Prueba Técnica para la empresa Master

Prueba de técnica - Empresa Inlaze

Repositorio: https://github.com/elvingonzalezb/master-test

Enlace para verificar pequeña App: http://44.218.139.126/

Enlace para verificar documentación servicio API:
http://44.218.139.126/api/docs

Enlace para verificar documentación servicio Extract:
http://44.218.139.126/extract/docs

Enlace para verificar documentación servicio Transform:
http://44.218.139.126/transform/docs

Enlace para verificar documentación servicio Load:
http://44.218.139.126/load/docs

Infraestructura general: * > Servidor EC2 en - AWS * > Ip Elastica - AWS * > RDS postgreSQL - AWS * > S3 – AWS * > AmazonMQ Rabbit - AWS * > Cluster Redis – Digital ocean * > Cluster MongoDB – Atlas * > Repositorio docker Hub * > GitLab - Github Actions * > Creación de servicios mediante docker-compose * > Creación de contendores mediante docker

Lenguajes en general: * > Python - FastApi - Servicio extract/transform/load/api * > Node js - Servicio auth * > React - Frontend - zustand - shadcn

1. Desarrollo con Python y SQL:

- > La clase DatabaseConnection es la encargada de conectarse a la base de datos postgres en una instancia de AWS - RDS, utiliza psycopg2, tiene varios metodos para conectar, abrir cursor, cerrar conexion, con respecto al manejo de errores captura excepciones si hay error al intentar conectarse OperationalError, tambien maneja adecuadamente y asegura que la conexion se abra y cierre corractamente.
- > PostgreSQL utilizando el módulo psycopg2 y realice una consulta SQL
- > Se crea funcion para obtener el número total de registros de una tabla de repositorios de gitHub.
- > Describe cómo manejarías los errores de conexión y consulta en
- > Código fuente se encuantra en el repositorio
- > Se crea la Documentación con Swagger

2. ETL con API pública utilizando microservicios:

- > Todo el proceso de infraestructura utiliza una arquitectura de microservicios, por lo cual estan descentralizados y cada servicio tiene su logica o modelo de datos, asi es posible tener independencia a la hora de desplegar.
- > Para esta prueba se crearon 6 microservicios y un nginx para redirigir el trafico y mapear los puertos de los servicios y asociarlos a un ip elastica de AWS.
- > Cada servicio es responsable de un unico proceso (claro dentro tienen varios procesos pero dependen en general del proposito del servicio)

- Servicio API: para realizar peticion a laAPI de github actions y obtener los repositorios aleatorios.
- Servicio Extract: encargado de realizar la extracción de los datos
- Servicio Transform: encargado de realizar la transformación de los datos, obviamente este proceso tiene varios procesos que no se llegaron a concluir debido a que es una de las capas mas complejas.
- Servicio Load: encargado de realizar la carga de los datos.
- Servicio Auth: encargado de realizar la autenticación para el frontend.
- Servicio Frontend: encargado de renderizar algunos componetes simples, a mode de visualización y pruebas
- Los servicios se pueden comunicar mediante peticiones sincronas y asincronas, mediante url que cada uno tiene asignada, todos tiene conexion con la capa de datos, pueden persistir informacíon y enviar eventos a colas para procesar peticiones asincronos.
- Cada servicio se empaqueta mediante contenedores docker, estas imagenes sunsubidas a un repositorio privado en DockerHub
- Para el despliegue de los microservicios utilice un servidor de AWS EC2, tiene activo los runners de gitHub Action y el proceso de CI - CD es automatico. Cabe destacar que otra opcion excelente para el despligue es en un cluster de k8s, no lo realice debido a temas de costos, aunque tengo un cluster en una cuanta free de digital ocean, pero actualmente lo tengo ocupado.
- Es posible alguna latencia en el servicio, ya que la instancia de EC2 es de la capa free por ende no tien mucha potencia.

3. Proceso automatico de CI-CD:

- Para este proceso utilice gitHub Actions.
- Existen dos stages y cada uno con difrentes jobs.
- El proceso incia al momento de realizar un commit, en una rama especifica, ya eso depende del flujo de trabajo de tareas.
- Se activa el stage build que contiene diferentes jobs, entre ellos:
  - Creación de imagen base y revison de recursos (proceso automatico perteneciente a github action)
  - Login con el repositorio privado de docker hub, para hacer push y pull de la imagenes
  - Buid y publish de todas las imagenes.
  - Copiar el archivo docker-compose.
- Se activa el stage deploy que contiene diferentes jobs, entre ellos:
  - Verificación y autenticación SSH a la instancia.
  - Ejecución en la instancia.

Observaciones del proceso:

- Es importante destacar que, aunque es un proceso válido, se deben tomar algunas medidas para mitigar el tiempo en que se realiza el deploy y el sitio puede quedar inactivo.
- Cada servicio puede estar en repositorios separados para mejorar el rendimiento. Esto depende también de la lógica del negocio y la cantidad de servicios.

- Es fundamental versionar y etiquetar las imágenes.
- Una mejor opción podría ser implementar esta misma solución en un clúster de Kubernetes. Esto permitiría aplicar balanceo de carga, controladores de ingreso, medidas de seguridad contra ataques como DDoS, límites de tasa, prevención de ataques lentos, circuit breakers, compresión de encabezados en las peticiones, uso de HTTP/2 y gRPC.
- Para este ejemplo, utilize algunos patrones como repositorio, una implementación de DDD (no al 100%, pero estable), principios SOLID en algunos casos, arquitecturas orientadas por eventos y microservicios. La lógica de programación está basada en mi experiencia en todo el ciclo de vida del software.
- Teniendo en cuenta que este puede ser un proceso simple en ciertos casos, puede llegar a volverse complejo debido a la gran cantidad de información en el proceso general de ETL. Aquí solo se realizó la extracción (E) de una API, sabiendo que puede haber múltiples orígenes de información.