiliza los siguientes componentes: base de datos PostgreSQL usando el servicio Cloud SQL de GCP, tres servidores o máquinas virtuales de Google Compute Engine para la ejecución del worker, file server y web server y servidor web creado mediante Flask .

El servicio Cloud SQL permite tener una base de datos PostgreSQL disponible y en una máquina diferente a donde se encuentra la aplicación Flask, lo que permite tener un mejor desempeño ante un alto número de solicitudes.

Flask, por su parte, fue utilizado para desarrollar la aplicación web y proporcionar una interfaz amigable y accesible para los usuarios. La arquitectura de Flask permite una fácil integración con otros componentes de la aplicación, provee un patrón de desarrollo MVC (Modelo Vista Controlador), lo que hizo posible una gestión eficiente de la presentación de los datos.

Por último, se utilizaron servicios REST con JSON para integrar la capa de datos y permitir una comunicación flexible y eficiente entre los componentes de la aplicación. Con esto se separa la aplicación web en el servidor presentación, del componente de data y procesamiento que posiblemente se pueda ir a una infraestructura distribuida que en futuros ejercicios sea utilizada.

De acuerdo con lo anterior, en la siguiente imagen se presenta el diagrama de despliegue de la aplicación web creada para la compresión de archivos con procesamiento asíncrono.

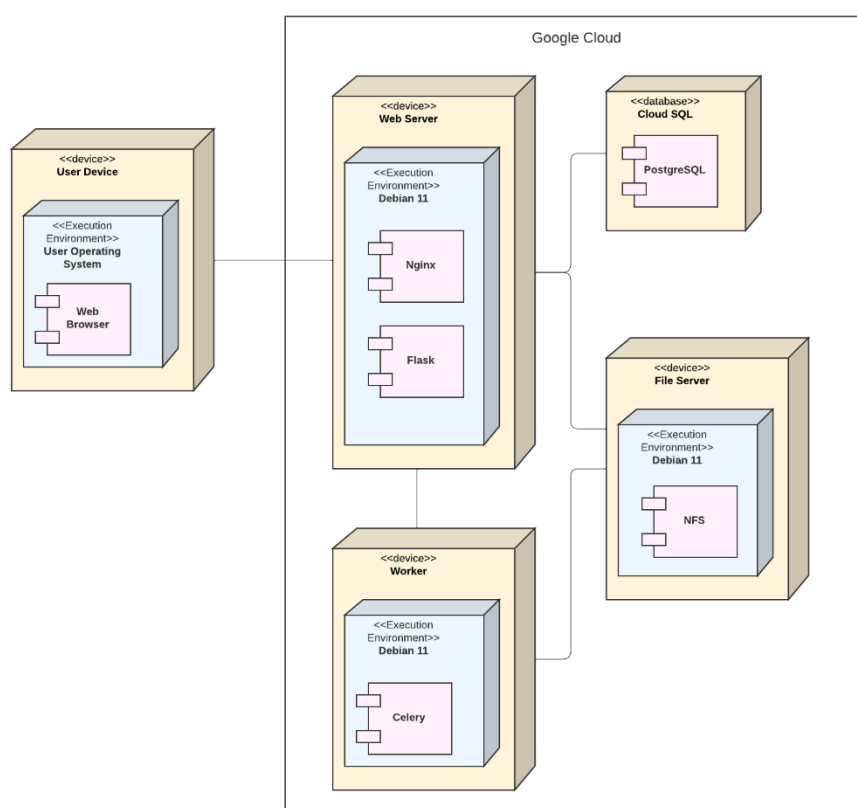
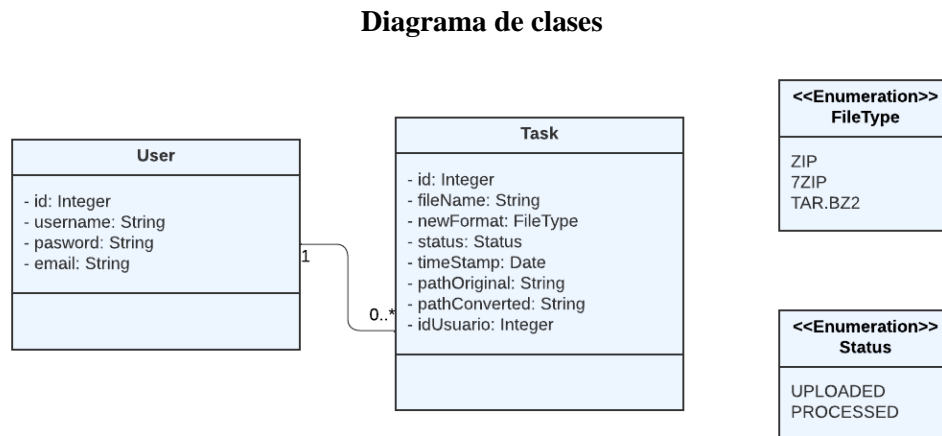


Figura 1. Diagrama de despliegue

Como se evidencia en el diagrama de despliegue, la aplicación utiliza Cloud SQL de GCP como servidor de base de datos y tres servidores para su funcionamiento: worker, file server y web server que expone los servicios requeridos y contiene la lógica de negocio, los cuales se implementaron usando el

framework Flask, una cola de tareas Redis y un broker Celery. El cliente accede a los servicios de la aplicación por medio de peticiones HTTP.

Por otro lado, la lógica de negocio (backend) y la base de datos siguen el modelo de datos que se muestra en el siguiente diagrama:



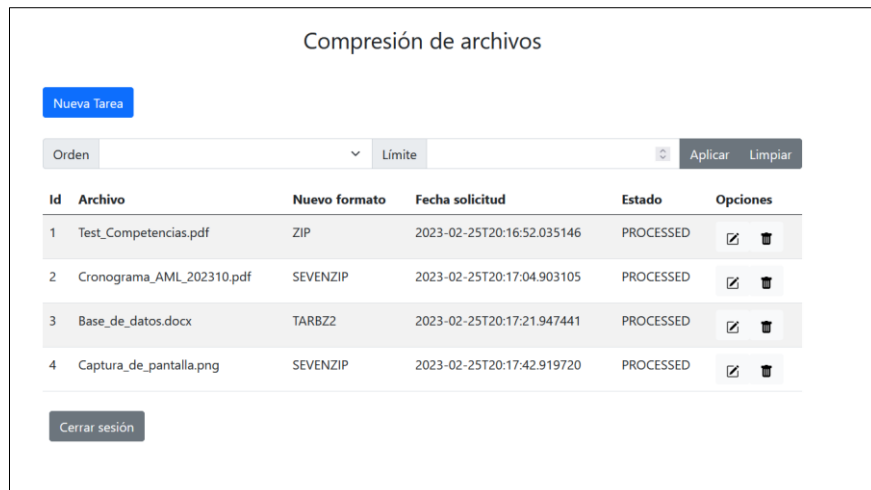
Es decir, el modelo de datos se compone de la entidad Usuario la cual se compone de sus datos básicos y puede o no tener múltiples tareas de compresión de archivos. Por otra parte, la entidad tarea tiene los datos del archivo a comprimir, su estado y tipo de archivo de compresión, modeladas con una enumeración.

Respecto a la capa de presentación, se implementó una interfaz de usuario con dos pantallas como se muestra a continuación.

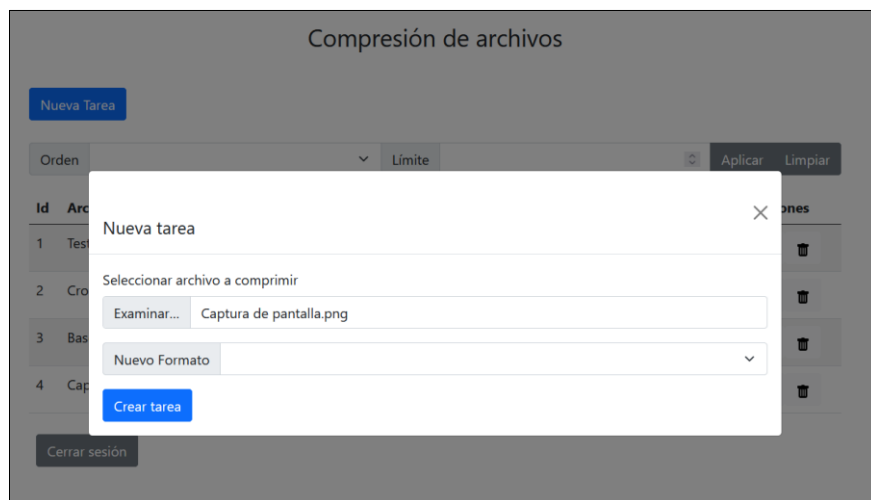
#### **Pantalla 1:** permite el registro del usuario e inicio de sesión

The screenshot displays the 'Compresión de archivos' (File Compression) interface. It features two input fields for 'Usuario' and 'Contraseña' (Password), followed by 'Iniciar sesión' (Login) and 'Registrarse' (Register) buttons. An arrow points from the 'Registrarse' button to a modal window titled 'Nuevo usuario' (New user). This modal contains fields for 'Nombre' (Name), 'Email', 'Contraseña' (Password), and 'Confirmar contraseña' (Confirm password), along with a 'Registrarse' button.

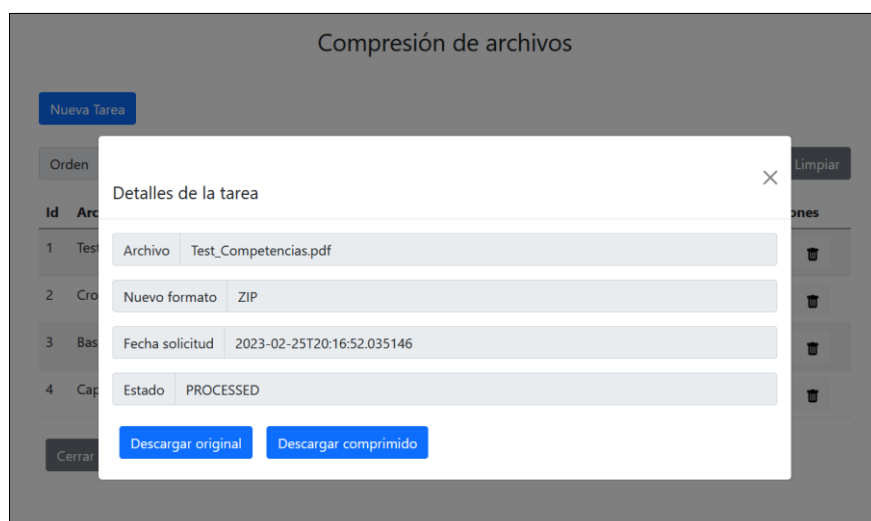
#### **Pantalla 2.1:** permite consultar las tareas de compresión de archivos o eliminarlos



**Pantalla 2.2:** también permite crear tareas de compresión de archivos



**Pantalla 2.3:** y permite la descarga del archivo original y comprimido



### 3. Instrucciones para la ejecución del entorno

Para iniciar la ejecución del entorno de la aplicación es necesario primero tener instalado Python en la máquina, es posible instalar Python usando Bash en la máquina con `sudo apt-get install python3.6`. A continuación, se debe instalar un ambiente virtual en el directorio base, en el cual se encuentra el archivo de requerimientos `requirements.txt`. Después de activar el ambiente virtual con `source {Nombre ambiente virtual}/bin/activate`, se deben instalar todas las librerías que usará la aplicación web, estas librerías y su versión se encuentran en `requirements.txt` por lo que al usar `pip install -m requirements.txt` se instalan todas las librerías necesarias en la aplicación web. Esta es la primera consola que debe estar activada, en la que se ejecuta la aplicación de Flask

Se activa una nueva ventana de la consola para correr comandos bash y esta ventana se usará para correr la instancia de Redis, esta instancia se corre con `redis-server -port 6360`. Este comando se utiliza para crear la instancia de Redis en el puerto 6360. En el archivo de `tasks.py` se define el puerto 6360 como el puerto en el que va a estar el broker de la aplicación de Celery

En este punto ya tenemos la instancia de Redis que se encarga de la cola de tareas y que se conecta con la aplicación de Celery en el puerto 6360 y necesitamos iniciar la aplicación de Flask que se encuentra que podemos activar en la primera terminal, en la cual tenemos el ambiente virtual activado. Debemos estar en la carpeta `“./backend/flaskr”` y usar el comando `Flask run` para iniciar la aplicación de Flask. La aplicación de Flask inicia la instancia de la aplicación de Celery cuando el `worker` de Celery sea inicializado.

Ahora bien, abrimos una nueva ventana de la consola para iniciar la aplicación de Celery, para su ejecución debemos estar ubicados dentro del directorio `“./backend/flaskr”` y tener el ambiente virtual instalado previamente activado. Para iniciar la aplicación de Celery usamos el comando `celery -A tasks worker -l info`. Este comando crea un `worker` de la aplicación de Celery definido en el archivo de `tasks`. En este punto todo el backend de la aplicación web está funcionando dentro de la máquina. La base de datos está configurada a partir del servicio Cloud SQL, por lo que no es necesario configurar nada de la base de datos.

Por otra parte, la instancia del frontend de la aplicación web se creó con la clase `VistaFrontEnd`, en este caso se usa el servidor web Nginx, por lo que debemos tener instalado en la máquina en la que vamos a correr la aplicación web. Una vez Nginx este correctamente instalado se debe configurar que la aplicación de Flask, que en este caso contiene el frontend se ejecute con Nginx. Para realizar esto se crea un archivo de configuración en el directorio `‘/etc/nginx/sites-available’` y se modifica el contenido, con el comando `sudo nano configuracion.conf`. El contenido del archivo recién creado debe ser:

```
server {
    listen 80;
    server_name example.com;
    location / {
        proxy_pass http://localhost:5000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
    location /static{
        /Desarrollo-cloud-grupal/frontend/;
    }
}
```

Una vez se haya creado este archivo es necesario crear un enlace simbólico usando el comando `sudo ln -s /etc/nginx/sites-available/myapp.conf /etc/nginx/sites-enabled/` y por último reiniciar el Nginx con `sudo systemctl restart nginx`.

#### **4. Consideraciones para lograr la escalabilidad y conclusiones**

Esta aplicación web es una aplicación distribuida, se desarrolló con una arquitectura que separa los componentes de lógica, almacenamiento y persistencia para la segregación de tareas y facilitar la escalabilidad ante escenarios de un aumento en la cantidad de usuarios y solicitudes generadas. El servidor web se base en microservicios con lo cual es posible escalar la aplicación web con mayor facilidad puesto que es posible replicar el servicio que no está cumpliendo con los requerimientos esperados de tiempo de espera para el tráfico actual de solicitudes.

Para esta fase del proyecto, la aplicación se migró a la nube de Google Cloud usando servicios como Compute Engine y Cloud SQL sin aplicar una configuración que permita la escalabilidad automática de la aplicación web. Durante el desarrollo de esta fase se evidenció una mejora al separar los componentes, sin embargo, al separar los componentes se tuvieron inconvenientes en la configuración de conexiones, firewalls y de permisos al utilizar múltiples usuarios para la configuración de los servidores.

Para permitir la escalabilidad automática de la aplicación será importante la configuración de un balanceador de carga que se encargue de medir y lanzar nuevas instancias cuando se requiera de mayor capacidad de procesamiento. Adicionalmente, como las instancias se habilitan de forma automática, es necesario mejorar la automatización del arranque y configuración de las máquinas a usar.