Pablo A. Martínez Follow
Entrepreneur & CEO at GreenLionSoft · Android Lead @MadridMBC & @Shoptimix · Android, OpenSource and OpenData promoter · Runner · Traveller
Feb 24 · 8 min read

Lessons learnt (the hard way) using Firebase RealTime Database

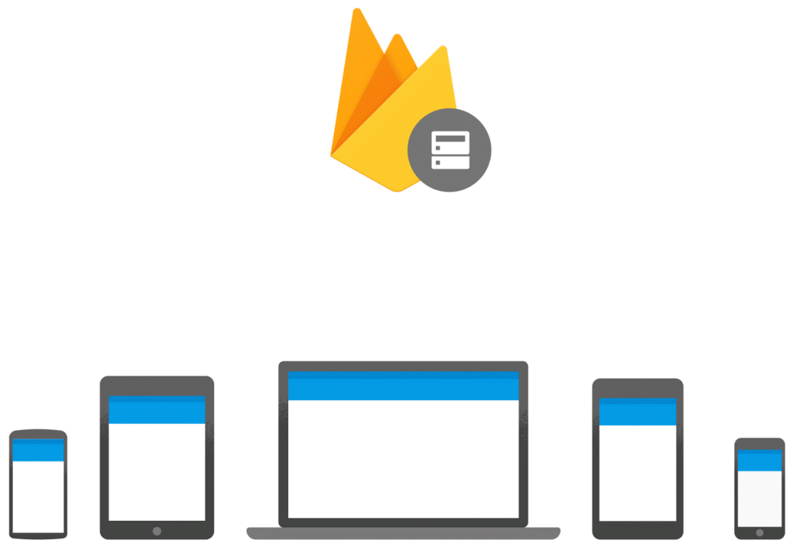
TL;DR

Last year at GreenLionSoft we decided to progressively ditch various of the goodies provided by AWS and go "all in" into the Google's Firebase suite.

In this article I want you to learn some critical tips to avoid the errors we committed while integrating Firebase Realtime Database that resulted in a unexpected monthly bill of more than **1.000€.**

· · ·

Firebase Realtime Database for what?



As you probably can imagine the “RealTime Database” is a solution that stores data in the cloud and provides an easy way to sync data/state among various devices.

Up to the date we had a Spanish Transit app (Madrid Metro|Bus|Cercanías) in our portfolio that allowed users to save different kinds of “favorites” like bus stops, routes, etc… Those favorited items were saved locally in the device serialized as JSON strings. Not a big deal.

One typical complain we received, specially during Christmas when lots of new phones are given as presents, was:

> *“hey guys, I have a brand new phone, Is there a way to migrate my favs from my old phone to the new phone?”*

So here Firebase came to the rescue, to accomplish what users were requesting we needed two things, being able to identify users and being able to store their favorites into the cloud. The deadline was set, we had to finish the implementation before the end of December 2017 (Christmas holidays)

Integrating Firebase

Us working on the integration of Firebase Auth and Realtime Database

1. Firebase Authentication

To be able to identify users we had to integrate Firebase Auth and as our app was only available only on Android we only enabled **Anonymous** users and **Gmail** as “auth providers”.

You may wonder why we allow **anonymous** users, the explanation is easy and we recommend you to do the same. When a new user installs your App, you don’t want him to be bothered by a login flow, you want the user to discover the app straight and let him realize that the features provided are worth enough to keep the app installed. Later on, when this user will need to migrate their favorites he himself will realize that he need to provide a mean to link his data to this identity and will login willingly.

We didn’t have major issues integrating Firebase Auth you just have to keep in mind that when the user executes the **Gmail** authentication you don’t have to create a new Firebase User but, link the provided credentials to your existing Anonymous Firebase User otherwise he will lose all his data.

2. Firebase RealTime Database

The database provided by Firebase is the typical NoSql schema where data is stored like a JSON Tree. Following the docs we structured the data in a flat fashion to avoid nesting data which would incur in downloading more data thus higher bills, we will talk about that later…

Here is an example of our database structure:

```
/users
    /userId1
       userEmail: "123email@gmail.com"
       userName: "Shally"
       ...
    /userId2
       userEmail: "xyzemail@gmail.com"
       userName: "John"
    ...


/favorites
    /userId1
       favoritesBusSection : "Serialized JSON object"
       favoritesMetroSection : "Serialized JSON object"
       ...
    /userId2
       favoritesBusSection : "Serialized JSON object"
       favoritesMetroSection : "Serialized JSON object"
       ...
    ...
```

Our existing implementation to retrieve and save favorites , following Clean Code Practices, was decoupled using an interface (contract), so we just needed to provide a new implementation of this interface using the Firebase SDK. **No hassle.**

If you don't know what I'm talking about you should consider looking for the topics "Clean Code" and "Dependency injection", here is an example of an interface that can be implemented by a class that uses Firebase or another class that uses e.g. local sqlite database.

```
interface IFavoritesProvider {

fun saveBusFavorites(favorites: List<BusFavorite>)

fun retrieveBusFavorites(): Single<List<BusFavorite>>

...

}
```

Later on you just have to choose which implementation your dependency injector should provide into your app. If one day Google decides to quit Firebase (**this has happened before with other projects**) your code will be ready to migrate to a new cloud provider with no pain (or at least with less pain)

So, we were at the beginning of December, the app was fully working and tested and the we decided to release it.

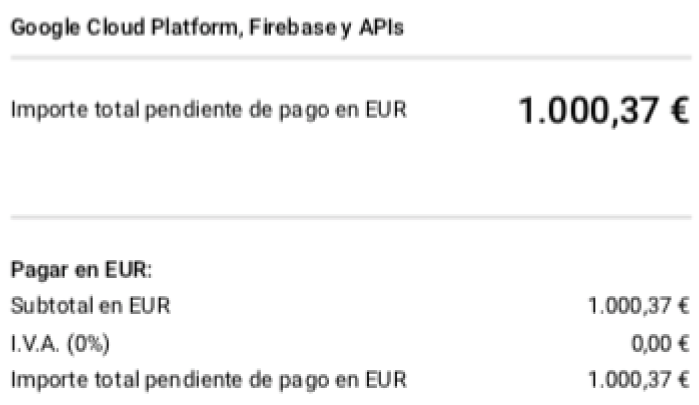
The day we released the app integrating Firebase Auth and Realtime Database

. . .

Everything is OK, or not…

December days were passing and little by little our more than **400K+** user base updated the app, authenticated and moved their favorites from the device to de cloud. No crashes, no issues, good user ratings, we went to holidays with no stress and ready to enjoy ourselves :)

Then the new year arrived, and with that our first Google Cloud Invoice: **1.000,37€**



Google Cloud Platform, Firebase y APIs

Importe total pendiente de pago en EUR	**1.000,37 €**

Pagar en EUR:

Subtotal en EUR	1.000,37 €
I.V.A. (0%)	0,00 €
Importe total pendiente de pago en EUR	1.000,37 €

1.000,37 € WTF!!

Our face when we got the invoice

You may think that 1.000 € is not much but we are a 2-guy self-funded start-up and to put it in perspective, in Spain, where we are based, that is more than the minimum salary that is 735€. If you think in other countries in South America, Africa or Asia the situation is even worst, because Firebase pricing is the same for everybody. Besides that, not all our user base had already migrated meaning costs would go higher and higher.

Learning the hard way

Firebase pricing

Pricing is quite simple $5 per GB stored and $1 per GB downloaded. (Note that SSL handshake, protocol and encryption is also counted in downloaded data). Uploading data to the cloud is completely free.

Our full database size was only 750 MB so 99,5% of the invoice was due to downloaded data. Aprox. 1000 GB of downloaded data. It meant that the full content of the whole database was downloaded up to 44 times everyday, that made no sense… but It finally did.

Wrong Firebase Database assumptions

We assumed that as most of our users usually have just one device, dowloaded data would negligible because data in the cloud would just work as a backup until the user changes of device.

Firebase Database SDK works offline if we enable persistence:

```
val database = FirebaseDatabase.getInstance()
database.setPersistenceEnabled(true)
```

Enabling persistence allows normal operation of the app while the user is not connected to the internet, Firebase keeps a local cache with the favorites and if the user makes any change, changes will be propagated to the cloud next time internet is available.

We also enabled an extra feature "**keepSync**", we thought that syncing the database node containing all user features will speed up app UX. That is probably true, **but this was our terrible error and the major culprit of the 1000€ invoice.**

```
database.getReference(getUserFavoritesPath(getCurrentUid()))
.keepSynced(true)
```

We thought that Firebase SDK was *intelligent enough* to avoid downloading data from the Cloud if the data hasn't changed, but what was happening was that a full copy of all favorites was dowloaded every time the user started the app. **And billed, of course.**

But the problem was even worse.

Discovering that the problem was even worse…

In any section that involved possible favorites interaction we requested the corresponding node of favorites to Firebase, (and if the user made any change then we sent back the new data to the Cloud)

This data request was also dowloaded from the internet, even it we had persistence enabled and already synced all favorite nodes. **Again, all this data was billed, increasing the final amount of the invoice.**

Database / Favorites size

Looking at the data itself stored in the database we realized there was an obvious an easy optimization:

```
[
  {
    "description":"Home",
    "stopId":"12345"
  },
  {
    "description":"Work To Home",
    "stopId":"45678"
  },
  {
    "description":"School",
    "stopId":"01357"
  }
]
```

Just renaming the serialized var names saved tons of bytes:

```
[
  {
    "d": "Home",
    "s": "16606"
  },
  {
    "d": "Work To Home",
    "s": "16606"
  },
  {
    "d": "School",
    "s": "16606"
  }
]
```

In this example from a serialized string of 131 chars we went to 86 chars which means a 34% size saving. **Savings directly translated to your invoice.**

Our solution

Then we came up with the following solution:

1. Optimize serialized data saved in the database, this can be achieved easily with GSON, a library to handle JSON data that allows changing the name of serialized entities with annotations. (There are many other libraries that can do the same)
2. We disabled "keepSync" on the favorites node, don't use it, never. (unless you are sure that syncing this data is worth enough)
3. We implemented a memory cache for used favorites that is kept only during the life of the App.

This cache works as follows:

- User opens the app and goes somewhere in the app where any kind of favorites is listed, let's say Bus Stop favorites.
- The App requests to the memory cache a list of Bus Stop favorites but as the App has just been launched it returns **null.**

- Then the App requests to Firebase only the node containing Bus Stop Favorites, which are downloaded from the internet and **then billed once.**
- A copy of this favorites is saved in the memory cache.
- User plays with favorites, adding, removing or editing them.
- Every time a modification is done the resulting favorites list is saved in the memory cache and also sent to Firebase (because uploading data is free).
- Now the user can leave the App, and unless it is explicitly removed from the recent App stack by the user or killed by Android, the user can come back to the App and continue playing around with the Bus Stop favorites without incurring in extra billing.

This implementation despite being "download/money saver" has a drawback to take in account:

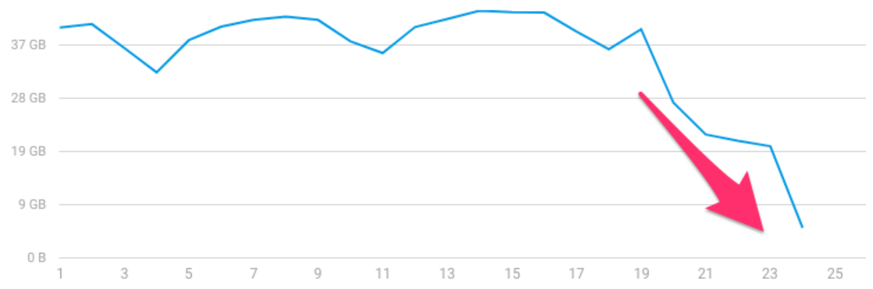
- If the user is in the unlikely scenario of editing same favorites in 2 or more devices at the very same time they won't actually sync until all Apps are fully restarted.

Some pseudo code to ilustrate :

Our FavoritesProvider Implementation would be like that:

The results

After implementing all this counter measures we could see straightforward in the Firebase Console how dowloaded data decreased drastically:



Downloaded data going from more than 40GB to less than 10GB

Can this solution be improved?
Of course, implementing a local disk cache would be the next step in parallel with some kind of timestamp or hashing system to check if remote data is fresher than the local one before downloading.

Summary of lessons learnt (the hard way)

- If your user base is huge, Firebase Realtime database can be costly if your database schema and retrieval logic is not optimized to minimize the amount of downloaded data.
- Try to minimize the size of data stored in the cloud, using small var names for your data classes / models can save you lots of money.
- When Firebase Persistence is enabled, Firebase only will use its local cache when internet is not available, otherwise it will get the data from the cloud even if it hasn't changed.

- Avoid using “keepSync(true)”, this will always download a copy of the synced node even if it hasn’t changed.
- Implement any kind of local cache to avoid requesting twice same data to Firebase.

We deserved some beer after all what we had learnt

• • •

The end!

Yes, you have finally reached the end of this long article! I hope you found it useful. If you find any error or you have any doubt please don’t hesitate to make any comment, I’ll do my best to try to help and if you liked it please share it, follow me or even clap me! Thanks!