INFORME

(Grupo A)

(Proyecto)

Juan Manuel Hoyos Contreras - 202380796
Juan Esteban López Aránzazu - 202313026
Víctor Manuel Alzate Morales - 202313022
Sebastian Cifuentes Florez - 202380764

Asignatura:

Infraestructuras Paralelas y Distribuidas

Docente:

Carlos Andres Delgado Saavedra

Ingeniería de Sistemas

Universidad del Valle – Sede Tuluá

26 de diciembre de 2024

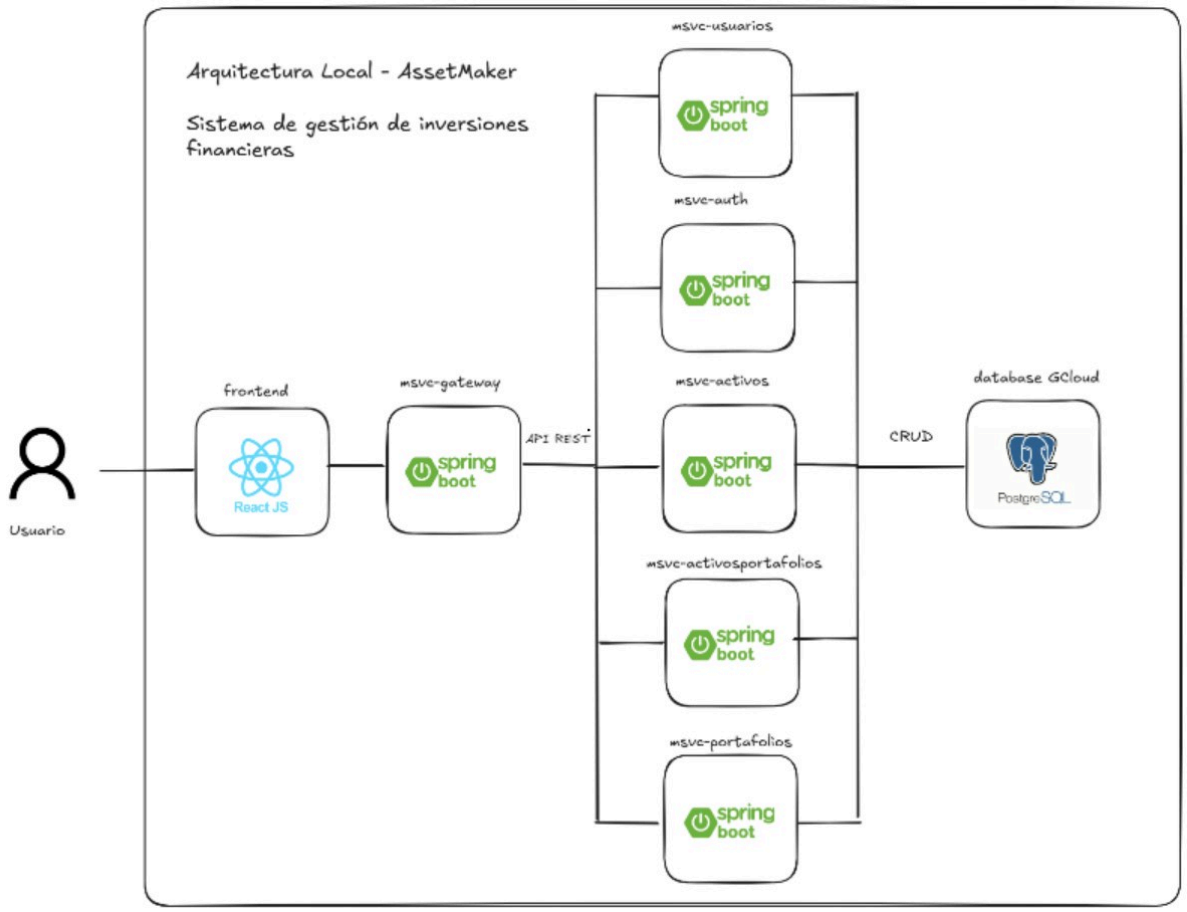
Tabla de Contenidos

Introducción

AssetMaker es una aplicación diseñada para gestionar portafolios de inversiones de manera eficiente y organizada. Con ella, puedes guardar y realizar un seguimiento detallado de los activos que has adquirido en otras plataformas, permitiéndote centralizar toda tu información en un solo lugar. Además, AssetMaker ofrece herramientas avanzadas para generar reportes personalizados, visualizar estadísticas clave y analizar el rendimiento de tus inversiones, brindándote una visión clara y completa de tu progreso financiero.

Las tecnologías utilizadas por el equipo de desarrollo para crear esta aplicación son:

- React en su versión **18.2.0** para el Front-End
- Java en su versión **22** para el Back-End
- PostgreSQL en su versión **16.6** como gestor de Base de datos



Objetivos

Objetivo General

Desarrollar una aplicación web que permita a los usuarios gestionar y centralizar sus inversiones mediante la creación y administración de portafolios, el registro de activos, la consulta de precios utilizando una API externa como Alpaca y la generación de reportes y estadísticas detalladas para optimizar la toma de decisiones financieras.

Objetivos Específicos

- Diseñar y estructurar un sistema eficiente que permita la creación de portafolios, el registro de activos y la integración de datos.
- Implementar una base de datos robusta que soporte la gestión de múltiples portafolios y activos por usuario.
- Incorporar funcionalidades avanzadas de reportes y estadísticas que permitan a los usuarios analizar el rendimiento de sus inversiones de forma clara y eficiente.

Solución local

La aplicación está diseñada bajo una arquitectura moderna basada en microservicios, con cada servicio implementado de manera independiente y empaquetado utilizando Docker. Tanto el backend como el frontend cuentan con su propio archivo **Dockerfile**, lo que facilita su despliegue y escalabilidad.

La base de datos de la aplicación está alojada en **Cloud SQL**, asegurando un almacenamiento centralizado, seguro y de alta disponibilidad. La comunicación entre el frontend y el backend se realiza mediante **API REST**, lo que garantiza una integración fluida y un manejo eficiente de las operaciones.

En el backend, cada microservicio está diseñado para ejecutar operaciones CRUD (crear, leer, actualizar, eliminar), permitiendo una gestión completa de los datos relacionados con los portafolios, activos y usuarios. Esto asegura una separación de responsabilidades y facilita la escalabilidad y el mantenimiento del sistema.

El frontend, por su parte, ofrece una interfaz intuitiva y amigable, que interactúa directamente con las APIs para presentar información sobre los portafolios y los activos, incluyendo precios y estadísticas.

Docker Compose

Dockerfile del Front-End

https://github.com/juanhcode/AssetMaker/blob/main/Dockerfile

Este archivo Dockerfile configura y construye una imagen Docker optimizada para una aplicación frontend desarrollada en React. Durante las pruebas iniciales, se evaluaron diferentes versiones de Node.js, como **node:21**, pero debido a problemas de rendimiento y compatibilidad por las dependencias de npm, se optó por la versión **node:20-bullseye**, que ofrece un equilibrio ideal entre ligereza y estabilidad. El Dockerfile sigue una estructura en dos fases: la fase de construcción utiliza una imagen base de Node.js para instalar dependencias y generar los archivos de producción con **npm run build**, mientras que la fase de ejecución utiliza Nginx para servir los archivos estáticos de la aplicación. Se incluye una configuración personalizada de Nginx para optimizar el servidor y asegurar un rendimiento eficiente. Finalmente, el puerto **4200** se expone para permitir el acceso a la aplicación desde cualquier navegador web local, lo que facilita su despliegue en entornos de producción.

Dockerfile del Back-End de los microservicios

https://github.com/juanhcode/AssetMaker-Microservices/blob/main/msvc-activos/Dockerfile

Este archivo Dockerfile configura y construye una imagen Docker para un microservicio desarrollado en Java, utilizando OpenJDK como base. Todos los microservicios comparten una estructura similar, diferenciándose únicamente en el puerto expuesto y el nombre del archivo .jar correspondiente al servicio. En el caso del microservicio activos, la imagen utiliza openjdk:23-ea-17-jdk para garantizar compatibilidad con las últimas versiones de Java. El archivo .jar del microservicio se copia al directorio de trabajo /app, y se configura el puerto 9003 como el punto de acceso. La ejecución del microservicio se realiza con el comando java -jar, asegurando que el contenedor inicie correctamente el servicio al ser desplegado. Esta estructura facilita la estandarización y el despliegue consistente de múltiples microservicios en el entorno.

Docker-compose

https://github.com/juanhcode/AssetMaker-Microservices/blob/main/docker-compose.yml

Este archivo **docker-compose.yml** configura múltiples servicios, cada uno representando un componente de la aplicación AssetMaker, incluyendo microservicios backend, un gateway, y el frontend. Utiliza la versión 3.8 de Docker Compose y define una red compartida llamada my-network para permitir la comunicación interna entre los servicios.

Microservicios Backend

1. **msvc-usuarios:**
    - Imagen: **juanhoyos/assetmaker:users-1.0.5.**
    - Contenedor: **msvc-usuarios**
    - Exposición del puerto **9001** dentro del contenedor hacia **8080** en el host.
    - Conectado a la red my-network.
2. **msvc-auth:**
    - Imagen: **juanhoyos/assetmaker:auth-1.0.5.**
    - Contenedor: **msvc-auth**
    - Exposición del puerto **9000** dentro del contenedor hacia **8081** en el host.
    - Incluye una variable de entorno FEIGN_CLIENT_URL que conecta este servicio al microservicio de usuarios (msvc-usuarios).
    - Depende de msvc-usuarios, asegurando que este último se inicie primero.
3. **msvc-portafolios:**
    - Imagen: **juanhoyos/assetmaker:portafolios-1.0.0.**
    - Contenedor: **msvc-portafolios**
    - Exposición del puerto **9002** dentro del contenedor hacia **8083** en el host.
    - Incluye una variable de entorno FEIGN_CLIENT_URL que conecta este servicio al microservicio de (msvc-usuarios).
    - Depende de msvc-usuarios.
4. **msvc-activos:**
    - Imagen: **juanhoyos/assetmaker:activos-1.0.0.**
    - Contenedor: **msvc-activos.**
    - Exposición del puerto **9003** dentro del contenedor hacia **8084** en el host.
    - Depende de msvc-usuarios.
5. **msvc-activosportafolios:**
    - Imagen: **juanhoyos/assetmaker:activosportafolios-1.0.0.**
    - Contenedor: **msvc-activosportafolios.**
    - Exposición del puerto **9004** dentro del contenedor hacia **8085** en el host.

    - Incluye dos variables de entorno para conectarse a msvc-portafolios y msvc-activos a través de sus URLs internas.
    - Depende de msvc-portafolios y msvc-activos.

Gateway

- **msvc-gateway:**
    - Imagen: **juanhoyos/assetmaker:gateway-1.0.0.**
    - Contenedor: **msvc-gateway.**
    - Exposición del puerto **8080** dentro del contenedor hacia **8082** en el host.
    - Actúa como punto central para enrutar solicitudes hacia los microservicios.
    - Depende de todos los microservicios backend, asegurando que estos se inicien antes del gateway.

Frontend

- **react-frontend:**
    - Imagen: **juanhoyos/assetmaker:react-frontend.**
    - Contenedor: **react-frontend.**
    - Exposición del puerto **4200**, permitiendo que el frontend sea accesible desde el host.
    - Depende del msvc-gateway, asegurando que el gateway esté operativo antes del inicio del frontend.

Red

- **my-network:**
    - Red de tipo **bridge** que permite la comunicación entre los contenedores, garantizando un entorno aislado y eficiente para la aplicación.

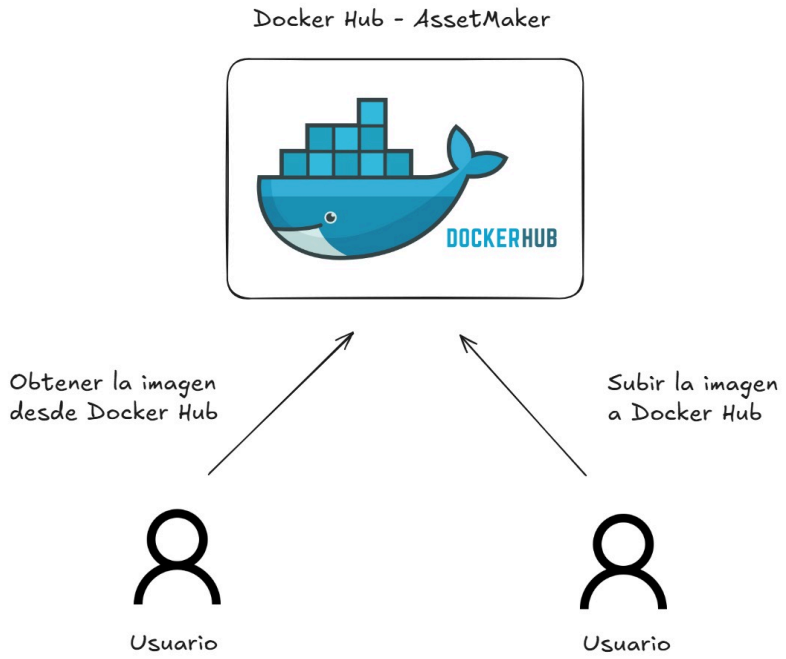
Este archivo proporciona una solución integral para desplegar toda la arquitectura de la aplicación, asegurando que cada componente esté correctamente configurado, se comunique eficazmente y se inicie en el orden adecuado.

Docker-compose con repositorio del docker hub

https://hub.docker.com/repository/docker/juanhoyos/assetmaker/general

En un entorno de Docker Compose con imágenes alojadas en Docker Hub, cada microservicio se empaqueta como una imagen Docker y se publica en un repositorio centralizado como Docker Hub, garantizando su disponibilidad para cualquier usuario o entorno sin depender de imágenes locales. Al subir las imágenes al Docker Hub, se puede usar el versionado para mantener un control estricto de cada actualización (por ejemplo, **juanhoyos/assetmaker:users-1.0.5**). Además, para admitir diferentes arquitecturas como ARM (usada en Mac con chips Apple M1/M2) y AMD (común en Windows y Linux), las imágenes se construyen y etiquetan para ambas plataformas, usando herramientas como **docker buildx** para crear imágenes multiplataforma. Así, los usuarios pueden descargar y ejecutar automáticamente la versión correcta según su sistema operativo, asegurando compatibilidad y portabilidad del proyecto.

La siguiente imagen ilustra el proceso de cómo un usuario o desarrollador sube una imagen a Docker Hub utilizando el comando **docker push**, así como el proceso de cómo obtenerla con el comando **docker pull**. El flujo comienza cuando el desarrollador construye y etiqueta la imagen localmente, luego la sube al repositorio de Docker Hub para hacerla accesible a otros. Posteriormente, cualquier usuario o sistema puede obtener la imagen desde Docker Hub, asegurando que todos trabajen con la misma versión y configuración sin depender de imágenes locales.



Kubernetes

Para la generación de los manifiestos, utilizamos la herramienta Kompose, que convierte los archivos Docker Compose en manifiestos de Kubernetes, automatizando así el proceso de creación.

Continuando con Kubernetes, presentamos los siguientes manifiestos:

frontend-service.yaml

https://github.com/juanhcode/AssetMaker-Microservices/blob/main/frontend-service.yaml
El archivo frontend-service.yml define un Service de tipo LoadBalancer en Kubernetes para exponer el frontend de la aplicación. Utiliza la API v1 y tiene como nombre frontend. Este servicio selecciona los pods que tengan la etiqueta io.kompose.service: frontend para dirigir el tráfico hacia ellos. El servicio expone el puerto 4200, y tanto el puerto externo (port) como el puerto interno del contenedor (targetPort) son el mismo, 4200. Al ser de tipo LoadBalancer, Kubernetes configurará un balanceador de carga externo para distribuir el tráfico entrante hacia los pods del frontend.

Frontend-deployment.yaml

https://github.com/juanhcode/AssetMaker-Microservices/blob/main/frontend-deployment.yaml
El archivo frontend-deployment.yml es un manifiesto de Kubernetes que define un Deployment para desplegar una aplicación de frontend en un clúster. Utiliza la API apps/v1 y establece que el nombre del deployment será frontend. Se especifica que habrá 1 réplica del contenedor de la aplicación (es decir, un solo pod), y el selector garantiza que los pods correspondientes a este deployment se identificarán con la etiqueta io.kompose.service: frontend. El contenedor que se ejecutará es la imagen gcr.io/infra-odyssey-437506-a3/assetmaker/react-frontend-1.0.1, y este escucha en el puerto 4200 utilizando el protocolo TCP. Además, se configura una política de reinicio Always, lo que significa que Kubernetes reiniciará el pod automáticamente si falla.

msvc-activos-deployment.yaml
https://github.com/juanhcode/AssetMaker-Microservices/blob/main/msvc-activos-deployment.yaml
El archivo msvc-activos-deployment.yml define un Deployment en Kubernetes para desplegar un servicio llamado msvc-activos. Utiliza la API apps/v1 y especifica que se ejecutará 1 réplica del contenedor. El contenedor utiliza la imagen gcr.io/infra-odyssey-437506-a3/assetmaker:activos-1.0.0, y está configurado para escuchar en el puerto 9003 utilizando el protocolo TCP. El selector asegura que los

pods de este Deployment se identifiquen con la etiqueta io.kompose.service: msvc-activos. Se establece la política de reinicio como Always, lo que implica que Kubernetes reiniciará automáticamente el pod si falla.

msvc-activos-service.yaml

https://github.com/juanhcode/AssetMaker-Microservices/blob/main/msvc-activos-service.yaml

El archivo msvc-activos-service.yml define un Service en Kubernetes para exponer el servicio msvc-activos. Utiliza la API v1 y tiene como nombre msvc-activos. Este servicio selecciona los pods con la etiqueta io.kompose.service: msvc-activos. Expone el puerto 8084 y lo redirige al targetPort 9003, que es el puerto interno donde el contenedor del servicio está escuchando. No se especifica el tipo de servicio, por lo que se asumirá el valor predeterminado de tipo ClusterIP, lo que hace que el servicio sea accesible solo dentro del clúster.

msvc-activosportafolios-deployment.yaml

https://github.com/juanhcode/AssetMaker-Microservices/blob/main/msvc-activosportafolios-deployment.yaml

El archivo msvc-activosportafolios-deployment.yml define un Deployment en Kubernetes para el servicio msvc-activosportafolios. Utiliza la API apps/v1 y especifica que se ejecutarán 3 réplicas del contenedor, lo que garantiza alta disponibilidad del servicio. El contenedor utiliza la imagen gcr.io/infra-odyssey-437506-a3/assetmaker/juanhoyos/assetmaker:activosportafolios-1.0.0, y está configurado para escuchar en el puerto 9004 con el protocolo TCP. Además, el contenedor tiene dos variables de entorno (FEIGN_CLIENT_URL y FEIGN_CLIENT_URL_ASSETS), que indican las URLs de otros servicios (msvc-portafolios y msvc-activos). El selector asegura que los pods se identifiquen mediante la etiqueta io.kompose.service: msvc-activosportafolios. La política de reinicio está configurada como Always, lo que significa que Kubernetes reiniciará automáticamente los pods en caso de falla.

msvc-activosportafolios-service.yaml

https://github.com/juanhcode/AssetMaker-Microservices/blob/main/msvc-activosportafolios-service.yaml

El archivo msvc-activosportafolios-service.yml define un Service en Kubernetes para exponer el servicio msvc-activosportafolios. Utiliza la API v1 y tiene como nombre msvc-activosportafolios. Este servicio selecciona los pods con la etiqueta io.kompose.service: msvc-activosportafolios. Expone el puerto externo 8085 y lo redirige al targetPort 9004, que es el puerto interno donde el contenedor del servicio está escuchando. No se especifica el tipo de servicio, por lo que se asume el valor

predeterminado de tipo ClusterIP, lo que hace que el servicio sea accesible solo dentro del clúster.

msvc-auth-deployment.yaml

https://github.com/juanhcode/AssetMaker-Microservices/blob/main/msvc-auth-deployment.yaml

El archivo msvc-auth-deployment.yml define un Deployment en Kubernetes para el servicio de autenticación msvc-auth. Utiliza la API apps/v1 y establece que se ejecutará 1 réplica del contenedor. El contenedor utiliza la imagen gcr.io/infra-odyssey-437506-a3/assetmaker:auth-amd-1.0.2 y está configurado para escuchar en el puerto 9000 utilizando el protocolo TCP. Además, se define una variable de entorno llamada FEIGN_CLIENT_URL, que apunta a la URL http://msvc-usuarios:8080, probablemente para la comunicación con otro servicio de usuarios. El selector asegura que los pods de este deployment se identificarán con la etiqueta io.kompose.service: msvc-auth. La política de reinicio está configurada como Always, lo que implica que Kubernetes reiniciará automáticamente los pods si fallan

msvc-auth-service.yaml

https://github.com/juanhcode/AssetMaker-Microservices/blob/main/msvc-auth-service.yaml

El archivo msvc-auth-service.yml define un Service en Kubernetes para exponer el servicio de autenticación msvc-auth. Utiliza la API v1 y tiene como nombre msvc-auth. Este servicio selecciona los pods que tienen la etiqueta io.kompose.service: msvc-auth. Expone el puerto 8081 y lo redirige al targetPort 9000, que es el puerto interno donde el contenedor del servicio está escuchando. No se especifica el tipo de servicio, por lo que se asume el valor predeterminado de tipo ClusterIP, lo que significa que el servicio solo será accesible dentro del clúster de Kubernetes.

msvc-gateway-deployment.yaml

https://github.com/juanhcode/AssetMaker-Microservices/blob/main/msvc-gateway-deployment.yaml

El archivo msvc-gateway-deployment.yml define un Deployment en Kubernetes para el servicio de gateway msvc-gateway. Utiliza la API apps/v1 y especifica que se ejecutarán 3 réplicas del contenedor, lo que asegura alta disponibilidad del servicio. El contenedor utiliza la imagen gcr.io/infra-odyssey-437506-a3/assetmaker/gateway-amd-1.0.3 y está configurado para escuchar en el puerto 8080 utilizando el protocolo TCP. El selector asegura que los pods de este Deployment se identificarán con la etiqueta io.kompose.service: msvc-gateway. La política de reinicio está configurada como Always, lo que significa que Kubernetes reiniciará automáticamente los pods si fallan.

msvc-gateway-service.yaml

https://github.com/juanhcode/AssetMaker-Microservices/blob/main/msvc-gateway-service.yaml

El archivo msvc-gateway-service.yml define un Service en Kubernetes para exponer el servicio msvc-gateway. Utiliza la API v1 y tiene como nombre msvc-gateway. Este servicio selecciona los pods con la etiqueta io.kompose.service: msvc-gateway. Expone el puerto externo 8082 y lo redirige al targetPort 8080, que es el puerto interno donde el contenedor del servicio está escuchando. El tipo de servicio es LoadBalancer, lo que significa que Kubernetes asignará una IP externa para acceder al servicio desde fuera del clúster. Esto es útil para aplicaciones que necesitan ser accesibles públicamente.

msvc-portafolios-deployment.yaml

https://github.com/juanhcode/AssetMaker-Microservices/blob/main/msvc-portafolios-deployment.yaml

El archivo msvc-portafolios-deployment.yml define un Deployment en Kubernetes para el servicio msvc-portafolios. Utiliza la API apps/v1 y específica que se ejecutará una réplica del contenedor. El contenedor utiliza la imagen gcr.io/infra-odyssey-437506-a3/assetmaker:portafolios-amd-1.0.3 y está configurado para escuchar en el puerto 9002 utilizando el protocolo TCP. Además, se establece una variable de entorno llamada FEIGN_CLIENT_URL, que apunta a la URL http://msvc-usuarios:8080, lo que sugiere que el servicio de portafolios se comunica con el servicio de usuarios. El selector asegura que los pods de este Deployment se identificarán con la etiqueta io.kompose.service: msvc-portafolios. La política de reinicio está configurada como Always, lo que significa que Kubernetes reiniciará automáticamente los pods si fallan.

msvc-portafolios-service.yaml

https://github.com/juanhcode/AssetMaker-Microservices/blob/main/msvc-portafolios-service.yaml

El archivo msvc-portafolios-service.yml define un Service en Kubernetes para exponer el servicio msvc-portafolios. Utiliza la API v1 y tiene como nombre msvc-portafolios. Este servicio selecciona los pods con la etiqueta io.kompose.service: msvc-portafolios. Expone el puerto externo 8083 y lo redirige al targetPort 9002, que es el puerto interno en el que el contenedor del servicio está escuchando. Esto permite que las solicitudes externas al puerto 8083 sean dirigidas al servicio de portafolios que corre en el puerto 9002 dentro de los contenedores.

msvc-usuarios-deployment.yaml

https://github.com/juanhcode/AssetMaker-Microservices/blob/main/msvc-usuarios-deployment.yaml

El archivo msvc-usuarios-deployment.yml define un Deployment en Kubernetes para el servicio msvc-usuarios. Utiliza la API apps/v1 y especifica que se ejecutará una réplica del contenedor. El contenedor utiliza la imagen gcr.io/infra-odyssey-437506-a3/assetmaker:users-amd-1.0.0 y está configurado para escuchar en el puerto 9001 utilizando el protocolo TCP. El selector asegura que los pods de este Deployment se identificarán con la etiqueta io.kompose.service: msvc-usuarios. La política de reinicio está configurada como Always, lo que significa que Kubernetes reiniciará automáticamente los pods si fallan.

msvc-usuarios-service.yaml

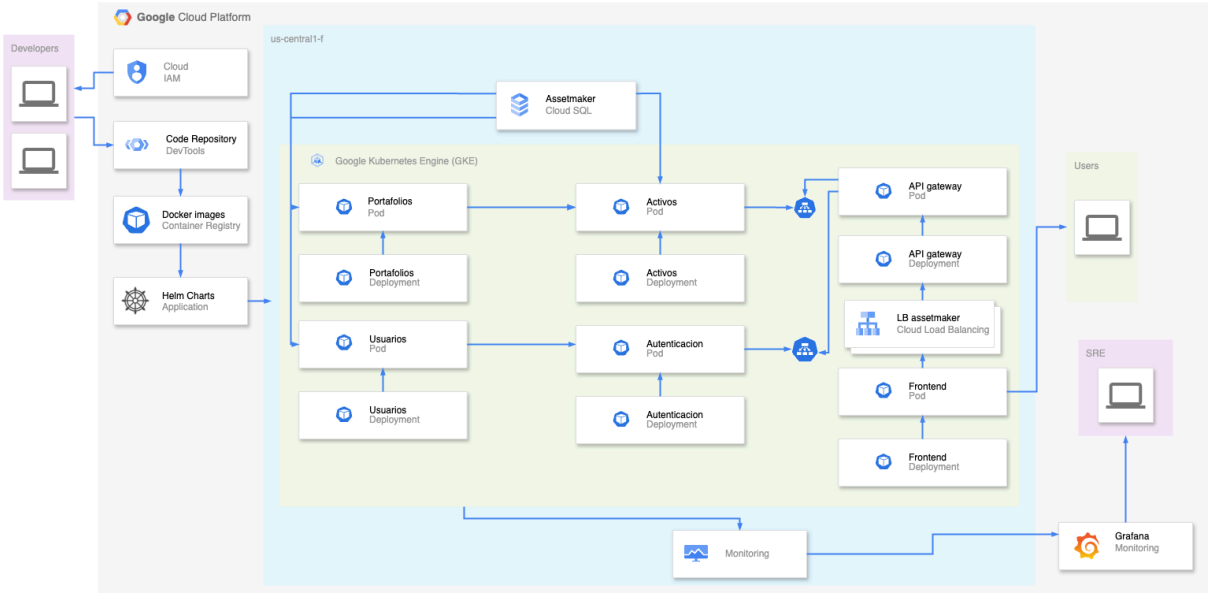
https://github.com/juanhcode/AssetMaker-Microservices/blob/main/msvc-usuarios-service.yaml

El archivo msvc-usuarios-service.yml define un Service en Kubernetes para exponer el servicio msvc-usuarios. Utiliza la API v1 y tiene como nombre msvc-usuarios. Este servicio selecciona los pods que tengan la etiqueta io.kompose.service: msvc-usuarios. Expone el puerto externo 8080, que redirige al targetPort 9001, el cual es el puerto donde el contenedor del servicio está escuchando. Esto permite que las solicitudes externas que lleguen al puerto 8080 sean dirigidas al servicio msvc-usuarios que está corriendo en el puerto 9001 dentro del contenedor.

Es importante destacar que las imágenes de los contenedores se obtienen desde Google Container Registry (GCR). Además, algunos microservicios están configurados para conectarse a la base de datos en Cloud SQL, mientras que otros no requieren dicha conexión.

Solución en la nube

Google Cloud Platform



Utilizamos Google Cloud Platform (GCP) para desplegar la aplicación, aprovechando diversos servicios que facilitan su escalabilidad, gestión y rendimiento, como se detalla en la arquitectura.

Este diagrama muestra la arquitectura de una aplicación desplegada en Google Cloud Platform. La aplicación está compuesta por varios microservicios que se ejecutan en un cluster de Kubernetes.

- Los desarrolladores utilizan herramientas como Cloud IAM, Code Repository y Docker images para construir y desplegar la aplicación.
- Los microservicios se comunican a través de una API Gateway, que utiliza un Load Balancer para distribuir el tráfico entre los Pods.
- Los Pods están agrupados en Deployments, que son los encargados de manejar la actualización y el escalado de la aplicación.
- La aplicación se conecta a una base de datos en Cloud SQL y también cuenta con un sistema de monitoreo con Grafana.
- La aplicación puede ser utilizada por los usuarios a través de la interfaz web, y es monitoreada por el equipo de SRE.

Configuración para el balanceador de carga:

Para la configuración de los balanceadores de carga en el despliegue, en este caso, el frontend se configura con el tipo LoadBalancer en el manifiesto del deployment. Al ejecutar este deployment en Google Kubernetes Engine (GKE), GKE reconoce automáticamente que se requiere un balanceador de carga y crea uno dentro del servicio de balanceo de carga de Google Cloud Platform (GCP). Esta configuración se maneja de forma automática, incluyendo la asignación de una IP pública que permite acceder al servicio desde el exterior, asegurando una distribución eficiente del tráfico y mejorando la disponibilidad y escalabilidad de la aplicación. Además, el Gateway se configura con el tipo LoadBalancer y 3 réplicas en el manifiesto del deployment.

Flujo para Subir la Aplicación a la Nube:

Para desplegar la aplicación en la nube, se creó un clúster en Google Kubernetes Engine (GKE) utilizando el modo Autopilot. Se enviaron los deployments y services correspondientes hasta que todos los pods estuvieran en estado Running. Una vez los pods estaban en funcionamiento, se activó la base de datos con el comando **gcloud sql instances patch assetmakerdb --activation-policy ALWAYS**, lo que garantizó que la base de datos estuviera siempre disponible. Finalmente, se verificaron los logs, los deployments y los services para asegurar que todo estuviera funcionando correctamente y en orden.

El estado de todos los pods corriendo

```
juanhoyos@MacBook-Pro-de-Juan assetmaker % kubectl get pods
NAME                                    READY   STATUS    RESTARTS   AGE
frontend-58c7dc4fc6-8z5kh               1/1     Running   0          49m
msvc-activos-785db66dc-kxvkx            1/1     Running   0          72m
msvc-activosportafolios-7b44546767-sgl4v   1/1     Running   0          72m
msvc-auth-869b48bc96-jwcrv              1/1     Running   0          71m
msvc-gateway-869dd5c9df-9h4lv           1/1     Running   0          46m
msvc-gateway-869dd5c9df-jz4s7           1/1     Running   0          46m
msvc-gateway-869dd5c9df-wp458           1/1     Running   0          46m
msvc-portafolios-7ccc656769-zgp2k       1/1     Running   0          70m
msvc-usuarios-5cd54dcf7-9ksqv           1/1     Running   0          31m
```

El estado de los services

```
juanhoyos@MacBook-Pro-de-Juan assetmaker % kubectl get services
NAME                  TYPE           CLUSTER-IP       EXTERNAL-IP     PORT(S)          AGE
frontend              LoadBalancer   34.118.238.169   34.29.171.118   4200:32044/TCP   18h
kubernetes            ClusterIP      34.118.224.1     <none>          443/TCP          37d
msvc-activos          ClusterIP      34.118.231.86    <none>          8084/TCP         10d
msvc-activosportafolios   ClusterIP      34.118.233.84    <none>          8085/TCP         6d7h
msvc-auth             ClusterIP      34.118.229.199   <none>          8081/TCP         6d14h
msvc-gateway          LoadBalancer   34.118.227.214   34.68.186.196   8082:30115/TCP   81m
msvc-portafolios      ClusterIP      34.118.225.104   <none>          8083/TCP         6d17h
msvc-usuarios         ClusterIP      34.118.239.193   <none>          8080/TCP         29d
```

El estado de los deployments

```
juanhoyos@MacBook-Pro-de-Juan assetmaker % kubectl get deployments
NAME                    READY   UP-TO-DATE   AVAILABLE   AGE
frontend                1/1     1            1           48m
msvc-activos            1/1     1            1           72m
msvc-activosportafolios   1/1     1            1           71m
msvc-auth               1/1     1            1           70m
msvc-gateway            3/3     3            3           46m
msvc-portafolios        1/1     1            1           69m
msvc-usuarios           1/1     1            1           30m
```

Análisis

El rendimiento local está limitado por los recursos del equipo del desarrollador, lo que impide realizar un escalado vertical, ya que aumentarlo implicaría costos adicionales para el equipo. Por lo tanto, el rendimiento de la aplicación en el entorno local depende tanto de la cantidad de réplicas como de cómo se distribuyen los recursos de CPU y memoria entre ellas. Por otro lado, el rendimiento en la nube está condicionado por el plan gratuito de los créditos de GKE, que asigna una cantidad limitada de recursos a un clúster dentro de ese plan. Sin embargo, al estar en la nube, se puede realizar escalamiento tanto vertical como horizontal de cada microservicio. Esto

permite mejorar el rendimiento, ya que aumenta la capacidad de respuesta y la carga del frontend al escalar las réplicas según sea necesario.

Uno de los principales retos fue la comunicación entre los microservicios, ya que, como se explicó previamente, muchos de ellos necesitan interactuar a través de HTTP. Para resolver esto, implementamos una tecnología llamada Feign, la cual nos permite especificar el DNS de la imagen junto con su puerto interno correspondiente. Otro desafío significativo fue la integración de una fuente de datos con Grafana para visualizar métricas tanto del IAM como de CloudSQL, lo que requería configurar adecuadamente la conexión para obtener información precisa y actualizada de ambas fuentes.

En cuanto a seguridad, a nivel de desarrollo implementamos el cifrado de contraseñas, autenticación y el uso de JWT tanto en componentes como en servicios. Al almacenar las imágenes en GCR de GCP, contamos con un control más estricto y la capacidad de analizarlas, ya que funciona como un repositorio privado de imágenes. Además, para acceder a cualquiera de los servicios, es necesario contar con permisos o roles específicos gestionados mediante IAM, lo que nos permite definir políticas claras sobre quién puede crear deployments, gestionar réplicas, entre otras acciones.

Docker y Kubernetes simplifican la implementación y administración de aplicaciones. Docker crea contenedores portátiles, mientras que Kubernetes automatiza su orquestación, escalamiento y disponibilidad, optimizando la infraestructura y mejorando la eficiencia operativa. Estas tecnologías simplifican el flujo de trabajo, mejoran la eficiencia operativa y facilitan la implementación continua, aunque requieren una inversión en aprendizaje y configuración para alcanzar su máximo potencial.

Conclusiones

Como conclusión, aplicamos los conocimientos adquiridos durante el semestre en el proyecto. El uso de Docker y Kubernetes nos permitió entender cómo las grandes aplicaciones en la industria tecnológica responden a la demanda de recursos, escalando automáticamente a través de réplicas o reduciéndolas según sea necesario. Además, la gestión de GKE en este proyecto nos brindó una valiosa experiencia sobre cómo crear y gestionar aplicaciones compuestas por múltiples microservicios, integrando varios servicios de GCP. Aprendimos a desarrollar de manera modular en lugar de monolítica, como se había hecho en proyectos anteriores. Docker y Kubernetes se consolidan como herramientas clave para la gestión de contenedores y la orquestación en entornos modernos.