**Proyecto de Sistemas Distribuidos**

Top Ten Gifs con arquitectura de Microservicios

**Integrantes**

José Viteri

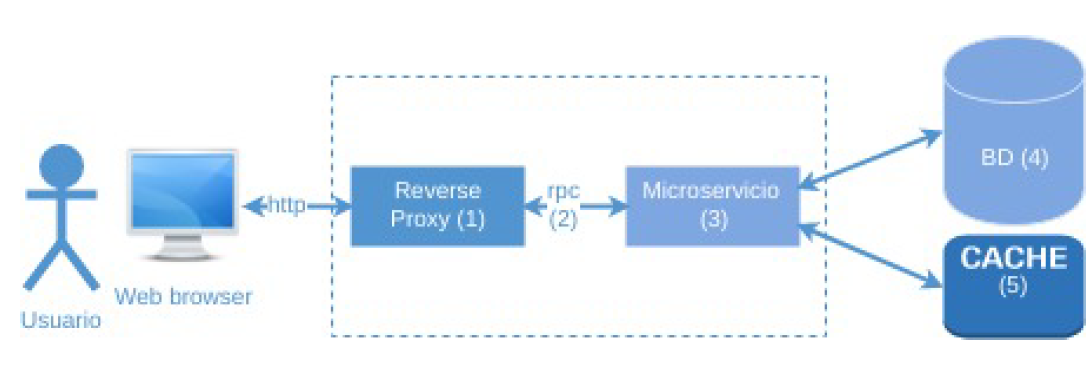
Edison Mora

Jaminson Reascos

Profesor: Dra. Cristina Abad

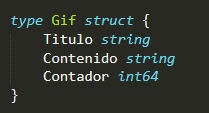
## Problema y solución propuesta

Para este proyecto tuvimos que diseñar un servicio web que sea capaz de darnos los 10 gifs más populares en determinado momento. Para eso se pensó en realizar una arquitectura de microservicios , con un servidor web, un microservicio, y dos bases de datos: una relacional, donde estarían todos los gifs,y una no relacional donde se almacenarán los top 10 gifs diarios y actuaría de caché. Tanto para el servidor web, como el microservicio, lo hicimos en el lenguaje de programación GO, específicamente el servidor web en un framework llamado Gin. Para la base de datos relacional se uso MySQL y para la caché Redis. Además se usó un reverse Proxy de Nginx.



## Estructuras de datos usados

Para almacenar los gifs en redis se transformó cada gif en base64, luego se armó la siguiente estructura:



Inicialmente se tenía pensado almacenar cada Gif (struct) en un hash en Redis, para tener la misma representación que en el microservicio. Para cargar los gifs de Redis simplemente se iban a seleccionar todos los hashes que estuvieran en Redis en ese momento, debido a que asumimos que siempre iban a estar solo el top 10 ahí.

Sin embargo, Redis no ofrecía ningún método para obtener todos los hashes de una sola llamada, para hacerlo había que obtener uno por uno en llamadas diferentes a Redis. Esta forma de hacerlo no iba a ser para nada eficiente, por lo que se decidió dejarla de lado.

Finalmente, se decidió serializar la estructura Gif y almacenar ese string en una lista en Redis. Cada gif serializado sería un elemento de la lista que tendría como key la fecha en la que fue creada.

Este método permite hacer una sola llamada para obtener los gifs de redis. Sin embargo, no es bueno para hacer otras operaciones como aumentar el contador de un gif o añadir/quitar un gif de la lista. Para estas operaciones se debían hacer dos llamadas a Redis, a diferencia de usar hashes, para lo cual hubiera sido una sola llamada.

Se decidió utilizar ese método a pesar de sus desventajas debido a que lo que se quería optimizar es el tiempo de búsqueda, ya que sería la operación más frecuente.

## Librerías/middlewares

Para la implementación del microservicio y el servicio web se hizo uso de las siguientes dependencias:

* Protocol buffers: Es una herramienta que facilita la serialización de estructuras de datos, el uso de este mecanismo es necesario para definir un estándar de representación de datos en la comunicación entre procesos
* gRPC: Es un framework especializado para implementar RPC, es necesario el uso de gRPC ya que Protocol Buffers se limita a definir la representación de los datos por lo que la manera en que estos los mensajes son enviados entre los servicios es una tarea delegada a gRPC
* Redis: Es una in-memory data structure store, que en nuestro proyecto hace las veces de memoria caché, para aliviar la carga sobre la base de datos
* Mysql: Es la base de datos relacional elegida para almacenar en disco la información a enviar
* Nginx: Para la api Gateway se hizo uso de Nginx, que servirá de Reverse Proxy , para redireccionar los pedidos http al servidor web hecho en Gin que corre naturalmente en el puerto 3000.
* Gin: Gin es un framework ligero web, que extiende ya las funciones web naturales de Go. Se lo eligió sobre todo por su simplicidad y performance dentro de otros frameworks de Go

## Manejo de errores

Cuando no existen los gifs dentro de Redis, lo que hace el microservicio es pasar el control a mysql y obtener los datos ahí, luego almacena esos gifs en redis.

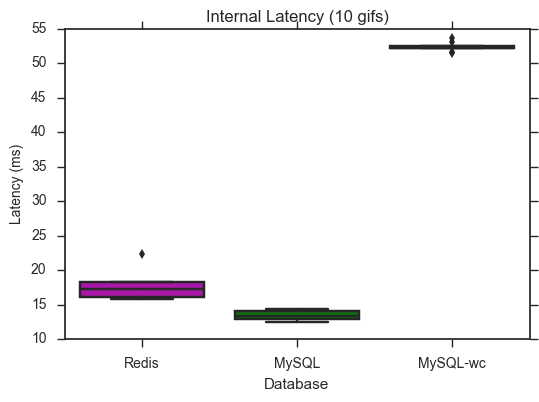
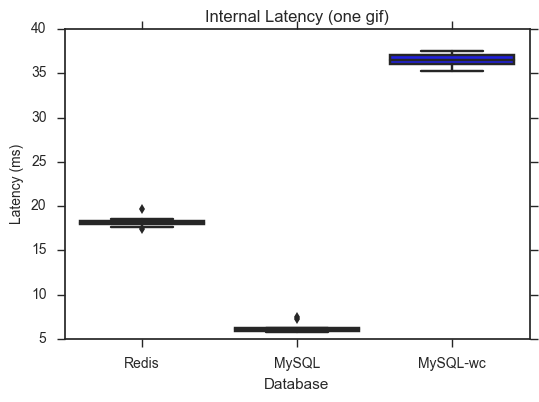
El error se da cuando la base de datos de mysql no tiene el top 10 de gifs tampoco.

En ese caso se envía un error al cliente del microservicio, y este se encarga de avisar del error al cliente web.

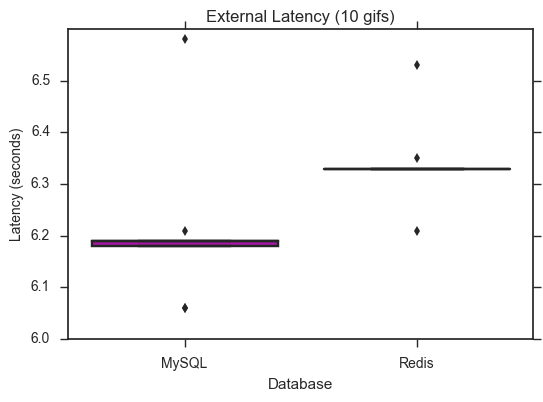
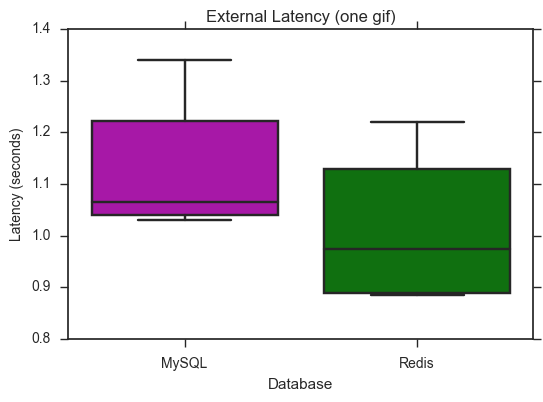
## Pruebas de rendimiento

Para las pruebas de rendimiento se hicieron tres pruebas, tanto para retribuir un solo gif como para retribuir el top de los mejores 10 gifs. Una de Latencia interna que calculaba la cantidad de tiempo que la base de datos (MySQL o Redis) tardaba en acometer el query respectivo. En esta prueba cada vez que existía una solicitud se guardaban los tiempos en el mismo microservicio. Se hizo una prueba de Latencia externa que media cuánto tardaba en dar el gif o los gifs desde que hacía la solicitud hasta que se cargaba en el navegador. Para esta prueba se hizo uso de la misma interfaz del cliente web y el modo desarrollador de Google Chrome donde se cogieron los tiempos. Una última prueba fue la de throughput donde se midió la cantidad solicitudes web a los respectivos endPoints que retribuyen el gif o los gifs. Aquí se hizo uso de bash y curl para las pruebas. Los resultados a continuación

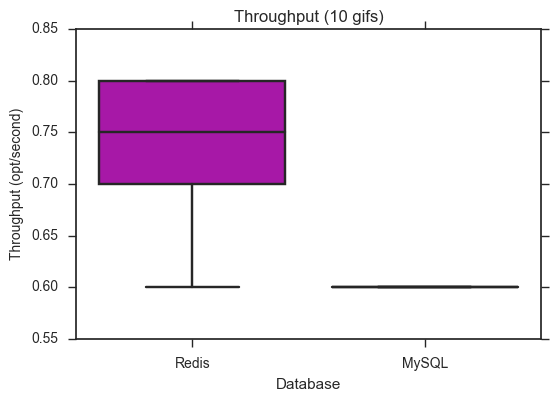
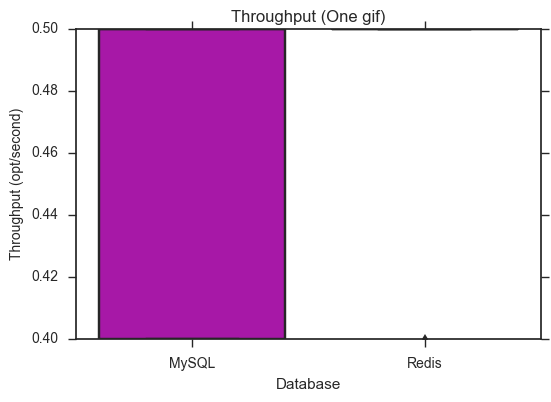
**Latencia interna:**



**Latencia externa**



**Throughput**



**Conclusiones**

Una de las cosas que se puede ver, es que a diferencia de lo que uno pueda pensar, MySQL se comporta en muchos casos mejor que Redis. Después de investigar nos dimos cuenta que la razón de ello es que MySQL implementa internamente un sistema de cacheo a los querys más repetidos. Dado que en este caso el query que se solicita es en gran medida el mismo, el rendimiento de MySQL mejora mucho. La prueba de Latencia interna se hizo justo para ver en cambio como sería el rendimiento sin el sistema de Cacheo de MySQL (el label que dice MySQL-wc). Como era de esperarse el rendimiento empeoro bastante y Redis fue mejor. Aunque en estas pruebas finales no se ha subido, cuando se hicieron pequeños tests en un servidor local ese rendimiento de MySQL sin cacheo fue incluso peor de lo que se mostró aquí, teniendo picos con latencias 10 veces superiores a las de Redis.

## Nube

Para proveer escalabilidad al proyecto y asegurar disponibilidad se decidió subir tanto el componente web como el microservicio a la nube, para esto escogimos usar Elastic Computing Cloud de Amazon Web Services que nos permite utilizar recursos computacionales remotos bajo demanda con un uptime garantizado del 99.99%. Gracias a esto nuestro proyecto es accesible desde cualquier computador conectado a internet sin necesidad de hacer ningún tipo de configuración especial.