Facultad de Ingeniería
Departamento de Ingeniería de Sistemas y Computación
Maestría en Ingeniería de Sistemas y Computación
ISIS 4426 – Desarrollo de Soluciones Cloud

Estudiante	Código	Correo
Juan Camilo Sanguino Pérez	201617597	jc.sanguino10@uniandes.edu.co
Julian Camilo Mora Valbuena	202012747	j.morav@uniandes.edu.co
Julian Camilo Garcia Escobar	201729543	j.garcia@uniandes.edu.co
María Alejandra Vargas Torres	201123148	ma.vargas73@uniandes.edu.co

Proyecto - Entrega 1

Contenido

1. Arquitectura

1.1. Descripción del proyecto

Frontend:
Este es el servicio que maneja la interfaz de usuario.
Tecnología utilizada: Node JS [v. 18] - React.
Está construido a partir del directorio ./front_cloud y se expone en el puerto 3000.
Se comunica con el backend a través de una API Rest

Backend:
Este servicio maneja la lógica de negocio de la aplicación.
Tecnología utilizada: Python [v. 3.11] – Fast API.
Utiliza Uvicorn como servidor web.
Está construido a partir del directorio ./back_cloud y se expone en el puerto 8000.
Este servicio depende de los servicios de redis y db.
Se comunica con nuestra base de datos PostgreSQL

Redis: Este es el servicio de mensajería que actúa como intermediario.

Celery Worker: Este es el servicio que maneja la cola de tareas distribuidas. Está construido a partir del directorio ./back_cloud y depende de los servicios redis y backend.
Flower: Este es un servicio para monitorear el estado de las tareas en Celery. Está construido a partir del directorio ./back_cloud y se expone en el puerto 5555. Este servicio depende de los servicios backend, redis y celery_worker.

DB:
Este es el servicio de base de datos de la aplicación.
Motor de base de datos utilizado: PostgreSQL
Expuesto en el puerto 5432.

1.2. Diagrama de componentes

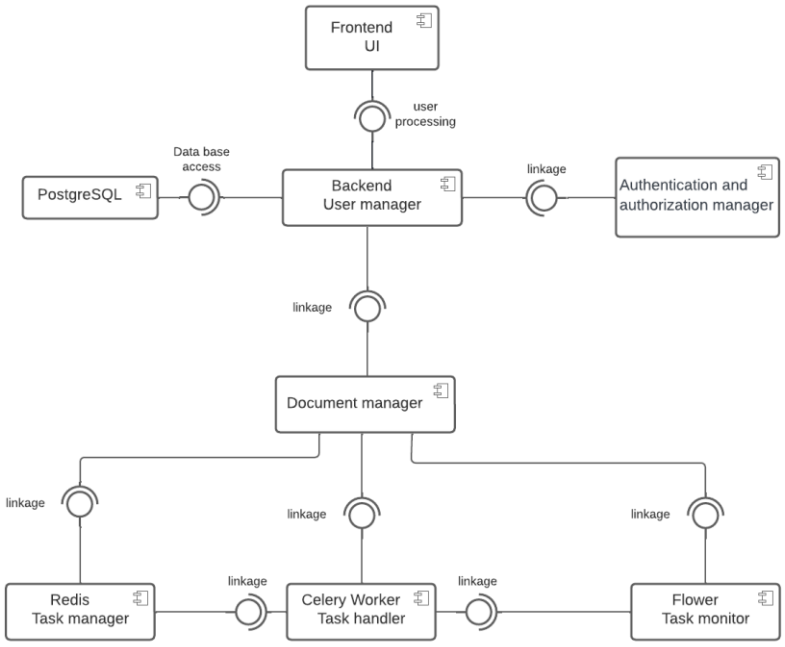

Figura 1. Diagrama de componentes

1.3. Diagrama de Despliegue

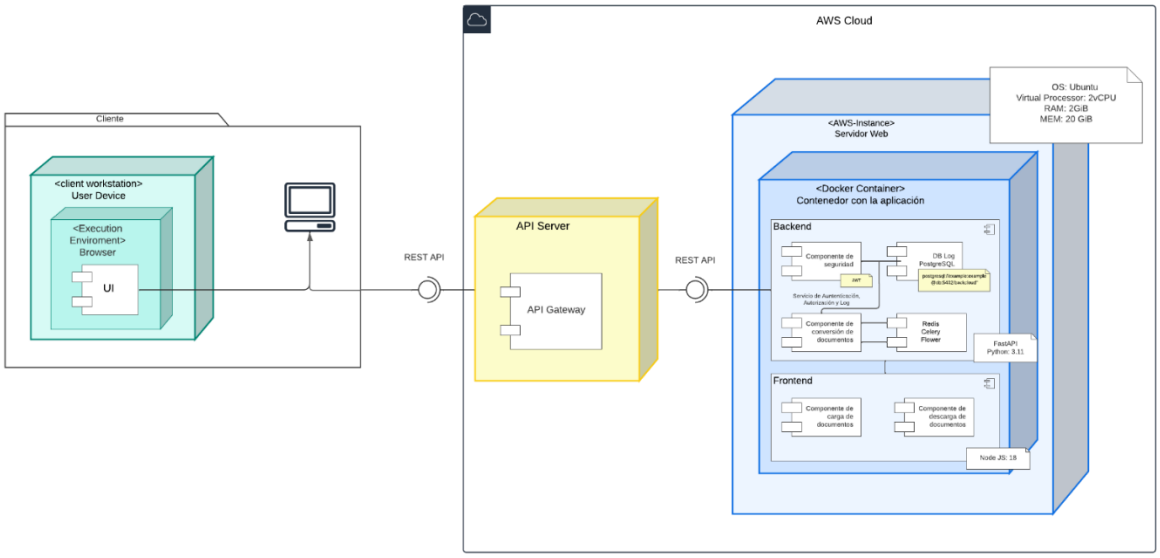

Figura 2. Diagrama de Despliegue

2. Conclusiones

- La implementación de este proyecto con Docker y Celery fue exitosa al garantizar una comunicación eficiente entre contenedores. La colocación de Celery y el backend en la misma red ha sido crucial para asegurar un funcionamiento estable y escalable del sistema. También se tuvo como desafío en comprender la necesidad de la comunicación entre contenedores.
- La integración de Celery en el backend para ejecutar la tarea de conversión de archivos periódicamente ha mejorado la eficiencia y escalabilidad del sistema, permitiendo automatizar procesos repetitivos y garantizar la disponibilidad constante de los servicios de conversión.

3. Consideraciones

3.1. ¿Qué se puede tener en cuenta para que la aplicación se pueda escalar para soportar cientos de usuarios finales de manera concurrente?

Una de las estrategias que se podrían tener en cuenta es considerar escalar nuestro backend y frontend de manera horizontal. De momento, nuestra aplicación solo utiliza un contenedor backend y frontend. Por lo cual, se podría aumentar el número de contenedores dentro del docker-compose para distribuir la carga de trabajo y mejorar el rendimiento.

Siguiendo en este orden de idas, si se opta por aumentar el número de contenedores del backend y frontend, también se debería agregar un balanceador de carga que ayudaría a distribuir el tráfico entre cada uno de los componentes, para que puedan cumplir con el objetico de nivelar la carga de trabajo.

3.2. ¿Cómo podemos determinar el número adecuado de contenedores a utilizar u otras estrategias de escalamiento?

Para determinar la manera más adecuada de escalar la aplicación se planea realizar las pruebas de carga con JMeter, con el objetivo de identificar el estado actual de la aplicación y cómo varia su rendimiento dependiendo de la cantidad de usuarios concurrentes.

Por lo tanto, se condiera pertinente iniciar con pruebas de carga, que contengan un número pequeño de usuarios, sobre la aplicación como se tiene al momento de esta entrega. Estas pruebas se realizarán en varios escenarios de uso de la aplicación. Entre estos: (i.) Login, (ii.) SignIn, (iii.) Creación de tareas, (iv.) Procesamiento de archivos y (v.) descarga de archivos.

A) <u>Objetivos de las pruebas de carga:</u>

- **General:** Medir el estado actual de la aplicación y cómo se comporta al aumentar el número de usuarios concurrentes.
- **Rendimiento:**
  - Medir el tiempo de respuesta de las operaciones en los escenarios.
  - Evaluar el impacto de aumentar la carga en el rendimiento del sistema.
  - Operaciones críticas: ¿Qué operaciones consumen mayor tiempo o presentan fallos inesperados?
- **Funcionalidad:**
  - Capacidad del tema para manejar la carga sin errores.
  - Identificar cuellos de botella.

<u>Carga objetivo de las pruebas (estado actual de la aplicación):</u>

- 10 usuarios
- 20 usuarios
- 50 usuarios
- 100 usuarios

Después de realizar un análisis del estado actual de la aplicación, se planea escalar la aplicación con estrategias como aumentar el número de contenedores del backend y frontend y un balanceador de carga, como se mencionó anteriormente. Cabe aclarar que las decisiones para escalar la aplicación pueden variar dependiendo de los resultados que se obtengan sobre las primeras pruebas de carga. De notar que la carga en las tareas del backend y frontend se maneja de manera satisfactoria con un solo contenedor, se tiene un mapeo de otras estrategias que se podrían realizar, entre ellas se encuentra un escalamiento vertical en el servicio PostgreSQL para la base de datos. Con esto presente, las siguientes pruebas de carga deberán seguir la siguiente estructura para poder evaluar el desempeño de nuestra aplicación bajo la demanda concurrente de cientos de usuarios.

B) <u>Objetivos de las pruebas de carga:</u>

- **General:** Medir cómo se comporta la aplicación al escalarla teniendo en cuenta las estrategias mencionadas anteriormente (Aumento del número de contenedores o escalamiento vertical en el servicio de PostgreSQL).
- **Rendimiento:**
    - Medir el tiempo de respuesta de las operaciones en los escenarios
    - Evaluar el impacto de aumentar la carga en el rendimiento del sistema
    - Operaciones críticas: ¿Qué operaciones consumen mayor tiempo o presentan fallos inesperados?
- **Funcionalidad:**
    - Capacidad del tema para manejar la carga sin errores
    - Identificar cuellos de botella

Carga objetivo de las pruebas (estado actual de la aplicación):

- 50 usuarios
- 100 usuarios
- 200 usuarios
- 500 usuarios

La última prueba con 500 usuarios depende completamente del desempeño que tenga la aplicación con 200 usuarios concurrentes. Si después de analizar los resultados de las pruebas con 200 y 500 usuarios se considera que se puede aumentar el número de usuarios en la aplicación, por lo cual, sería prudente hacer una prueba con más usuarios para tratar de llevar la aplicación al límite (Número posible de usuarios para esta prueba: 1000).

3.3. Posibles limitaciones

Una de las limitaciones que se tienen en este proyecto es que la infraestructura requerida para el despliegue especifica una máquina virtual con las siguientes características:2 vCPU, 2 GiB RAM, 20 GiB en almacenamiento. Al revisar con detenimiento los valores de CPU y RAM, es evidente que pueden representar una restricción al momento de soportar cientos de usuarios concurrentes. Por ende, si en futuras iteraciones del proyecto es necesario seguir trabajando con las mismas especificaciones técnicas, será necesario contemplar soluciones de escalamiento que potencien al máximo los recursos de la máquina virtual. Estas decisiones deben estar apoyadas en el análisis de las pruebas de carga descritas anteriormente en el documento.

Finalmente, algo para tener en cuenta en las siguientes iteraciones del proyecto es que el proyecto es compatible con las versiones de Docker Compose posteriores a la 2.