75.61 – Taller de Programacion III

Trabajo Práctico 3: Taxi Tracker

1er cuatrimestre 2018

1ra entrega – 26/04/18

Padron: 95470

Nombre: Gabriel Gayoso

Email: ga-yo-so@hotmail.com

Facultad de Ingeniería

Universidad de Buenos Aires

Objetivos

El objetivo principal es establecer un sistema lo suficientemente completo tal que cumpla con los requisitos pedido, lo suficientemente flexible como para que se pueda adaptar a requisitos que surjan en el futuro, y lo suficientemente escalable como para soportar volúmenes altos de carga. Además es deseable poseer medios de monitoreo de los flujos de información en el sistema.

Por la naturaleza de los actores del negocio la escalabilidad no es igual para todos los servicios que se deben proveer: es seguro que el número de taxistas interactuando con la aplicación será notablemente mayor que el número de administradores. Por esta razón es necesario mantener lo mas separado posible los distintos servicios.

Para cumplir con lo pedido, se considera apropiado el desarrollo de la aplicación como servicios sobre Google App Engine. Esto permitirá escalar cada parte del sistema por separado, siempre que el crecimiento del negocio lo requiera y se cuente con el capital necesario.

Esto impacta en la implementación del sistema, ya que se trata de un servicio del tipo PaaS que encapsula todas las partes de la aplicación. Por ejemplo las partes del sistema quedarán separadas en lo que se conoce como servicios, la persistencia se manejará con la estructura Datastore (y Memcache para agilizar los accesos), y se cuenta con entorno de programación pre configurado y optimizado pero limitado.

Suposiciones

1 - No se requiere en esta etapa del desarrollo un sistema de accesos seguros a las distintas partes del sistema

2 – Los dispositivos HTTP siempre informan correctamente sobre los viajes

Esto quiere decir que si la conexión al sistema falla durante alguna operación (subida/bajada de bandera), el dispositivo persiste la operación para repetirla más tarde. De esta forma se reduce notablemente la posibilidad de registros incompletos en el sistema.

Diseño

Casos de Uso

A partir de los requisitos planteados se pueden distinguir una serie de casos de uso frente a distintos actores: los taxistas, los administradores y los ciudadanos. A continuación se presenta el diagrama que los expone.

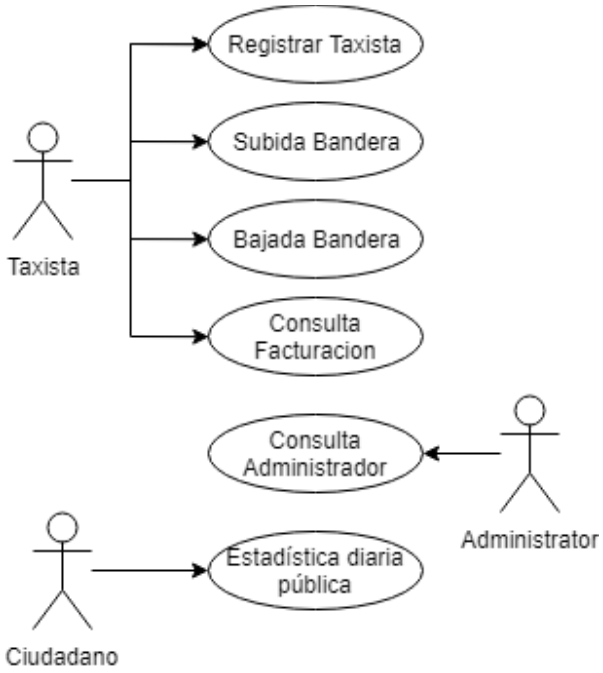

Figura 1: Casos de uso

De aquí en adelante se considerará como una unidad al taxista junto con su dispositivo que habilita la comunicación con el sistema. De esta forma se ve que los primeros tres casos de uso son automáticos al configurar el dispositivo en el taxi, y luego al realizar una subida o bajada de bandera. Los demás son consultas realizadas activamente por alguno de los actores del sistema.

El primer caso ocurre solo una vez por cada taxi, y afecta solo a ese taxi. Los siguientes dos son los que más frecuentemente se darán en el sistema, y afectan solo a su taxi. La consulta que puede realizar el taxista es solo sobre su información propia, mientras que la consulta de administrador puede abarcar a todos los taxistas. La consulta de ciudadano se trata de un caso de muchos individuos queriendo acceder a la misma información que además eventualmente es constante (al terminar el día). Seguro será conveniente tenerla pre calculada.

Vista Lógica

Lo primero que se puede hacer es tomar la información anterior para definir qué datos necesitan ser persistidos. A partir de esto, se concluye que hacen falta las entidades:

- Taxi: con un identificador único y campos de descripción del taxista y/o vehículo.
- Trip: con un identificador único, una referencia al taxista que lo realizó, y campos de descripción como ubicación y fecha de subida y bajada de bandera, distancia recorrida, etc.
- DailyStatistic: utilizando el día como identificador, deberá contener todos los campos que se quiera exponer al público como estadística diaria.

Además se puede encapsular cada uno de las funcionalidades pedidas en una clase de manera que sean independientes. Pero existe una funcionalidad más que debe definirse: la operación de cálculo intermedio para llegar a la estadística de viajes por día pedida. Esta operación debería correr periódicamente manteniendo la estadística actualizada a medida que se realizan nuevos viajes.

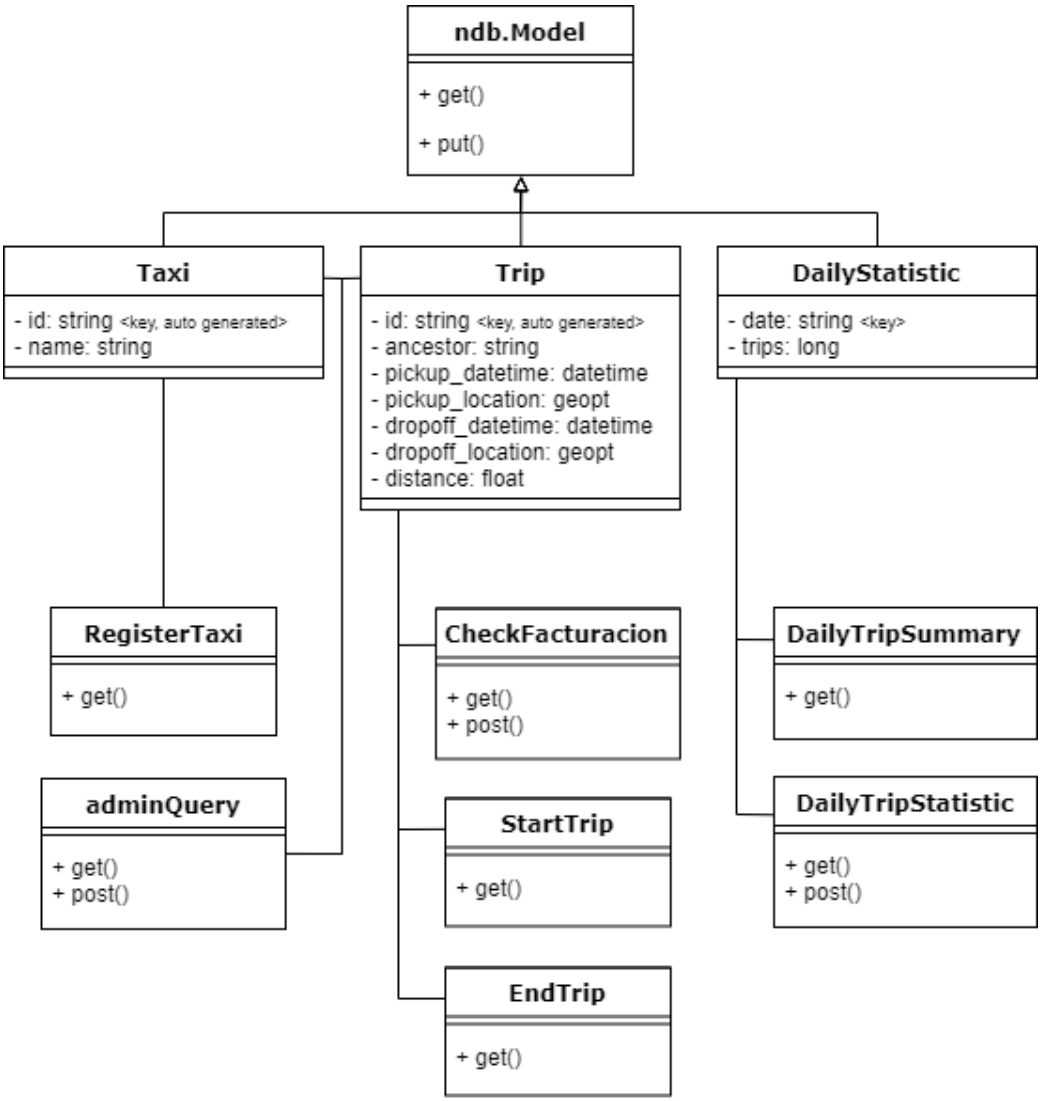

Figura 2: Clases

Los métodos “get()” y “post()” existen simplemente porque desde su planteo inicial (y por la elección de GAE) el sistema expone una API REST para exponer sus funcionalidades.

La clase “ndb.Model” se incluye para explicitar que estas entidades serán persistidas de alguna forma a través de los métodos “get()” y “put()” de esta clase (distintos de los mencionados anteriormente).

Vista de procesos

Las consultas se basan en recibir a través de una interfaz web una serie de filtros y obtener las entidades correspondientes para devolverlas.

La consulta de administradores podría llegar a abarcar todos los viajes de todos los taxis. Este resultado se vuelve más grande a medida que pasa el tiempo y ocurren más viajes. Por esto es necesario realizar un paginado de los resultados para mostrarlos de manera iterativa.

En el caso de consulta de taxistas, considerando que esta filtrada por día no se considera necesario un paginado. Si un taxi realizara 6 viajes por hora durante las 24 horas de un día aún llegaría solo a 144 viajes.

Lo mismo pasa con la consulta de ciudadanos.

Todos estos procesos al igual que el registro inicial del taxi son bastante simples. El proceso que tiene un poco de complejidad y debe separarse en partes es el de registro de un viaje. Como ya se dijo, cuenta de un inicio, un fin, y una etapa de actualización de estadística.

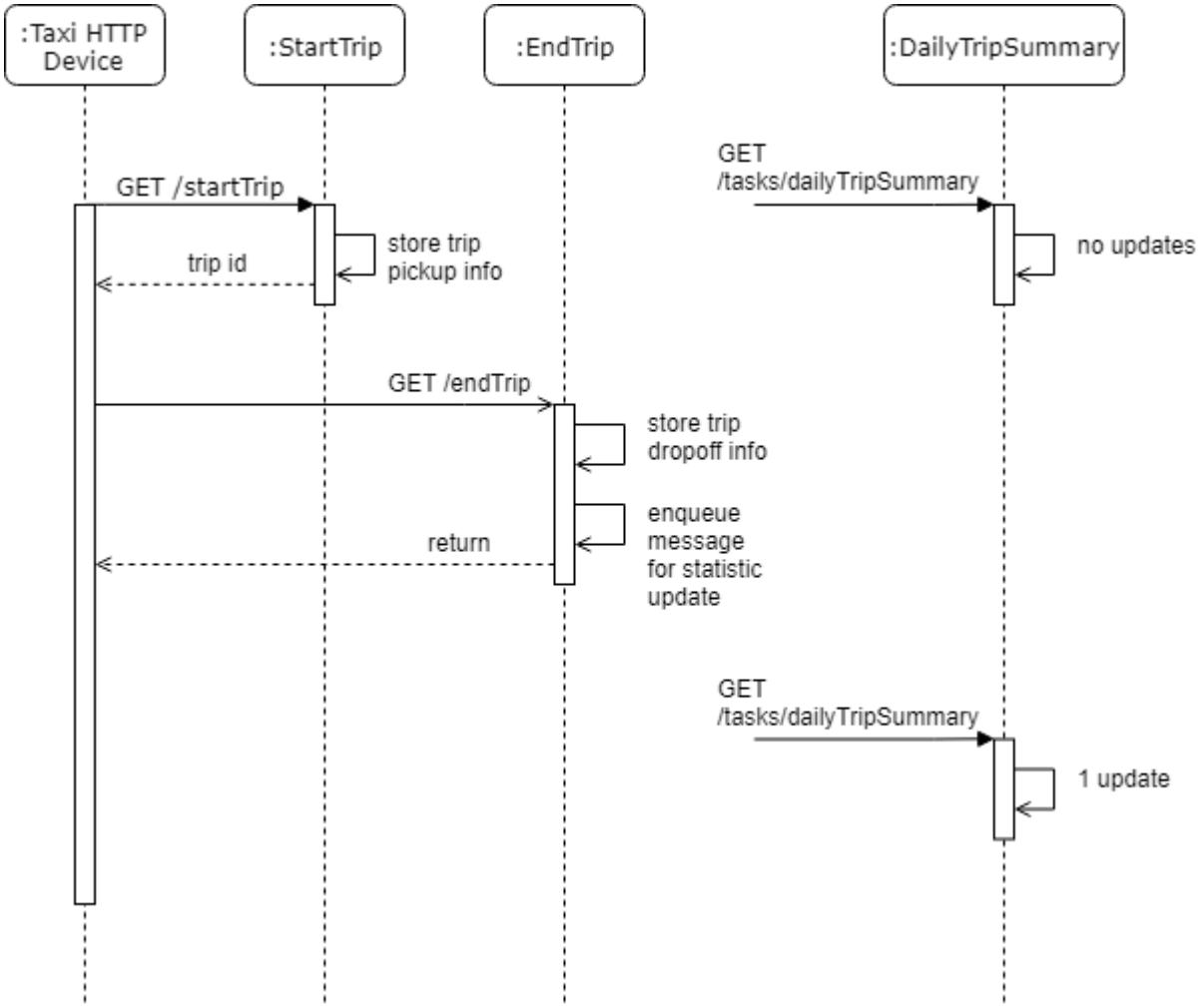

Figura 3: Secuencia de un viaje

El viaje se inicia durante la operación de bajada de bandera, cuando el dispositivo HTTP del taxi automáticamente realiza la llamada al sistema. En ese momento se crea una nueva entidad que representará ese viaje y se llenan los datos de inicio de viaje. También se devuelve el identificador del viaje. Al terminar el viaje el dispositivo vuelve a informar al sistema el cual almacena en la entidad los datos de bajada de bandera y además encola un pedido de actualización de la estadística en alguna estructura predefinida. Durante todo este tiempo corre asincrónicamente el DailyTripSummary que se encarga de recolectar estos pedidos de actualización y llevarlos a cabo. Como se puede ver al inicio, la operación se realiza aún si no hay pedidos de actualización (en cuyo caso no hay cambios).

Vista de desarrollo

El desarrollo del sistema está guiado por las buenas prácticas en la plataforma Google App Engine. Las unidades de procesamiento son llamadas *módulos*, *servicios* o *micro servicios* que escalan independientemente. Por esta razón se decidió separar las funcionalidades según su necesidad de escalamiento.

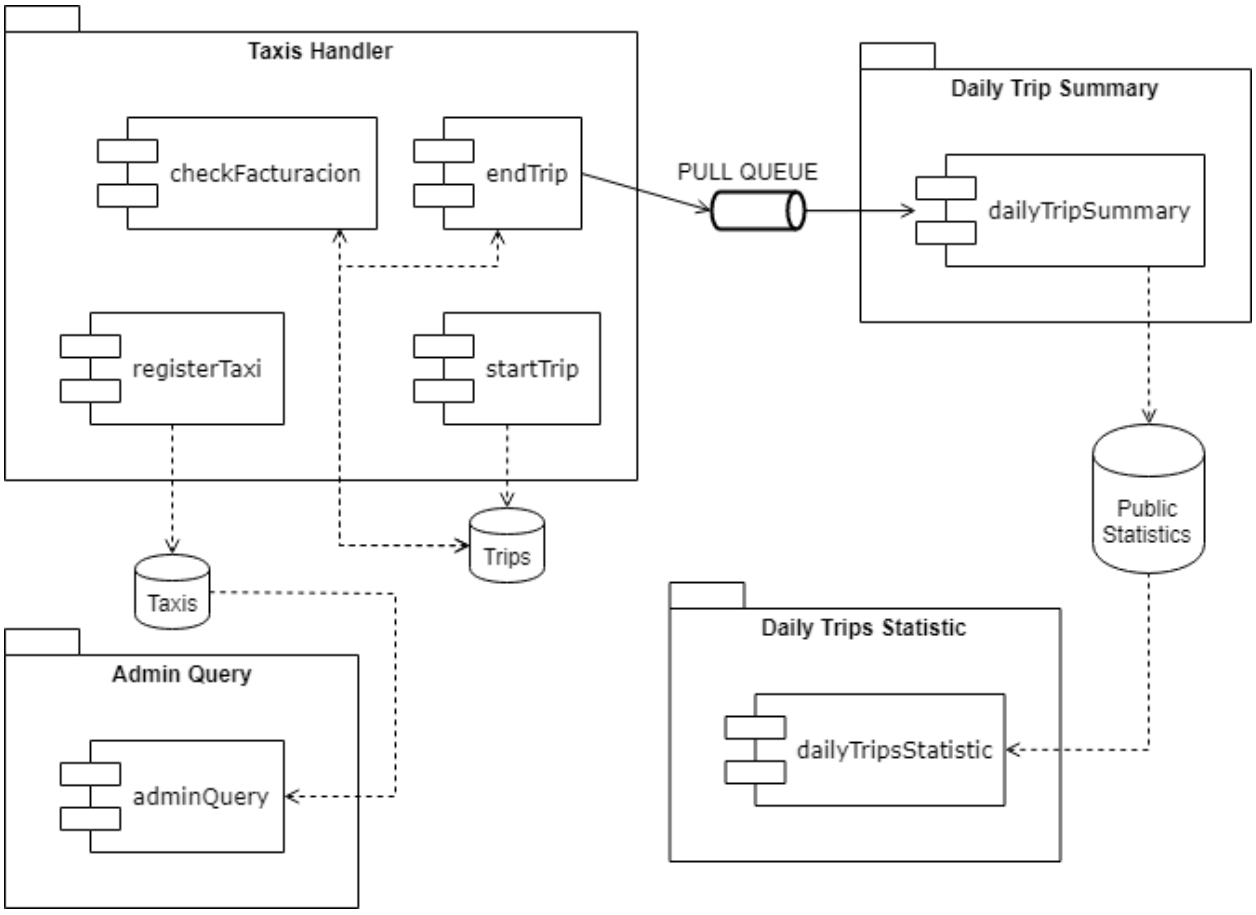

Figura 4: Componentes

Cada paquete del diagrama es accedido solamente por un actor del negocio. Para que puedan escalar de la misma forma que sus usuarios cada uno será un servicio separado en GAE.

La interacción entre “endTrip” y “dailyTripSummary” ocurre a través de otra herramienta de GAE: una pull queue. Al registrarse un fin de viaje se almacena en la cola un mensaje especificando la fecha del viaje como tag. Estos mensajes van acumulándose con el tiempo. Periódicamente se llama dailyTripSummary que intenta agarrar un número predefinido de mensajes que tengan el mismo tag. Luego, actualiza la estadística del día indicado por el tag acorde a cuantos mensajes obtuvo. Finalmente se liberan esos mensajes para que no vuelvan a la cola.

Vista física

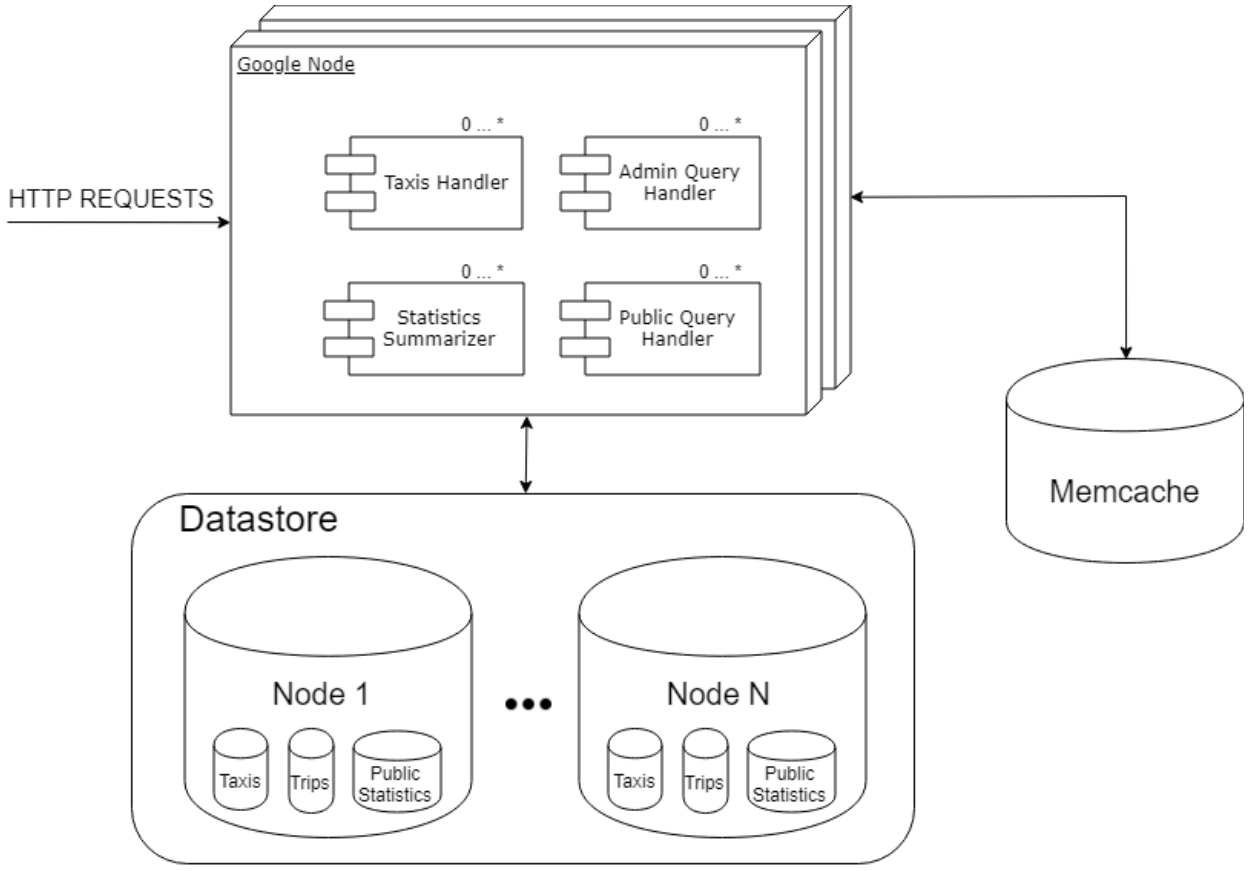

Figura 5: Despliegue

Al utilizar GAE el sistema se abstrae de su ubicación física real en cuanto a servidores. GAE cuenta con clusters de nodos en los que se levantarán los servicios tantas veces como sea necesario frente al volumen de pedidos que se reciban.

En cuanto a la persistencia, ocurre a través del Datastore. Este también puede ser distribuido en nodos, según como se defina la estructura de entidades. Dado que se tiene una entidad Taxi de la que depende la entidad Trip GAE mantendrá siempre todos los viajes de un taxi en el mismo nodo, pero intentará

separar los distintos grupos (taxi, viajes) en los distintos nodos disponibles. Lo mismo ocurrirá con las estadísticas, se intentara separar en nodos las estadísticas de los distintos días, pero sin dividir las de un mismo día. Esto permite el acceso concurrente a cada taxista minimizando el costo de performance.

Por último, los siguientes diagramas resumen el funcionamiento de los servicios definidos:

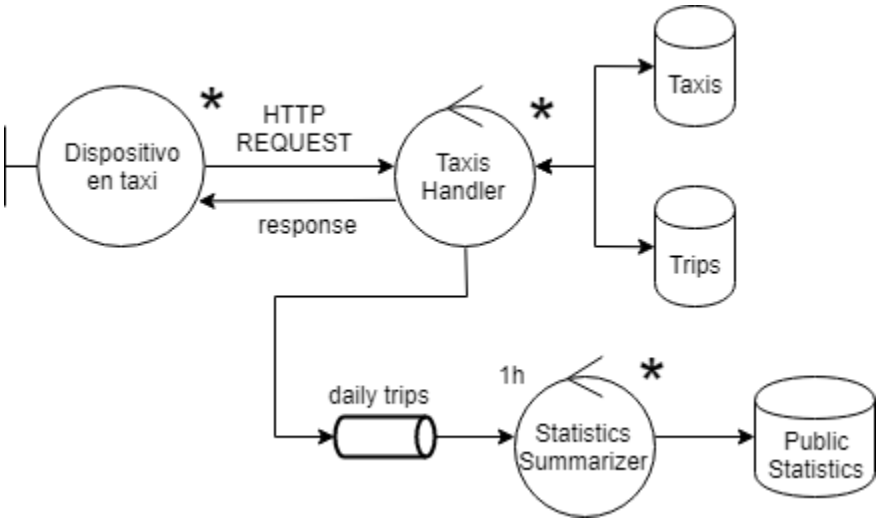

Figura 6: Robustez de Taxis Handler y Statistics Summarizer

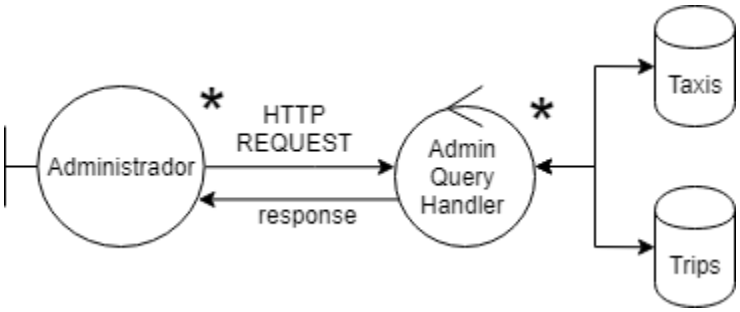

Figura 7: Robustez de Admin Query Handler

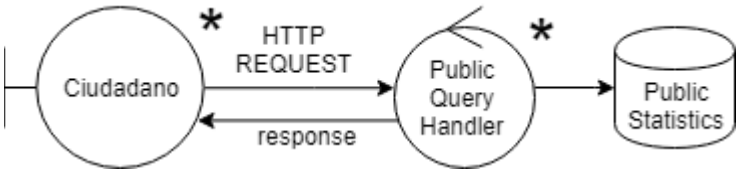

Figura 7: Robustez de Public Query Handler

Testeo

Para el testeo inicial del sistema se utilizo el servidor local de desarrollo que ofrece Google. Es posible levantar la misma estructura que se tendrá en la nube pero en la computadora local para realizar

cualquier tipo de testeo preliminar. El rendimiento no es el mismo, pero el objetivo no fue hacer pruebas de carga y performance sino verificar que el sistema funcione correctamente. Durante esta etapa se probaron manualmente las funcionalidades a medida que se iban desarrollando.

Una vez completado el diseño se subieron los servicios a un proyecto de Google App Engine. Al hacer esto es la plataforma la que decide en que nodos y cuantas instancias de cada servicio levantar. Para automatizar el testeo y poder realizarlo a una escala mayor se utilizó la herramienta Jmeter. Se desarrolló un test configurable con tres componentes: taxis, administradores y ciudadanos.

Se obtiene de un archivo una lista de nombres para los taxistas, y se crea un grupo de threads que se encargan de simularlos. Su ciclo de vida consiste en registrar su taxi y proceder a loopear un proceso de inicio de viaje, fin de viaje y chequeo de facturación durante algún día al azar. La información de los viajes se obtiene de otro archivo. Tanto la cantidad de taxistas como la cantidad de viajes que realiza cada uno son configurables.

Los administradores simplemente repiten consultas aleatorias sobre los taxistas, y los ciudadanos consultan continuamente la estadística pública en días al azar. Tanto la cantidad de administradores como de ciudadanos son configurables.

Además se dedicó un thread a simular el Cron y activar constantemente el servicio DailyTripsSummary ya que Google no permite una frecuencia de activación menor a 1 minuto.

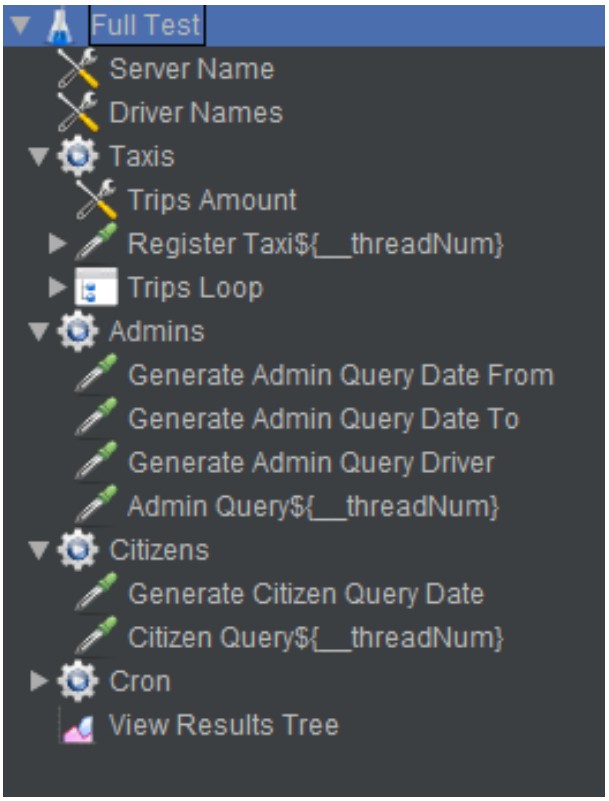

Figura 8: Esquema de testeo automático con Jmeter

Este test se corrió con los siguientes valores:

- Taxis: 100
- Viajes por taxi: 100
- Administradores: 15
- Ciudadanos: 100

El sistema tardó menos de 30 segundos en procesar los mensajes. Google ofrece muchas estadísticas de uso de su plataforma que permite ver de forma sencilla y gráfica como responde la aplicación:

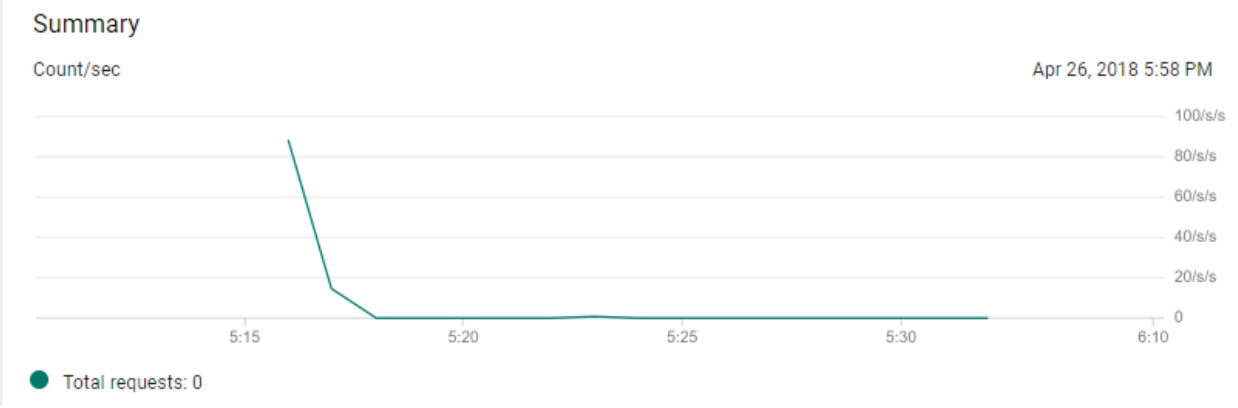

Figura 9: Request por segundo en todos los servicios

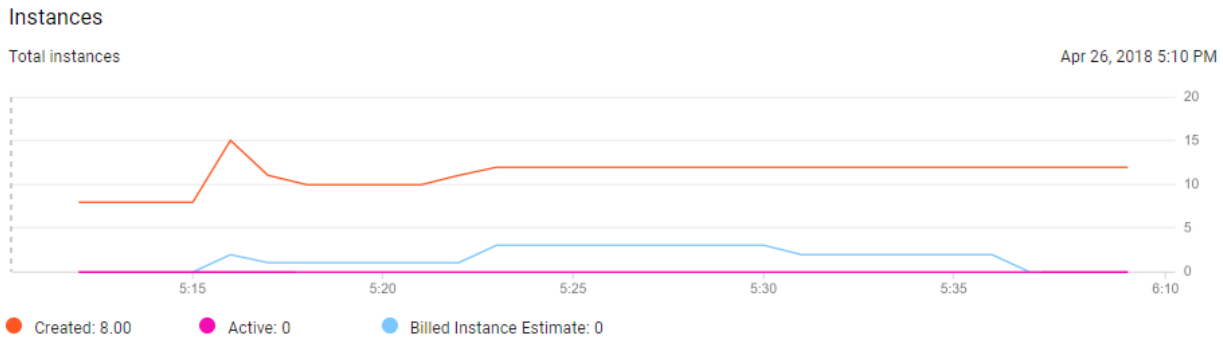

Figura 10: Instancias totales en todos los servicios

Se puede ver un pico de casi 100 requests por segundo lo cual parece correcto ya que había 100 taxis en la prueba. También es notable que el sistema solo necesitara de 15 instancias del servicio para suplir la demanda. Si bien 100 taxistas es un número bajo para una ciudad, en un caso real haría falta muchos taxistas para que la probabilidad de que 100 de ellos inicien o terminen viajes en el mismo instante fuera significativa.

Desglose de actividades

Tareas	Duracion (hs)
Investigation sobre google app engine	4
Diseño inicial del sistema	2
Implementation del sistema	6
Testeo local	1
Corrección de errores y revisión de código	1
Despliegue y configuración en la nube	1
Investigation sobre Jmeter	1
Testeo automático con Jmeter en la nube	2
Diagramas, documentación y redacción de informe	3

El desglose de actividades que se presenta no ocurrió exactamente en el orden expuesto ni las tareas fueron realizadas de manera separada. En general el desarrollo fue iterativo pero se muestra un estimado de las horas sumadas de cada tarea.