

MANUAL TÉCNICO

Código Fuente del proyecto:
<https://github.com/ldecast/P1-SOPES1-G7>

TABLA DE CONTENIDO

1. OBJETIVO
2. PRESENTACION
3. DESARROLLO DEL MANUAL TECNICO

1. OBJETIVO

Informar y especificar al usuario la estructura y conformación del sistema con el fin de que puedan hacer soporte y modificaciones o actualizaciones al sistema en general.

2. PRESENTACIÓN

El siguiente manual guiará a los usuarios que harán soporte al sistema, el cual les dará a conocer los requerimientos y la estructura para la construcción del sistema, en el desarrollo de programa de escritorio y aplicativo móvil conectados mediante una base de datos en la nube, el cual muestra las herramientas necesarias para la construcción y la funcionalidad del sistema.

3. DESARROLLO DEL MANUAL TECNICO

Procesos Procesos de entrada

□ Programa de escritorio Ingresar al programa de escritorio (acceso).

Ingresar datos para el registro.

Ingresar datos para registros de usuarios.

□ Aplicativo móvil o web Ingresar al aplicativo móvil (acceso).

Registrar datos para el registro de usuarios.

Procesos de salida

□ Programa de escritorio Consulta.

Consulta de usuarios.

Generar formatos

□ Base de datos MYSQL Exportar copia de seguridad de la base de datos en la plataforma (nube).

Requisitos del sistema

□ Requerimientos de hardware Equipo, teclado, mouse, monitor, dispositivo móvil. Memoria RAM 2 GB (equipo y dispositivo móvil) Tarjeta de red LAN y/o Wireless Procesador 1.4 GHz.

□ Requerimientos de software Sistema operativo (Windows 7 en adelante). Java 8.0. Sistema operativo móvil (Android 5.0. en adelante) Conexión internet local y móvil. Adobe Reader.

Herramientas utilizadas para el desarrollo

Servidor de base de datos (MySQL) El servidor de base de datos MySQL es uno de los más característicos y por tener la opción de código abierto a nivel mundial, siendo una de las más populares antes ORACLE y Microsoft SQL Server principalmente en entornos de desarrollo web.

Servicio de Comentarios y Notificaciones.

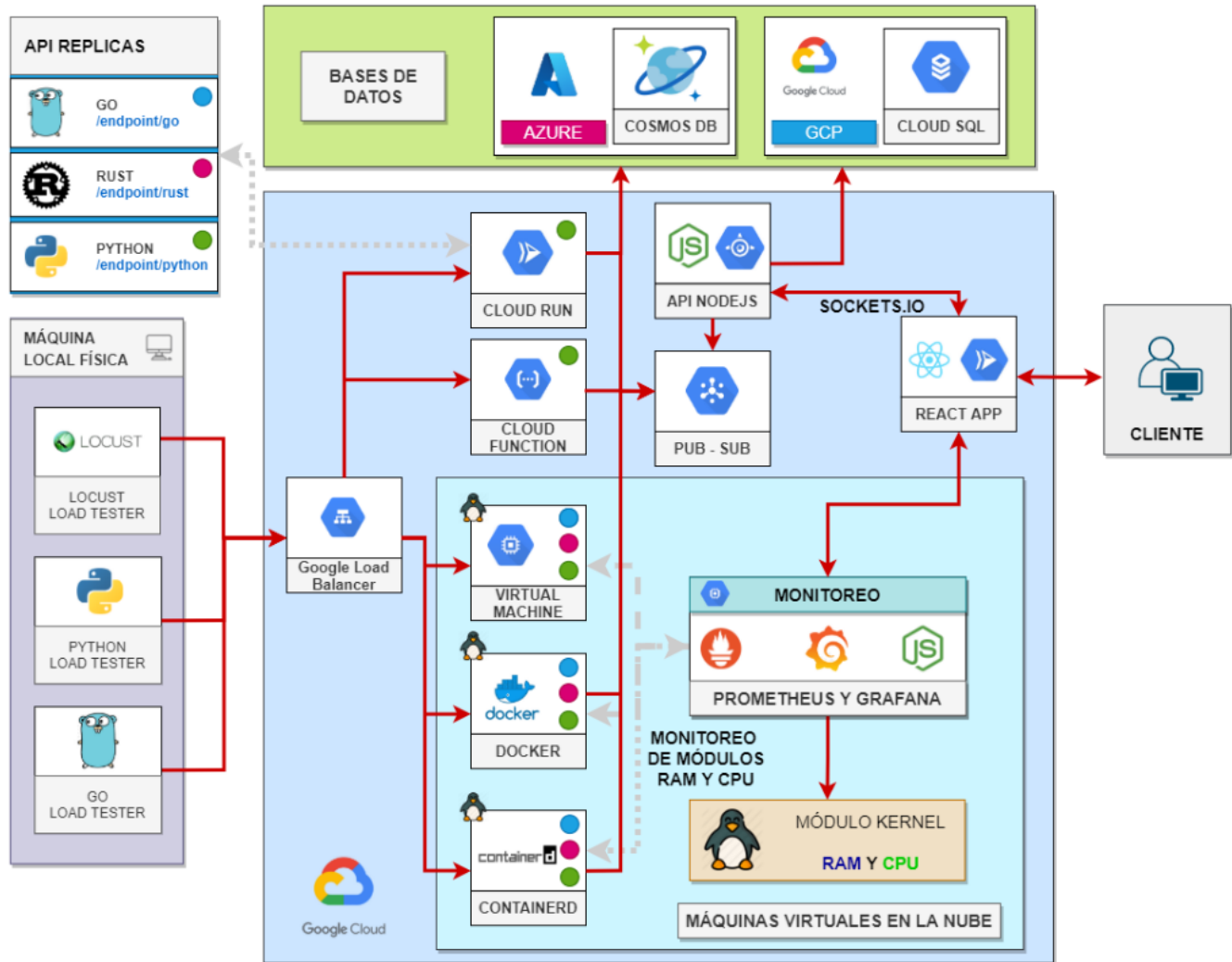
Es un sistema computacional distribuido, cloud native, utilizando diferentes servicios de Google Cloud Platform, virtualización a nivel de sistema operativo con Docker y Containerd y generadores de tráfico para ser aplicado a un tema actual.

DESCRIPCIÓN GENERAL

A partir de la finalización de los juegos olímpicos Tokio 2021, se tiene un visualizador de las noticias y comentarios de las olimpiadas que están realizando los espectadores.

El sistema está totalmente en la nube, utilizando diferentes servicios de Google Cloud Platform y una base de datos de Cosmos DB en Microsoft Azure. cuenta con una carga masiva de datos a partir de diferentes generadores de tráfico, la información a mandar será detallada más adelante. Además de este sistema, se cuenta con un modo "Administrador" en el cual se visualiza gráficas y métricas relevantes de las noticias, y de la información de la RAM y procesos de las máquinas virtuales que se tendrán en la nube.

DIAGRAMA DEL FLUJO DE LA APLICACIÓN:



La finalidad de las aplicaciones que están en esta parte es enviar tráfico al Load Balancer de Google Cloud (detallado más adelante). Básicamente estas aplicaciones tienen las siguientes funcionalidades:

- Leer un archivo .json que contenga un array de datos (los datos respectivos a cada generador de tráfico).
- Procesar el archivo y enviar tráfico a un endpoint.
- Mostrar cuantos datos fueron enviados y cuantos datos tuvieron error.

La información a enviar por los tres generadores de tráfico es la siguiente:

```
{
  "nombre": "Leonel Aguilar", // Nombre del publicador de la noticia o comentario
  "comentario": "El día de hoy el atleta [...]", // Comentario realizado
  "fecha": "24/07/2021", // Fecha en la que se realizo el comentario
  "hashtags": ["remo", "atletismo", "natacion"], // Hashtags o etiquetas del comentario
  "upvotes": 100, // Cantidad de personas a las que les gusto el comentario
  "downvotes": 30 // Cantidad de personas a las que no les gusto el comentario
}
```

GOOGLE LOAD BALANCER

Cloud Load Balancer recibe todo el tráfico creado por Locust y los otros dos generadores de tráfico, este redirecciona el tráfico a los diferentes servicios a implementar.

API REPLICAS

Se realizarán tres APIs que funcionarán de exactamente la misma manera (por ello el “réplica”) pero serán realizadas en diferentes lenguajes. La funcionalidad de estas APIs es recibir el tráfico que se genere y pase por el Load Balancer, para luego pasar esta información a la base de datos y crear una notificación utilizando Google PubSub. El estudiante deberá considerar qué rutas son las necesarias para pasar este tráfico, pero como inicio y sugerencia se tienen las siguientes:

- **/iniciarCarga:** Una ruta que podrá conectarse a la base de datos y esperar los datos a cargar.
- **/publicar:** Una ruta que tendrá la opción de publicar la información a la base de datos.
- **/finalizarCarga:** Una ruta que cerrará la conexión a las bases de datos y mandará una notificación a través de Google PubSub.

Los tres lenguajes a implementar serán:

- Python
- Go
- Rust

En cada uno de estos lenguajes deberán crear la misma funcionalidad de la API para conectarse a la base de datos y enviar información enviada desde los generadores de tráfico. Además, se debe crear una notificación de cuantos datos fueron cargados en Google Pub/Sub.

CLOUD RUN – APIS

En Google Cloud Run se cargará el servicio de la API de Python y de Go.

CLOUD FUNCTIONS

En Google Functions se cargará el servicio de la API de Python y de Go.

MÁQUINAS VIRTUALES EN LA NUBE

Se tendrán tres máquinas virtuales en la nube en Google Compute Engine, cada una con las mismas APIs que se detallaron anteriormente. La única diferencia es la forma de instalar y configurar las APIs,

MÓDULO RAM

Montar un módulo que lea la RAM del sistema. Este módulo debe de proveer de la siguiente información:

- Memoria RAM total (en MB)
- Memoria RAM en uso (en MB)
- Memoria RAM libre (en MB)
- Porcentaje de memoria RAM siendo utilizada.

Para validar que sea correcto, debe mostrar los mismos valores que el comando:

\$ free --mega -t

MÓDULO LISTA DE PROCESOS

Montar un módulo que lea la información del CPU del sistema. Este módulo debe proveer la siguiente información:

- Porcentaje de CPU utilizado
- Cantidad de procesos ejecutándose

Para validar que sea correcto, debe mostrar valores parecidos a los obtenidos con el comando:

\$ free --mega -t

Los módulos Kernel a montar en el sistema operativo deberán ser desarrollados en lenguaje C como fue visto en el laboratorio. Deberán ser montados en las tres máquinas virtuales que contienen las APIs. La forma de leer este archivo y mandarlo al módulo de monitoreo queda a discreción del estudiante. Se recomienda que:

- Se implemente un endpoint en la API de su elección (la más sencilla sería en Python) para leer el contenido de los archivos desde el directorio /proc y enviarlo por una petición HTTP al módulo de monitoreo.

MÓDULO DE MONITOREO

En este módulo se pretende generar un Dashboard utilizando Prometheus y Grafana donde se observe la información de la RAM (toda la que aparece en el Kernel) y la información del CPU y cantidad de procesos ejecutándose. La manera de obtener esta información queda a discreción del estudiante. Se recomienda:

- Implementar una API NodeJS que pida la información por medio de una petición HTTP a las máquinas virtuales donde se encuentran los módulos Kernel y leer la información desde NodeJS hasta prometheus.

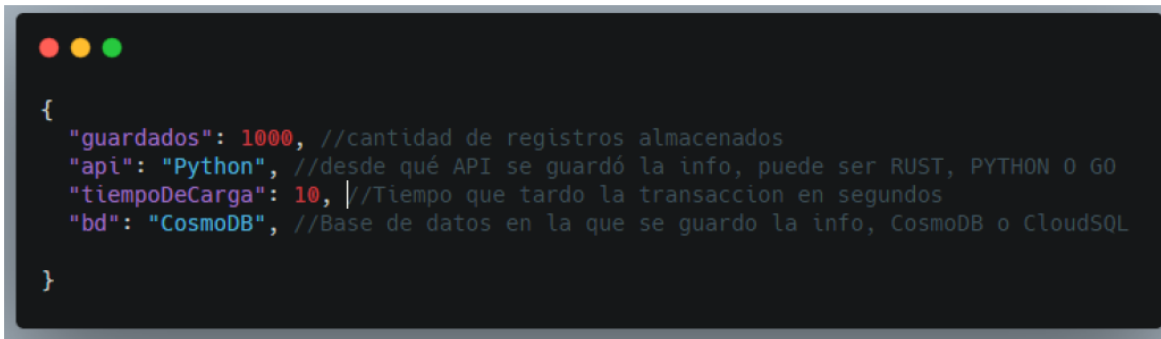
Para más información consultar acá:
<https://codersociety.com/blog/articles/nodejs-application-monitoring-with-prometheus-and-grafana>

El módulo de monitoreo debe estar en una máquina virtual, ya sea una dedicada únicamente para esto, o una de las que ya tienen de las APIs. La manera de implementar el dashboard de grafana queda a discreción del estudiante, sin embargo se recomienda que se utilicen Dashboards previamente realizados y únicamente acoplarlos a la información manejada de la RAM y CPU.



GOOGLE PUB/SUB

Se utilizará Google Pub/Sub para enviar notificaciones acerca de la carga de información desde las APIs hasta la API principal de NodeJS. La información de esta notificación es la siguiente:



```
{
  "guardados": 1000, //cantidad de registros almacenados
  "api": "Python", //desde qué API se guardó la info, puede ser RUST, PYTHON O GO
  "tiempoDeCarga": 10, //Tiempo que tardo la transaccion en segundos
  "bd": "CosmoDB", //Base de datos en la que se guardo la info, CosmoDB o CloudSQL
}
```

La notificación se disparará cada vez que se finalice la carga de uno de los generadores de tráfico, es decir, no llegará una notificación por cada registro guardado, sino una notificación por el total de los registros. Si desde el generador envió 200 registros, se deberá mostrar una única notificación con el total de guardados de 200.

APP ENGINE

Esta API estará escrita con Javascript utilizando NodeJS. La finalidad de esta API será recolectar los datos para ser mostrados en la aplicación del cliente. Además de realizar reportes con la información que se tiene almacenada (detallado posteriormente). **Esta API estará publicada en el APP ENGINE de Google Cloud.**

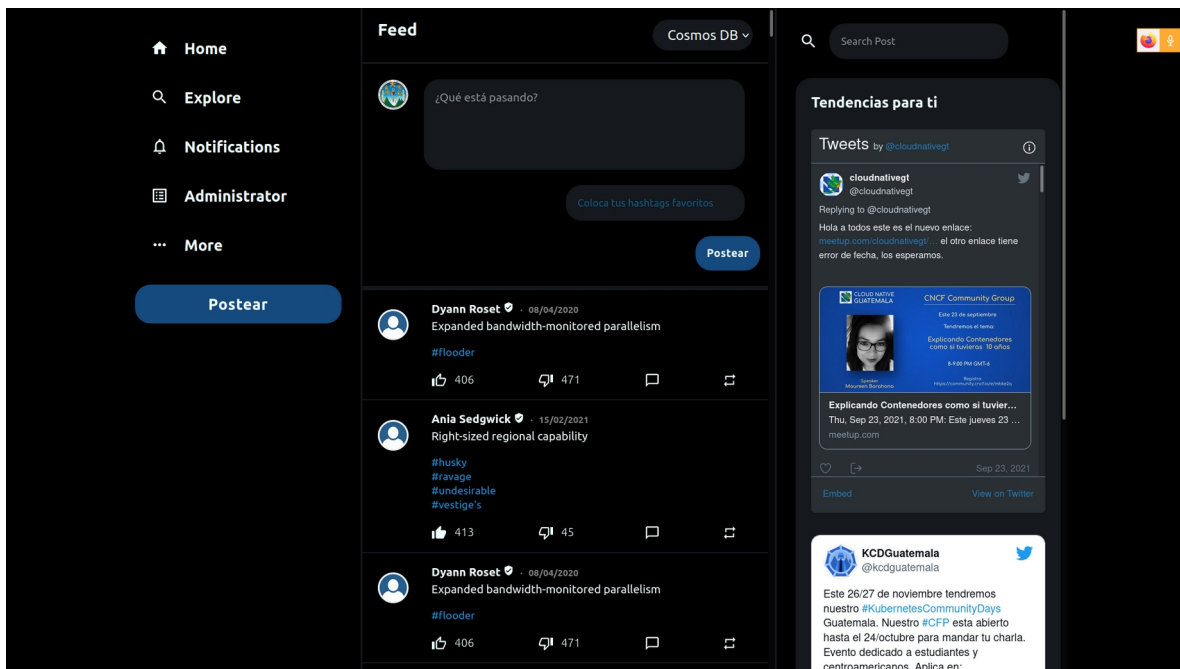
Importante: Esta API se conectará a la APP a través del módulo **Sockets.IO**. para simular una aplicación en tiempo real.

APP WEB

Realizar una app web utilizando el framework React. Esta mostrará las diferentes métricas a partir de los datos guardados en el servidor de CosmosDB y de CloudSQL. Se tomará en cuenta en la calificación la aplicación de conceptos de UX/UI y de responsiveness para el diseño de la app. Por último, la APP realizada en React debe ser publicada utilizando Google Cloud Run.

VISTA DE INFORMACIÓN

La información debe ser mostrada en su totalidad. Las vistas en las que se mostrará la información queda a discreción del estudiante, sin embargo tomar en cuenta que se calificará la estética y usabilidad de esta.

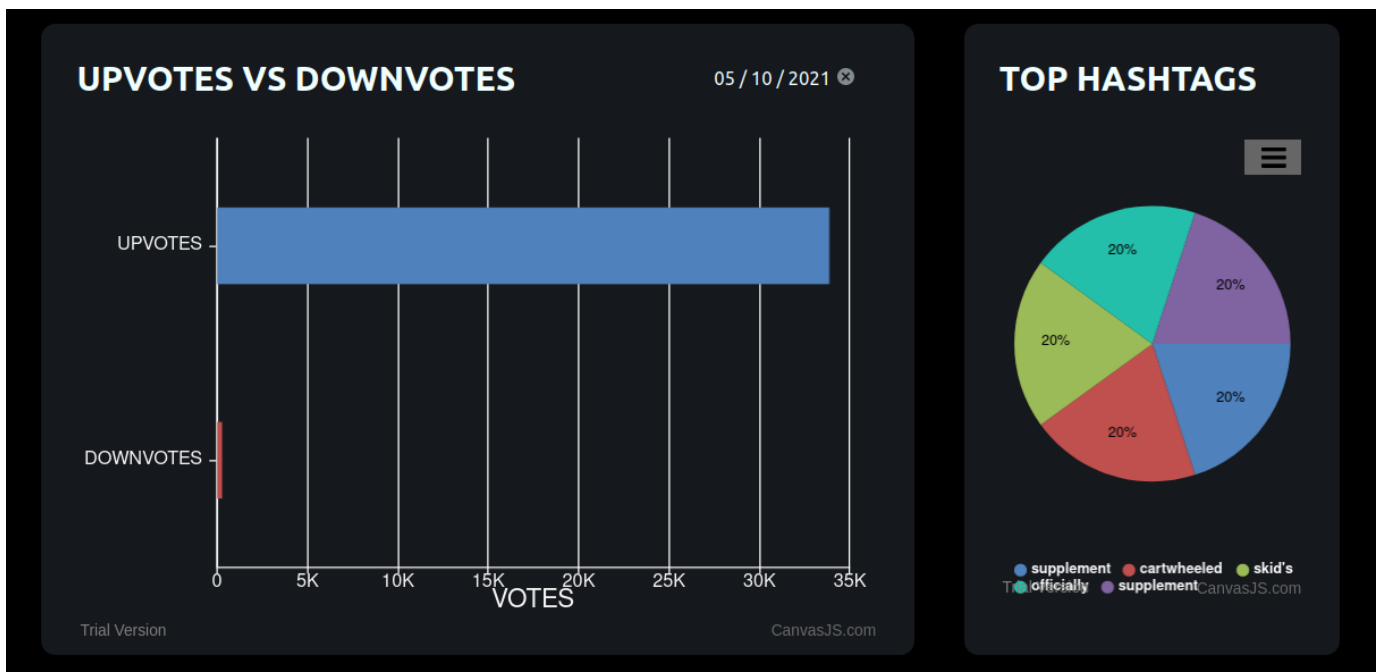
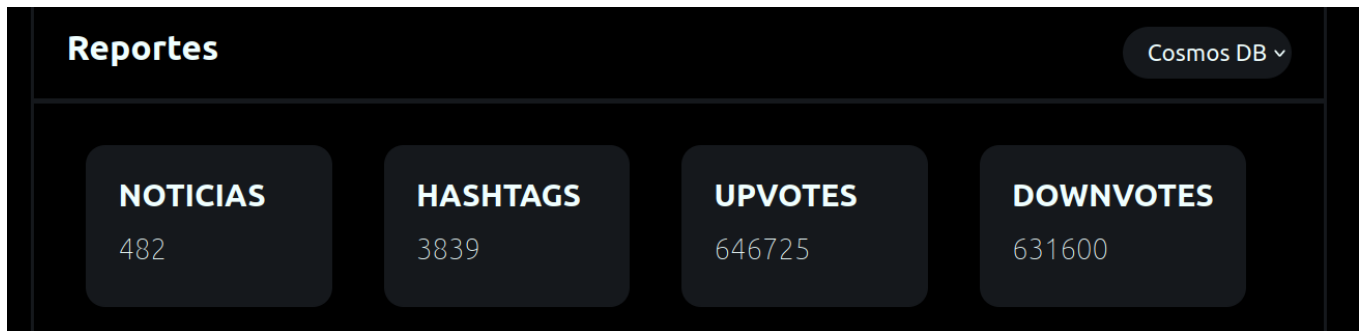


VISTA DE REPORTE

Aparte de la información mostrada de la base de datos, se le pide realizar los siguientes reportes en forma de tablas, estos reportes **deberán de ser en tiempo real utilizando Sockets.IO**.

- • Mostrar el total de noticias, upvotes y hashtags (diferentes) que hay en el sistema.
- • Tabla con los datos que están en el sistema.
- • Gráfica circular del top 5 de hashtags. Se calculará a partir de la cantidad de upvotes que tiene cada hashtag.
- • Gráfica (de barras, líneas, stackeada, etc...) que compare la cantidad de upvotes y downvotes por día.

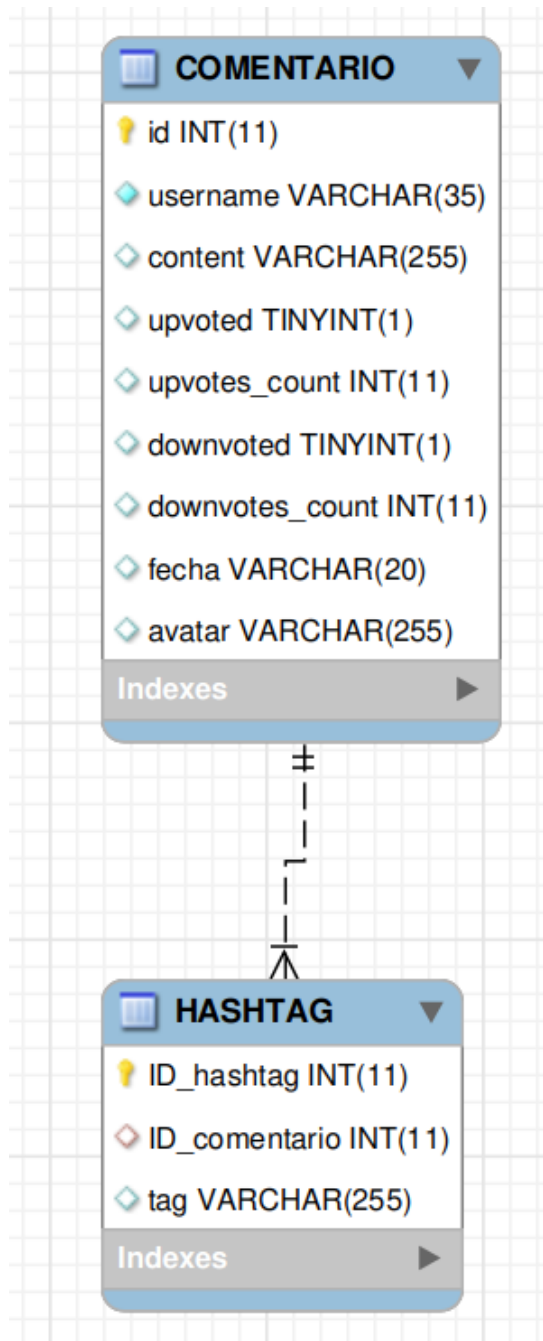
También se agrega un botón o filtro con el cual se pueda decidir desde qué base de datos realizar los reportes.



ENTRADAS RECIENTES

Usuario	Fecha	Comentario	Hashtags
Luis Castellanos	05/10/2021	Probando...	#sopes1
Dyann Roset	08/04/2020	Expanded bandwidth-monitored parallelism	#flooder

MODELO ENTIDAD RELACIÓN PARA LA BASE DE DATOS RELACIONAL



PREGUNTAS DE REFLEXIÓN

✓ **¿Qué generador de tráfico es más rápido? ¿Qué diferencias hay entre las implementaciones de los generadores de tráfico?**

El generador de tráfico que encontramos más óptimo fue el de Python gracias a la simplicidad con la que se puede escribir el código y la eficiencia de tener menos sentencias y declaraciones ofrece un menor tiempo de ejecución y sin librerías externas, las cuales fueron las principales diferencias con Go. Locust es bastante similar en rendimiento ya que se construye también en lenguaje Python.

✓ **¿Qué lenguaje de programación utilizado para las APIs fue más óptimo con relación al tiempo de respuesta entre peticiones? ¿Qué lenguaje tuvo el performance menos óptimo?**

El lenguaje que fue de preferencia gracias a su rendimiento y accesibilidad de uso fue el de la API programada en Python. Rust fue el más complicado pero tuvo un performance similar al de Go. La velocidad de inserción en Go es algo lenta para la base de datos en CloudSQL y no existe documentación para implementar inserciones en una API de SQL de CosmosDB en Azure. Sin embargo, el módulo de "net/http" de Go es más ligero y en ocasiones se percibió un mayor performance que el paquete de Flask de Python y el servidor de Rust.

✓ **¿Cuál de los servicios de Google Cloud Platform fue de mejor para la implementación de las APIs? ¿Cuál fue el peor? ¿Por qué?**

El mejor servicio para la implementación de la API, en este caso la principal de Noje Js, como backend de la aplicación Web, fue el servicio de Google App Engine. La implementación de la aplicación en App Engine fue de una manera bastante sencilla y optimizada ya que el único requerimiento es tener un archivo de especificación y configuración (app.yaml) en donde se establece el lenguaje de programación, versión y qué tipo de instancia de está utilizando, el resto de configuraciones se pueden dejar por defecto y con un comando sencillo se puede hacer deploy de la aplicación, insertando buckets de datos automáticamente y finalmente se genera una url en donde correrá el servidor. Tiene un panel de control muy efectivo.

✓ **¿Considera que es mejor utilizar Containerd o Docker y por qué?**

Ambos cumplen con los requerimientos que se deseaban para la aplicación, alojamiento e instalación de módulos Kernel, aunque por temas de documentación en la web y practicidad en la implementación de los contenedores encontramos levemente mejor y preferible el uso de Docker, ya que al utilizar Docker también se está usando containerd pero de una manera interna, y containerd viene de Docker que implementa el Container Runtime Interface (CRI) pero que fue separado del proyecto de Docker como tal para volverlo más modular.

✓ **¿Qué base de datos tuvo la menor latencia entre respuestas y soportó más carga en un determinado momento? ¿Cuál de las dos recomendaría para un proyecto de esta índole?**

Definitivamente la base de datos con mejor rendimiento para la implementación de este proyecto fue CosmosDB, bajo la API de SQL Core, de Azure. Los tiempos de inserción a las bases de datos entre MySQL y CosmosDB fueron bastante distintos, siendo la de Azure mucho más rápida. Para la visualización de los registros solicitados en la aplicación Web también fue mucho más rápida.

Para poder insertar y seleccionar los datos, se realizan menos instrucciones por iteración al no contar con un modelo relacional, los datos se manejan directamente como se ingresan en el archivo de entrada (esto gracias al formato JSON del archivo de registros).

El único problema que encontramos al trabajar con CosmosDB fue que no existe documentación actualizada y funcional para ciertos lenguajes como Go, cuyo SDK para crear y manejar contenedores con la API de SQL Core no existe, ya que sólo se encuentra de la API de Mongo.

✓ **Considera de utilidad la utilización de Prometheus y Grafana para crear dashboards, ¿Por qué?**

Sí, es de utilidad ya que Prometheus trabaja con datos de series de tiempo identificados por nombre métrico y claves/ valores pares, cuyo lenguaje de consultas permite aprovechar esta multidimensionalidad para consultar los datos de manera simple y flexible con nodos que colaboran a la colección de series de tiempo ocurre a través de un modelo de extracción, utilizando HTTP. La compatibilidad que presenta Grafana es muy recomendada ya que trabaja bien con soluciones en Cloud y la imagen mejorada de Docker, lo que permite suministrar Grafana con configuración a través de archivos.