CKTrends

A system for tracking your CloudKit database

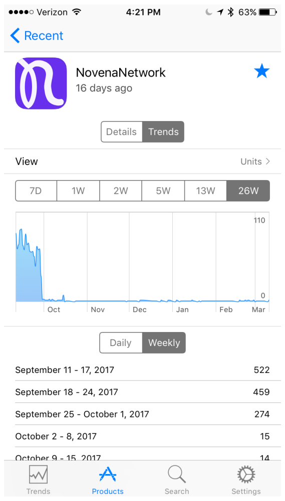
Marissa Le Coz
Dartmouth College
Advisor: Dr. Charles Palmer
Fall 2017 / Winter 2018
COSC 297

I. Motivation

In May 2017, my first app, NovenaNetwork, was accepted by the App Store. As is the case for any new iOS developer, I was very excited to monitor my app's downloads and watch my user base grow. I soon discovered Apple's "iTunes Connect" app, which allows you to see the number of downloads your app gets each day. Although iTunes Connect was slow and inconsistently updated, I was still able to get a snapshot of how NovenaNetwork was doing. That is, until late August 2017.

One day in August, I opened iTunes Connect and found that NovenaNetwork had apparently been downloaded over 1000 times in one day in China. This was a major outlier compared with the rest of my statistics. Had NovenaNetwork gone viral in China? This seemed unlikely, so I did some research.

Apparently there are cheater companies that some iOS developers pay to download and rate their apps. These cheater companies have tons of virtual devices that they use to download an app many times. To mask what they are doing, the cheater companies additionally download a bunch of free apps, like NovenaNetwork. Sure enough, when I checked my CloudKit [1] database for NovenaNetwork, there most certainly were not 1000+ new User records, supporting the hypothesis that the "new users" I saw in iTunes Connect were not real users at all.



September 11 - 17, 2017	522
September 18 - 24, 2017	459
September 25 - October 1, 2017	274
October 2 - 8, 2017	15
October 9 - 15, 2017	14

Figure 1 iTunes Connect app

Once NovenaNetwork's statistics began to be skewed by the cheater companies' nefarious work, I could no longer rely on iTunes Connect for trustworthy information. Whenever I wanted to gauge user activity on NovenaNetwork, I had to go straight to my CloudKit database dashboard online and execute a query for new users, new posts, etc. This was not ideal for a multitude of reasons. First, I really needed to use a laptop to properly view the CloudKit dashboard, so I could not check my statistics on the go, as I could with iTunes Connect. Second, drawing conclusions from the database required a lot of mental processing on my part; databases on their own do not display data in any visually enlightening way. Third, query results in the dashboard were limited to a certain number of records (which I did not realize until much later), so it was impossible to get a full snapshot of activity on NovenaNetwork.

II. Solution

I realized that the only way that I could get a good representation of NovenaNetwork's true trends would be to create a visualization app that would somehow have access to NovenaNetwork's CloudKit database. CKTrends is this app.

[1] CloudKit is Apple's built-in cloud database for apps built on Apple platforms. CloudKit database are, for the most part, one-to-one: one app, one CloudKit database.

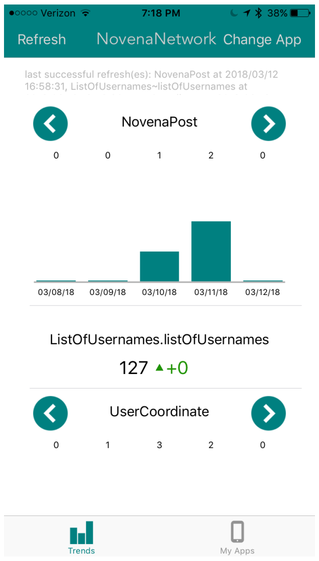


Figure 1 CKTrends interface for NovenaNetwork

CKTrends allows developers to track any custom record types in their CloudKit database. For each record type, CKTrends displays a bar graph showing the number of instances of that record type created on each day. Developers can scroll through dates all the way to the very first date that an instance of that record type was created. Not only can any number of record types be monitored for any given app, but also any number of apps (each with their own CloudKit database) can be tracked by CKTrends.

Additionally, developers can track the number of elements in a list that is an attribute of some record type, assuming exactly one instance of this record type in the database. This is useful if the developer has structured his/her database such that there is some record that contains a "master list" of some important elements.

The CKTrends concept sounds straightforward enough, but now onto the real question: how does the data get to CKTrends? Herein lies the bulk of the system.

III. System Design

Developers who wish to use the CKTrends app must install the CKTrends API, available on Github - https://github.com/mlecoz/CKTrendsAPI - and integrate it into the code for the app they wish to track. The README on Github walks developers through exactly how get their app set up to use CKTrends. The API calls themselves are very simple – just an instance variable declaration in the project's AppDelegate file (the code entry point for apps that have just been opened), an initialization of that variable that includes the record types and lists that the developer wants to track, and a function call that kicks off the refresh process (discussed in detail shortly).

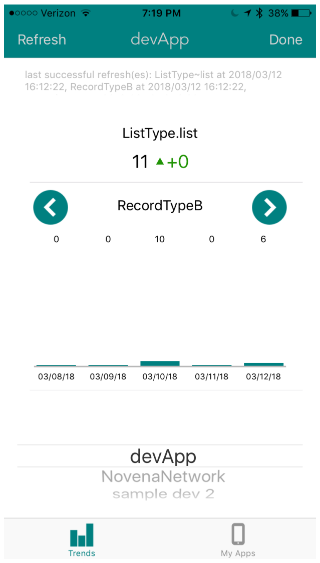


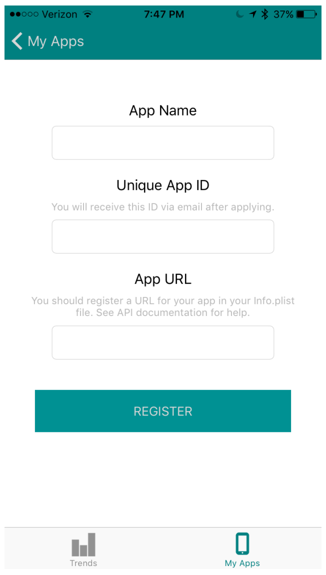
Figure 2 CKTrends supports developers with multiple apps.

Once developers have used the README on Github as a guide to configure their project to use CKTrends, they must "refresh" the trends for their app for the first time. To do this, they open the CKTrends app, ensure that the app they wish to refresh is the currently selected app (using the "Change App" button), and tap "Refresh."

During the set-up process, the developer had generated a local URL for the app s/he wished to track. Local URLs are how apps are able to open other apps on a device. As part of the onboarding process in the CKTrends app (described in detail in the README), the developer had to enter this local URL, which was then stored for later use. Well, the time for later use is now.

Upon tapping Refresh, the CKTrends app finds the local URL for the currently selected app and opens it. When the developer's app is opened, a certain method in the AppDelegate is invoked. (This method is one that Apple created. Its use is optional, but if added to the AppDelegate, it is invoked every time the app is opened by another app. Its parameters provide the local URL of the calling app.) The developer was instructed in the README to add this method to his/her AppDelegate, and in its body, to make the `appWasOpened` CKTrends API call. The `appWasOpened` method checks that CKTrends was the calling app.

If CKTrends was the calling app, then `appWasOpened` kicks off a series of function calls that read information from the developer's CloudKit database. If it is the developer's first time refreshing CKTrends for this app, every record is queried for each record type that the developer wants to track. If, on the other hand, the developer has refreshed CKTrends for this app before, only the records created after the last refresh will be queried. (Hybrid situations are also accounted for—consider the case where the developer updates their API call to include tracking of additional record types in subsequent releases of their app. The CKTrends API knows when it should query all records of a type and when it should just query records after a certain date/time, as the date/time of the last refresh is stored on a record-type-by-record-type basis. The CKTrends API can also handle the removal of record types as well as record type names that do not exist.)



App Name

Unique App ID

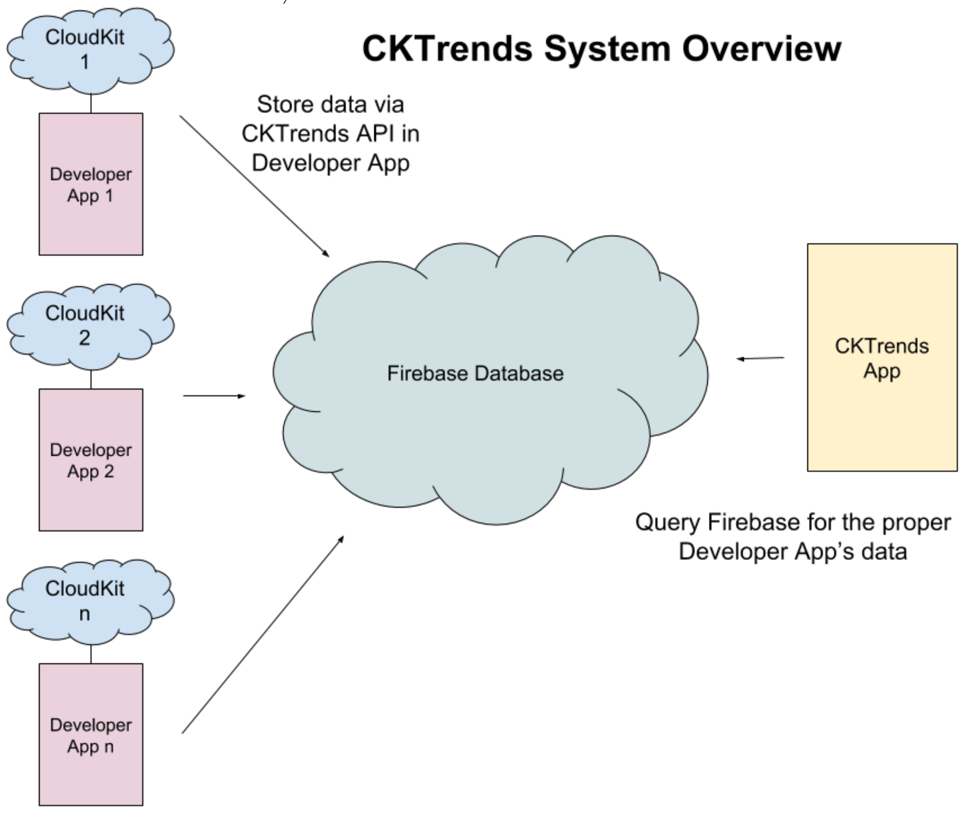
You will receive this ID via email after applying.

App URL

You should register a URL for your app in your Info.plist file. See API documentation for help.

REGISTER

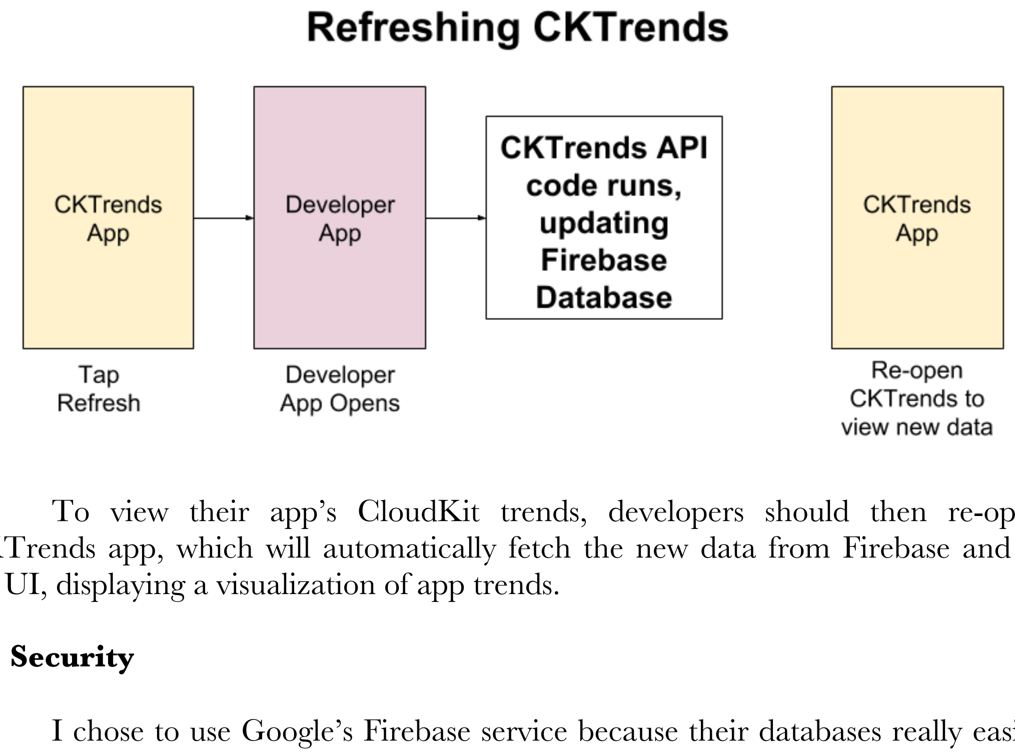
Figure 4 Users must enter their app's local URL (among other information) as part of the onboarding process.



Minimalistic data are stored in a Google Firebase Realtime database based on these queries. For each date on which a queried record was created, the record type and its total number of instances created that day are stored. As metadata of sorts, the following information is also stored:

- "LAST_CHECK" – For each record type, the date/time when CKTrends was last refreshed.
- "MAX_COUNT" – For each record type, the maximum count for any one day. This is used to generate the bar graph scale for the UI in the CKTrends app.
- "EARLIEST_DATE" – For each record type, the first date on which an instance was created. This is used to define the x-axis limits of the bar graph in the CKTrends app.
- "TRACKING" – The record types and list types being tracked. This is used for figuring out how many sections to display in the CKTrends UI.
- "TOTALS" – For each list type being tracked, the number of elements in that list.
- "DELTAS" – For each list type being tracked, the net gain or loss since the last check. This is displayed in the CKTrends app UI.

If an error occurs while the CKTrends API is working, a simple error message is displayed in an alert that overlays the developer's app's UI, in order to interfere only minimally with the developer's app. Complex error handling steps are not taken, again, to avoid interference. In these cases, the developer should attempt the refresh again, or make the changes indicated in the error message.



To view their app's CloudKit trends, developers should then re-open the CKTrends app, which will automatically fetch the new data from Firebase and render the UI, displaying a visualization of app trends.

IV. Security

I chose to use Google's Firebase service because their databases really easily lend themselves to being shared by many apps. After all, this database needs to be shared by the CKTrends app and all the developer apps that are being tracked. This may lead to

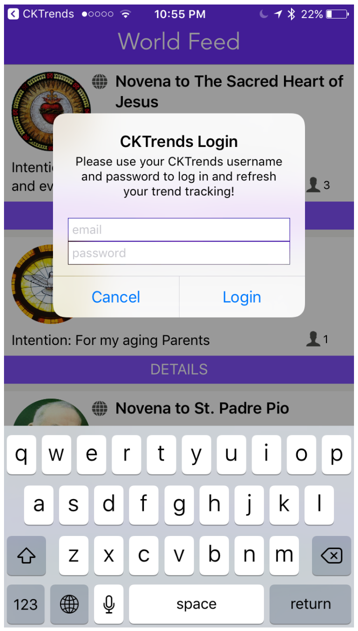

CKTrends 10:55 PM 22%
World Feed
Novena to The Sacred Heart of Jesus
CKTrends Login
Please use your CKTrends username and password to log in and refresh your trend tracking!
email
password
Cancel
Login
Intention: For my aging Parents
DETAILS
Novena to St. Padre Pio
space
return

Figure 5 CKTrends authentication in NovenaNetwork. The CKTrends API authenticates users in order to write to and read from the Firebase database.

security concerns—how can you ensure that Developer A cannot access Developer B's trends?

Firebase's Realtime Database, used for this system, has a nested data structure, and the admin is able to write rules securing different parts of this nested structure. The Firebase database for CKTrends is organized such that each user's trends data are nested under his/her unique user ID (uid) generated by Firebase. CKTrend's Firebase database has security rules in place that stipulate that, in order to read or write data nested beneath a uid, you must be authenticated as that user (via Firebase's authentication API). For this reason, when tapping Refresh in the CKTrends app opens the developer's app, s/he is prompted in an alert box to enter his/her CKTrends username and password. This authentication is what enables the CKTrends API to write to the Firebase database. Similarly, users are authenticated when they open CKTrends, which is what enables the CKTrends app to read data from the Firebase database.

V. Evolution of Ideas

The way I have presented the CKTrends system thus far has perhaps made it sound like the system design was straightforward to piece together. It was not. There were plenty of hiccups on the road to this ultimate pipeline, and I would like to share the most significant one, just to give a sense of how this project has evolved.

At the start of this project, I for some reason had it in my mind that I would use a push model rather than a pull model to get new data from CloudKit. (I had not even considered the possibility of a pull model.) My plan was to set up one CloudKit "Subscription" for each record type being tracked. These Subscriptions would send push notifications to the Developer App whenever an instance of the corresponding record type was created. On a surface level, this seemed ideal for my purposes. I planned to have a handler method that would receive these push notifications and store the appropriate count data in the Firebase database.

This model fell apart when I realized at the tail end of the fall term that push notifications can only be handled when the receiving app is in the foreground. This meant that, if I wanted to run the handler code that saved record counts to Firebase, the developer's app would have to be opened. Realizing this issue led me to reassess what I was doing, and I realized that a push model using CloudKit Subscriptions was actually really bad for the following reasons.

First, suppose you set up a CloudKit Subscription to the creation of RecordTypeA instances. If UserX creates an instance of RecordTypeA, then all users ***except*** UserX gets a push notification. Obviously, for my purposes, I would have wanted to handle only one of these pushes, lest I over-count newly created records. Presumably, the handler code that saved counts to Firebase would have only run on the developer's instance of the app. This leads to an obvious problem: the developer would not receive pushes for his/her own changes to the CloudKit database, meaning that these would not be displayed in the CKTrends app. A corollary issue was the fact that so many extraneous pushes would be sent out to so many devices. Unnecessary work is never good.

Second, this push model would not be able to account for records created before integrating the CKTrends system into one's project, so this model would not provide a complete picture of an app's CloudKit trends.

Third, if a device gets too many push notifications at the same time, they will be "batched." Batched push notifications require special handling. I was worried that an unsuspecting developer who was already using push notifications of their own would add the CKTrends API to their project, start getting batched push notifications, not handle them properly or even realize they were being batched, and not understand what had happened to their own push notifications. I did not want to disrupt the developer's pre-existing push notifications setup or raise the barrier of entry to using CKTrends.

Given these issues, I was forced to rethink the push model I had assumed was the best approach. I sketched all the theoretical possibilities I could think of and assessed them for feasibility, eventually arriving upon the pull (manual refresh) model I described earlier in this paper.

VI. Outcome

I submitted CKTrends to the App Store in late February. After over a week (which is a long time for an app review), I received a call from an App Store reviewer. I was told that CKTrends violated section 3.3.11 of the Apple Developer Program License Agreement, which states:

"Neither You nor Your Application may perform any functions or link to any content, services, information or data or use any robot, spider, site search or other retrieval application or device to scrape, mine, retrieve, cache, analyze or index software, data or services provided by Apple or its licensors, or obtain (or try to obtain) any such data, except the data that Apple expressly provides or makes available to You in connection with such services. You agree that You will not collect, disseminate or use any such data for any unauthorized purpose."[2]

In other words, the App Store reviewer told me, the team felt that I was misusing Apple's CloudKit database by "scraping" it. While the news of CKTrend's rejection was initially disappointing, I decided that this reason for rejection gave me some street cred as an iOS developer. It seems to me that I was rejected not because my app was bad or because my design was poor, but because CKTrends was too clever. I can still use the CKTrends app to track NovenaNetwork and any other apps I develop, which is certainly satisfying in its own right.

VII. Conclusion

CKTrends is a system that allows developers to track their CloudKit databases, providing more accurate and customizable information than what iTunes Connect can

[2] Apple, Inc., "Apple Developer Program License Agreement," https://download.developer.apple.com/Documentation/License_Agreements__Apple_Developer_Program/Apple_Developer_Program_License_Agreement_20170919.pdf.

provide. Developers configure their project to use CKTrends, call the CKTrends API in their code, and tap Refresh in the CKTrends app to view the most up-to-date trends in their CloudKit database.