UCAB Universidad Católica ANDRÉS BELLO
Facultad de Ingeniería
Escuela de Ingeniería Informática

Proyecto FBCache

Firma
Luis Fuentes

Caracas, 15 de octubre de 2020

Índice General

Índice de figuras / tablas / apéndices

- Figuras

- Tablas

- Apéndices

Sinopsis

Para los clientes de los gestores de datos de Firebase existe un problema, el cual es la política de cobro que ejerce Firebase para mantener el servicio, dicha política establece que no habrá cargos por las operaciones de escritura que se hagan sobre estos, sino que habrá cargos por el volumen de datos extraídos por las operaciones de lectura, lo que puede traer gastos difíciles de estimar y que pueden llegar a ser muy elevados.

El Proyecto FBCache surge con el objetivo de Desarrollar una librería en NodeJS que implemente un almacenamiento en caché de la información consultada de los sistemas de gestión de datos de Firebase que permita minimizar la cantidad de información descargada de Firebase sin afectar la disponibilidad de la misma.

El desarrollo del proyecto se realizó siguiendo una metodología en cascada usando un flujo de trabajo en paralelo, siendo conformado por las fases de diseño, desarrollo y pruebas. Dejando como resultado una librería capaz de realizar operaciones de lectura y escritura sobre los gestores de datos que ofrece Firebase, y que internamente maneja un caché para ciertas rutas y/o colecciones indicadas en un archivo de configuración, que permite disminuir la cantidad de consultas directas que se realicen a Firebase, pero sin afectar a la disponibilidad que se tenga de la información almacenada en dichos servicios.

Planteamiento del Problema

Firebase es una plataforma en la nube propiedad de Google que ofrece distintos servicios para el desarrollo de aplicaciones, entre los servicios se encuentran 2 gestores de bases de datos, Firestore y Real Time Database; Google cobra a los clientes de estos por descargar la información que se encuentra almacenada en ellos, pero no cobra por la cantidad de información que se almacene, esto trae como consecuencia que el costo por mantener estos servicios dependa únicamente de la cantidad de consultas que se realicen. Si bien, que el costo dependa solamente de la cantidad de información que se extraiga se ve bastante atractivo, puede generar un costo muy variable en el mantenimiento; lo que puede traer como consecuencia que no se pueda hacer una estimación muy precisa de cuánto puede costar el servicio de almacenamiento, o que en un momento en específico se puedan generar muchas consultas que incrementen considerablemente el costo.

A todo esto, se suma que al ir creciendo la concurrencia de usuarios que usen alguna base de datos que se haya implementado en alguno de los gestores ofrecidos por Firebase, se van incrementando los efectos señalados anteriormente, llegando a ser un problema, ya que al no poder estimar bien el costo del mantenimiento del gestor de base de datos que se esté usando, y dicho costo poder ser muy elevado según el uso que se le dé, puede generar un pago excesivo.

La empresa Synergy Vision está en búsqueda de herramientas que permitan regular el costo del uso de Firestore y Real Time Database como gestores de base de datos, por lo que se quiere implementar una forma de disminuir la cantidad de consultas que se hacen directamente a estos, pero que no afecte a la disponibilidad que se tiene para acceder a la información almacenada.

Objetivo General

- Desarrollar una librería en NodeJS que implemente un almacenamiento en caché de la información consultada de los sistemas de gestión de datos Firebase

Objetivos Específicos

1. Diseñar una librería pública para la implementación de una caché de datos consultados sobre los sistemas de gestión de datos de Firebase
2. Diseñar un archivo de configuración donde se describa a qué rutas se le hará guardado en caché y especifique el tiempo de refrescamiento de la información almacenada en caché
3. Desarrollar métodos de consumo de la información almacenada en caché y refrescamiento de la información
4. Implementar un servidor de pruebas que muestre los logs de las consultas

Descripción de la Empresa

Synergy Vision es una empresa que se especializa en proveer la formación, la tecnología y la consultoría en las áreas de especulación (Trading), inversión, economía, finanzas y riesgo.

La oferta de servicios de Synergy Vision consiste en asistir a empresas con sus canales electrónicos con aplicaciones web y móvil para ofrecer servicios en las áreas descritas anteriormente. Así como el desarrollo de sistemas especializados para facilitar la gestión de inversiones, portafolios, trading, simulaciones, modelos, plataformas de trading, finanzas internacionales, valoración, optimización, riesgo, etc.

Cuenta con un equipo multidisciplinario que combina el conocimiento sobre Economía, Riesgo, Ciencias Actuariales, Finanzas, Matemáticas, Estadísticas, Probabilidades y las Ciencias de la Computación, con una Visión única sobre servicios para el sector financiero, aplicando Ciencia de los datos y Algoritmos inteligentes para el sector financiero; siendo además un equipo abierto y sensible, atento a la tecnología para mantenerse actualizado, que escucha a las necesidades de sus clientes para proveerles una solución de calidad.

Metodología Empleada

Se usó una metodología cascada con un flujo de trabajo en paralelo, esto porque, al ser una metodología que ofrece un marco de trabajo claro y simple, permite tener bastante control sobre los tiempos que se manejan para cada tarea, y más teniendo en cuenta la duración de 6 semanas del proyecto; además, al usar un flujo de trabajo en paralelo, hace que lo que se implemente durante una fase pueda ser probado en la siguiente, y en caso de necesitar algún ajuste, este se puede realizar sin romper con lo planificado.

El proyecto constó de 3 fases principales que abarcarían las 6 semanas de duración de este, las cuales fueron diseño, desarrollo y pruebas. La fase de diseño se llevó a cabo desde la primera hasta la segunda semana; la fase de desarrollo inició en la segunda semana y terminó en la sexta semana; y la fase de pruebas fue ejecutada durante la sexta semana.

Para llevar control de que se estuviera siguiendo la planificación, se utilizó un tablero de proyecto en el repositorio de GitHub donde se almacenó la librería, en este, semanalmente, el Tutor Empresarial abría Issues relacionados con las tareas a realizar durante la semana según lo planificado; además, se realizaron reuniones semanales con el Tutor Empresarial para mostrar los avances que se tenían durante el desarrollo del proyecto.

Desarrollo

Para la primera semana, según la planificación descrita anteriormente, se tenía el inicio de la fase de diseño del proyecto; durante esta se realizó una investigación a las librerías o dependencias de NPM que pudieran ayudar a la implementación de la librería y el servidor de pruebas de está, esta investigación dio como resultado que se decidiera usar en las siguientes dependencias:

- Dependencias de la librería FBCache
    1. firebase-admin (versión 9.1.1): Librería oficial de Firebase que permite conectarse a un proyecto en Firebase desde una aplicación de servidor.
    2. node-cache (versión 5.1.2): Librería que se implementó para facilitar el manejo de la memoria caché; esta se encarga de almacenar la información en la caché y asignarle un tiempo de vigencia (TTL).
    3. moment (versión 2.27.0): Para la implementación de políticas de refrescamiento de la información, se requería de un manejo de fechas un poco más avanzado que lo que brindaba el tipo de dato Date de JavaScript, entonces, se tomó la decisión de usar moment para facilitar el manejo de las fechas en la librería.
    4. uuid (versión 8.3.0): node-cache guarda la información en caché en un formato clave-valor, es decir, exige que se le asigne un identificador único a la información que se quiera guardar en caché para facilitar su consulta; entonces, para evitar problemas con identificadores repetidos, se implementó uuid para asegurar la asignación de un id único a cada ruta indicada para hacerle un almacenamiento en caché.
- Dependencias del servidor de pruebas:
    1. express (versión 4.17.1): Framework diseñado para crear aplicaciones web del lado del servidor.
    2. winston (versión 3.3.3): Librería para el manejo de logs.
    3. morgan (versión 1.10.0): Middleware que permite implementar logs del resultado de cada petición HTTP que se realice al servidor.

Además, tanto para la librería como para el servidor de pruebas se usó babel en su versión 7.11.

A partir de la elección de estas dependencias, se comenzaron a realizar los flujos que deberían seguir cada uno de los métodos de la librería para realizar cualquier operación de lectura y escritura sobre Firestore, Real Time Database y la memoria caché, además de realizar la firma de estos métodos y un diseño inicial del archivo de configuración de la librería, la firma de los métodos obtenidos se muestra a continuación:

- Inicialización: FBCache.init(config, projectURL, credentialType, credential)

  Este método permite inicializar la librería, leyendo desde el parámetro *config* el contenido del archivo de configuración, y obteniendo la información necesaria para conectarse al proyecto en Firebase en el resto de los parámetros; el url del proyecto a conectarse en *projectURL*, el tipo de credencial en *credentialType* (este pudiendo ser "file" en caso de que se tenga un archivo de credenciales del proyecto obtenido desde Firebase, o "token" en caso de querer conectarse a través de un token OAuth), y en *credential* la credencial con la que se quiere autenticar al momento de conectarse a Firebase.

- Consulta: FBCache.get(dbms, route)

  Este método permite realizar consultas sobre el gestor de datos indicado en el parámetro *dbms*, en caso de ser sobre Real Time Database, se consultaría la ruta indicada en el parámetro *route*, en caso de ser sobre Firestore, el parámetro route indicará el nombre de la colección a consultar. El método consulta primero si la información puede ser extraída desde la memoria caché, y en caso que no se pueda, consultará directamente al gestor de datos indicado

- Inserción: FBCache.insert(dbms, route, data, id)

  Este método permite realizar inserciones sobre el gestor de datos indicado en el parámetro *dbms* y *route* cumplen la misma función que en el método de

consultar, en el parámetro *data* se obtiene la información que se quiere guardar y el parámetro *id* indica el identificador que se le quiera asignar a la información, en caso de no indicar nada en este último parámetro, el gestor de datos asignará un identificador automáticamente. Dependiendo de si la ruta o colección indicada se le está realizando un almacenamiento en caché, se actualizaría automáticamente lo almacenado en caché.

- Actualización: FBCache.update(dbms, route, data, id)

  Este método permite realizar modificaciones sobre la información almacenada en el gestor de datos indicado; los parámetros *dbms* y *route* cumplen la misma función de los métodos anteriores, *data* contiene la información que se quiere actualizar y el parámetro *id* contiene el identificador a donde se quiere realizar la modificación. Dependiendo de si la ruta o colección indicada se le está realizando un almacenamiento en caché, se actualizaría automáticamente lo almacenado en caché.

- Eliminación: FBCache.delete(dbms, route, id)

  Este método permite eliminar registros almacenados en el gestor de datos indicado; los parámetros *dbms* y *route* cumplen la misma función de los métodos anteriores, *id* en este caso contiene el identificador que se quiere eliminar. Dependiendo de si la ruta o colección indicada se le está realizando un almacenamiento en caché, se actualizaría automáticamente lo almacenado en caché.

Para los métodos de consulta, inserción, actualización y eliminación se realizaron unos diagramas para ayudar a la implementación de estos y complementar la documentación de la librería. En la parte de apéndices puede conseguir estos diagramas.

La segunda semana de pasantías marco el fin de la etapa de diseño y el comienzo de la etapa de desarrollo. Para el final de la etapa de diseño se terminó el diseño del archivo de configuración de la librería, el cual termino con la siguiente estructura:

```
{
    "read_only": boolean,
    "firestore": Array[{}],
    "realtime": Array[{}],
    "max_size": string,
}
```

Figura 1 - Estructura general del archivo de configuración

read_only es un booleano que indica si se quiere habilitar las operaciones de escritura sobre los gestores de datos de Firebase, si se indica como “true”, las rutas en general tendrán activada la política de solo lectura, en caso de ser “false” se podrán realizar operaciones de escritura, y si no se indica por defecto tomará el valor “true”.

max_size tiene la función de establecer un límite de almacenamiento a la librería, el cual mientras no sea sobrepasado se podrán realizar actualizaciones a lo almacenado en la memoria caché, y cuando se sobrepase no se podrán realizar ninguna escritura sobre caché hasta que lo que esté almacenado en caché sea menor al límite establecido, y si no se indica este atributo la librería no tendrá limites en lo que pueda almacenar en caché; para indicar un límite se le asigna un string que tenga la unidad de almacenamiento que se desee, siempre y cuando esa unidad sea bytes (B), kilobytes (kB) o megabytes (MB), junto con la cantidad que se desee asignar, es decir, si se quiere asignar un límite de 50 megabytes, el string debe tener la siguiente forma: “50 MB”.

Los atributos *firestore* y *realtime* tienen la misma estructura, un arreglo de objetos, en estos se va a almacenar las distintas colecciones o rutas a las que se les quiere realizar un almacenamiento en caché. Los objetos que se pueden almacenar en estos arreglos deben contener la siguiente estructura:

```
{
    "name": string,
    "refresh": string,
    "period": string,
    "start": string,
    "read_only": boolean
}
```

Figura 2 - Estructura de los objetos del archivo de configuración

name indica la ruta de Real Time Database o la colección de Firestore a la que se le quiere realizar el almacenamiento en caché. *read_only* tiene el mismo funcionamiento que el atributo *read_only* de la estructura general del archivo de configuración, solo que este se aplica específicamente en la ruta, en caso de estar definido, la ruta tendrá la política de escritura que indique su atributo *read_only* particular, en caso que no, toma el valor del *read_only* general.

period y *refresh* indican el tiempo de refrescamiento que va a tener la información guardada en caché de dicha ruta; con la única diferencia de que cada indica una política de refrescamiento distinta, si se define el refrescamiento de la ruta con *refresh*, indica que cada vez que se tenga que actualizar la información, se asigna esa cantidad fija de tiempo a la información que se va a guardar en caché; en cambio, *period* indica que el refrescamiento se realizará basado en un periodo de tiempo (cuyo inicio se define en el atributo *start*), es decir, se define un inicio del periodo, y según la cantidad indicada, cuando se vaya a actualizar la información en caché, se determina cuanto falta para el siguiente momento que defina el periodo y establece el tiempo de vigencia de la información en esa cantidad de tiempo faltante. Cabe acotar que si en una ruta se define *refresh* no se podrá definir ni *period* o *start,* y viceversa.

Basado en lo realizado en la fase de diseño, se comienza la fase de desarrollo en la misma segunda semana con la implementación del método de inicialización de la librería, método el cual fue probado más a fondo en la tercera semana, momento en el que se implementó el método de consulta y el servidor de pruebas; para facilitar el

seguimiento del Tutor Empresarial del desarrollo del proyecto, se decidió hacer el despliegue de la implementación del servidor de pruebas en Heroku.

Entre la cuarta y quinta semana se implementó los métodos faltantes de la librería, teniendo lugar en la cuarta semana la implementación del método de inserción y en la quinta la de los métodos de actualización y eliminación junto a la implementación de los logs en el servidor de pruebas. Sin embargo, en la quinta semana se decidió implementar una mejora a la librería, la cual consistía en adaptar la firma de la librería a una similar a la implementada por la librería firebase-admin, con el objetivo de facilitar la adaptación de la librería en proyectos donde se implemente la librería firebase-admin; por lo que, para no afectar el diseño que se había realizado de la librería, se implementó una fachada que implemente esa firma similar a firebase-admin, y que internamente usé los métodos ya desarrollados; para entender este punto de mejor forma, se realizó el siguiente diagrama que retrata la estructura que fue implementada en la librería y la relación entre la firma original de la librería y la firma implementada en la fachada:

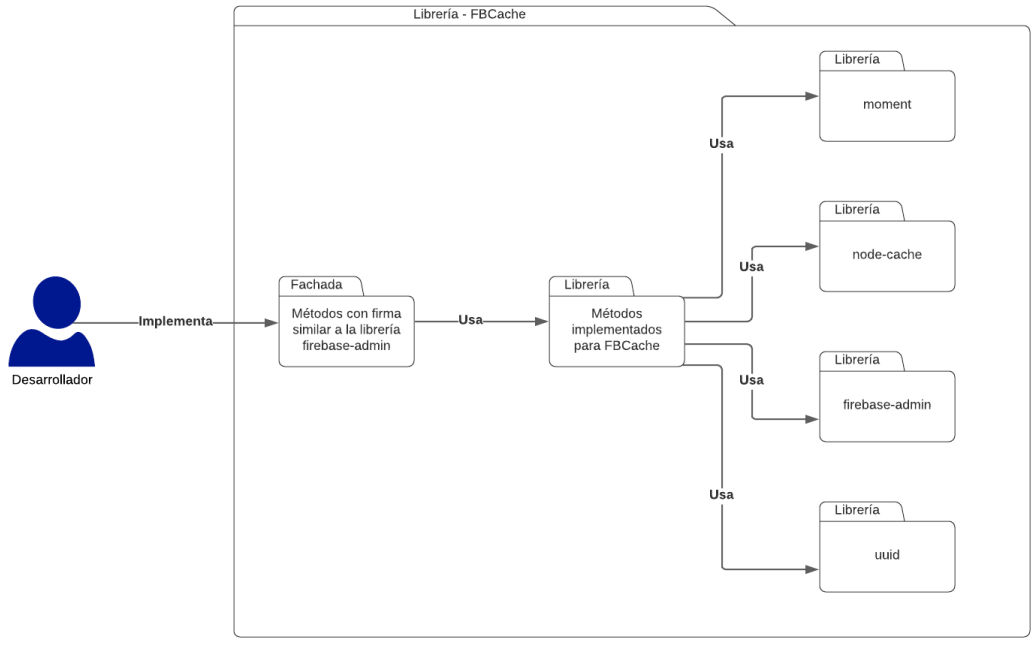


Figura 3 - Estructura de la librería FBCache

	Firma antigua	Firma nueva
Inicialización	FBCache.init(config,projectURL,credentialType,credential)	initFBCache(config,projectURL,credentialType,credential)
Consulta	FBCache.get(dbms,route)	FBCache.database().ref(route).once()
		FBCache.firestore().collection(route).get()
Inserción	FBCache.insert(dbms,route,data,id)	FBCache.database().ref(route).child(id).set(data)
		FBCache.database().ref().child(route).push(data)
		FBCache.firestore().collection(route).doc(id).set(data)
		FBCache.firestore().collection(route).add(data)
Actualización	FBCache.update(dbms,route,data,id)	FBCache.database().ref(route).child(id).update(data)
		FBCache.firestore().collection(route).doc(id).update(data)
Eliminación	FBCache.delete(dbms,route,id)	FBCache.database().ref(route).child(id).remove()
		FBCache.firestore().collection(route).doc(id).delete()

Tabla 1 - Comparación entre la firma original de la librería y la implementada

Para la última semana del proyecto, se decidió usar un framework para la automatización de pruebas que nos permitiera realizar pruebas E2E; el framework elegido fue Jest, y después de realizar las distintas pruebas se obtuvo el siguiente resultado de cobertura de código en la librería:

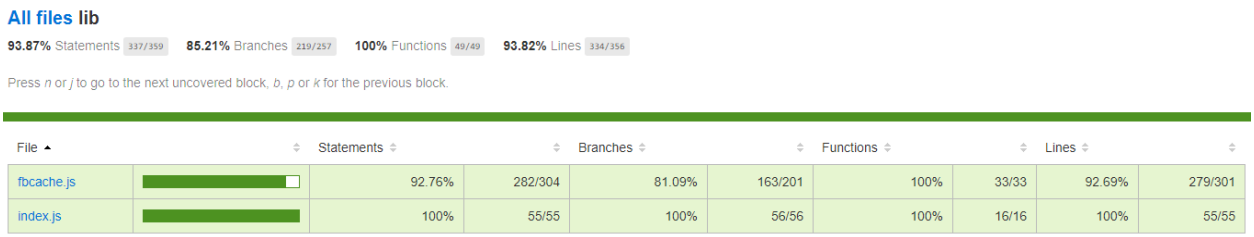


Figura 4 - Resultados del coverage de las pruebas E2E implementadas

Además de estas pruebas, también se empezaron a realizar pruebas E2E manuales usando el servidor de pruebas, y en estas pruebas se vieron unos pocos detalles que se arreglaron rápidamente para culminar la fase de desarrollo y la fase de pruebas al final de la sexta semana.

Resultados

Se obtuvo una librería para aplicaciones del lado del servidor implementadas en NodeJS, que permite establecer conexión a un proyecto en Firebase y realizar consultas a los datos almacenados en Real Time Database o en sus colecciones en Firestore y escribir información en estas; con el agregado de que la librería maneja un almacenamiento en caché interno de rutas a Real Time Database o colecciones de Firestore que se indiquen en un archivo de configuración para la librería, cosa que permite que la información se consulte desde la caché en los casos en que la información esté vigente, y que con cada consulta realizada a una ruta indicada, automáticamente se refresque la información, esto ocasiona que se pueda minimizar la cantidad de consultas directas a los gestores de datos de Firebase, y a su vez los gastos en los que se pueda incurrir por la cantidad de consultas realizadas sin afectar la disponibilidad de la información almacenada en estos gestores. Además, se pudo implementar la librería en un servidor de pruebas, que a su vez fue desplegado en Heroku que muestra el funcionamiento de la librería.

Conclusiones y Recomendaciones

Se realizó el diseño de la librería durante las 2 primeras semanas del proyecto, esto sirvió como base para la etapa de desarrollo, en la cual se realizó la construcción de la librería y, a su vez, se implementó y desplegó una aplicación de servidor en donde se hace uso de la librería para probar su funcionalidad, esto fue fundamental en la etapa de pruebas, en donde, además de probar las funcionalidades de la librería en el servidor de pruebas, también se implementaron pruebas E2E usando un framework de pruebas automatizadas que ayudaron a comprobar que el comportamiento de la librería era el esperado. En general, se pudo obtener una librería que puede ser descargada desde su repositorio público en GitHub que cumple con los objetivos previstos para el proyecto realizado y cuya documentación sobre su funcionamiento y método de uso está reflejada en el mismo repositorio de la librería.

Pensando en un futuro, sería recomendable implementar en la librería un manejo propio de la memoria caché y de las fechas que no dependa de las librerías implementadas, esto con la intención de que la librería sea lo más autónoma posible y su funcionamiento no se vea comprometido por cambios que se le puedan implementar a las dependencias usadas.

Referencias Bibliográficas

Babel. (15 de octubre de 2020). https://babeljs.io/

Express. (15 de octubre de 2020). https://expressjs.com/

Firebase. (15 de octubre de 2020). https://firebase.google.com/docs?hl=es

Jest. (15 de octubre de 2020). https://jestjs.io/

Moment. (15 de octubre de 2020). https://momentjs.com/

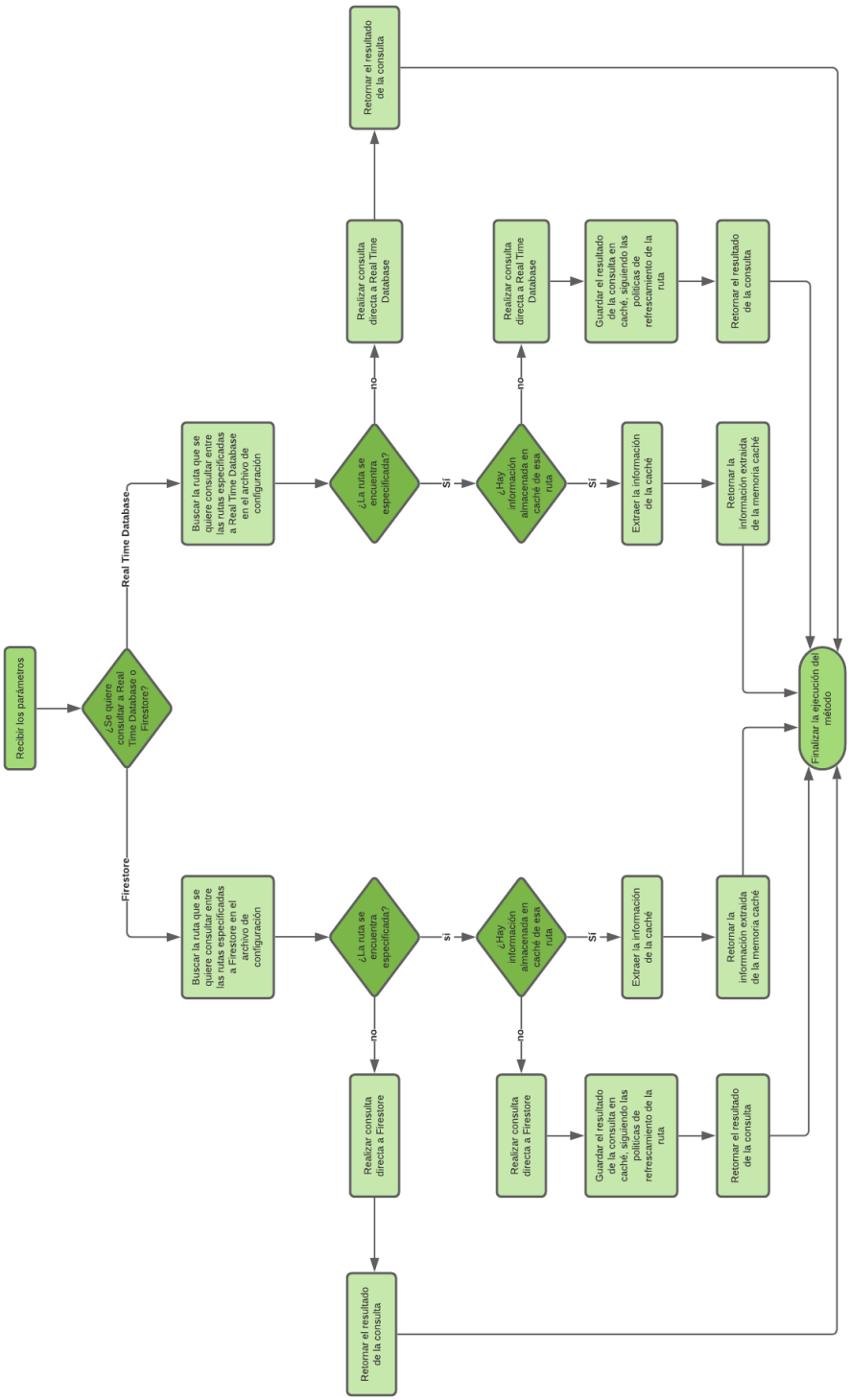
Morgan. (15 de octubre de 2020). https://www.npmjs.com/package/morgan

Node-Cache. (15 de octubre de 2020). https://www.npmjs.com/package/node-cache

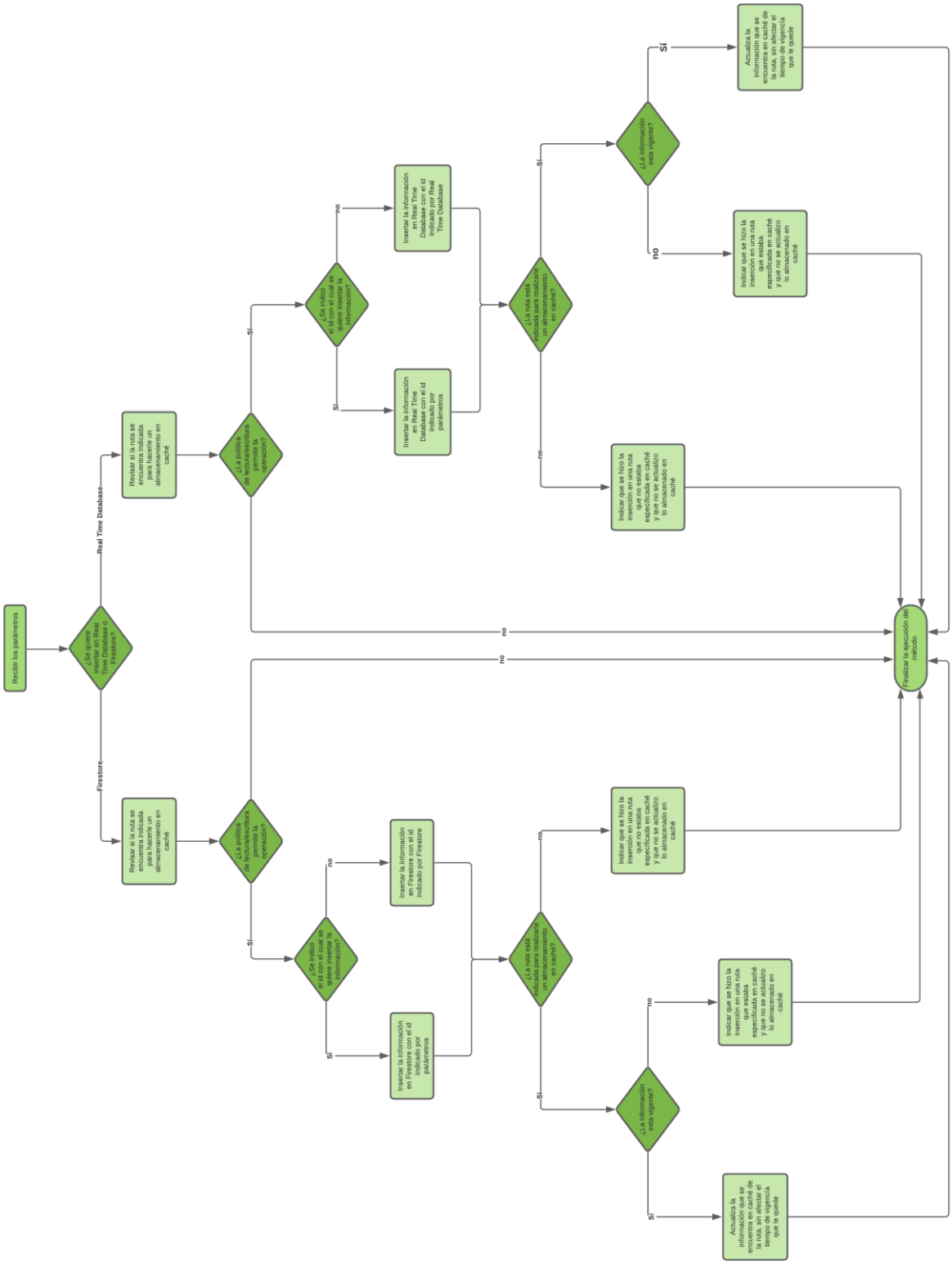
uuid. (15 de octubre de 2020). https://www.npmjs.com/package/uuid

Winston. (15 de octubre de 2020). https://www.npmjs.com/package/winston

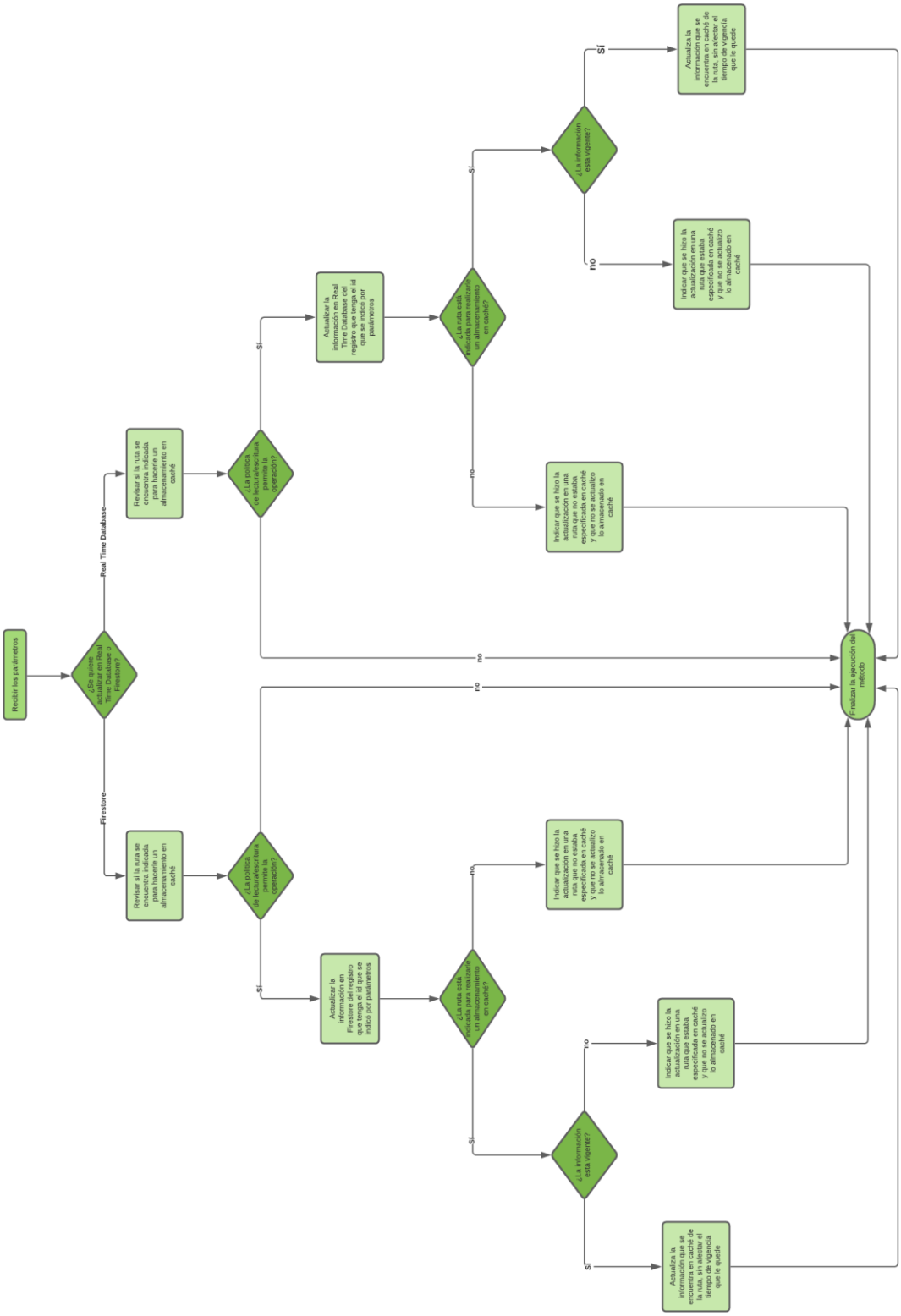
Apéndices



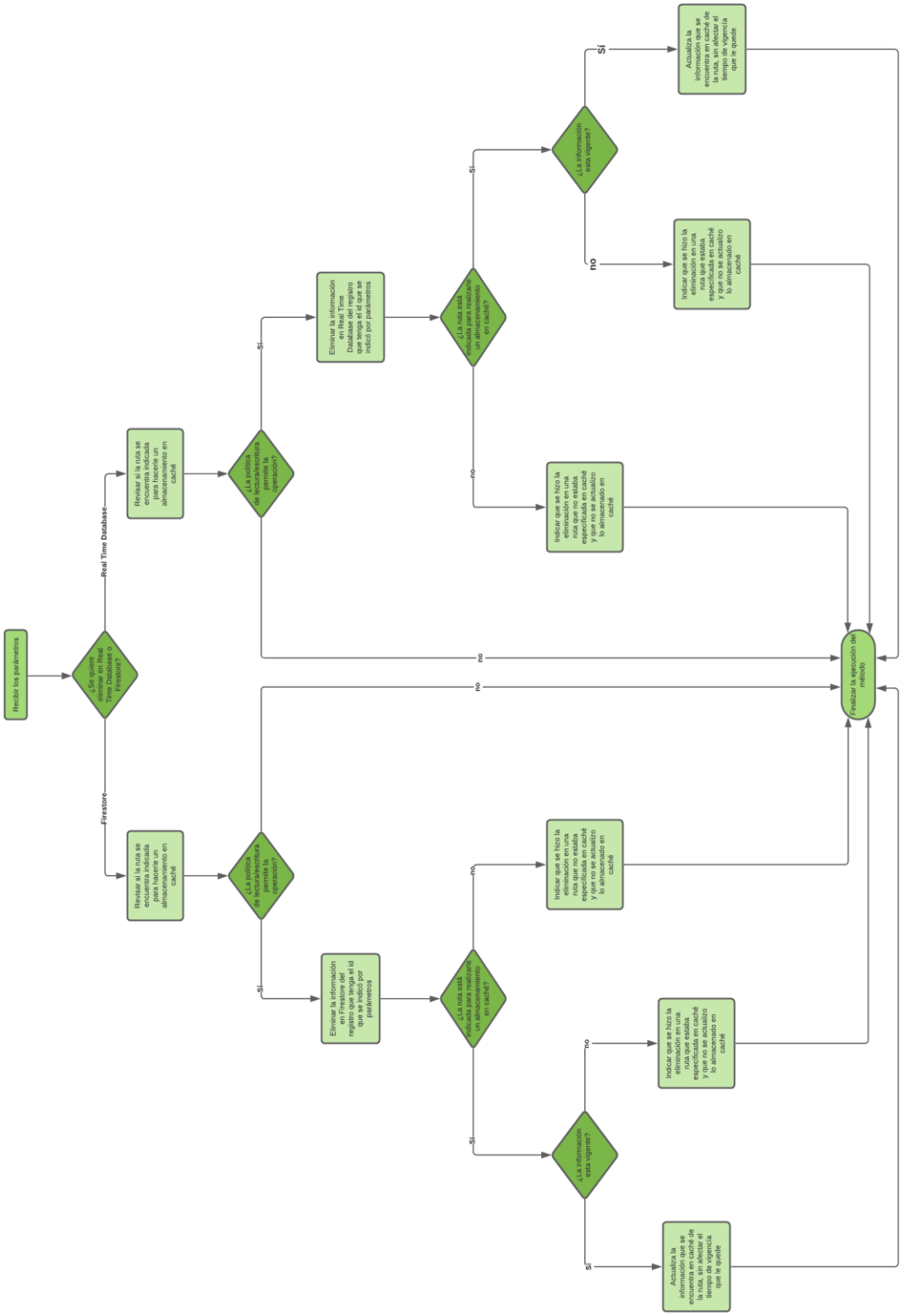
Apéndice 1 - Flujo del método de consulta



Apéndice 2 - Flujo del método de inserción



Apéndice 3 - Flujo del método de actualización



Apéndice 4 - Flujo del método de eliminación