Trabajo Fin de Máster

Behind the App: Software Architecture

Dinesh Harjani

La Laguna, on Monday January 22nd, 2018.

D. **José Luis Roda García**, with Spanish D.N.I. 43356123-L, being Associate Professor, belonging to the Departamento de Ingeniería Informática y de Sistemas at Universidad de La Laguna, as tutor

C E R T I F I E S

That the present TFM titled:

“Behind the App: Software Architecture”

has been written under his direction by D. **David Dinesh A. Harjani Harjani**, with Spanish D.N.I. 42220646-Y.

So that it may be stated, in compliance with current legislation and to the appropriate future effects, that the present has been signed in La Laguna as of Monday January 22nd, 2018.

Acknowledgements

Prof. José Luis Roda García

ABSTRACT

Modern-day software is anything but complex. From the early days of school, be it at Higher Education, professionally-taught courses or even as self-taught developers, we all begin learning how to write simple, straightforward code, that does one thing, and one thing only. This is a complete far-cry from what modern software development requires, which pushes so much complexity into the developer's hands, that building small pieces of functionality without tying them up together in higher-level structures will prove to be one of the greatest dangers to any project. With far greater risks that those proposed by any technical challenge.

Keywords: Modern, Software, Architecture, iOS, Apps, Development, Complexity, Programming.

Table of Contents

Index of Figures

1.Introduction

1.1 Preface

Figure 1: A MacBook Pro mirroring Xcode's code editor into an iPhone.

The clock has just striked the year 2018, which means, we're already a decade past the App Store revolution introduced by Apple in the Summer of 2008. It seems like it was just yesterday, when everyone was up in arms about Apple not allowing 3rd party developers to write native apps for the iPhone, back when Android resembled more a BlackBerry than what it eventually ended up shipping[1]. But it launched, and the App Store was such a hit with consumers, that developers flocked to the new mobile platforms hailing it as the Gold Rush of the 21st century. Today, what remains of said Gold Rush is the dreams many of us have to become full-time indies, or in other words, to make enough money writing apps that we love.

What many don't give enough credit for though, is to the shocking amount of progress we've made in another field of computing: the cloud. Ten years ago, "the cloud" resembled more a trap everyone had fallen into, rather than a solution onto itself. It was a necessary consequence to the ongoing demands of the web, still recovering from the dot-com bust of 2001, but it was far in everyone's minds from **becoming the de facto solution**. There were many pioneers back then, but the situation was nothing like it is now.

Make no mistake: everything you're using from your smartphone right now, is powered by the cloud. You might happen to call those colorful icons Instagram, Twitter, LinkedIn, Reddit, Snapchat, WhatsApp, Telegram, Yahoo!, Netflix, YouTube,

Hulu, Spotify, TripAdvisor, or any other belonging to this endless long-tail of etceteras. The secret to most of them though, is that their *clouds* are powered by a backbone comprised primarily of Amazon, Microsoft and Google servers. It's true that they all keep some amount of servers in-house, but scalability throughout the globe is ultimately provided by these big-whales, willing to exchange money, for the use of their large-scale commodity-based[i] server hardware plugged directly into the spine of the Internet. Few, like Facebook and Apple, have the economic power to build top-to-bottom walled-gardens of their own, to keep absolutely everything under their corporate umbrella. Everyone else, specially startups, rely on the big three to power their ideas at a reasonable cost.

But the consequences of all this progress is a great cost, known to mankind from the very early days of engineering, in the form of complexity. If we want to build bigger, better and cheaper entities, something's got to give; as nature has shown us, everything we can imagine is perfectly achievable, but getting there requires a certain sense of balance. You can't have everything, and if what you want is better, bigger and cheaper, and you're willing to throw as much manpower as possible to the problem, then **out-of-bounds complexity, is what you get**.

It wasn't always like this. In the early days, **hardware was very slow**. Therefore, we prioritized making software as simple and efficient as possible, to make the most of whatever silicon power we had. Even programming languages were written to focus on computer efficiency, rather than human efficiency[ii]. But when the 2000s came around, and Intel began its rain of ever-increasing performance products with its tick-tock strategy[2], coupled with the rise of smartphone app development, one thing became clear: hardware was now the commodity, and together with large-sums of money plunged by investors in order to not miss "the next big thing", developers became the key to increase the company's value. Complexity is no longer a problem, as long as you can keep everyone working together towards the company's goals.

1.2 How Is Complexity Solved?

---

i. There was a time when servers relied on completely different architectures in order to meet customer demand. After the rise of Intel's multi-core solutions, hardware to serve hundreds of requests per second became bargain-cheap, and with it came the benefits of knowing your software was always running on x86-based hardware.

ii. Think programming languages like Assembly, Fortran, C and C++. Even Objective-C, which predates Bjarne Stroutstrup's own.

Complexity isn't new to us humans. It's been a staple of many other branches of engineering: cars, ships, buildings, bridges, and really all kinds of construction. Even in the computer industry, back in the days when software represented the minimum amount of microcode necessary to make the hardware run, had to pull from a very common, simple, and yet highly efficient concept accross all forms of engineering: **architecture**.

As a word, "architecture" involves completely different concepts and ramifications depending on where it's placed, but at its core, it comprises a set of values that have been fundamental in enabling us, humans, to work together and build bigger and more complex structures than those we could build on our own, at a faster pace. How? It begins with a vision for the complete structure of what we want to make, how it should work, and what requirements it should have. Then, we break it up into smaller components, each of which meets a subset of the criteria required by the bigger structure. Once we've broken it down to the point where we can differentiate two distinct levels, we continue this process recursively, until we reach the bottom of the pyramid: where everything we need to do, is comprised of problems we already have a solution for.

Sounds easy enough, but there are caveats. You need leadership to provide a vision, and you need to be able to make people work together in the smaller components, in order for the project as a whole to come to fruition. Once you've solved that set of human-related problems, another one comes your way, and that is: "did we miss something?" Because if you did, you need to re-architect and re-engineer your solution to fit whatever new requirements you've discovered. This does not necessarily mean that your entire architecture needs to be thrown out of the window, but it might mean that you'll need to make certain tradeoffs in order to meet your deadline. Again, as nature has taught us, **balance is at the core of any great achievement**.

1.3 How Is Architecture Reflected In Our Industry?

If you perform a quick Google Search, most of the results on software architecture will either be platform architectures, like those based on Operating Systems or programming environments, or, the most common one in vogue today, which is **back-end architecture**. The first one has its origin back in the early 90s, when the Computer Industry hadn't yet settled on Windows, Linux and the Mac, and everyone wanted to believe the market for a mass-market Operating System wasn't

decided yet[iii]. In turn, the promise of being able to write great applications through new software platforms was the bait set in place for software developers to flock to the new platforms, which was the root for those conversations on what made great System Architectures. History would repeat itself again 15 years later with the rise of the smartphone[iv].

Today though, architecture in the software realm is mostly reserved to the myriad of complex systems keeping alive the "clouds" we spoke about earlier, powering all the apps that currently sit on your smartphone's homescreen. Why is this the case? Suppose, for example, that you're powering up the Facebook app in your phone. The first thing you'll see is a strip of blue at the top, and a sea of white in the middle of the screen. What do you think is happening here, while you're waiting to see content? That's right - after powering up all of the app's internal components, a request is being made to fetch items for the app to show your News Feed. Who's birthday is it today? What happened in your life 6 years ago? Where are your friends travelling to on holidays? What party did your ex go to last night? The app on your phone, the Facebook app, makes a single request to its server, and gets everything that it needs to show in the appropriate order for the current user, meaning the app only needs to know *how to show content*, but not how to aggregate it.

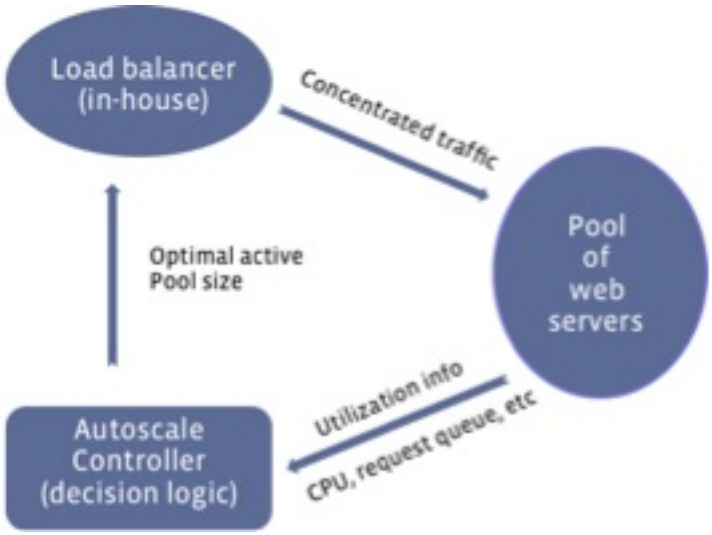


Figure 2: Facebook Autoscale Architecture.

But how does this request really reach your phone? Independently of whether we're using an Android or iOS device, using the OS' Developer API, the Facebook app

iii. Before Microsoft landed on its feet with Windows 95, companies like Commodore, NeXT, and Be thought the market didn't have to belong to MS-DOS and the Mac.

iv. BlackBerry and Palm would become the Commodores of the smartphone era, together with other failed mobile endeavors such as FirefoxOS, Windows on mobile and MeeGo.

performs an HTTPS request that hits Facebook's DNS Servers first. When it does, the request will be redirected to one of Facebook's load-balancers[3], which will decide which physical server should serve said request, and transfer it for processing. Once the request has been processed, and a response has been sent back to the user, your phone will display your News Feed[v]. At the same time, the chosen physical server that served your request will inform another component in Facebook's network about the work it just performed, so that the software running the servers can perform administrative duties on their farms. This allows Facebook, for example, to automatically expand and contract its pool of physical machines on demand, in order to save as much money as possible, but without ever rejecting or delaying a user's request. Facebook, together with other industry giants, makes it look extremely easy to adapt server infrastructure costs with on-demand network requests to a back-end, and in their case, Instagram and WhatsApp also use Facebook's back-end infrastructure, including release engineering[4].

But what happens inside the physical web server, once the request reaches it? What software is responsible for processing and returning a response? That's a bag full of complexity. You see, Facebook's services aren't written in any one language: they can be written in Java. In C++. Or even, and actually most likely, in PHP, which as we all know, is a slow interpreted language. So slow in fact, that Facebook spent 18 months writing a PHP Compiler to translate its PHP services into C++ code[5], which is then re-compiled and executed using a standard compiler like GCC[vi]. And, because a service can be written with any of these three languages, Facebook has found itself using Apache Thrift[6] to dynamically call each service independently of the language it was written in. And below the application layer?[7] We have services like logical storage, which is handled through MySQL, Hadoop's HBase, and of course Memcached[vii]. Back-ends also clearly differentiate between logical storage and mass storage, which is where massive binary files like our Facebook pictures and videos are stored. If we were the developers of the server's application, it shouldn't be our responsability to know in exactly which data center node the photo with ID 90210 happens to be in, right? In an ideal world, we would just like a quick

v. Which, let's be honest, we both know it's mostly filled with friends on holidays, getting engaged, or taking selfies with their cats.

vi. We don't have accurate information, but we hope that Facebook has since moved HipHop to use LLVM instead of GCC.

vii. Memcached is an in-memory cache with a hit-rate upwards of 90% that stores both MySQL queries and full request response objects to greatly speed up read-only requests and aid scalability.

method to obtain the Facebook CDN link[viii] to the image so that we can return it to the request client in JSON form. And this is exactly what Facebook built, in the form of an entirely new software called Haystack[8]. And to be honest, after learning a bit about what they need to do in order to keep all those petabytes of data in a reasonable amount of physical space[9], I really can't blame them for rolling-their-own solution here.

As you can imagine, I've left out lots of other potential problems companies deal with on a constant basis when building back-ends. But compounding the fact that every single company wants to build their back-end their own way, and given the massive scalability and reliability requirements of modern web-services, no one could blame them for thinking that their back-end architecture is truly important; modern businesses live & die by their normalized service uptime throughout the year, lest they become literal whales[10]. Problem is, this is only the back-end; at some point, **front-ends need to be written too, or users won't be able to reach these services**. And even though back-ends handle a lot of complexity, front-ends aren't complexity-free either. You can't jump in and write a front-end by gluing code together just because you think the back-end is handling most of the complexity for you. Before you know it, that mountain of code will come back, and bite you. To prove it, let's have a look at a real-life example.

1.4 The Effects Of Poor Software Architecture

It was the Summer of 2011, and I was at the height of my programming powers - *in my mind only*, of course. Back then, I believed that I was the best programmer in the world; a sentimient I think others might've also felt at some point in their careers, too.

I was at a crossroads: I'd had my first job as a mobile developer, but the company I was working for lacked a viable business, so they couldn't renew my contract. When that was over, I had a ticket to Silicon Valley for Google's Developer Conference[11] lined up, so I went there, enjoyed the trip, and returned home inspired. I had two options: I could look for a job, which I kind-of had in my hand with an upcoming local game development studio, or, I could build an app. Filled

viii. CDN stands for Content Delivery Network; it's where most photos and videos are stored in distributed nodes across the world to minimize data loading times. Big companies like Facebook, Twitter, Netflix, Apple, YouTube and so on rely on them for instant-streaming and downloading of content.

with the confidence of being capable of building absolutely anything I set my mind to, I chose the second.

In 2011, I was in awe of the iPad; I'd finally understood the point of tablets: they were *a lot more than just big phones*. Instead, they could be an empowering tool - a way to break from how the world perceived serious software - which still today resembles Windows 95 apps, I might add -, and create a new beginning, in which technology was the enabler, not a hoop through which you had to jump, with the iPad at its core. And at the center of my idea, besides the iPad, were the people sorrounding it: *the iPad as a people management solution*. I viewed it as a tool for the everyday businessman, managing a low-tech enterprise such as a shop, a package delivery service, an accounting firm, and maybe even perhaps a small startup, where the number of tasks was reduced, and the amount of workers to manage was between 7 and 15. Not too few that you could all hold it in your head, and not too many that you required a much more complex solution. My app was supposed to be an "employee performance tracker" of sorts, allowing you to define tasks and assign them, keep track of meetings and their developments, set reminders for work to be done, notes and ideas regarding your business, and more. If you wanted to know what someone was working on, you simply had to open the app, find the employee on the list, tap on their name, and you'd see everything they're associated with at the moment, plus automatically-generated data on their performance. Sounds a lot like Asana[12], and like a dumbed-down version of Jira[13] locked on your personal iPad[ix], if you ask me now. But back then, I knew better than to simply Google-check if such a product existed. And if it did, I'd do it better by kilometers. It was me we were talking about. In fact, I was so high in my own fever, that I called it **I am CEO**, and made sure, no one could ever take that name away from me, by locking it in Apple's Developer Console[x].

ix. In 2011 I didn't view "the cloud" as a good solution to build a product around. Now I think it's rare you don't need a server component to implement a business idea. Even Racing Tweets v2.0 has a server component, and it's a Twitter client!

x. To this day, it still receives the name of "iTunes Connect".

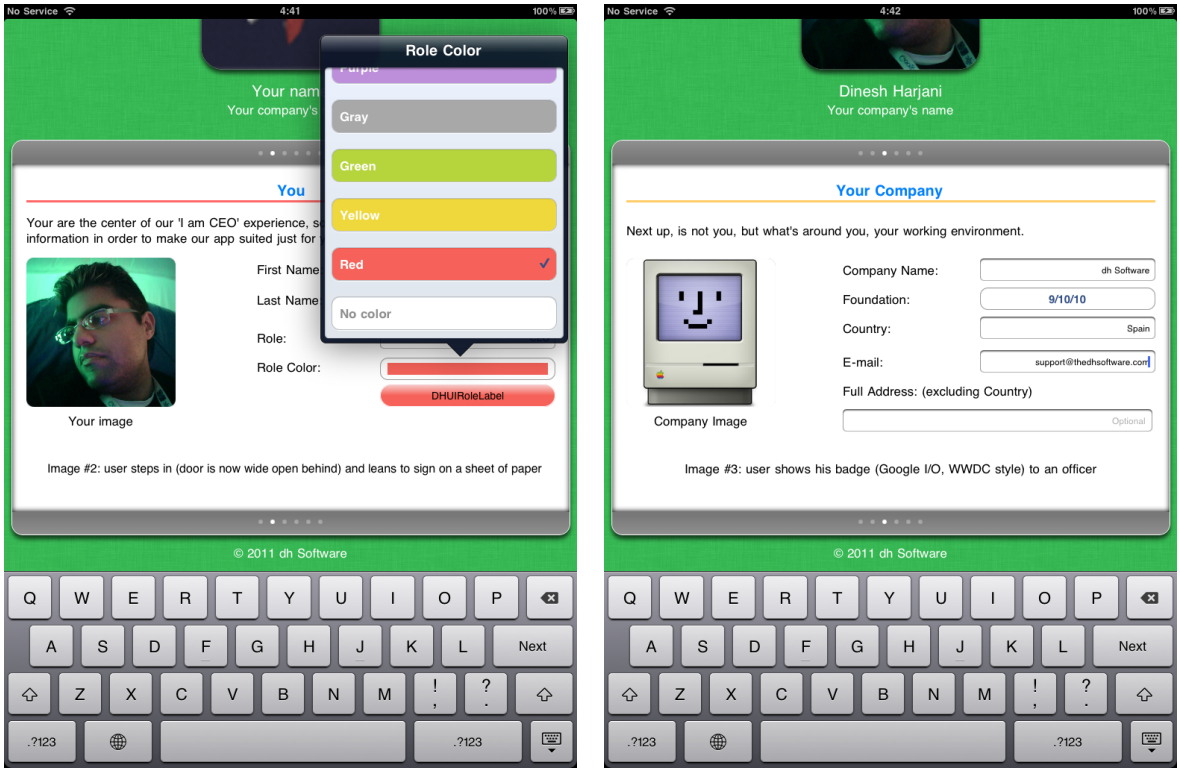


Figure 3: I am CEO Phase 1 company creation flow, circa September 2011.

Development was supposed to be split in three phases: **Phase 1** would be about building the basic framework for the app to be based on, plus the company creation flow. **Phase 2** would build most of the app's features regarding people/employee management, and **Phase 3** would involve a testing/feedback cycle to perfect the app before it was time to sell it to businesses. I later added a **Phase Zero**, to "protect my creation", with a password locking screen[xi]. **Phase 1** took months, about 3-4, and by the time I deemed it finished, rather than working on **Phase 2**'s features, I kept working on the company creation flow, which kept failing and crashing, because the code was a complete and utter mess. One of the company creation flow steps was to *pull* employess from your Address Book, or a Social Network such as Facebook or LinkedIn. The Facebook class clocked in at more than 3.000 lines of Objective-C code[xii], and what was worse: the code functions within the same file jumped around the file all the time, without any clear codepath. Meaning **the code was not written to logically perform a series of steps**, so instead, I just added code wherever I saw fit inside the class as I discovered more and more things were needed to perform the task. And achieving the list of fully parsed Facebook friends wasn't easy, either: first we had to obtain all our Facebook

xi. I'd later do the same with the Standard Engine. Until I realized it was pointless and I removed it.

xii. I wish I were kidding.

friends' IDs, and if memory serves me right, we had to chain multiple requests because only up to 50 friend IDs could be returned per request. Then we had to chain another set of requests, this time for all of our Facebook friends' JSON Objects, in sets of 20. And parse them. And after parsing them, we had to write another parser for the Facebook User profile images, which were also JSON Objects.

Things got worse as time progressed. Because I'd invested so much brainpower into fetching the list of Facebook friends, making them show up in the app, and load their images asynchronously depending on the scrolling position the user was in[xiii], whenever the system failed I couldn't help myself to not move on and fix it instead. This was a big source of dissapointment, because even once I deemed **Phase 1** finished, I kept going back to it, without being able to fully fix it once and for all, instead of working on **Phase 2** features. And how could I build the "company features", if I couldn't make my users reach that point flawlessly? This situation was not emotionally sustainable for me, so as the Summer faded and my University course-load picked up, I began to spend increasingly less amounts of time on the project, until I signed for a new job, and didn't look back at it.

If only I'd stopped, and tried to understand the problems I was going through… It is clear to me now though, if we look over the business-side of the equation, that most of my problems had roots in me not studying what I wanted my code to do and how I could achieve it. Just a few notions on Software Architecture would've allowed me to split the problem in parts and isolate them, allowing me to work on them independently, without fear of breaking something due to small changes. It didn't even require the scale of architecture we'd apply for the Standard Engine/Racing Tweets, but it needed something more that I couldn't provide at the time.

1.5 What Is Software Architecture, Then?

Far from being a magic bullet to solve development problems, **Software Architecture is a tool necessary to build any kind of software-based product that exceeds the length of a handful of code project files**. It does not have any

xiii. When you have a UI element that can scroll, your current scroll position forms "a window" of content, which is the currently visible set of items to the user. Because we wanted to "write it once and forget about it", we made the code listen to changes in the scrolling window's position and only load the images for the Facebook friends the user was currently seeing. We implemented this in a much better way for Racing Tweets.

rules, nor frameworks, and sadly, no public figure to rally behind either[xiv], telling you exactly what to do to ensure your project enjoys from a healthy architecture. **What Software Architectures is, is a way of thinking, of breaking up your problem in many smaller pieces**, so that when you have to code 150 business requirements plus another 100 User Interface features, everything has its place, and nothing that's completely unrelated breaks when you make a change. If you've realized people can't keep in their minds how everything in a big project works, or how all the components are inter-related, and have taken steps to improve the way your team visualizes tasks and the code they produce, you have taken steps toward implementing a better Software Architecture. You might not have realized it yet, but the important part is that you realized there was a problem, and rather than work against it like me in **I am CEO**, you decided to turn it into an opportunity.

Sadly, there are many misconceptions regarding Software Architecture. Many people believe they know Software Architecture, just because they took a class on Design Patterns at school and a couple of years later realized they could refactor an intricate web of source files using one of them. As I've explained before, good Software Architecture isn't based on recipes. Instead, it is a way of thinking, of allowing Object-Oriented Programming to shine through our design choices, rather than treating classes as boxes in which we simply paste code. Software Architecture begins by modelling your problem into real-life elements of your business, and making them actors in the form of Classes and Objects, in your code. Good Software Architecture has the downside of requiring some amount of documentation to communicate how the system is structured, but, it has the massive benefit of enclosing logic into a single component: when a change is required, you know exactly which class to touch, and can check for usages in that class to find any code ramifications from your change. Plus, keeping this documentation up to date will also aid you in ramping-up new hires, who will enjoy the added comfort of knowing the system is well compartimentalized, and will not fear modifying complex sub-systems because they will know how everything works together.

xiv. "Uncle Bob", also known as Robert C. Martin, comes close. Though he's more a proponent of Clean Code than good Software Architecture. Clean Code however, has the side benefit of promoting good Software Architecture.

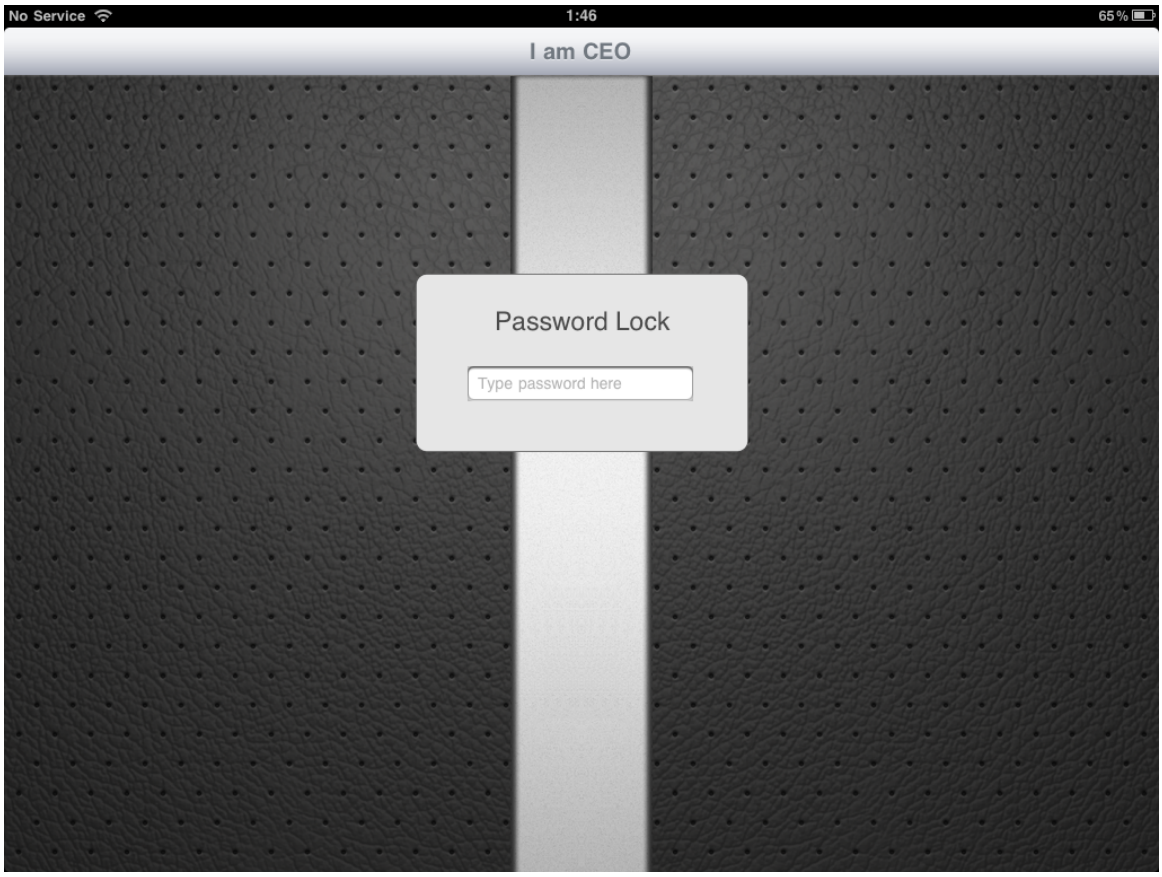


Figure 4: I am CEO's password protection screen. Upon successful unlock, the doors would unhinge in two distinct steps at different speeds, stopping in between for an increased dramatic effect.

Obviously, what I explained above didn't apply in any way to how I worked during the **I am CEO** Summer. But the question now is, what can we learn from my failure? I learnt that Software Architecture is not only fundamental for back-end applications, but also, for front-ends. In the back-end, people already separate their services in multiple elements: load balancers, databases, mass storage handling, event queues, offline processing, and so on. But in the front-end? Everything is just an app; it's just code. From my experience, there usually is some kind of organization, but no one is overseeing what the codebase looks like, or how it's evolving, and in which direction. And this is not scalable, especially in an era where employers are asking us to write different versions of the same screen in an app for cases like A/B testing[14]. Can you imagine having to solve many of the issues again, by having to copy-paste code? It's not only about being able to reuse parts of code; this is software, **we should be able to reuse everything but the parts that change**.

So, how would we go on and write a properly complex app, using good Software Architecture as our guiding principle?

2.Case Study: Racing Tweets's Architecture

2.1 What Is Racing Tweets?

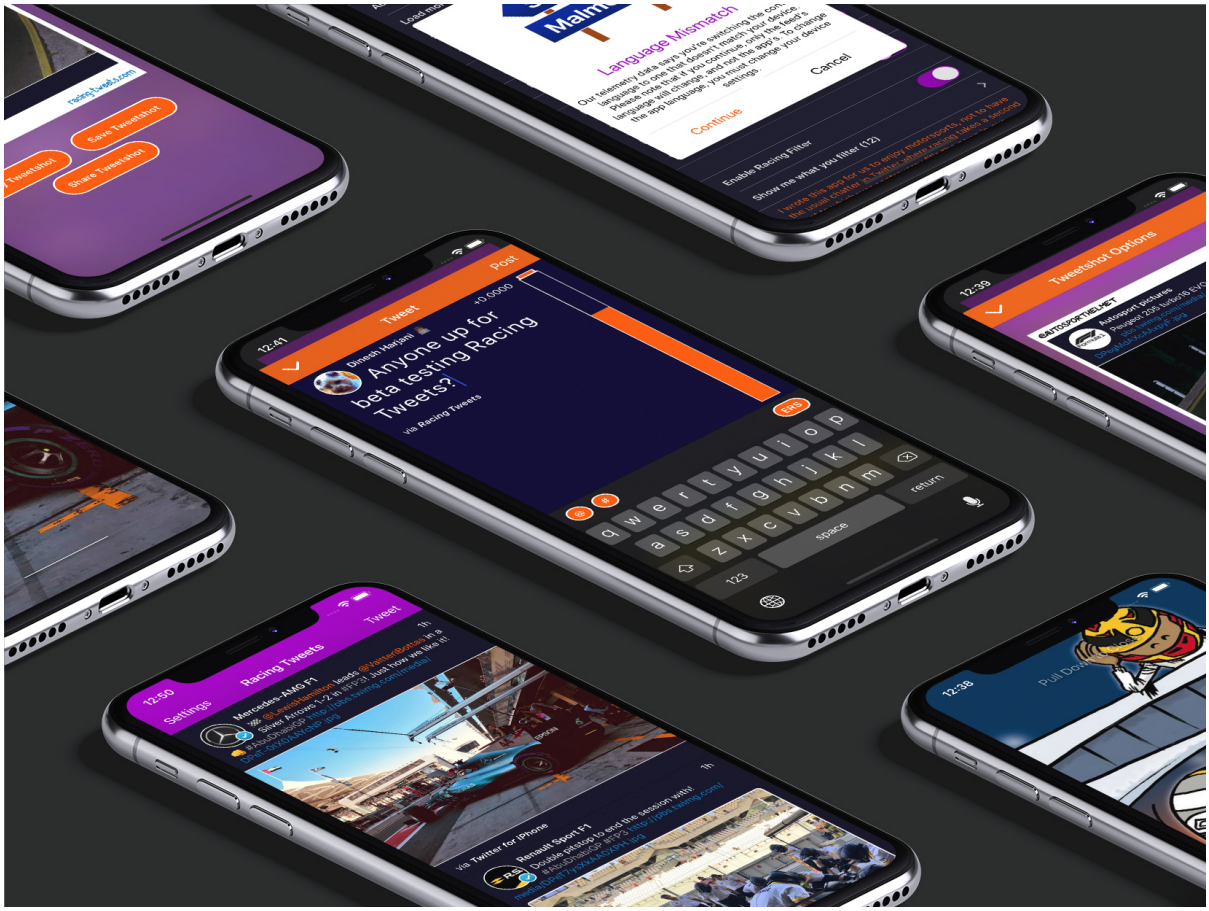


Figure 5: Promotional artwork for Racing Tweets v2.0.

The main course for this chapter will be an iOS App, specifically the unreleased version 2.0 of my personal project, **Racing Tweets**. We will not go through the full history of the app here, though you can find it in the Annex I chapter of this paper. In a sentence, Racing Tweets was meant to be a Twitter Client aimed at motorsport enthusiasts, capable of de-multiplexing multiple streams of tweets by combining them into a unified timeline, where everything the user saw was motorsport-related content. To complement and enhance its user-perceived status as "just a packaging of Twitter lists I can do on my own"[15], it offered innovative features such as:

- A **striking Design**.
- **Choosing which motorsport(s)** you want to follow, and in which language.
- A **Racing Filter** feature to prevent the user from seeing non-motorsport related tweets.
- **Motorsport-themed** touches everywhere (racing wheels as loading bars,

Formula 1-lookalike “grids” for quoted content, an Energy Recovery System battery load graphic to show available characters for a tweet, etc.)

- **Tweetshot(s)**, or snapshots of tweets that made them look like Polaroids.

This concept, and many of its ideas, have roots on past projects, which you can read about in the Annex. But there are two main points you need to know before we proceed, the first being that version 1.0 was released in late 2015, and did not meet expectations. The second, is that there’s an underlying component responsible for handling all of the data, which I built, and which I named the **Standard Engine**[i]. And because Racing Tweets is build atop of it, **speaking about what powers Racing Tweets requires speaking about the component responsible of powering its main functionality**. In fact most of what you’re about to read next, concerns directly or indirectly to the Standard Engine. Many features written specifically for Racing Tweets either plug-in to the Standard Engine, or were written directly in the layers above it. However, due to word-budget constraints, we won’t be able to talk about everything today.

2.2 Layers

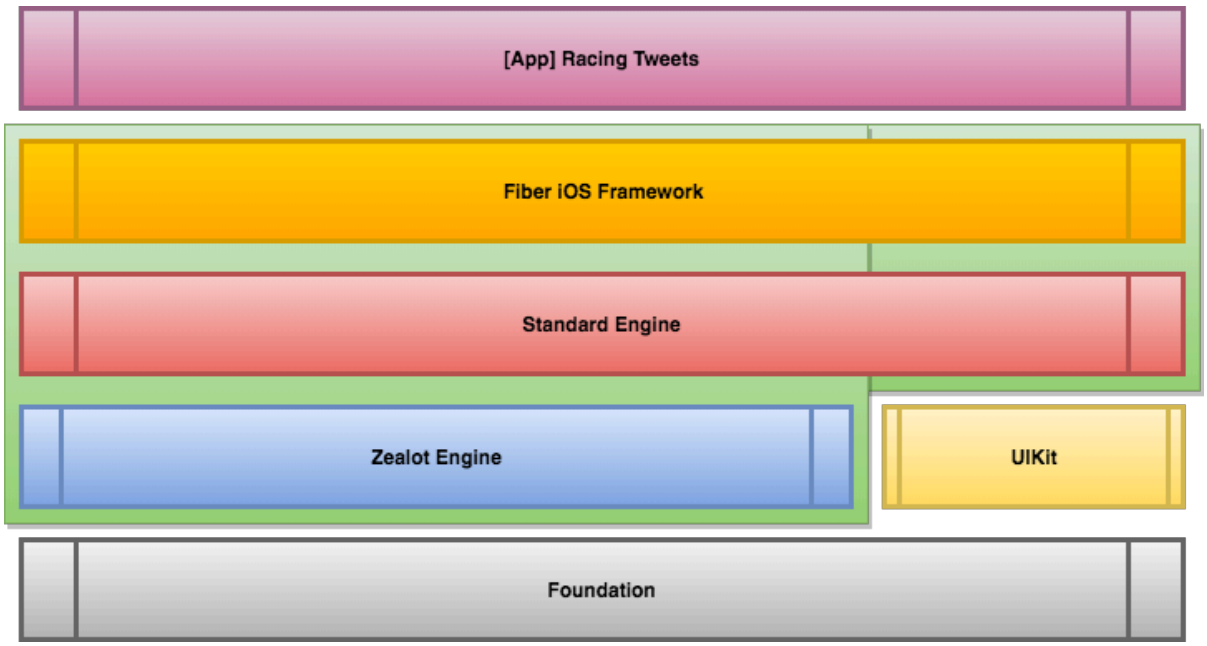


Figure 6: Racing Tweets v2.0 Software Architecture.

We’re going to begin peeling back the layers that make up the complete product. If

i. You can read the entire story it in the aforementioned Annex I, including the inspiration for the name.

you look closely, you'll see that the Standard Engine is highlighted with a green background, together with two other components. This is because **the Standard Engine was architected from the very beginning to be comprised of two layers, with a third one being made formal later on**, when we decided that we might want to build multiple apps around it. In any case, whenever we refer to the Standard Engine outside of this section, we will do so without distinction of the three individual layers that comprise it, to protect the reader's sanity.

Of all of the aspects that make up Racing Tweets, the Engine powering the core feature of the app itself is very important, but it's not the only piece of major engineering required to make it all work. There are more, because the Engine itself is built upon frameworks provided by our platform of choice: iOS.

- **Foundation**: This is the underlying framework, built by Apple, upon which all **MacOS** and **iOS** apps[ii], are built upon. It includes basic elements for any programming platform such as string management, mathematical functions, date & time functions, formatting classes for dates, monetary amounts, languages, and a lot more elements like data structures and their related functions. Foundation is only available on Apple platforms for the **Objective-C** and **Swift** languages[iii].

- **UIKit**: It's the User Interface layer for application development in **iOS**. It includes more than just UI-Level classes though, since it is responsible for drawing all graphical elements on the screen, screen hardware management, touchscreen event processing, animations, text management, accessibility, and a lot more. This is the layer through which the developer accesses most of the Operating System's functionalities. And together with `Foundation`, `UIKit` forms **Cocoa Touch**, a higher level framework containing everything necessary to build iPhone, iPad, AppleTV and Apple Watch applications.

- **Zealot Engine**: When we first sat down on a table, and thought about how we could solve the problem of mixing different social media feeds and unifying them into a single list of ordered items, we came up with a blueprint comprised of the major elements needed for this concept to happen. This later grew into a toolbox attached to a set of concepts & interfaces that

ii. There are two OS'es deriving from iOS: watchOS and tvOS. Both also rely heavily on Foundation and UIKit, though they do so on a subset of their features.

iii. In the Apple community, enabling Swift for server-side uses is an important thing. Therefore, a port of Foundation is available to support the development of Swift-based servers. Racing Tweets' own is one of them.

allowed for enough freedom, so as to not determine how the critical parts of the system should be written. Together, all of these parts form the Zealot Engine, which **specifies what needs to happen, without specifying how.** In code, the Zealot Engine's elements prefixed "ZE", or "DHZE"[iv].

- **Standard Engine**: As the layer directly above the Zealot Engine, **the Standard Engine is an implementation of the Zealot Engine concept**. It adds its own web of complexity on top of the tools supplied by the ZE, and aims to offer a plug-and-play solution at the UI level, whilst keeping 90% of its internal workings completely hidden to a 3rd party developer. It encompasses the data, network and user interface layers, in a way that allows the API user to keep their custom code highly modular, efficient and clean in an architectural sense. Standard Engine components are primarily prefixed "DHSE", but some remain as simply "SE".

- **Fiber iOS Framework**: As development of a working prototype for an app concept called **Fiber** continued past the complex logic encapsulated within the Standard Engine, complex User Interface features like a customizable background wallpaper for the app, were written using Standard Engine elements. As more and more UI code was written that could be reused between different SE-powered apps, these elements were architecturally placed within a third layer above the other two, retroactively named the Fiber iOS Framework.

- **[App] Racing Tweets**: Last, but not least, is all the code required to make the app's features come true. In our case, this includes all of our custom `UIViewController`(s), our own Standard Engine hooks, features like our Racing Filter, allowing the user to switch between content languages and motorsports, custom animations, custom user interface elements, custom transitions, icons, etcetera. Most classes follow the "DHTdF1" naming scheme at this level, since the app's original name was "F-1 Tweets", and the shortened moniker in Spanish sounded better to us than the English one.

With this introduction in place, we can now begin to analyze how **Racing Tweets** is structured, using some of the app's features as our guiding light.

---

iv. In the Objective-C era, namespaces didn't exist, so classes needed to be prefixed. After I began working on the project, I discovered two-letter prefixes belong to Apple by coding convention, so I had to rename the classes. This, however, has the problem of losing most of your Git history, therefore some classes weren't renamed.

2.3 Timeline

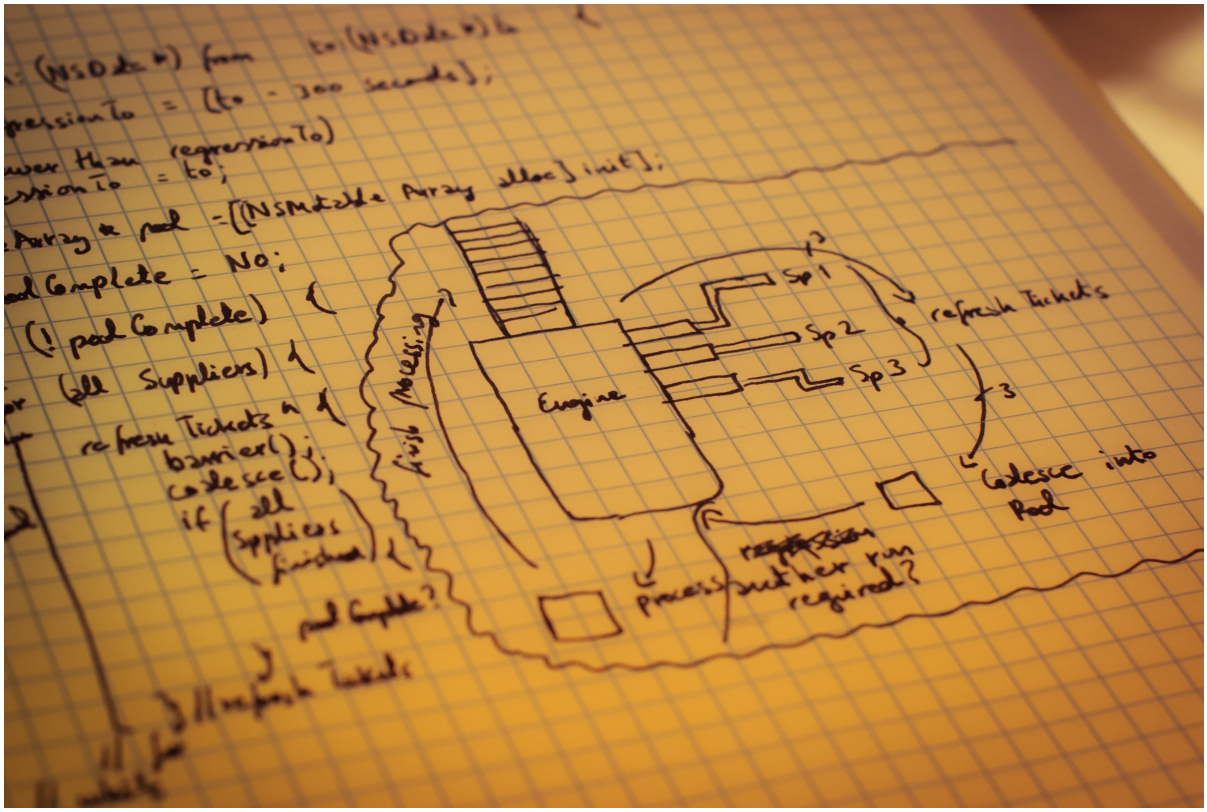


Figure 7: Early Standard Engine Concept drawing, circa 2013.

We cannot begin designing any system, if we don't know what the problem, or set of requirements, the system needs to meet. So before we enter into how **Racing Tweets** is structured, and how the Standard Engine works, we need to picture how complex everything the app needs to do really is. After all, Software Architecture isn't about introducing tons of boilerplate nor extra layers of indirection; it's a tool. And before picking up the hammer, we want to thoroughly analyze our nail[16][17]. So let's start from scratch: **imagine the Standard Engine doesn't exist, and that we're simply going into Xcode, and creating a new project from a template**.

We're going to start with the main interface for showing a list of tweets. As a Twitter client of sorts, everything in **Racing Tweets** revolves around this main screen, from where the user will be able to access the app's content. But, contrary to how I've done things in the past, **we're only going to design our solution top-to-bottom, and write our code from the bottom-up**. We know the main screen or timeline is comprised of a list, which we will need to fill out somehow, with the latest tweets from different kinds of motorsports. We're going to deal with the complexity of acquiring those tweets later, and for now, we're going to **suppose we already have this list, in the form of a simple array of Tweet Objects**, to focus on the interface.

Now that we've acquired this array of tweets, let's put them on the screen. To make these tweets visible and interactive to the user, we need to start using elements of iOS' User Interface framework, `UIKit`. Specifically, we're looking for an object called `UITableView`, which is capable of displaying a list of `UITableViewCell`(s). These "cells", represent information we want to display in this `UITableView`, and for our project, each one of our tweets from the array maps exactly to a `UITableViewCell`. Consequently, this is how 99% of Twitter clients, including Twitter's own, work.

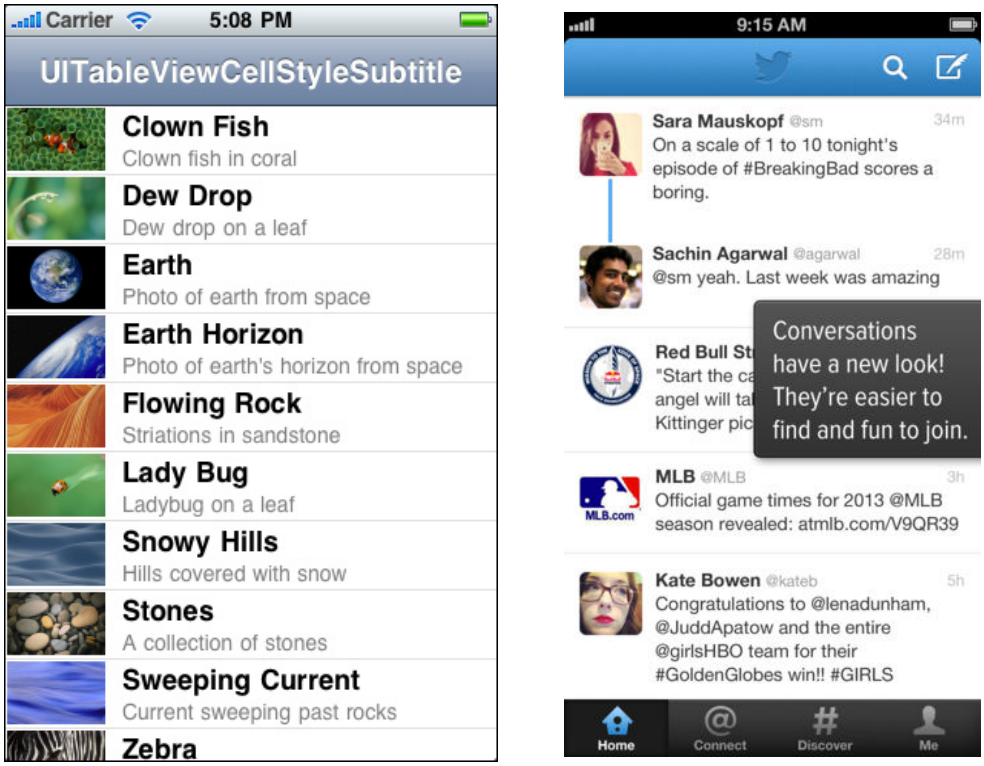


Figure 8: UITableView sample application from Apple (**Left**) and an old screenshot from the official Twitter for iPhone application (**Right**).

We have the list of tweets, and we know that, to show them, we need to put them in a `UITableView` instance. But how do we get this `UITableView` to be drawn on the iPhone's screen? Each "screen" in iOS is managed by a subclass of another object, `UIViewController`, which is responsible for each screen's lifecycle, as well as setting up the visual elements that make it up, including our `UITableView`. So we need one of those.

We now have the `UIViewController`, we have our array of tweets, and we have a `UITableView` for displaying our content. But how do we "plug in" our array of tweets into the `UITableView`? Someone - an object - needs to be responsible for communicating all of this information back to the `UITableView`, of being an adapter

between them[v]. Furthermore, our problems extend a lot further than just telling the `UITableView` how many elements there are in the array; **we also need to transform every tweet into a corresponding `UITableViewCell` so that it shows that tweet's information**. In the iOS world, this task can be carried out by any object, as long as it conforms to the `UITableViewDataSource` protocol[vi]. If you check iOS sample code online, you'll see that the majority of projects put this responsability into their `UITableViewController`, making it conform to the `UITableViewDataSource` protocol and implement those methods. For the sake of simplicity and good Object-Oriented Programming, we'll put this task in a different object, appropriately called `UITableViewDataSource`, whose sole responsability is to be whatever kind of delegate the `UITableView` instance needs.

There's more. We don't want to just show a list of tweets, because **the user needs to be able to interact with them**. Say, for example, that we want to allow the user to retweet a tweet: this means that when an element in the `UITableViewCell` is tapped, we need to call an object to perform the request, and then report us back so that we can show that the tweet was in fact, retweeted, in the UI. And lest we forget, **when the tweet has been retweeted**, **we need to update the (data) tweet array** from which we're displaying the list of tweets. Otherwise, if the user happens to scroll up, and then back down to the tweet they tweeted, it might not show up as retweeted, since the underlying data structure wasn't updated. Again, adhering to our principles, we'll give this responsability to a different object, which we'll simply call the "Twitter Controller".

So, to gather all of our ingredients, here's an interaction diagram with what we have so far:

v. By adapter we mean the Design Pattern, not a USB-C adapter.

vi. In both Objective-C and Swift, protocols are synonyms for interfaces.

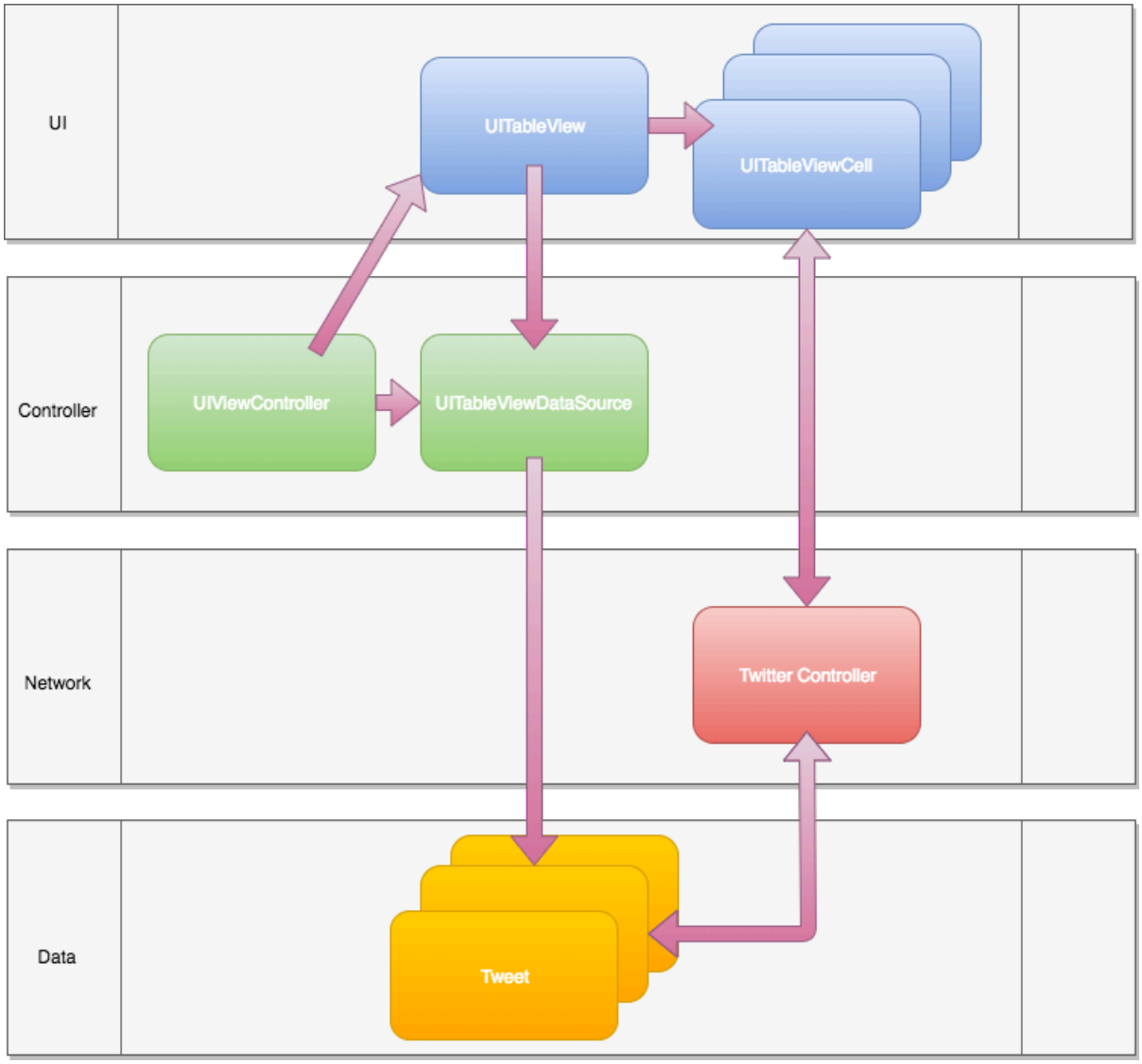


Figure 9: Twitter Client Timeline Interaction Diagram.

We've only done the simplest of things, and yet, as you can see, this has already gotten a bit more complicated than you would've expected it to. What happens in real life is that most people dive head-first into writing code, and just put everything they can into their `UIViewController` subclass, making it perform `UITableViewDataSource` duties as well as the Twitter Controller's ones, no to mention it might even be responsible for updating our tweet array. And, the `UIViewController` might also be the one instancing `UITableViewCell`(s) and styling them to represent each data tweet object. This kind of code-bloat is no different than what happens in other platforms like Windows or Android, where the Form and `Activity/Fragment` objects end up being God-level controllers with thousands of lines of code.

This really is a problem. Take into account that the Twitter network access needs to happen on a different thread, so as to not cause the User Interface to get

blocked, and we also need to deal with returning this information back to the UI thread, and handling error cases. Plus, **in modern software, users expect apps to serve them first, independently of the hidden complexity underneath**. So good apps are expected to enqueue multiple operations and do them in the background, making the user come first, and technology second. For example: if the user tapped the retweet button in 5 different tweets and then favored another 10, **we can't *not allow* the user to perform the next action until the previous one is finished**. Instead, our job as developers is to solve this problem, by way of some sort of operation queue allowing us to perform these tasks serially or in parallel, while making the user believe we've performed their action. Behind the curtain though, we will let the user know if the network operation failed through the corresponding UI update (returning the modified tweet to the previous state, for instance). You could even consider that we should automatically schedule the user's task to be performed later, if say Twitter itself went down, rather than being annoying, as everyone else does[18]. I wasn't kidding earlier when I said that modern software development is no piece of cake.

With all of this in mind, and more, I had a very clear idea, when I began to work on the first app that would use the Standard Engine, of **what I didn't want: a God-level `UIViewController`**. I wanted the UI, including all attached controller objects, to have a clean break from whatever was needed to process all the social media information and making sense of it. So this is what I pictured instead:

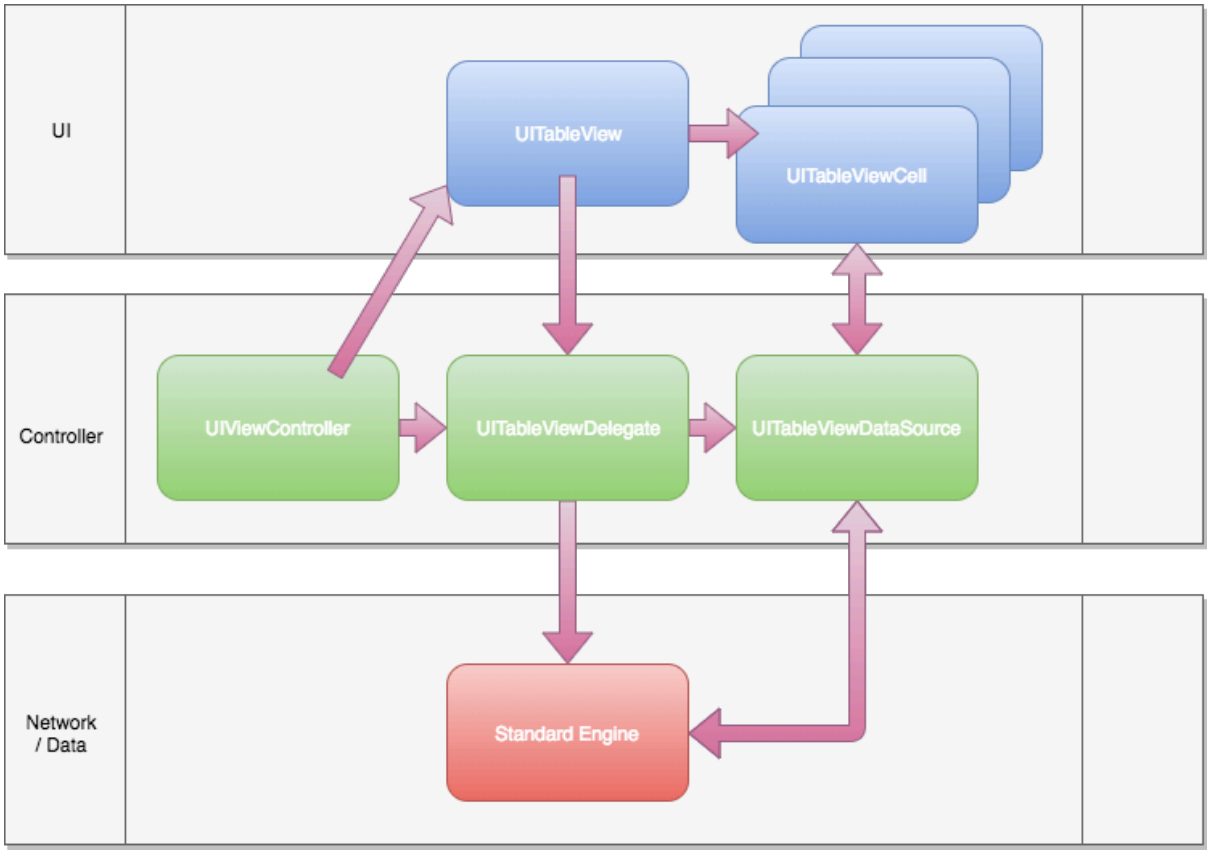


Figure 10: Standard Engine-powered Twitter Client Interaction Diagram.

To be fair, in reality the Standard Engine doesn't provide as good as a clean-break as it looks from over here, but it does show how the `UIViewController`'s task is reduced to merely being the glue between the Standard Engine, and the Interface components. The `UIViewController` makes no requests of its own, it merely connects what the user sees with what the Engine delivers, including management of error dialogs and navigation to other screens from it. The Engine is also responsible for delivering the `UITableViewCell` instances representing each data element or tweet, reducing the `UIViewController`'s `UITableViewDataSource` implementation to just 3 lines of code in each method. You can read about it in Annex II if you're interested.

Finally, to express how good Software Architecture helps you in ways many have not realized yet, consider how big the `UIViewController` class for **Racing Tweets**, which follows the above pattern, might be. Any guesses? The absolute truth is that it encompasses only about 650 lines of code, split into 5 different implementation files[vii], where the biggest one clocks at around 370 lines, and the rest average 65 lines. **This is important**. **Keeping files small allows our minds to focus a lot**

vii. We're using different UIViewController class extensions for this, allowing code to be reused by different UIViewController(s) that might use the Standard Engine, too.

more on the code that's in front of us, instead of having to create an underlying map of how everything is connected, before being able to dive in and work. This is exactly the reason why, when opening a big file, many of us have felt how our mind just lost focus at the sight of how long its contents are. As rational beings, we can't begin to solve an equation if we don't understand what the variables mean, or how they might be related to each other. It's the same with code.

2.4 Navigation

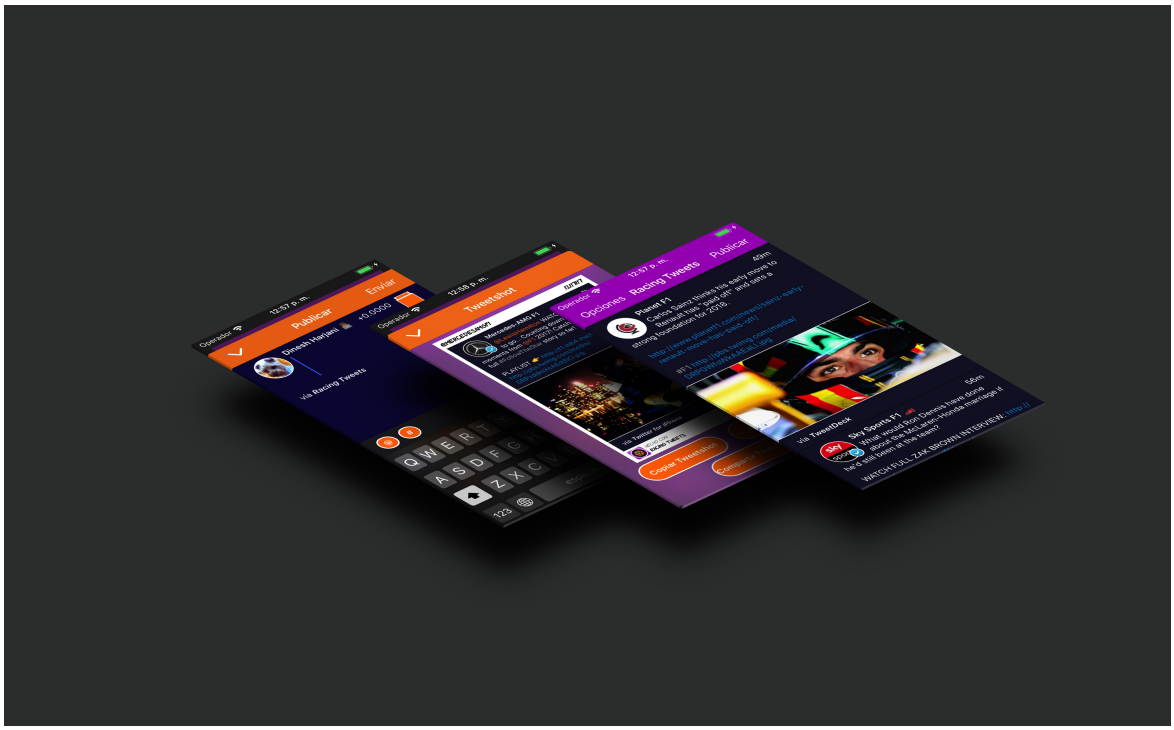


Figure 11: Tweet Editor, Tweetshot Preview, and Timeline screens from Racing Tweets v2.0.

We've figured out, from the User Interface perspective, what we want the Standard Engine to handle for the `UIViewController`:

- **Asynchronous Data Fetching** (including Network Requests)
- **Data Visualization** (in `UITableViewCell` format)
- **Asynchronous Data Updating** (including Network Requests)

I don't know if you can tell just from reading the bullet points above, but the Standard Engine is going to end up being pretty massive in size, given the large number of responsabilities it has. And we're not even counting with all the other

requirements it has by itself, which we'll see in the next section.

But before we get into that, there's more we need to dicuss. So far, we've only looked at **Racing Tweets** from the perspective of what its main screen, the one showing the combined list of tweets of different motorsports, will require. But is there more to any app, besides delivering its main feature, regardless of how complicated that feature might be? Well, of course!

2.4.1 Settings Screen

Not pictured at the beginning of this section, is an integral part of **Racing Tweets** which the user absolutely needs access to, and that is the Settings screen. From there, the user can choose which motorsports to follow, which language the press and commentary Twitter accounts should be in, and whether or not one of our marquee features, the Racing Filter, should be enabled or not. But besides all of this, the Settings screen also serves as the hub for all the Developer options which, once enabled, opens up the possibility of changing Racing Tweets' server address, looking at Standard Engine stats like the number of tweets currently loaded, reading the engine's logs, and more. In version 1.x, you could even enable a Navigation Bar extension - the big purple stripe across the screen reading "Racing Tweets" - updating the amount of memory used by the app, the timeline's length, and the current rate limit on the Twitter API[19].

But there's more. Some tasks, within the Settings screen, need screens for themselves for things like looking at the Standard Engine's logs, checking which tweets we've filtered for the user, or providing feedback to the developer[viii]. For all of these and more, we need to navigate the user to a different screen, or `UIViewController`, from within the Settings screen. This is something we need to take into account when thinking about which `UIViewController`(s) "talk" or interface with which other, since **it allows us to place mental boundaries on the communications within them, and thus, structure them and architect them properly**. For example: since the Settings screen needs to show the Standard Engine's logs, and also, allow the user to switch motorsports and control whether the Racing Filter is turned on or off, it stands to reason that **the Settings screen needs access to the Standard Engine instance being used in the main screen's** `UIViewController`, and that it must be passed on to it as a reference.

viii. That's me!

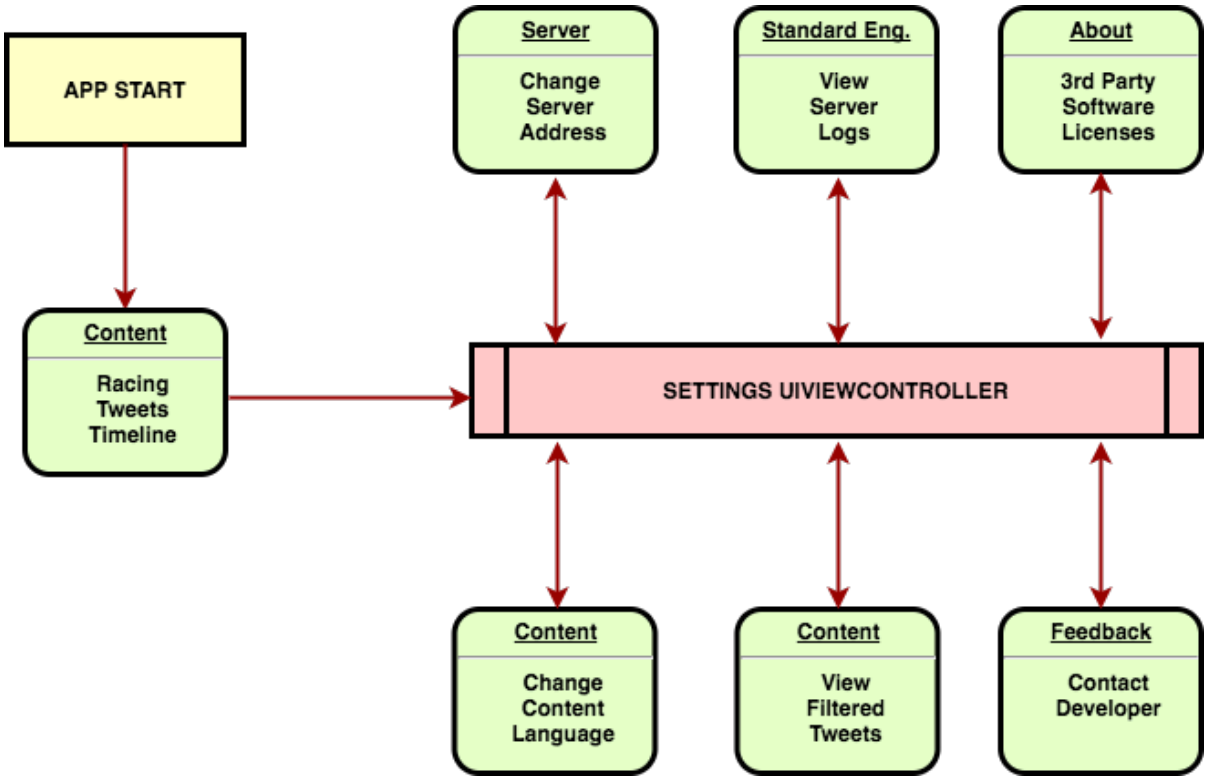


Figure 12: Racing Tweets' Settings User Flow Diagram. The red and green boxes represent "screens", or UIViewController(s), in some shape or form.

The Settings screen represents an adventure in and of itself, though at a different level to the complexity that the **Racing Tweets** Timeline screen represents. We need a clean way to separate these two, both at an architecture, and at a user perspective level, so the user also understands that they're in the Settings area, navigating screens belonging to the Settings user interface. So now we need to solve two distinct problems:

- **Architecturally separate Settings-derived** `UIViewController`(s), with access to the main timeline's Standard Engine.
- **Inform the user they're no longer in the Racing Tweets Timeline when entering the Settings area.**

The first bullet point is not really a problem. It's simple: all the **Racing Tweets** Timeline needs to do is fire up the Settings `UIViewController`, pass on a Standard Engine weak reference[ix], and then let the Settings `UIViewController` communicate whatever is needed with its own `UIViewController`(s) as it sees fit. But the main

ix. Remember: the Timeline UIViewController is the owner of the Standard Engine, so it can pass a weak reference to it.

`UIViewController`, showing Twitter content when the app starts, should remain completely decoupled from these other `UIViewController`(s), because **it has no need to directly communicate with them**.

As for the second point, we could argue a lot about how to structure navigation in modern iOS Apps, but that's not our goal here, so we'll just hop straight into our solution, and explain the thought process behind it.

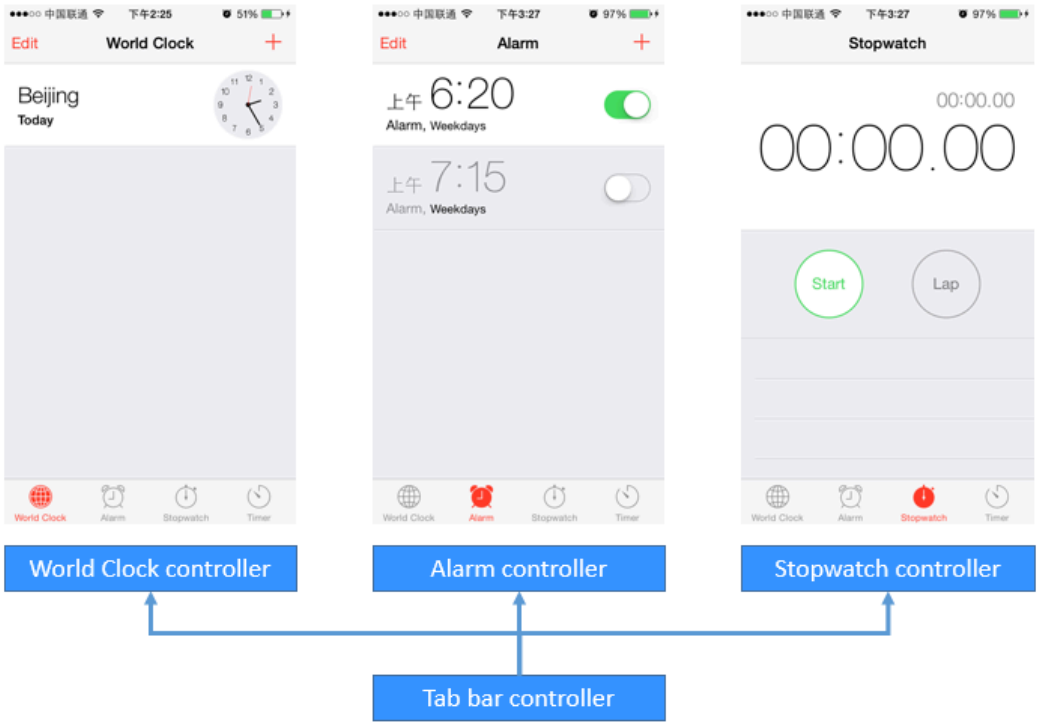


Figure 13: UITabViewController example.

We are not fans of "tabs", those buttons at the bottom of the screen that allow you to drammatically switch contexts within the same app. We think they take away precious screen real estate from the user, specially in an app like **Racing Tweets** where we aim to show as much content as possible. So we never really considered adding Settings as a tab option, or any tab control whatsoever. We are fans of the `UINavigationBar`, which is the name of the piece of UI common to most iOS apps standing right at the top of the screen[x], so we added a Settings button over there. When tapped, the Settings screen springs above the Twitter timeline with an animation, giving the user a clear visual cue that they're no longer in the same User Interface plane as they were before. This is called a modal-style transition in the land of iOS development. From there on, the user can navigate in an out of different

x. The strip of gray at the top of the different screens in the graphic above reading "World Clock", "Alarm" and "Stopwatch" is the UINavigationBar.

screens, or `UIViewController`(s), using the standard push and pop navigation (screens moving right to left to enter, and left to right to exit), powered by the common `UINavigationController` element.

2.4.2 Gluing It All Together

Other screens, like the Tweet Editor `UIViewController`(s), also appear above the main timeline. Accessing the Tweet Editor is as easy as opening the Settings screen: it's the other button in the `UINavigationBar`, visible from the main screen. And since, from the Tweet Editor, the only thing the user can do is either cancel or post their tweet, there's no new screens to navigate from there. Similarly, we have two other modal screens the user can access from the **Racing Tweets** Timeline: the full-screen photo viewer, and the Tweetshot screen. Both of these show up modally, but the first one covers the entire screen, and doesn't animate the Timeline to appear below it, since photo viewing needs to be immediate, and not necessarily cause a shift in the user's perspective[xi]. Tweetshots, on the other hand, represent a screen capture of a tweet the user is either long-pressing, or applying pressure to (3D/Force Touch)[20], so we do animate it like the Tweet Editor and the Settings screen. But again, these are very simple screens, and unlike the more complex Settings `UIViewController`, the user can't navigate anywhere else from there.

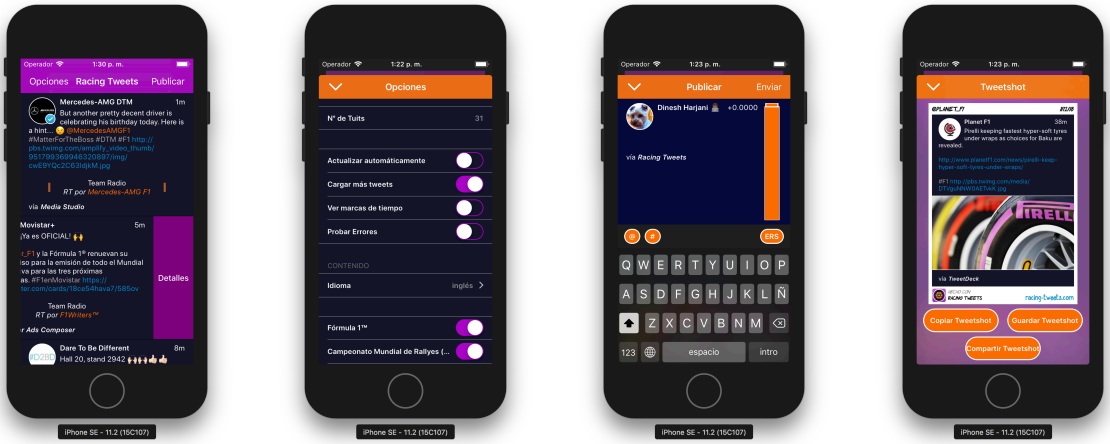


Figure 14: Timeline, Settings screen, Tweet Editor and Tweetshot preview screens from Racing Tweets v2.0.

For the 2.0 release, another screen was part of the specification: a tweet "details"

xi. The only reason we we fill the screen with a photo is to allow full user immersion in their content.

screen, accessible by sliding a tweet to the left. It borrows heavily from concepts introduced by major iOS apps, including the official Apple Mail app, and instead of relying on a 3rd party implementation, it uses new APIs introduced with iOS 11. However, the screen was not completed before the project was terminated, and is only used to view specific tweet data for debugging purposes. Still, the underlying navigation structure was put in place, to allow the user to navigate past that tweet, similarly to how the user moves through the various Settings' screens, relying on another `UINavigationController`.

With all the pieces on the board, we can finally start picturing what our app's navigational structure looks like.

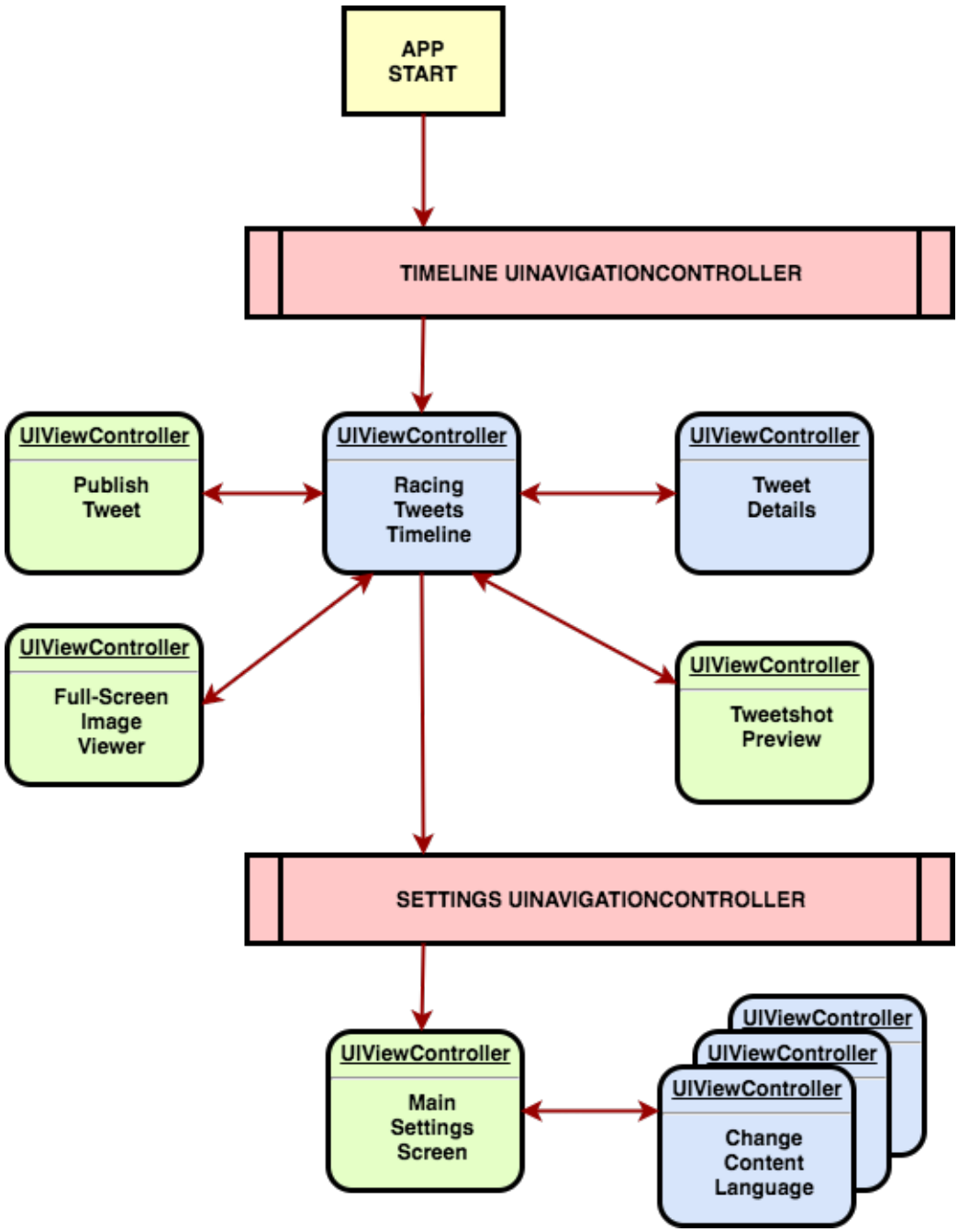


Figure 15: Racing Tweets' Complete User Flow Diagram**.**

Legend: Red boxes are `UINavigationController` instances, the `UIKit` element responsible for backing the right-to-left (to enter or push) and left-to-right (to exit or pop) transitions for `UIViewController`(s). There are only two places in our whole app where we want to navigate between screens this way: the main Timeline screen and in Settings, so those are our two `UINavigationController`(s). Green boxes trigger modal transitions, causing a new `UIViewController` to show up above the View Controller they're transitioning from, while blue boxes stay inside the same navigational level, by "pushing" or "popping" screens.

2.5 Standard Engine

2.5.1 Component Overview

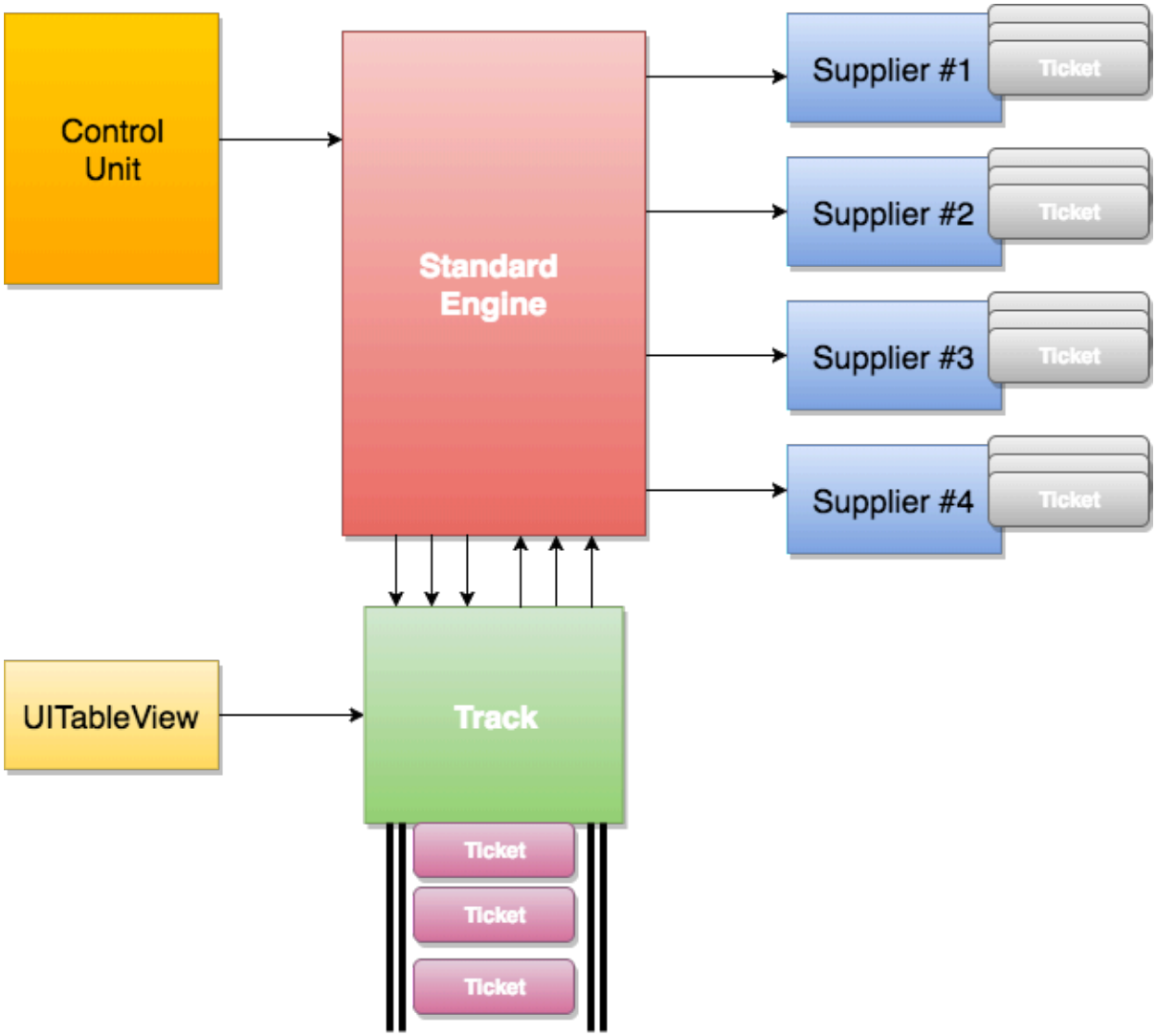


Figure 16: Standard Engine Overview**.**

To begin our deep dive into the Engine, let's introduce these new concepts:

- **Ticket**: it represents a single item of information that needs to be displayed to the user. It contains a lot of varied information like a User object, a text section, a timestamp, an id, a weak link back to its Supplier, multimedia (URL) links such as photos, videos or weblinks and so on. **In our context, every Ticket represents a tweet**. But a Ticket could also represent a Facebook post, an Instagram photo, a Pinterest pin, an RSS feed item, and

so on.

- **Supplier**: it represents a stream, an input or source, of Tickets. The concept behind a Supplier can be as coarse as fetching data from an entire social network, such as Twitter, Mastodon[21] or micro.blog[22], or it can be as fine as the tweets from a single user on a single social network. Generally, we associate a Supplier with a single social network. In the case of **Racing Tweets**, which targets 2 different languages and 4 different motorsport branches, up to 12 Suppliers can be active at any single time delivering new content. The requirements **the Engine expects from the Tickets by a Supplier are that they must be ordered logically and have unique IDs**; the Engine will take care of about making sense of all the Supplier's streams.

- **Track**: it was envisioned as the data source for the `UITableView`. It is the processed, chronologically ordered and logically correct sequence of Tickets returned by the Standard Engine upon a request for new Tickets. Every Ticket must have a unique combination of string ID, Supplier ID and timestamp, which means **no two Tickets from the same Supplier can have the same ID nor the same date & time**. The Track is not mutable from a 3rd party perspective, so only the Standard Engine can make changes to its Track. **There is only one Track per Standard Engine instance**.

- **Control Unit**: originally named *Commander*, **the Control Unit is the means through which a developer can alter the behavior of a Standard Engine instance**. Its most important role is returning the correct Suppliers to the Engine when it requests them, in order to add new Tickets to the Track. But it is coarsely-aware of the Engine's lifecycle, so it can be used to plug-in external behavior to the Engine, in conjunction with custom Suppliers. For instance, the Racing Filter feature which allows the user to see which Tickets have been filtered, is implemented through the Control Unit, which picks up any Tickets filtered by the various Twitter Motorsport timeline Suppliers.

- **Standard Engine**: it is a single instance of a Standard Engine, an object. It is not a Singleton, so there can be multiple Engine instances running in the same application. It requires a Control Unit in its constructor to perform its operations, and the Control Unit cannot be changed once the Engine is instanced. Its API is mostly a façade[xii], wherein all operations take place in one or more background threads, so as to not block the UI from ever

xii. By this we mean the Façade Design Pattern.

handling any kind of user input.

There are many, many more elements and concepts that comprise the Engine as a whole, but going through all of them here would not be practical.

What we do find interesting, is the fact that all of the above elements were shaped in pen & paper before ever opening Apple's IDE, Xcode. At the time, we thought this made sense, since we didn't know how complicated the algorithm for intertwining Tickets from different Suppliers and ordering them would look like. So we thought it'd be appropriate to explore these issues outside the bounds of a code editor, and instead trust drawings, blueprints and pseudocode. This was an important step, as it provided us with a very valuable standing ground from which to begin the coding phase, once we were certain that the our logic for the code block adding new Tickets to the Track was sound. However, the development phase hit multiple rocks every time we discovered simpler concepts we'd not accounted for that were needed. This meant that our existing concepts became bloated, absorbing more responsabilities than originally intended, causing a lot of the logic to became messier[xiii].

A quick example was that of the Timeframe; **a Timeframe is a block of time, from A (start value) to B (end value)**, where B can be `nil`, in which case the Timeframe represents *everything that comes after A*. Timeframes, like the Track's Tweets, work in descending order in the Standard Engine, so if a Timeframe begins today at 9:41 AM and lasts 10 minutes, the end value is 9:31 AM. If the end value were to be `nil`, the Timeframe would cover from 9:41 AM until the beginning of time. Before we made them a class, the constant use of Timeframe(s) for key pieces of logic meant we were constantly carrying around a couple of time objects, or `NSDate`(s), representing a Timeframe's start and end values, plus related information about that object, like whether it was infinite (no end value), or its timespan (amount of time between its start and end values, measured in seconds). Once we decided to refactor all Timeframe-related code, our logic became a lot easier to read, and many other concepts evolved naturally from there, like asking: "how many Tickets does a Supplier have within this Timeframe?" Questions like these, in turn, allowed us to package other concepts into classes too, simplifying our code and improving our

xiii. We never reached I am CEO levels of code disorder, but it wasn't pretty. A lot of our logic contained ad-hoc'ed concepts that needed to become classes onto themselves, complicating readability.

architecture[xiv].

If there's a lesson to be learnt from this process, is to **never drop the pen & paper attitude from your mind**; always be on the lookout for concepts you didn't realize were needed before, but may have been relying on. Don't just assume that, because you've carefully analyzed what you need to do by hand first, that you haven't missed anything. In our case, we focused on what we thought was the most complicated problem, and didn't bother to even look to see if there's something else that needed our attention. Or even if we were making assumptions about concepts that needed to be coded into a class, but that we weren't, despite the amount of times we reused them. The DRY principle[xv] really helped us in realizing how many concepts we were missing, after we began converting repeated code into classes. It is not easy to admit publicly that you've made mistakes, but we think it's important because once you do, you can take action, correct it, and open up yourself to wonder if you're missing other things, too.

2.5.2 Requirements / Features

xiv. To do this, you need to be willing to refactor your code constantly, whenever you understand how a concept can be packaged into a class to improve your architecture. TDD is very helpful for this workflow, to ensure you don't break your use cases.

xv. DRY stands for Don't Repeat Yourself.

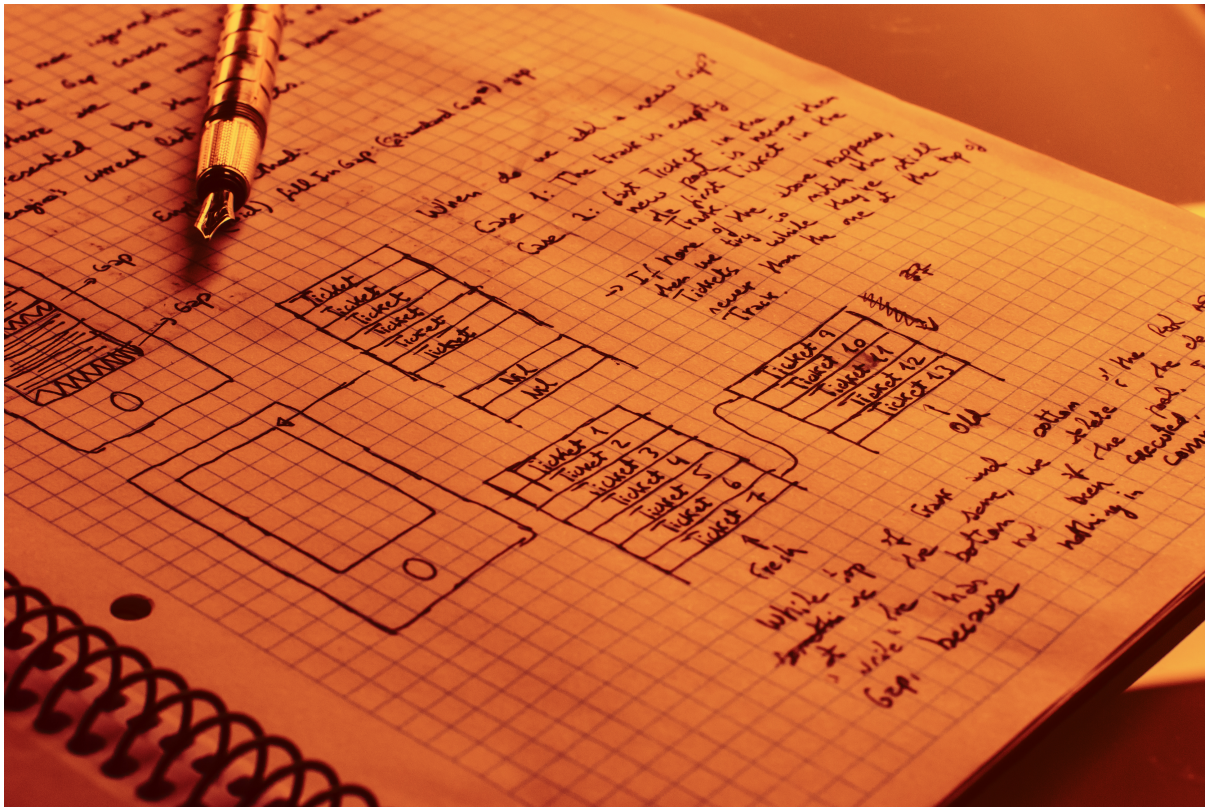


Figure 17: Early drawings of Standard Engine Gap Support**.**

Thus far in this work, we've alluded to how the Engine itself had a set of logic requirements, or features, it had to meet in order to fulfill the engineering task it set out to achieve. For the purposes of diving even deeper into the inner workings of the Engine, we thought now was the best time to lay out exactly everything the Engine needs to do, and add some hints as to how those problems have been resolved.

But before you go through them, I'd like to add that, in hindsight, it's no wonder for me that it took so long to fully implement the Engine, given the massive amount of features it had, and how worried I was about building something that, again, could break with a single change here or there. For sure, **there are weak spots in the Engine**, which are outside the scope of this work, **and it is not a perfect design,** but it meets its most important goals, and it works. Hopefully, you'll also take away from here that it's important to measure the scope of your projects before commiting to them, so that your "God-level Developer complex" doesn't win the battle against yourself. Having said that, I think there's something to be said for actually achieving whatever it is you set out to do, even if in our case the Standard Engine hasn't powered any commercially successful apps yet. After all, sometimes you do need to be a little bit crazy, to make the impossible come true[23].

2.5.2.1 Logical Coalescing

The Standard Engine's main purpose is to make sense of all the disparate content, in the form of Tickets, produced by the Suppliers attached through the Control Unit. This means they must be chronologically ordered and coalesced into a single, unified, list or array, **without ever losing any of the user's content in the process**. This is critical, because if the end-user perceives the Standard Engine is not performing well its duties and gives the impression of misplacing or losing user content, by either changing its position in the list, or removing it but adding it later through a different operation, it can seriously affect the perceived quality of any product based on the Standard Engine.

Specifically for us, it was obvious that if we managed to build and release a break-through product combining multiple social media feeds, anyone could verify the validity of our technology by simply looking at the feeds individually and then checking back with our app. This feature was also the main reason why we were building the Standard Engine, so there was no way we wouldn't be willing to do anything it took to ensure this requirement was fully met.

2.5.2.2 Time-awareness

One of the first curve-balls[xvi] we found a couple of months inside development is that tests and fake data don't represent the chaos of the real world, wherein a subset of data can span widly different amounts of time, going from mere seconds, all the way up to complete decades. For example, if we plug into a trending topic in Twitter, we might have thousands of tweets in the same second. Yet, on a well-known but hardly updated blog like Hypercritical[24], there are maybe only a hundred posts going all the way back to 2005.

This has big consequences, because it means the Engine must be capable of both cutting new content down to a reasonable amount of new items, as well as being able to navigate backwards in time through history to find new content if there's nothing new. Timeframes were key to implementing this feature, as well as the idea of the exponential backoff implemented in the TCP protocol[25], which we shamelessly adapted into the Standard Engine.

xvi. "Trouble with the Curve"?

2.5.2.3 Multi-threading & Responsiveness

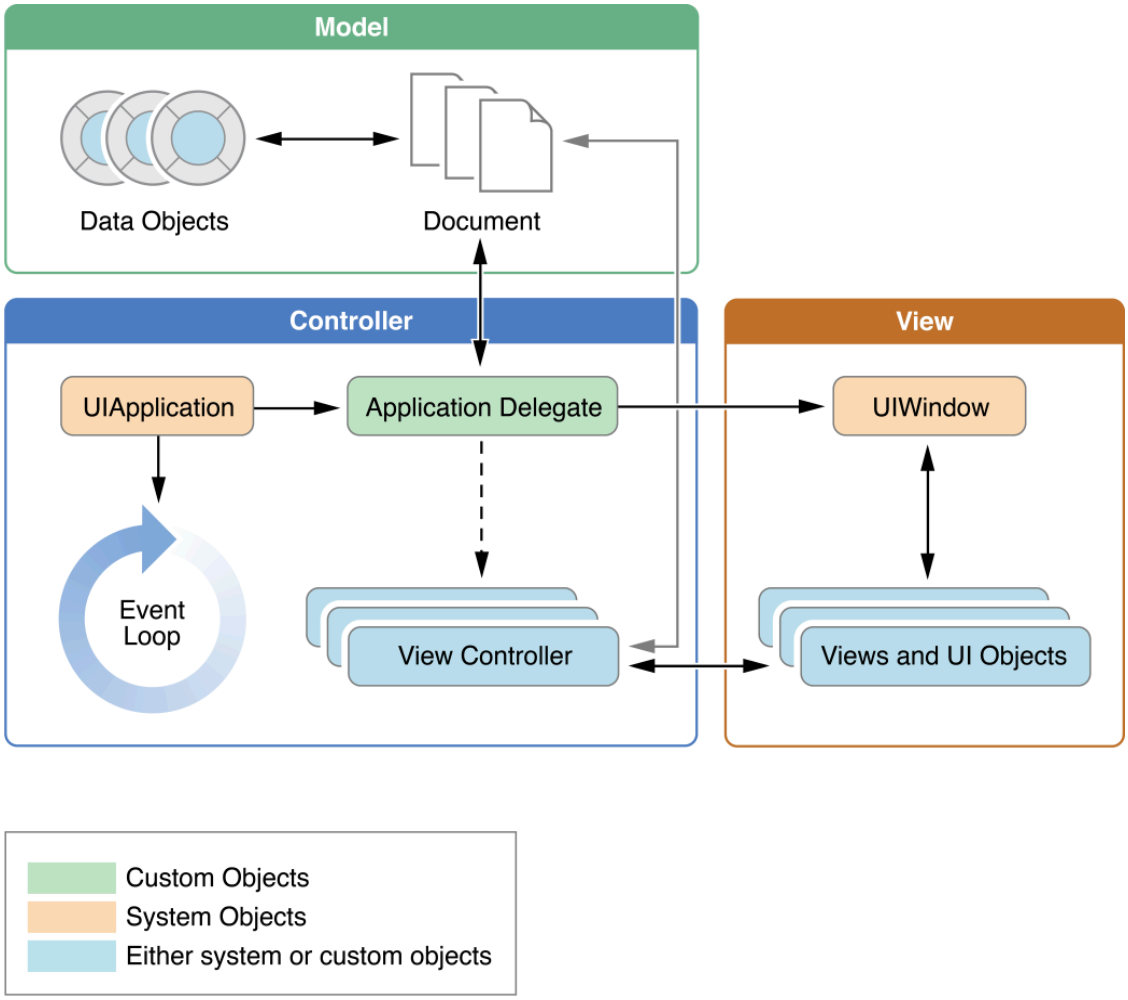


Figure 18: iOS Event Loop Diagram. **Source**: Apple.

Modern software development revolves around the concept of EDD, or **Event-Driven Development**. This means that most of the code is executed in response to an event, such as a user tap on a button, instead of applications being a set of instructions executed serially like in a bash script. When the user taps on a button in the interface, a call is made inside our codebase, and from there we can launch as much complexity as is needed into the CPU/GPU to achieve the user's desires. To this end, the Standard Engine set out, from the very beginning, to have an architecture based around these events, rather than fighting against them. We want the Engine to exploit the ever-increasing parallelization abilities of modern hardware as well, and support unobtrusive communications back-and-forth between the User Interface layer and the Engine itself.

2.5.2.4 Multi-queued

This was not met. Neither back when the Engine was deemed finished in 2014 to start building **Racing Tweets**, nor now, when we're onto version Racing Tweets v2.0, more than 3 years later. The idea behind this was to not only allow the Engine's operations to be backgrounded, but to also give Suppliers a "pipeline API" for events, such as a status change in the UI after a Twitter retweet, a Facebook Like, a Pinterest Pin or an Instagram comment. It was dropped due to the ever-increasing complexity of the codebase, but on hindsight, it sounds like many hard-coded features in **Racing Tweets** could be simplified if the Engine were to add this feature. Future work?

2.5.2.5 Gap Support

This one's pretty big. Like many other features, it emerged from inspiration of my favorite iOS Apps from that time, one of which was the popular 3rd party Twitter client Tweetbot. Back in 2010-2012, developers used to test their abilities and prove their worth by writing their own take on a Twitter client[xvii], before Twitter severely restricted access to its API[26]. Tweetbot emerged in 2011[27] causing a huge impact, implementing complex features with a much better end-user experience, and leaving the official Twitter client for iOS quite behind. One of the key features it implemented better was "gaps", or `UITableViewCell`(s) showing up in the timeline that represented missing content and, upon being tapped, triggered a background request causing said gap to be filled with new tweets. Even though Gap Support was always part of our concept, the Engine had to be nearly scrapped in its entirety in 2013[28] in order to properly implement it. This is because Gaps and the *time-aware* requirement are intimately related, since content that has been cut from being added to the Track must be covered, or included, inside a Gap, to ensure no content is ever hidden or mismanaged by the app, and therefore not shown to the user. We explain this a bit better in the Looking Down section.

2.5.2.6 Bulletproof

xvii. The trend seems to have shifted towards podcast clients now. The newest example being Jared Sinclair's 'sodes app.

From our prior experience attempting to write **I am CEO** back in 2011, we learnt that constant refactoring and tests were essential to ensure the long-term health of any project. With this in mind, we wrote tests for the Engine every step of the way, attempting to both lock-in the behavior of small components like the Timeframe class or the Ticket Containers[xviii], as well as testing the Engine as a whole, to ensure refactoring didn't make it break or stop covering a particular use case. This has had many reassuring moments, such as when large-scale changes were made. But it came at a price of keeping the tests themselves up to date, and properly taking care of them like any other part of the app's codebase. As such, tests required building their own separate framework to ensure the asynchronous Standard Engine calls could even be tested[xix], as well as the complicated subject of how the Engine internally asks its Suppliers for Tickets, requiring the development of a test Supplier stack as complicated, if not more, than real-life requests to a service like Twitter or Facebook. Note that **our tests cover the Engine, and not the UI**.

2.5.2.7 User Protection

From the very beginning, we envisioned how, for the **Fiber** app, multiple Suppliers would hit different servers across the world, and that any one of them might fail, either because that service failed, or because the device suddenly lost connectivity. It could even be that the developer wrote some bugs into the Supplier. With this in mind, the Standard Engine not only controls the Tickets added to the Track, but it also sets safeguards against Suppliers that don't return any Tickets, or that take indefinitely to return them, in which case the Track update is cancelled and an error is bumped up to the user, in the form of a Standard Engine Notification, for the `UIViewController` to show.

2.5.2.8 Highly Extensible

The Standard Engine exists to solve the complicated problem of unifying information from completely different sources. To that end, specific API parts were built to allow easy extension, such as the Supplier API. However, the Control Unit,

---

xviii. They're what you think they are: classes that aggregate Tickets and enforce invariants, such as ascending or descending order.

xix. From our experience, most testing frameworks, including Apple's own, are designed to test synchronous APIs, not asynchronous. We wrote our own framework, using semaphores.

as it name implies, is the easiest and most important way a 3rd party developer has to influence how the Engine works, without having to deal with all of its internal workings. Callbacks within the Control Unit API ensure it is always aware of the internal Engine cycle, so different, unthought of features can be built on top of this "plug-in" concept.

2.5.2.9 User Interface Customization

Even though from the start the Standard Engine was built to fill in `UITableViewCell`(s), mirroring the style of a Twitter client, the API around it evolved as the presentation became more and more complicated, allowing for the creation of an API focused sorely on the presentation of Tickets. The level of customization here can be as fine as having completely different presentation for Tickets from the same Supplier, or as coarse as having all Tickets, even though they are technically produced by different Suppliers, share the exact same presentation code. The latter is actually the case with Racing Tweets, but wasn't the case for **Fiber**, where we expected to use different presentation styles for each Supplier. You can read more on this topic in the Annex II chapter.

2.5.2.10 Developer-focused

The Standard Engine was built to solve a problem for the developer, and in this predicament, we included making the Supplier API as friction-less as possible, as well as complimenting the Standard Engine with tools that can reduce the amount of work required to use it. For example, `UITableView`(s) accept being told the exact rows that are about to change in the data they're showing to the user, and to make the job easier for the developer, there's an API in the Track that returns which rows have been removed and added, to enable easy animations for each case. Furthermore, calls to the Standard Engine object itself are automatically enqueued, so if the user calls for a Gap to be processed, and then also asks for a refresh operation asking for the newest Tickets, the developer doesn't have to add this extra control layer above to ensure the operations are performed in the right order. Instead, the Engine will do this on its own, and return back to the caller with the appropriate state to keep the `UIViewController` as simple as possible. This is also where the 2.5.2.4 Multi-queued facet, which was not implemented, starts to make sense, as it was intended primarily to simplify the developer's tasks and keep as much complexity as possible within the Engine itself.

2.5.2.11 Adaptable

Combinining feeds from multiple sources and coalescing them logically and chronologically involves complexity, which might not be required, for example, if all we want to do is use the Standard Engine to query a tweet's replies. See what I mean? We already have code that connects the Track to a `UITableView` and shows our tweets just as we want, but now we might want a different screen where **we only need to show a simple set of tweets that doesn't require coalescing nor ordering work from the Engine**. How would you solve this? One way could be instancing a new Standard Engine and attaching a Control Unit that only returns a single Supplier for said tweet's replies.

But, if we could engineer a way for the Standard Engine to shed all the systems required to coalesce and order multiple feeds, we could have a much more efficient solution for simpler tasks like showing a tweet's replies. Making the Engine adaptable enough to support such an internal algorithm swap was indeed made, but we didn't effectively work on what we coined the *Single-Supplier Standard Engine*. Like many other things from this list, we considered it to be an "easy task", but became well-versed enough in Software Development to know we had to account for unforseen details. Just like the Multi-queued requirement, it holds promise for future work though.

2.5.2.12 Track-based Supplier Filtering

Not to be confused with the Racing Filter feature, this is fully implemented and has been working for many years now. It is also a good example of both implementing features before needing them, and in doing so, implementing them the wrong way. The idea behind this is to allow the user to filter out content from all Suppliers except for one, which made sense in the context of **Fiber**, where we believed filtering out all other social network Tickets and leaving those from just one Supplier, was a feature the user would appreciate[xx]. In truth, this is not a hard feature to implement, but the consequences for us are that this code is not effectively used, because for the purposes of **Racing Tweets** it doesn't make any sense. As stated before, **Racing Tweets** works by combining more than 10 Suppliers, so filtering out

xx. For the Fiber app, we envisioned the use of one Supplier per social network.

all of them and leaving just one isn't coherent because each motorsport/language combination is comprised by at least 3 individual Suppliers. The feature could be adapted to **Racing Tweets** to filter out all available motorsports except for one, but due to the way the feature was implemented, it isn't viable and would require some architecture work to fix it. At this juncture, we've simply kept it going as it was originally written, without investing more time in it other than ensuring we don't break its tests.

2.5.3 The Developer's Perspective

If we had to make any guesses, we'd say that at this point, your perception of actually using the Standard Engine in one of your projects is pretty much none. So, to convey what it's like to work with the Standard Engine in an app, we're going to do a walk-through of what it takes to set it up. Be it to power a social media app like **Racing Tweets**, or some other feature in a completely different kind of software. We will go through as many details as possible within our word budget, while also keeping the conversation light, so as to not turn this section into an excercise you, the reader, have to follow paying careful attention[xxi]. **We don't want to bombard you with details; we just want to give you an outline**.

There are three critical components to our discussion: **Tickets**, **Suppliers**, and the **Control Unit**. There's a fourth one, the **Presenter**, which we wrote about, but had to move out into the Annex II chapter. In any case, dropping that section does not affect us, since it does not change how the Engine itself works. What you do need to know, is that Presenters are the API mechanism in the Engine where Tickets are transformed into `UITableViewCell`(s). Having said that, let's answer the following question: what do we need to do to have a working Ticket?

2.5.3.1 Data Items

Nothing, really. The Standard Engine supplies the `DHSETicket` class, which has been tested to within an inch of making me lose my sanity, and there's nothing we have to do other than to actually use it. Like most of the Standard Engine components, it has been tested to comply with both Facebook and Twitter's APIs, and we can choose to either use it as-is, to extend it, or to dial it back as much as

xxi. I love reading technical books at night, too.

we need: we can use the `DHZEBareTicket` class, which is a barebones implementation of the `DHZETicket` protocol that spares us the details of having to implement Ticket equality and ordering[xxii]. As an API, `DHSETicket` provides:

- **Everything included in the `DHZEBareTicket` implementation**: String ID, a weak link back to the Ticket's Supplier[xxiii], a timestamp (`NSDate`) and Ticket equality and ordering implementation.
- **Text field**: A String, which represents the Tweet text itself, a Facebook post's text, an Instagram photo's text, etc.
- **User field**: a weak link[xxiv] to a `DHSEUser` object, which is not a requirement within the `DHZETicket` protocol, but an addition of the `DHSETicket` class. It contains basic user information such as an ID, a name field, a username field, and a URL link to a profile image.
- **Application Name field**: a simple String with the name of the application or Supplier that produced this Ticket.
- **Image URL field**: a URL object, or `NSURL`, pointing to an image that might be attached to this Ticket.

Because both **Fiber** and **Racing Tweets** were very tied to the use of Twitter, a `DHSETicket` subclass, called `DHSETwitterTicket`, is also included as part of the Engine. It adds support for basic items such as number of favorites, number of retweets, attached media items (URLs), weak links to another Ticket representing a quote or retweet (which themselves are `DHSETwitterTicket` instances), and a new class, `DHSETweetEntity`, which is used to help format the eventual text shown to the user, such as username (**@dinesharjani**) syntax coloring, hashtag (*#RacingTweets*) coloring, URL (www.racing-tweets.com) coloring and highlighting, etc. The engine also includes a Tweet parser, and a Twitter Supplier that can be

---

xxii. Ticket equality and order revolves around a unique combination of Ticket ID, timestamp and Supplier ID.

xxiii. This prevents a reference cycle (see next footnote); the Supplier is the Ticket's owner, not the other way around.

xxiv. In languages like Java, (post-ARC) Objective-C and Swift where the developer does not need to manage memory, objects are kept alive in memory by referencing one another. If two objects reference each other, neither can be evicted from memory, even if they're not used, because they're both being strongly referenced. To this end, weak links are used, to clearly define an object owner, and therefore break these "reference cycle(s)".

reused for many API queries.

2.5.3.2 Data Source

For the purposes of our example, we will assume what the `DHSETicket` class provides is enough for our purposes, so we must now follow on to a more important subject, the Supplier itself. Suppliers have a very simple API, wherein they must have a Supplier ID, a name, a Presenter, and they must conform to a couple of functions which belong to the `DHZESupplier` protocol, in which the following function is key:

Language: Objective-C

```
- (void) fetchTicketsFrom:(id<DHZETicket>)sinceTicket until:
(NSDate*)until withBlock:(TicketsDeliveryBlock)block;
```

Figure 19: DHZESupplier API required function.

This is the function the Standard Engine will call to request new Tickets from the Supplier. The last argument is the least important one: it's a callback in which we return to the Engine the Tickets it's requesting from us. Which Tickets is it requesting from us? Let's look at the first argument, in which the Engine provides a reference Ticket[xxv] that can, and will be, `nil` on accasion. If it is `nil`, the Supplier must assume the Engine is requesting the newest content (Tickets) available, and **if it's not** `nil`**, the Supplier must return the Tickets that immediately follow the** `sinceTicket` **argument**. The second argument is when things start getting tricky: the Engine is also asking us, the Supplier, to return all of the Tickets that belong in the Timeframe starting at the `sinceTicket` argument's timestamp, and end in the `until` argument timestamp. This means the Supplier is not only responsible for fetching the content, but that it must also do a small amount of work for the Engine.

I didn't like this design. But I understood too late that I was adding more work to the Suppliers than I had already envisioned, once I began to consider actually implementing a Facebook or Twitter Supplier, instead of the test Supplier for my tests, which generated Tickets programmatically in intervals of one second. To solve it, rather than altering my protocol definitions, I added more tools to the Zealot Engine layer:

---

xxv. id<DHZETicket> is Objective-C for "a reference to anything conforming to the DHZETicket protocol".

Language: Objective-C

```
//
//  DHZESupplierCacheManager.h
//  FiberiOSFramework
//
//  Created by Dinesh Harjani on 10/13/13.
//  Copyright (c) 2013 Dinesh Harjani. All rights reserved.
//

/*!
 This protocol must be implemented by any Supplier who wishes
to have its Cache management to be taken care of by an
instance of DHZESupplierCacheManager.
 */
@protocol DHZEManagedCacheSupplier <DHZESupplier>

/*!
 Returns an array of hard-fetched Tickets
 This method is a requirement for all users of the
DHZEManagedCacheSupplier class. It requires Suppliers to hard-
fetch new data happening after the given Ticket or, In its
absence, from the given reference Date.
 Under no circumstances is the callee asked to perform any
form of optimizations with this data: the caller guarantees
these calls will be kept to a minimum since it assumes
fetching new Data is expensive.
 */
- (NSArray*)fetchTicketsSince:(id<DHZETicket>)sinceTicket
withDateReference:(NSDate*)reference;

@end
```

Figure 20: DHZEManagedCacheSupplier protocol definition.

In other words, if we write a Supplier that conforms to this new `DHZEManagedCacheSupplier`, we only have to worry about a single callback, `-fetchTicketsSince:withDateReference:`, and always **return Tickets that follow `sinceTicket`**, without worrying about calculating which Tickets belong to the requested Timeframe. If more Tickets are needed, we will be called again for them - no additional logic is required to keep track of which Tickets we've been previously asked for accross different calls to this function.

In practice, there's another class, the `DHSEBaseSupplier` abstract class, which forces us to implement that callback, and it also provides us with `DHSETicket` presentation for free, in the form of the `DHSETicketPresenter`. What we don't get

for free is management of the `DHSEUser` pool of objects, because it might not be necessary; in the context of Twitter, Facebook or Instagram, it is quite likely that the same user will post multiple items, so we might want to reuse User objects to reduce our memory consumption. But in a personal blog's RSS feed, all items *belong to the same user*, so we didn't feel the need to add this in. Other tasks, such as data parsing and authentication, also need to be handled by the Supplier subclass we're writing.

2.5.3.3 Engine Control

This is the `DHZEControlUnit` API:

Language: Objective-C

```
//
//  DHZEControlUnit.h
//  FiberiOSFramework
//
//  Created by Dinesh Harjani on 4/10/13.
//  Copyright (c) 2013 Dinesh Harjani. All rights reserved.
//

@protocol DHZEControlUnit <NSObject>

- (NSDate*)suppliersTimeThreshold;

- (void)prepareForNewPool;

- (NSArray*)suppliersCoveringTimeframe:(DHZETimeframe
*)timeframe;

- (void)cleanupAfterPool;

- (void)restartSuppliers;

@end
```

Figure 21: DHZEControlUnit API.

As was the case with Tickets, we don't really have to strictly implement any of these. We can just compose our own `DHZEControlUnit` protocol-compliant class with a `DHSEModularControlUnit` instance, which we can simply call-through for each of the above calls, and also set up using its own API:

Language: Objective-C

```
//
// Created by Dinesh Harjani on 7/12/14.
// Copyright (c) 2014 Dinesh Harjani. All rights reserved.
//

@interface DHSEModularControlUnit : NSObject<DHZEControlUnit>

- (void)addSupplier:(id<DHZESupplier>)supplier forKey:
(NSString*)supplierKey;

- (id<DHZESupplier>)supplierForKey:(NSString*)supplierKey;

- (void)removeSupplierForKey:(NSString*)supplierKey;

- (BOOL)supplierInstalled:(NSString*)supplierKey;

@end
```

Figure 22: DHSEModularControlUnit API.

Essentially, these four methods allow us to dynamically add and remove Suppliers whenever we need to[xxvi], whether it's only once on app startup, or multiple times during app execution, and not have to worry at all about any kind of logic whatsoever[xxvii]. Still, the complete power of the underlying API is available if we ever need to use it. And for the sake of adding yet a little bit more light into the magic that happens within the Standard Engine, we're going to cover two items of the `DHZEControlUnit` API:

Language: Objective-C

```
@protocol DHZEControlUnit <NSObject>

- (NSDate*)suppliersTimeThreshold;

[…]

- (NSArray*)suppliersCoveringTimeframe:(DHZETimeframe
*)timeframe;

[…]

@end
```

xxvi. You could even change a Control Unit's Supplier while the Standard Engine is fetching new Tickets, but why would you want to do that?

xxvii. By logic we mean keeping track of whether a Supplier has been already added or not. That's what the Supplier keys are for in the DHSEModularControlUnit API.

Figure 23: DHZEControlUnit Supplier Time Threshold API.

We really hope you don't actually hate us for this, but, there's a piece of the Supplier API we've been hiding from you, which is their **time threshold**. Now, to understand what it is, we're going to look at it from the perspective of the Control Unit itself. As we've mentioned before, **the Engine asks the Control Unit for Suppliers to query for new Tickets**, and it does this specifically with the second API call, – `suppliersCoveringTimeframe:`. This call basically means *give me the Suppliers I can ask for content within this Timeframe*, which the Control Unit is expected to return. Failure to do this, effectively means the Engine cannot work properly, since its safeguards will prevent it from adding Tickets to the Track belonging to a Timeframe it isn't asking Tickets for. However, this has a reverse implication you can use to your advantage: **as the Control Unit developer, you can alter where and when the Standard Engine tries to pull content from the Suppliers**.

Of course, the latter is not the intended use case scenario; if you wanted to license from me the Standard Engine for use in one of your products, you'll most likely just want it to work as it was intended to do. And to do that, you can safely rely on the aforementioned `DHSEModularControlUnit` to handle all of this for you, but, **your Suppliers still need have to properly implement their time threshold property**. Essentially, a Supplier's time threshold is the limit back in time at which point it is known that there won't be any new content (Tickets) available.

Language: Swift 4

```
@objc override open var timeThreshold: Date! {
    let dateComponents = DateComponents(calendar:
Calendar.current, timeZone: TimeZone.current, year: 2006,
month: 1, day: 1)
    return Calendar.current.date(from: dateComponents)
}
```

Figure 24: Supplier Time Threshold Sample implementation.

As a practical example, consider an account that only has 3 tweets, but our user expects to see at least 30-40 Tickets on each query for new Tickets. The Engine will attempt by all means possible to extract those 30-40 Tickets from its Suppliers, but it'll be impossible to find more than those 3 Tickets. And once it backtracks in time past January 1st 2006, it's impossible for any Twitter Supplier to deliver any Tickets, because 2006 is the year in which Twitter was founded. By checking against the time threshold, the Engine can enjoy a quick exit in its algorithm to fetch new Tickets knowing it isn't missing any user content, because there isn't any. Starting

from there, we can choose how efficient we want our Suppliers to be; an RSS feed Supplier might return as its time threshold the timestamp of the feed's first post, instead of returning March 1st, 1999[29], to cover its back. Conversely, the time threshold can get tricky if Facebook suddenly elects to support backdating posts all the way back to the 1800s, but that complexity will be handled by the `DHSEModularControlUnit` class for you, as long as you set proper values for your Suppliers.

2.5.3.4 View Controller API

This is the easiest part! All we have to do, is essentially copy and paste existing code from Racing Tweets :) We're going to begin with instancing the Standard Engine, and setting up some of its initial parameters, from within our `UIViewController`:

Language: Objective-C

```
DHTdF1EngineControlUnit *controlUnit =
[[DHTdF1EngineControlUnit alloc] init];
self.standardEngine = [[StandardEngine alloc]
initWithControlUnit:controlUnit];
self.controlUnit = controlUnit;
```

Figure 25: Standard Engine instantiation.

Those are all of the steps required to set-up a working Standard Engine instance, provided we have a Control Unit with the appropriate Suppliers. To start filling its Track with a fresh batch of new Tickets, this is all we have to do:

Language: Objective-C

```
[self.standardEngine fillAt:[DHSERefreshMarker ticket]];
```

Figure 26: Standard Engine Refresh call.

That only covers the logic side of things. But what about actually showing those Tickets? Well, provided we have a working `UITableView` instance, all we have to do is properly implement the functions defined in the `UITableViewDataSource` API:

Language: Swift 4

```
override open func tableView(_ tableView: UITableView,
cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let ticket =
self.standardEngine.track.ticket(atTrackIndex: indexPath.row)
    let presenter = ticket.supplier().presenter
    let cell = presenter.cell(for: ticket, for: tableView, at:
indexPath)
    return cell
}

override open func tableView(_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
    return self.standardEngine.track.count
}
```

Figure 27: Standard Engine UITableViewDataSource protocol implementation.

Barring some logic safeguards, this is all you need to do to see your Tickets show up in your iOS app's `UITableView`. Complexity begins when we need to start thinking about UI State management, error states, bi-directional communications between the Engine and the `UIViewController` and so on which, to the best of my knowledge, is complexity we will always have to deal with. Until I get angry enough that I write my own "Standard Engine for User Interface complexity".

2.5.4 Looking Down (Pool Cycle)

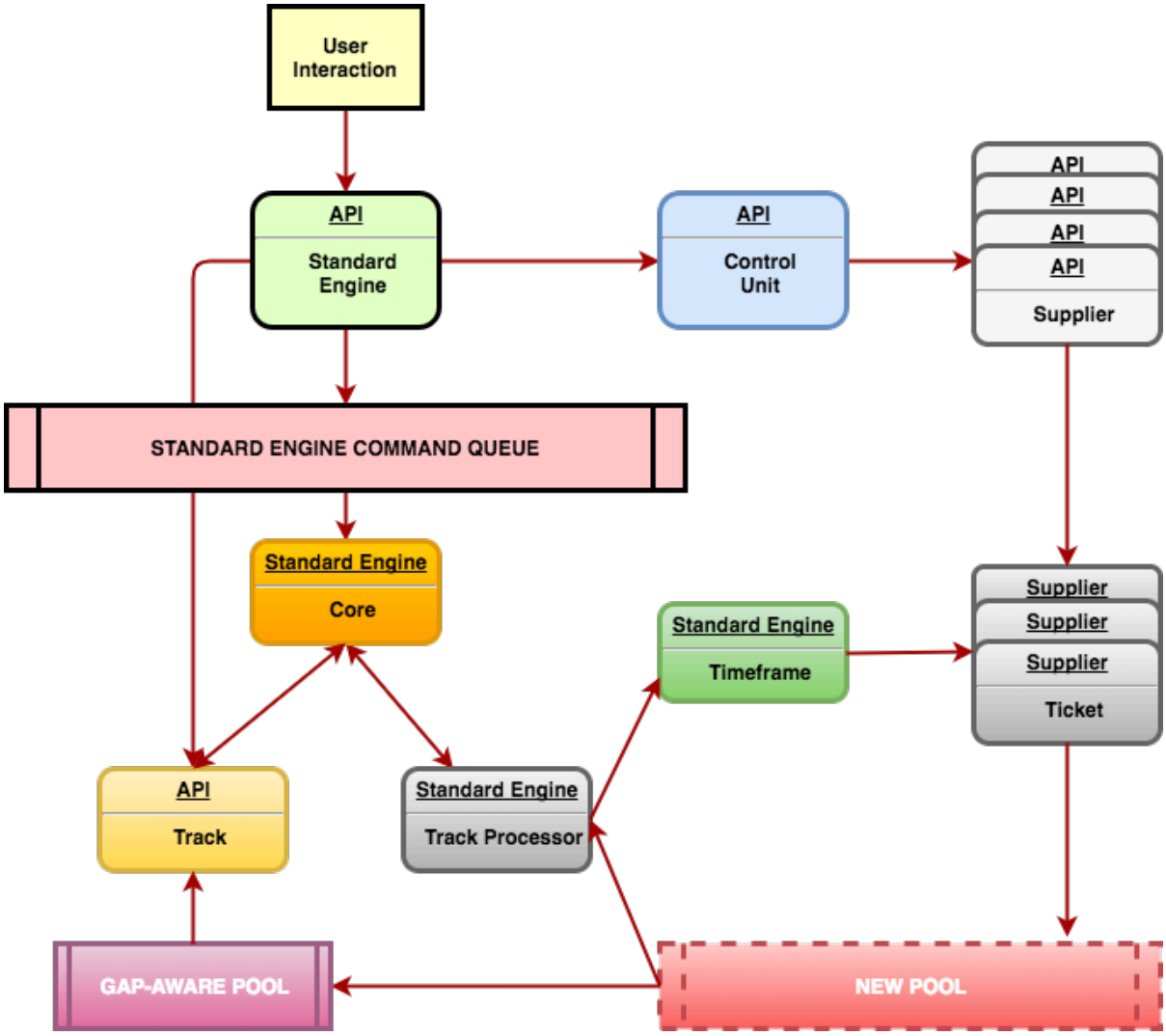


Figure 28: Standard Engine Pool Cycle Diagram.

We'd like to finish our discussion about the Standard Engine giving you a rough idea of how it all comes together to produce new Tickets and adding them to the Track. This explanation will not be perfect, since we will not cover exactly every single function at hand coordinating this effort. But if we do our job well, you'll be able to walk away from this section knowing, more or less, what the Standard Engine's internal algorithm is.

We begin with the *User Interaction* box at the top, which essentially stands for a callback in the `UIViewController` or similar class, which is part of the topmost presentation layer of the app. This component has access, directly or indirectly, to the Standard Engine instance. For this example, we're going to assume we've already set up our Standard Engine with our `UITableView`, as seen in the previous section. When our User Interface code is called, we presume the user wants to

either add new Tickets[xxviii], or to fill in one of the two types of Gaps supported: a **Standard Gap, which has a beginning and and end**, and is *sandwiched* between two sets of Tickets. Or an **Infinite Gap**, which is always the last Ticket in the Track, and **represents *old* Tickets we've not loaded into the Track yet**.

When we perform a call to the Standard Engine, said call is actually enqueued, and then the code directly returns to the User Interface Thread. The real execution for the requested operations happens from within a serial background queue, so we are free to make multiple calls from the User Interface, and have the Engine work through them in succession, while informing us at the User Interface layer, of all the changes as they're processed. The key call to perform all Gap filling operations happens upon the **Standard Engine Core**, which is a small class co-ordinating the efforts between all the appropriate actors to correctly add new Tickets to the Track. The true owner of both the Track and the Control Unit, in programming terms, is the Core; not the outer layer Standard Engine instance. The Core is also the owner of the Track Processor, which is the "pluggable" component responsible for implementing the Standard Engine algorithm[xxix]. The Core is also responsible for handling the Engine's state (has the operation succeeded? Are we cancelling it?), and of communicating back to the User Interface Layer, through a different component, whether the operation was successful or not.

In order to move forward though, we need to add a new concept: **the Pool**. And the easiest way to explain what the Pool is, is to have a look at the Track, which we view as our source of truth, and the final result of any new Standard Engine operation involving pulling content from the Suppliers. The Track needs to be logically perfect, with no duplicate Tickets of any kind, nor any missing Tickets, and said Tickets must also be ordered in descending timestamps. But handling this ever-increasing succession of Tickets is complicated, in the sense that the more Tickets get added, the more complex it is to keep everything in order. That's what the Pool concept aims to do, but in reverse: the Track is static and doesn't change, other than when new Tickets are added. This means we only have to worry about maintaining the Track's logical requirements when new Tickets are added, and we only need to worry about the new Tickets, not the old ones. So in truth, **the structure that enforces all of the Track's invariants is the Pool, which represents the new Tickets we want to add to the Track, and not the Track**

---

xxviii. This would be a "Refresh" operation in Standard Engine parlance, hence the DHSERefreshMarket ticket class seen before.

xxix. Our "Single Supplier Standard Engine" concept, would actually only need to swap over this Track Processor component to completely change the Engine's behavior for filling the Track.

itself. And **the component responsible of filling the Pool with new Tickets, is the Track Processor**.

Upon the Standard Engine Core's request, the Track Processor begins a loop that stops when at least one of the following two conditions is met:

1. **The Pool has more Tickets than the defined maximum number of Tickets per Pool**[xxx]. This usually happens due to two reasons:

    - **The initial batch of Tickets returned by the Suppliers covers our initial Timeframe** and the Pool's length already exceeds the maximum number of Tickets per Pool.

    - The Track Processor has iterated through the loop, **extending the Timeframe covered** by the Pool, until **the length of the Pool exceeds the maximum number of Tickets per Pool**.

2. It is impossible for the Track Processor to add more Tickets to the Pool, because **all the Time Thresholds of the Suppliers returned by the Control Unit have been met**.

xxx. This number can be programmatically changed easily by a 3rd party developer.

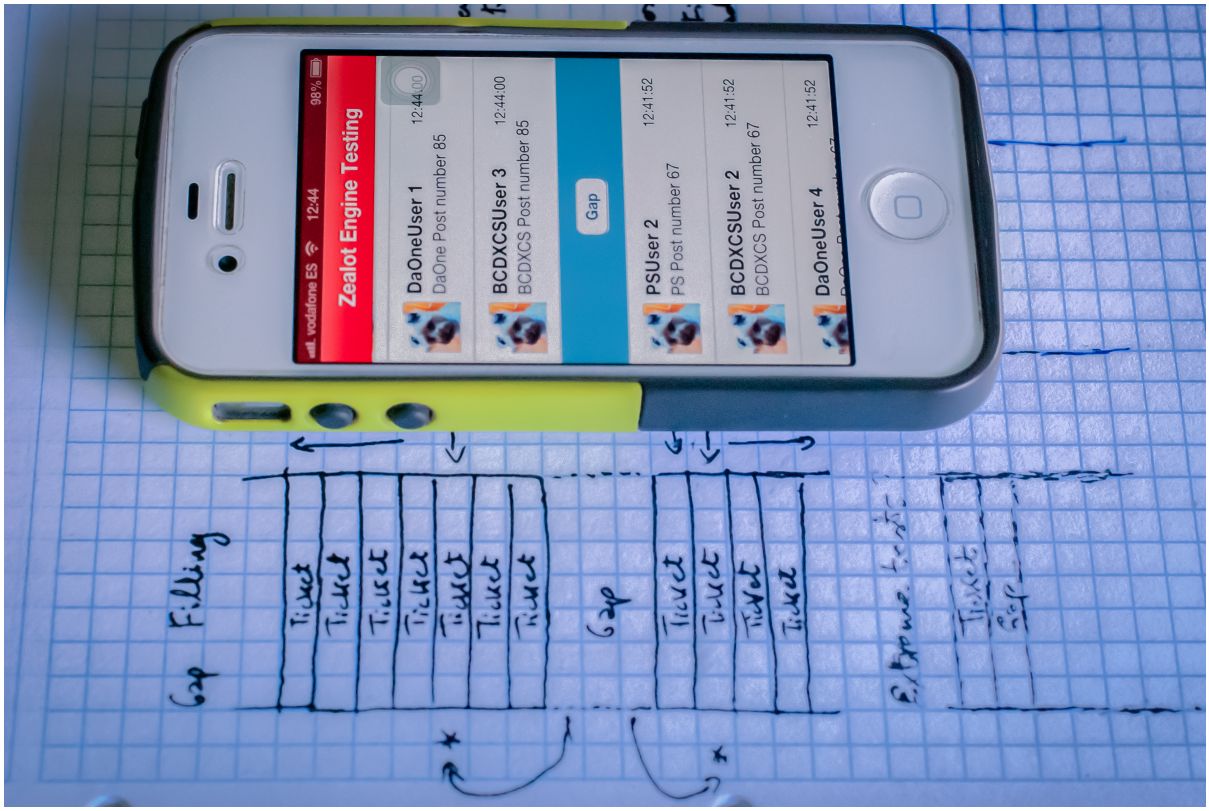


Figure 29: Drawing showcasing programmatically generated Tickets for Gap Support testing, circa 2013.

The Track Processor's guiding principle is to **fill the Pool**. It does this by defining a standard Timeframe of 5 minutes, and by filling the Pool with the Tickets that logically belong into that Timeframe. If the first few Tickets from the Suppliers don't cover that Timeframe or don't hit the minimum number of Tickets required (both need to be true), we iterate again extending the Pool's Timeframe and requesting new Tickets from our Suppliers, until we've met both requirements. **Only if we reach all of our Suppliers' Thresholds does the Track Processor drop the minimum number of Tickets required**, since there aren't any more, and **we can be certain, from a logical standpoint, that we're not missing any of the user's content**. Remember: the Track Processor knows which Suppliers have valid content (Tickets) for the Pool it is trying to fill because it can ask them, through the Control Unit. Also note that, in practice, there's more logic involved regarding the Pool's Timeframe, since the Pool might be trying to fill a Standard Gap whose Timeframe is less than 5 minutes.

Once finished, the Track Processor will return said Pool back to the Standard Engine's Core. At this point, **the Pool contains both logically and chronologically ordered Tickets from all of the Suppliers**; we know they're logically correct because Suppliers need to worry only about returning Tickets that follow each other

logically. And we know they're chronologically correct because both the Track Processor and the Pool enforce this invariant[xxxi].

When control is handled back to the Standard Engine's Core, it will call upon the Pool Object itself to perform what turned out to be one of our nightmares: **Pool Cutting**, also known as the *Gap-Aware Pool* box in the above diagram. **Pool Cutting is the process of fitting the new Pool into the Track**; this includes making sure the continuity between the Track and the Gap we are filling isn't broken, and that no duplicates are added in the process. To do this, we first cut down the Pool to be our maximum number of Tickets per Pool, or lower. This is necessary because when the Pool is filled, the Track Processor stops looping when the Pool size exceeds the minimum, so the Pool might be larger than we intended it to be. Once cut, we now face another problem, which is that **the Timeframe covered by the Pool is not deemed complete**; in other words, we don't know if we have all the content between the newest and the oldest Ticket in the Pool. For example; our Pool may now have Tickets ordered chronologically between 16:00.00 and 16:05.20, but **we don't know if we have all the Tickets that belong to the second between 16:05.00 and 16:06.00, because they may have been cut**; all we know is that we have a Ticket in our Pool with a timestamp of 16:05.20[xxxii]. As a solution, **during Pool Cutting we cut again down to the second**, so in our example, our Pool's Timeframe would be clocked with a start value of 16:00.00 and an end value of 16:05.00, because we are once again certain that we're not missing any content within its bounds. We know because the Pool is chronologically correct, and because we had Tickets in the Pool past 16:05.00, therefore, we have everything in between 16:00.00 and 16:05.00.

Now that our Pool is both logically and chronologically correct, smaller than our maximum Pool size, and the Timeframe it covers is complete, we need to fit it into the Track. This involves continuity checks regarding what comes after and before the new Tickets we're adding to it, and more checks to see whether the Pool sufficiently covers the Gap, or if it adds another one. This is the reason why we keep Pool Timeframe(s) and Gap(s) independent of each other, and in different steps of the process: **a Pool might initially cover a Timeframe, but once it's been cut down, it might no longer do so**.

---

xxxi. The Pool itself is a TicketContainer class.

xxxii. Complicating matters is the fact that we had to write Timestamp comparisons of our own, because Foundation's NSDate class wouldn't return true to 16:05.19 being older than 16:05.20, and so on.

Finally, when the Track has been updated with the Pool's Tickets, and the Ticket row changes calculated, the Core sends a `DHSENotification` to the User Interface layer, informing it that the operation has been completed, and our background queue operation concludes, allowing for the next operation, if there is any, to be performed.

2.6 Grab Bag

Figure 30: More Racing Tweets v2.0 Promotional Artwork.

Unfortunately for us, there are many, many, many things we don't have time to talk about. But I want to do more than just say "I want you to know they're there"[30], and bail out. Instead, what we're going to do is close this chapter by going through some interesting tidbits we didn't get to talk about in detail, to further extend your knowledge of what is required to build a modern mobile application.

2.6.1 Self-Sizing UITableViewCell(s)

This one has to be the holy-grail for all iOS Developers. `UITableView`(s) have been a part of iOS Development since the very beginning, much like the `ListView` and its `Adapter` classes have been for Android. However, making the `UITableViewCell`(s) have different sizes has always been a bit of a nightmare, and has required a lot of different tricks over the years. Back in the day for example, one possible strategy was to split a `UITableViewCell`'s height into the height you knew about, and the height of the elements you didn't know about, which was usually some text element. Then, you'd insert an invisible `UILabel` item into your UI, outside the `UITableView`, and every time a new cell needed to be drawn, you'd set its text into the invisible `UILabel`, force it to resize, measure it, and then return the corresponding `UITableViewCell's` height. And again, **you'd do this for every cell**. It wasn't pretty, but the upside was that once you got it to work, there were no messy hairs to contend with.

Nowadays, `UITableView` has some magic[31] to avoid these games, but the downside is that `UITableView` never ends up behaving the same way across iOS releases, since the underlying API class, and its cell height calculation algorithm, are constantly being tweaked. In turn, you end up having to hack the API forcing it to present itself like you'd expect it to[xxxiii].

2.6.2 Aspect-ratio consistent UIImageView(s)

Because I'm never happy, I decided that solving variable-length text-based `UITableViewCell`(s) wasn't enough for my ego, so I decided to tackle having full-bleed images, going edge-to-edge, that were resized to always keep their aspect ratio. This proved to be an utter nightmare, until I decided I couldn't go against a simple fact: `UITableViewCell`(s) need to know their height, for a given Ticket, **before they are shown**. Notice the bolded text, because that is the key here; when we had to size the cells for ourselves in Racing Tweets 1.0, our images were of a fixed height, and we managed to calculate the height of the text, and of the cell, before it was shown to the user. What doesn't work is what we tried for a long time: to show a cell, and then dynamically change its height to that of the image we have fetched after showing the cell to the user. In this case we were constantly seeing cells have the wrong height for the image they're trying to show, because they were assuming the height of the wrong image. Then, we'd fight the API to make the cell adapt to a new image's height and wonder why it didn't have the correct height in

xxxiii. We admit to not being the most knowledgeable people on the subject, but if only Stack Overflow wasn't full of similar complaints…

the first place. We were not very smart going about this problem to be honest, and much energy was spent complaining about the API, rather than thinking about where the problem really was.

Thankfully, Stack Overflow came to our aid in explaining this to us, and Twitter made its part in making it possible, by providing in its API the dimensions of each image entity in the Tweet object itself. Once we parsed that data and added it to our `DHSETwitterTicket` class, we knew the image's aspect ratio, and together with the screen size, we had the corresponding aspect-ratio corrected height for each image before it was downloaded from the CDN, meeting `UITableView`'s requirements.

2.6.3 Text flowing around UIImageView(s)

Figure 31: Comparison with other Twitter clients. **Left to Right**: Twitter for iPhone, Twiterrific for iPhone and Tweetbot 4 in their dark trim, followed by Racing Tweets v2.0.

Apple loves to pride itself in its APIs, some times for good reason. Android, for example, had to hack a way together **upon release** to recicle `View`(s) and show them in a `ListView`, lest the Garbage Collector kicked in and caused scrolling to stall, making the developer responsible for implementing this needed optimization. iOS had this from day one, and it was also the first modern mobile platform that was launched. This means Apple can really provide great APIs, as they often do. The problem comes when you try to use shiny Apple APIs outside the realms of the cool demo they used to announce them, like the TextKit API that allows you to set boundaries around which your text elements should be able to flow[32]. To get the `UITableViewCell`'s text to flow around the user's image (see above), we had to

place the UI element underneath the title and make it avoid the `UIImageView` to its left, rather than placing the text element at the top, and giving it the username and the profile picture's boundaries for it to avoid on its own. I can confidently say that **no other Twitter client, nor social media client, does this.** For what it's worth, we've done our part by trying to share and spread the knowledge[33].

2.6.4 Hacking UITextView to not accept user input

The API to achieve flowing text mentioned above, can only be used with a particular class that is designed for user input (writing text with the soft keyboard). We didn't want to enable the user to write on top of our Standard Engine Tickets' `UITableViewCell`(s), and we couldn't just make the UI element non-user interactive, because otherwise the user can't tap on any links, nor press down on a `UITableViewCell` to take a Tweetshot. We tried so many things that[34], to this day, we still haven't figured it out completely how to solve all the small issues, but we moved on past a certain point, and now we're not sure if we'll be able to come back and finish it completely.

2.6.5 Custom-Drawn UIView(s)

We are incredibly strong proponents of keeping app sizes as small as possible, and one of the culprits of bloated apps is the amount of resources, namely images of different kinds of shapes and sizes suited for all kinds of devices[xxxiv], that are required in a modern app. Our solution for this is one being adopted by high-profile indie developers in the industry: using code to draw, primarily through tools like PaintCode[35]. At a practical level, it's not that different from loading a bitmap into memory, uncompressing it, and sending it over to the GPU to show in the display. The only difference is, instead of loading an existing image, we use code to draw the image in a buffer, convert it to a bitmap, and from there on the process ramains the same. For us, the advantages of using code rather than bitmap graphics far outweighed the inconvenience of having to use custom software to draw them:

---

xxxiv. Contrary to what many believe, the iOS ecosystem does suffer from device fragmentation: iPhone SE, iPhone 6/7/8, iPhone Pluses, and now the iPhone X, all need their own sets of resources. The issue is further excarberated by API changes through OS releases, and the iPad's own fragmentation.

- No increase in app package size[xxxv].
- No need to add new resources every time there's a new type of device introduced.
- Images are always drawn in memory to fit the size, in pixels, they're going to occupy, so they're both pixel-perfect and a lot more efficient than their bitmap counterparts.

On our Timeline, the blue tick sign signaling Twitter Verified User(s) and the circle around each user profile image are both drawn from code. Other elements drawn in real-time using code include the image loading graphic, which is an F-1 tyre with accompanying rim, the ERS battery[xxxvi] for the Tweet Editor, the downward facing arrow for modally-shown screens, the rounded buttons, and more.

2.6.6 Asynchronous background loading of UIImageView(s)

We could go miles and miles about this one, and we're kind of embarrassed to say that we haven't completely nailed it yet, but we at least have a much better solution than what we hacked together for **I am CEO**. The subject at hand is that whenever a `UITableViewCell` is about to be shown, we fire a background request to fetch any images it might have. When the images arrive, we need to set them in the right `UIImageView`, which gets incredibly tricky because `UITableViewCell`(s) are recycled within their parent UITableView[xxxvii]. So the user might be flicking through the list of tweets, firing lots of requests, stop at some point, and then multiple images will be returned for the same `UITableViewCell`, because while the user was scrolling, that `UITableViewCell` represented many different Tickets. And once the background queue starts returning multiple images for the same `UITableViewCell`, how do we know which one is the one we need to show right now?

---

xxxv. Compared to images, code does not add any significant amount of weight (size in MB) to our app.

xxxvi. Under current regulations, F-1 cars harvest their wheels' kinetic energy to charge the Energy Recovery System's battery. This battery energy can then be deployed to the vehicle's rear wheels.

xxxvii. UITableViewCell recycling also affected us when we were trying to have full-bleed images that kept their aspect-ratio.

Our current solution is to use the tag field of the `UIImageView` class, and assign it the hash of the latest image's URL. When multiple images are requested for the same `UIImageView`, all of these requests will have multiple hashes, but only one of them will have a hash matching the tag field of the `UIImageView`. Said image is the one we set.

2.6.7 UIImageView Cache(s)

We wrote a full blog post covering most of this subject[36], so we'll be even more sparse than usual on the details here. What we want here is to keep a good cache of all the images we've had to fetch for the user, so they may not be re-fetched, and thus use more battery power and network data, if the user scrolls back up through the Track to reveal an image we downloaded earlier. Also, we want to avoid having to reload all the images if the user leaves our app, and then returns. Initially, and going as far as the first two releases of **Racing Tweets**, we implemented a cache ourselves, until we tried a 3rd party library, and never looked back since.

2.6.8 Tweetshot(s)

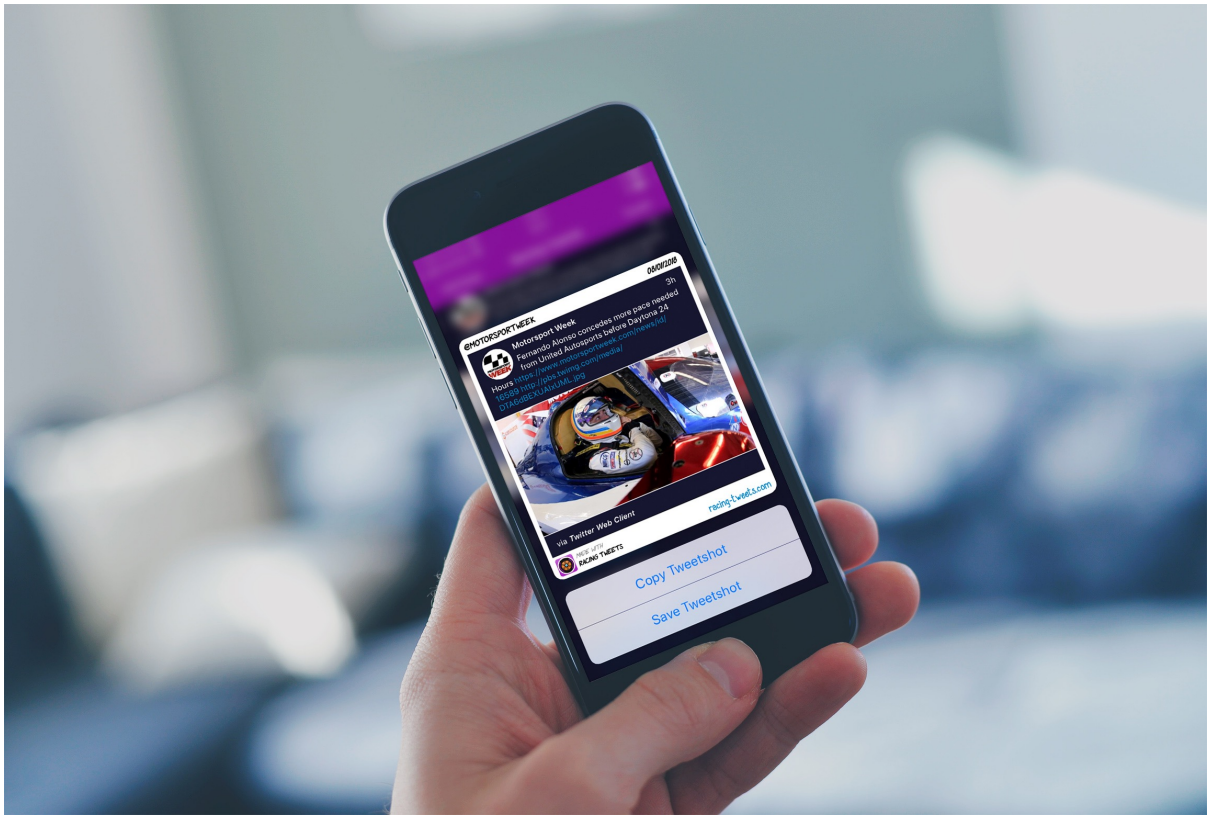


Figure 32: When a UITableViewCell is 3D-Touch'ed, the above UI element pops above said cell. This use case was the root of the Tweetshot idea.

We would really, really love to show you how these were implemented. They we such a beautiful headache; beautiful because even though the **Tweetshot** as such exists on Product Hunt[37], we don't think people really know what they are. Maybe for good reason, since **Racing Tweets** hasn't caught on either. Tweetshots were our way of showcasing how the user could customize **Racing Tweets** via custom backgrounds, as well as a quick and easy way to share tweets not via links, but via images. Even though links are cool, images still have this raw feeling of showing others exactly what we've seen and how we experienced it, therefore containing more meaning. This is something you lose when sharing a link, where usually, the other person needs to open the link for themselves.

In any case, photographing a `UITableViewCell`, wherever it may be in the UITableView[xxxviii], in a way that allowed for their background to also be drawn into the image buffer, was a wonderful technical experience, knowing very few have actually attempted, and been successful, at something like this. I'd even go as far as saying that very few would be able to produce a good knock-off given the

xxxviii. Even if you hard-pressed on a Ticket's UITableViewCell that only had its title visible, we'd be able to snapshot the whole Ticket and show it to you in an instant.

months of engineering it took to reach our level of polish[xxxix].

Breathe Dinesh, breathe.

xxxix. To give you an idea, in the iPhone SE, a UITableViewCell might be taller than the space the user has to see the Ticket in their screen, specially if said tweet has a tall image. We also solved that ✌🏾.

3.The Closing

3.1 What Does Software Architecture's Future Look Like?

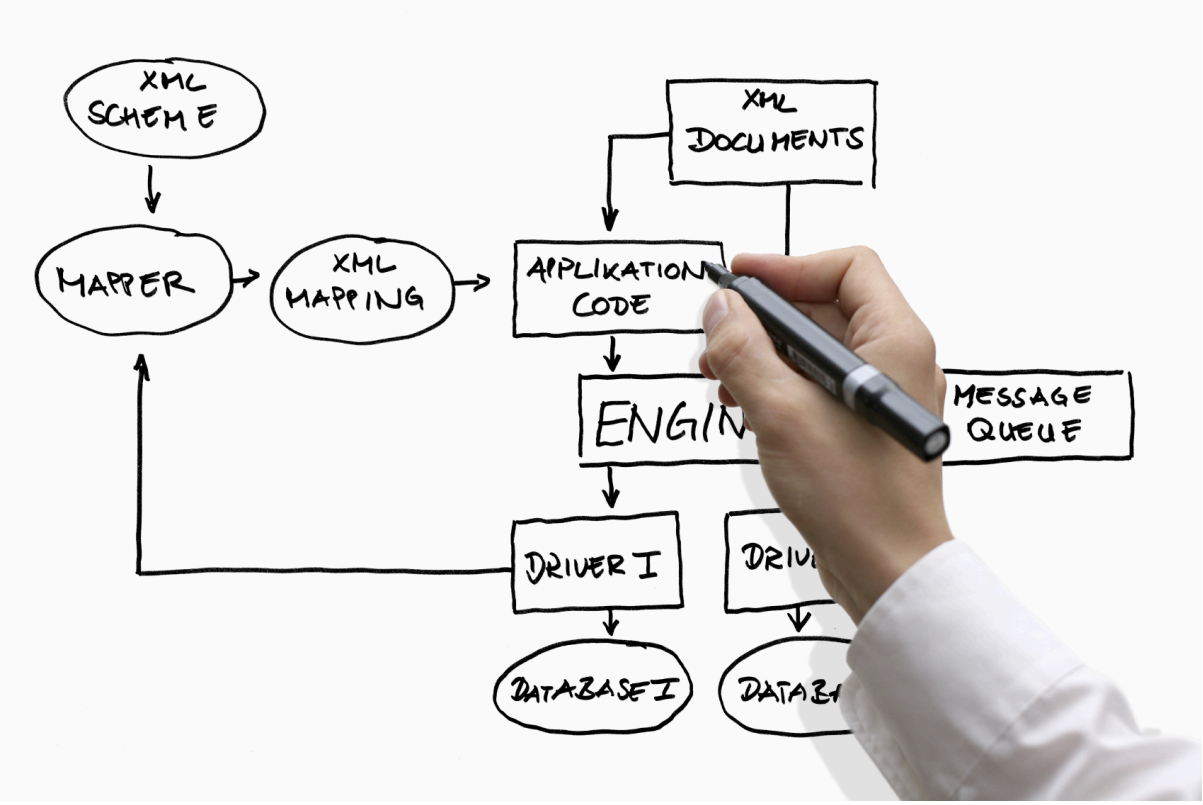


Figure 33: Software Architecture.

It is hard to tell. One would hope that, as time goes on and our craft grows, together with our systems' complexities, that our knowledge and understanding of **how to work better with code**, grows too. After all, this isn't too much different compared to how airlines have learnt that training their crews as individuals isn't enough, and that they must be trained too in the human arts. To help them enable each other to work together towards a common goal, rather than everyone pulling in their own direction under wraps[38]. We face the same problem: code is not written in a vacuum by computers, it is (mostly) written by people working in teams, each one building a piece of the bigger puzzle. It's true that you can't prepare for how all the pieces should fit together - God knows I've had to rewrite tons of code around the Pool, Track Processor and Track classes to adapt - but when the new knowledge hits you, you've got to take a step back, and see the big picture. And if you're afraid to do it, you need to speak up and find someone else to do it, or your project will crash at some point.

It's actually not that hard to understand: there's hardware, and then there's software.

Everybody knows not to mess up with hardware[i], because it's really scary for many. But gluing code together *just because you can*, turns everything into a pile of strapped-together code codepaths that, in consequence, make code not malleable, but as static and immovable as silicon itself. Have you ever wondered if, perhaps, that thing your Product Manager thought would be easy to do, like changing your business model's logic, should actually be easy to change, instead of you complaining about how he doesn't understand software engineering? It's in the name itself: **software should be *soft*, and therefore lend itself to change**. That's what good Software Architecture truly enables, as well as a lot of other side benefits, like component de-coupling, error contention, sub-systems division and reuse, pain-free open-sourcing of code with continous integration, and so on.

You're simply not a professional if you don't pay attention to how the code you write fits into the larger system. And even if you don't want to do it for your legacy within the company you're currently working at, or for your co-workers, **do it for you**. Yes, do it for you: do it for the next time a crazy bug no one has any idea about gets assigned to the one who last touched that file, and you were the unlucky one who last modified that file because you opened it by accident, it got modified, and you didn't bother removing it from your pull request. *Git blame*, you know?

3.2 Racing Tweets' Future

I guess many of you might be wondering *what happens now to the impressive structure you built?* I guess my answer has two distinct parts, one for the product we've been talking about, and another one for the technology underpinning it.

Racing Tweets has a very special place in my heart - it has helped me regain the confidence in myself to actually build products from start to finish, instead of dropping them midway through development. It's tricky because, I'm not sure I can just stop using it; for the purposes I use it for, which is to follow motorsports, it provides a much better experience than my previous solution, which was using a Twitter client and attaching it to a massive Twitter list[ii]. And if I'm using it, chances are that I might start Xcode from time to time to fix something, but it's not my goal. **My goal is to move to my next product, and build something that has an audience larger than just me, willing to pay for it, whilst remaining an indie**

---

i. Case in point: 20 years after researchers published papers on the dangers of CPU speculative execution, we got Meltdown and Spectre.

ii. Called "formulaOne".

developer. Some have managed to make this work at a small scale, but I, well, I just want more.

As **for the Standard Engine, I find it hard to believe that I won't find something for it to do in my upcoming projects**. After all, something the Standard Engine does pretty well is to interface data with one of the most used structures in all of iOS: `UITableView`. And as soon as things get even a tiny bit complicated, I'm sure the ease of use of the Presenter API will allow me to gain significant speed and leverage using everything the Standard Engine has to offer. It won't go the way **I am CEO** has thus far. I'm even thinking of improving its internal structure just for the sake of it, inspired by the great insights I eventually had into software development through working on it. Plus, like I said: I'm willing to license the Engine if you're interested :)

3.3 By the way

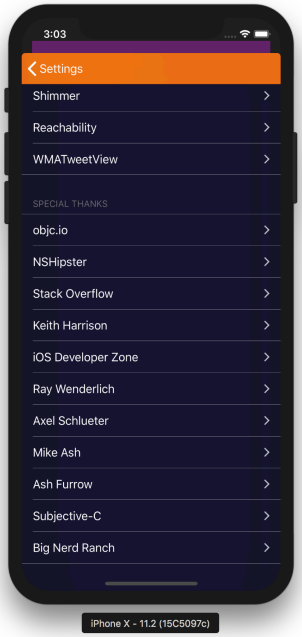


Figure 34: Racing Tweets' *Standing On The Shoulders Of* screen.

There's one last thing I'd like to say, and that is, **thank you**. Thank you for everyone belonging to this immense community, for helping me out in arriving at this destination. Because really, without being able to stand atop the works of many others who've come before us, plus, those who are more knowledgeable than us and can save us significant amounts of time (and money), we wouldn't be able to craft any kind of products at the speed we currently do. From the bottom of my heart, thank you for helping out; whether it was through a tweet, a blog post, a Medium post, a Stack Overflow comment or even a real-life conversation that has sparked an idea within me. Thank you.

4.Annex I: Racing Tweets' Story

4.1 WPF Origins

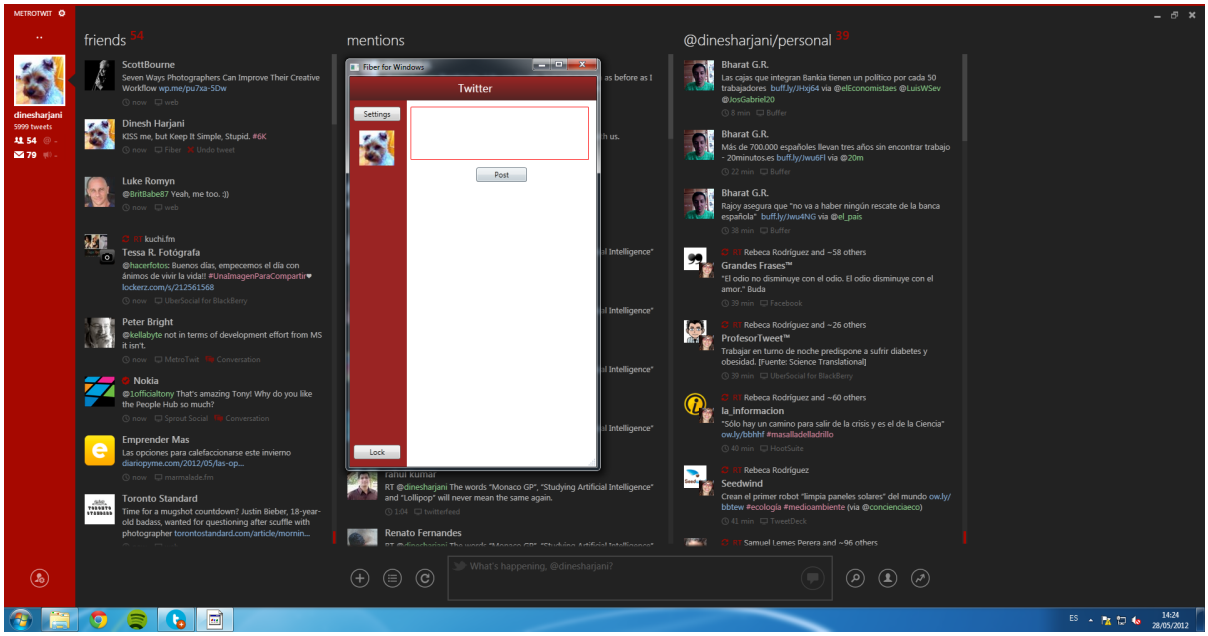


Figure 35: Windows for Fiber prototype, with MetroTwit on the background to verify a tweet was posted.

The journey begins a few months after **I am CEO** stopped being a thing, back in 2012, when I felt very close to a group of friends. To keep in touch, I'd have Facebook, Twitter, Instagram and maybe even LinkedIn, open every day using a combination of both browser tabs and apps, to check for any new posts from any of them. This was easy to do, because back then social networks prioritized content from those you knew and cared about, and with this group of people every post and comment meant something to me. The alternative being having conversations online with the big void that represents the Internet. As a true believer in native apps, it started dawning on me that no one had done an app with that focus: filtering out all the noise, and showing you only the content of the people you really cared about.

This meant that, for example, whenever this fictional app notified you that there were 1 or 2 pending posts to read, reading them when taking a break from work actually meant something to you, instead of making you constantly scroll through lists of endless content, leaving you unsatisfied after spending all of that time scrolling[i]. And if there wasn't any new notifications, you would know there was nothing important, and therefore you were free to invest your valuable break time in something else. But when you did have notifications, you'd feel connected with the people you cared about, instead of you having to do the work of a computer: being

i. To this day, this still happens.

the information filter.

It'd be a challenge to write, I knew. But I loved challenges! And besides, there was a successful app that was able to kind-of-do the same thing, but only with Twitter: Hootsuite[39]. Hootsuite rose to power by allowing you to combine multiple Twitter streams into one single feed, but it never - to my knowledge, at least - attempted to combine content from other sources[ii]. I knew that I was capable of doing it. After all, the APIs were there: to improve my own Facebook experience, I'd created a *favorite list* with all the people I cared about, and all I ever did when opening the dark blue button in my iPhone was check for any new posts in that list. Twitter had explicit lists, so that was a done deal. LinkedIn, Pinterest or Instagram didn't make it easy to follow specific people with list-like structures, but since I believed Twitter and Facebook were the main focus, I could either tackle that locally, by filtering out posts within the app, or I could figure it out later with a more complex solution, such as with my own feed server.

I knew doing this would require a significant time investment, but I was busy with both a full University course load and a 40-hour day-job as an Android Developer. My energy and excitement for the project were at its peak between April and May, when a big personal setback meant the only thing I had to cling onto was this idea. So I began to write it… in Windows. Why Windows? Because all of my potential testers were Windows users, and despite me being a converted Apple-person through and through, I still recognize good software when I see it, and the MetroTwit client for Windows[40] made me believe it was possible to make cool software for Redmond's OS. I got to the point where, as you can see on MetroTwit's leftmost column, I was able to tweet, *from **Fiber***.

Fiber was the name I came up with, but there were many others. Most notably, I can remember Socialité, or simply *Socialite*[iii], but it was already taken by another social media app more focused on dating. For me, the word **Fiber** meant strength, but also, it meant union: a fiber is composed of multiple individual strings which, separately, are weak and easy to break. But, if you intertwine many strings together, eventually you'll end up with something a lot stronger, capable of withholding a lot more strain. That was what the name meant to me, but it wasn't how I was going to sell it to the press; instead, **Fiber** is composed of multiple strings, because each

ii. Perhaps Hootsuite never had the capability of coalescing multiple streams, and simply used Twitter's API to do it for them? I wonder.

iii. I squarely blame this name on a teen drama taking place on the Upper East Side.

string[iv] is a social network whose feed we're pulling in and combining with others, to have a **Fiber**.

4.2 What Happened?

What you saw in the previous section is as far as the Windows iteration of **Fiber** made it. I don't remember all of the details of what pushed me back, but on the non-technical side, lack of motivation was one of them. I do remember thinking that the project was really daunting, in part because WPF and C# were incredibly complex compared to what I knew about Android and iOS development: it took houndreds of lines of code to add a UI panel and make it slide to one side, because you had to write the animation code yourself. I wanted to pull my hair out back in 2012 - no wonder all Windows apps were (and are) so damn horrible. Even worse was the code to show a list of tweets on the screen; I don't remember the specifics, but I remember not feeling like it would be any worth it, even if my low-levels of motivation fueled me that far.

And last, but not least, I had the feeling that the hardest part was going to be building the Engine. Because if I got through all the other stuff, and I was able to animate stuff on the Windows Presentation Form[v], and also set up a list or table of items of sort to go through, I still, after all that, had to figure out the most complex part: being able to pull data from all of the aforementioned social networks, including the task of dealing with their OAuth 2.0 interfaces, and then figure out a way to mix them, and support loading the items in small batches. And because this was me, **I also wanted all of this to work in an efficient way**: I wanted to do multi-threading in Windows to use all the cores a modern PC had to offer. And to this day, I still think Microsoft and Google need to make this a lot easier for developers than just offering Thread Pool APIs[41][42].

So that killed it. And apparently, I tossed the source code somewhere, because I can't find it even if it's just to reminisce about it.

---

iv. A "fiber" is comprised of multiple strings, just like the Standard Engine's Track, is comprised of Tickets from multiple Suppliers 😄

v. In both WinForms (classic Win95 apps) and WPF, a "Form" is a "window", just like in iOS a UIViewController is a "screen".

4.3 The Revival

Figure 36: Standard Engine prototype pulling Formula One tweets, circa August 2014.

The project's idea never truly left me. After giving up on WPF in the Summer of 2012, life called me to Silicon Valley, and although I picked up the phone, I didn't do a good enough job telling them how much they needed me at the time. But I started believing that I could do so with other companies, so I made up my mind on quitting my employer once I resolved some issues. Once I did that, I was the one who began calling for jobs away from home in early 2013, and in the meantime, bored of Android - sorry! - I decided to pick up iOS development again, through Ray Wenderlich's fantastic set of books and articles[43].

While I re-learnt iOS development, I was also sending CVs for Android-based jobs, since that was where I had the most amount of experience. But after going through enough tutorials and not feeling motivated to do more, I wondered what could I do to improve my skills, and paired with blogging, I came up with the idea of retaking the **Fiber** project, but this time doing it the other way around: instead of starting at the interface level, and then finding out what I needed to do, I'd work on the core, or engine of the project first, and then build an eventual product around it. **The engine was eventually named the Standard Engine, in honor of the Higgs Boson discovery a year earlier, but first confirmed in 2013**[44]. I feel too embarrased now

to tell you where the Zealot[vi] Engine's name came from, so I'll just try to hide it as a footnote.

Come Summer of 2013, I got a job in Germany, and the Standard Engine was in early stages, working under some very shaky tests; I wanted to make sure that, the day I plugged in Facebook and Twitter content, I wouldn't have to doubt the Engine's results. That eventually wouldn't happen until almost a full year later, but when it did, it lit my heart on fire. In the meantime, I'd moved to Germany and resumed my job as an Android Developer, taking every chance I could get to snatch a few commits into the inhouse iOS App that was being developed over there.

I was scheduled to travel to Italy in September 2014. Just a few weeks before, I started seeing Facebook and Twitter content being actively processed by the Standard Engine. It dawned on me then, with all the complexity it took to switch on and off the couple of social networks that I had, and the problems I was having to style them properly, specially with my concept of custom backgrounds, that maybe I could walk down an alternative solution: why not release a simpler app, powered by the Standard Engine, and once this simple app was out of the way, and we've come to find all of the bugs and quirks of the engine's design, then release **Fiber**?

Enter **Racing Tweets**.

4.4 But, What Was Racing Tweets all About?

vi. Between 2011 and 2013, Starcraft 2, a strategy-game by Blizzard Entertainment, was known as "Silicon Valley's Chess", which included leagues and championships between all the major Silicon Valley companies. The Zealot is an infantry unit from one of the game's three races.

Figure 37: Promotional artwork for **Racing Tweets** 1.0 release.

Fiber began with the idea of *open one app, and see what the people you care about are doing across all social media*. **Racing Tweets** never had a business idea, it was just a package around the Standard Engine that I could sell. Its seed also dates back to 2013, when I discovered how Twitter allowed for much closer following of my favorite sport, Formula One. I was already a pretty good F-1 geek, but Twitter opened me to lots of insights: communications (tweets) straight from the teams, immediate links to the best analysis and reporting articles, insight from journalists and lots of theories and gossip that, most of the time, would have a kernel of truth in them, but that were not reported by bigger news outlets like the ones broadcasting the Grand Prix races. Not to mention live tweets from the track on racing weekends, as well as spy shots on all the car development happening throughout the seasons. This offered a lot of value to me personally, and since millions of people watch Formula One, both inside and outside Spain, I thought, surely this must also be valuable to all of them. What could go wrong?

In October 2014 I commited myself to making **Racing Tweets** a product in the App Store, and proceeded to work feverishly on it until the Xmas holidays. During the festivities, I kept the project mostly under wraps, because I had this stupid feeling in me that if I said it out loud, then some of the magic would be lost, and that my drive to keep working on it would somehow vanish in a thin puff of smoke. Come early

2015, I rejoined work in Germany and was beginning to suffer from a poor working environment, which in turn pushed me to focus more on me, on **Racing Tweets**. The Summer of 2015 was very tense, because many things needed to happen, very quickly, for me to release on the same date as iOS 9, when all major apps would launch in the App Store. I didn't make it in time for iOS 9, and instead during the month of September I was caught in the web of trying to push people to beta-test it for free instead. I tried, I really tried to get as many testers as possible. And I only recently remembered how much I tried to contact people & press to try it out, when I started going through all of my past Twitter DM conversations in order to do the same thing for v2.0. Most of my attempts to gather interest, both then and now, ended up with people turning me away, including a majority that wouldn't reply back. Some were extremely funny, and said that "they didn't believe in any product made for Apple platforms", or even better, that they "didn't believe in it, but wanted to be told when it was released". I don't remember how I did it, but eventually I found a group of 10-ish people to test it, and then I hit another rock, or rather, a fruit.

Version 1.0 was cleared by App Store Review for release, to the point where all I had to do, was press a button in iTunes Connect. Then, I don't remember how, a colleague of mine[vii] who was an iOS Developer and tested my app, found a bug. Try as I might I can't gather the details about it, just the feeling I had that the bug we'd found affected a common scenario, and would cause crashes for a lot people. Of course, I was counting on the app to be a moderate success, and since I wanted to charge 3.99€ for every sale, I thought the appropriate thing to do was to delay the launch, fix it, and upload version 1.0.1 for App Review. My colleague told me to release the app anyway, but I wanted to do the right thing. So I delayed the release, fixed the bug, and when I uploaded v1.0.1, it was rejected.

You see, the original name for **Racing Tweets**, was not **Racing Tweets**. It was *F-1 Tweets* in English, and *Tweets de Fórmula 1* in Spanish, abbreviated to *TdF1* on your homescreen. Version 1.0 was approved with these names, but 1.0.1 was inspected by a different reviewer at Apple, and was rejected. I remember thinking months before, when I first came up with the name, that *this shouldn't be a problem*. Because sure enough, a quick App Store search detailed many apps with "F1" or "F-1" in their names, having absolutely no official license from the sport. The reviewer didn't say as much, but it seemed like the name was the only issue holding us back. I had a couple of options: change the name, or be sneaky, upload it again,

---

vii. This colleague was actually my iOS mentor throughout the last year of Racing Tweets v1.0's development. I'll never forget the moment he took me through the process of "filing a radar", which is Apple-speak for reporting bugs back to the mothership.

and pray that a different reviewer approved it. I chose the first, since the second didn't seem a good way of doing business; after all, what prevented them of blocking a very important release, or another bug fix, down the line? It was left as **Racing Tweets** in English, and "Tweets de Vuelta 6"[viii] in Spanish.

4.5 Phoenix Rising

Figure 38: A close-up of Racing Tweets 1.0. Notice the muted (purple) color, the blurry background partially showing a user-defined wallpaper, how the text doesn't flow around the user profile image, and how the tweet's image doesn't fill the screen edge-to-edge.

After one more App Review rejection, this time because "images can't show trademarked content"[ix], the app went live on October 29th, 2015 at 18:55 GMT. It was not a success. No reviews on websites, besides one that I paid for[45], and barely any purchases. I pushed hard with discounts and targeted advertising

viii. I was frustrated by the fact that my best name was rejected, after it'd been approved. I tried to come up with something with meaning towards F-1, and "Vuelta 6", which meant "Lap 6" in Spanish, had the double meaning of "V6" when shortened, which is the engine architecture of current F-1 cars.

ix. This meant that none of my promotional artwork was allowed to reference Formula One directly.

through Facebook, to no avail - **no one purchased it**. Over Xmas 2015, I set it free on the App Store and managed some high-profile retweets with people familiar with the sport in Spain, garnering 83 downloads. Sadly, most people either didn't open it, or opened it once, never to do so again.

It hurt a lot - to this day, it still does, making it incredibly hard to write down these words. But back then, I was full of energy - I'd made it! The Standard Engine was now out in the open! It didn't matter that no one was buying it, I still had a lot of potential work to do! So many features to implement! Maybe then people would care? I only lasted two releases, a bug-fix and a minor release (1.0.3 and 1.1), before I completely gave up in March 2016, which coincided with my return home from Germany, and me not wanting much to do with tech anymore.

But, things turn around. Nine months later, the spark was beginning to burn within me again - *maybe I did it wrong?* It didn't have enough features. It had nothing to go for it - you couldn't choose which sports to follow, the background idea was good, but ultimately hideous and extremely complicated to do with adaptable theme colors. The main timeline showed clear signs of being completely rushed; the margins were all wrong and not standardized across the app. The colors were absolutely horrible, muted and lacking coherence and personality. And lastly, nothing made the app feel special, other than the background feature; **nothing made a petrolhead feel like this was their app**.

So I set out to change that with 2.0: gorgeous colors, text flowing around images in the main timeline, *Team Radio* retweet signs, spinning tyres with official Pirelli-colors as loading signs, support for multiple images within a tweet, motorsport-inspired images in every dialog, multiple language support, support for multiple motorsports, a complex filter to ensure only racing-related tweets show up, a tweet editor with an ERS battery sign as a tweet length counter, semaphores to show notifications… When I hit the point where I had to stir interest in it for a beta-test, I basically got the same result, and I didn't remember how hard I'd tried the first time. In my mind, I just did it wrong the first time and thought, *now is when I'm going to make it truly special*. But I was kind of wrong.

It's not that I don't believe it is worth releasing - I think it is, but with the right expectations. Right now, I can't find a way to make people interested in it, even though millions of people hit both Twitter and motorsport sites to check daily about Formula One, Rallying, the World Endurance Championsip and more. This means **Racing Tweets** isn't a viable business as the main income source for an indie developer like me, but it could find a home if made ad-based with some people. We'll see.

5.Annex II: More On Racing Tweets' Tech

5.1 Shared State

Figure 39: Another promotional image for Racing Tweets 1.0.

Once we'd settled on a navigational structure, a different issue we need to consider was how to share state accross all the `UIViewController`(s) that need it. Like the Standard Engine, the main screen's ViewController also needs to share a common object with the Settings VC, so that from Settings we might alter the behavior of the full application. However, the concept of shared state encompasses a lot more quirks that what we haven't covered so far.

Take, for instance, a marquee feature that heavily powered me through the development of the first release of **Racing Tweets**. It was to be a feature intended for **Fiber**, making it stand out accross all other social media apps, but given how without it, 1.0 would've been just a twitter list aggregator, it became something that the app couldn't ship without, and that was custom backgrounds. At the point where **Racing Tweets** became an actual project back in 2014, *Dark Mode* themes were not yet commonplace in most social media apps, so in a way, the UI design of **Fiber**/**Racing Tweets** was two steps forward in this regard, wanting to provide even more flexibility to the user, while drawing most of the text in white over a darkened background, like the dark themes of today. But this effect was not meant to be just for the main timeline: it had to show up all over the app, providing users with the

sense that the app was truly theirs. As you know, it was also meant to showcase Tweetshot(s), when we came around implementing them.

So how did we solve this? We built our own custom `UIViewController`, incorrectly named DHSENavigationTableViewController[i], to handle all features related to the standardization of multiple screens sharing the same background wallpaper, enabling the user to change said wallpaper at any moment. To explain how this works, we'll illustrate the process in a series of steps:

1. Whenever a `DHSENavigationTableViewController` is instantiated, it embeds itself within a standard `UINavigationController`, so the API's caller does not need to perform this extra step[ii].

2. `DHSENavigationTableViewController` adds a `UIImageView` instance, a `UIKit` object for displaying images, as a background image for the `UINavigationController` in which it has embed itself.

3. `DHSENavigationTableViewController` removes the background from itself, as a `UIViewController` subclass of its own[iii], allowing the user to see the image set in the parent `UINavigationController`.

4. From now on, all `UIViewController`(s) pushed from the initial `DHSENavigationTableViewController` instance do not need to set again a background image; they only need to remove their default backgrounds.

5. `DHSENavigationTableViewController` can update its background to the current one with an API call.

Even though a `DHSENavigationTableViewController` and its children classes can update their background, **they cannot change it**. They can only read what their background should be, and change it if the one they currently have isn't a match. Instead, backgrounds are a setting, so no `DHSENavigationTableViewController` subclass has the functions, nor the code, to change said setting. This is where the `DHSEUserSettings` class enters the picture.

---

i. I should rename it the DHFiberNavigationTableViewController.

ii. UINavigationController(s) are "true controllers", in the sense that barely add any UI, and instead provide a layer of control for the behavior of the underlying UIViewControllers. Therefore, we always need a starting UIViewController.

iii. A UINavigationController exists as part of the view hierarchy, since it's the owner of the UINavigationBar. UIViewControllers control their own background, and by making these use the NavController's background instead, they all share the same one.

5.1.1 Settings Management

Though in its name you can read the letters "SE" for Standard Engine, there are no dependencies whatsoever between a Standard Engine instance and a `DHSEUserSettings` instance. Like the aforementioned `DHSENavigationTableViewController` class, `DHSEUserSettings` should replace the letters "SE" with "Fiber" in its name. In any case, `DHSEUserSettings` is a Singleton class holding, as its name implies, a variety of settings. These include the path for the user's background, and Engine-related settings like whether the (Settings) Debug section is enabled, if we want to show exact timestamps for all the tweets, and more. In truth, `DHSEUserSettings` forms the backbone for a lot of user data, since it internally holds a dictionary class, which is then flattened into a Property List (XML) file. Once saved to disk, this file is then set as fully protected by the Operating System in case the FBI tries to steal our user's settings[46]. Whenever a setting is changed, `DHSEUserSettings` writes itself to disk again, avoiding us having to worry about whether any changes have been saved or not.

Swift, like Objective-C before it, has a very interesting property, called **class extensions**. A class extension is simply an import of a file that adds behavior to existing classes. In very simple terms, **this allows anyone to extend any class, without subclassing, including Apple framework classes**. So you can basically add a new method to the String class that returns the same text but made bold, in the form of an `NSMutableAttributedString`, which is how you can add font attributes to text in Cocoa[iv]. But there are limitations, or otherwise subclassing would see its value significantly reduced: class extensions cannot override functions, nor can they add stored variables. Both of which can be overcome through mastery of the language, of course.

In fact, we wrote `DHSEUserSettings` with class extensions in mind: `DHSEUserSettings` offers API functions for loading and storing both strings and numbers, which enables us to write functions in our user settings' extension to write and read "new variables". This is because most properties can be reduced to either strings or numbers, and complex objects can be serialized and deserialized from strings as necessary. And, through the use of syntactic sugar, we can even make these look like proper variables, using Swift's computed properties, which allow us

iv. Cocoa, without the "Touch" prefix, is the Cocoa Touch equivalent for the MacOS platform. UIKit derives from its MacOS counterpart, AppKit. And AppKit itself, derives from NeXTSTEP.

to define the getter and setter of a property. And since we can use our API to read said properties, it all becomes quite clean from the perspective of the API's user:

Language: Swift 4

```
extension DHSEUserSettings {

   @objc public var receiptValidated: Bool {
         get {
             return
DHSEUserSettings.shared().getBooleanSetting(WithKey:
Constants.receiptValidatedKey, andDefaultValue: false)
         }
         set(newValue) {

DHSEUserSettings.shared().setSettingWithKey(Constants.receiptV
alidatedKey, toValue: NSNumber(booleanLiteral: newValue))
         }
     }
}
```

Figure 40: Code example for extending DHSEUserSettings class, adding a new property representing a **Racing Tweets**-specific setting.

This structure allows us to turn `DHSEUserSettings` into the foundation for most user-related data storage in the app, excluding any kind of images[v]. To that effect, **Racing Tweets** defines its own `DHSEUserSettings` extension, called `DHTdF1UserSettings`, which adds an API to read and write many user-facing settings specific to the app, such as which motorsport categories are enabled, which language the user prefers their content to be in, whether the Racing Filter feature should be enabled or not, etc. Even **Racing Tweets**' own server's web address[vi], is stored here, since it can be changed for debugging reasons. Besides `DHTdF1UserSettings`, another class, `DHTdF1ServerCommunicator`, leverages the `DHSEUserSettings` API under wraps to store other kinds of logic. Even though they're tied at the storage level, using different singletons to access different information allows for a clean separation of intents. So, if in the future we switch `DHTdF1ServerCommunicator` to use a different storage system for its contents, the rest of our code won't change, just this class. This strategy would've served us well in the past, when we decided to place the background-resizing logic into

v. Not by storing their binary data in XML, but by storing their relative path in the filesystem. This was used to store the current background image set by the user.

vi. It has never made it past being a localhost. Still, we had some fun blurring out its address from the UI.

`DHSEUserSettings`, since it alreaded pointed to the resized image on disk, and it allowed for a clean interface. But, since backgrounds are no longer a feature of **Racing Tweets** v2.0, we ripped out most of that code.

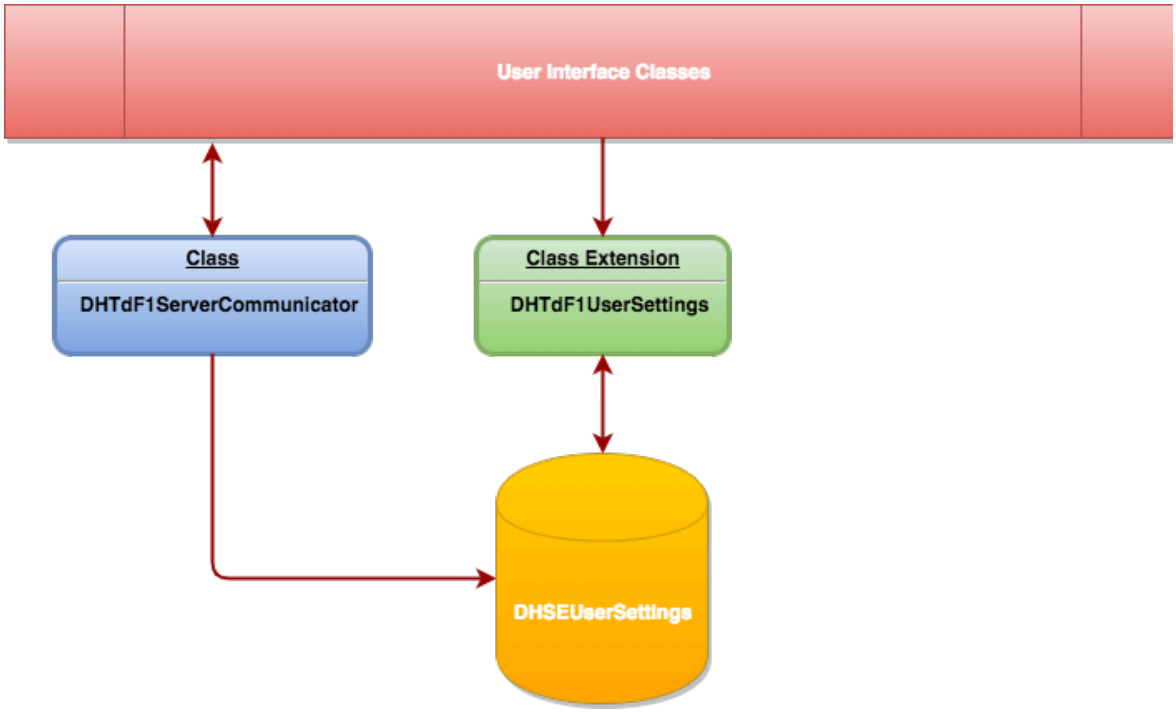


Figure 41: DHTdF1ServerCommunicator and DHSEUserSettings relationship diagram.

It's very important to note though, that when a user updates a setting, this does not trigger any kind of User Interface update, as the graphic above might imply. The reason for this is that they're mostly not necessary: some settings, like changing which motorsports tweets should be fetched for the user, trigger a full reload of all the data. So from a logical point of view, all we have to do from the Settings screen is trigger a data reload, and when the user returns to the timeline View Controller, that screen's code will handle the rest. This behavior also applies for a change in the content's language, and a change to the status of our Racing Filter. Another type of settings don't trigger dramatic changes when changed, but are instead read constantly wherever they might take effect, which is technically slower, but it saves developer time. A quick example would be the *Show Timestamps* setting, which is checked against every time a tweet is shown to the user in the form of a `UITableViewCell`, and depending on the setting, a tweet's timestamp is either shown exactly (eg. 09:41:00) or relatively (9h ago), where the latter represents the standard, user-facing behavior. This means that if the user changes the timestamps setting from the Setting screen, upon returning to see their list of tweets, they'd be automatically be refreshed, and the setting would be read and the timestamp value

updated accordingly. No other changes required.

5.1.2 Simplicity Caveats

In Engineering, there is no perfect solution. There are highly desirable traits, where most of them counter-balance each other, meaning good engineering is an act of balance. Our pursuit of pure simplicity when updating a `DHSEUserSettings`-backed setting, and triggering an immediate XML flattening operation and subsequent write to disk mean we'll never have to worry about some of our settings not being saved, nor about a potential difference between what's on memory and what's on disk. This works well most of the time, since the user rarely changes any setting, and becomes less critical as the user spends more and more time with the app, at which point they've already tuned it to their liking. But there's a very important case where this strategy plays completely against us, and the user.

When the app starts right after being installed, it checks for the settings file written on disk, and if it can't find it, it triggers a new save with some defaults. This is useful when making big changes to our settings structure between app updates, since we can just trigger a default write to disk that adds the new missing settings after the update. But, when the app starts and begins checking its behavior against all of the default settings, it also triggers more default writes for settings that haven't been set with a default, but that are also missing, which greatly slows down the app startup process, because contrary to popular belief, writing to flash storage isn't always fast[47]. Moreover, a settings change can happen from any thread, so different threads can potentially schedule a write operation that removes a setting change for another. This is a great risk that burdens the sole developer in the project with remembering to always update settings from the User Interface thread.

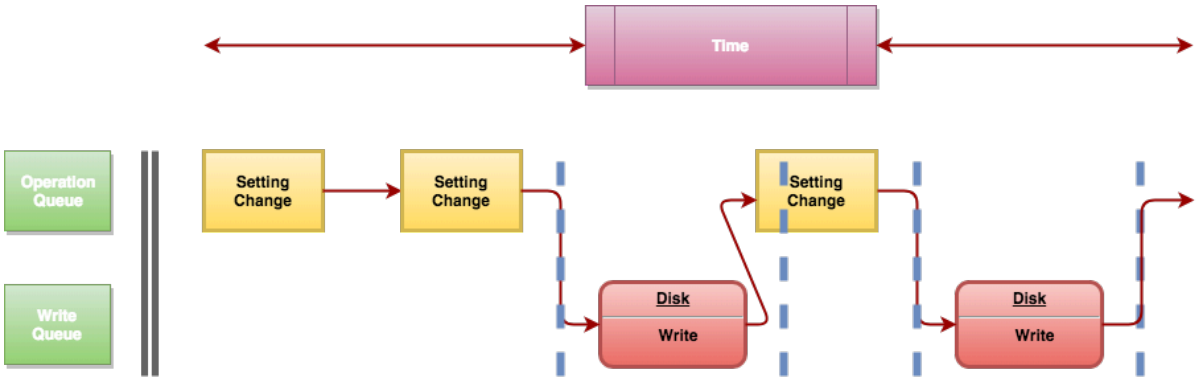


Figure 42: DHTdF1ServerCommunicator Write Coalescing algorithm.

The solution we came up with is to perform **Write Coalescing**. This means that all setting changes enqueue a write operation at some point in the future, on a different thread. This write operation is enqueued further down if a new setting change happens, so as to encourage multiple setting changes to only require a single write. Once we hit a certain threshold, a write happens, independently of whether new setting change happens while we're writing to disk or not, and if it does happen, the new changes are enqueued for a second write which, may or may not be delayed if more changes happen again, like the first time.

This strategy was not implemented in `DHSEUserSettings`. It was, however, implemented in the new `DHTdF1ServerCommunicator` class we spoke about earlier, as a means of protecting the app against multiple calls for server data from different threads. In particular, one of the app's features was a one-line commentary on the most important Twitter accounts, powered by the our Racing Tweets Server[vii]. This information is required from multiple threads, so all of them will request an update of said information whenever the user requests new tweets. And to prevent multiple requests, we set up the above structure to coalesce server requests and disk writes for the same information, with the hope of retro-fitting the new code back into `DHSEUserSettings`, for the benefit of both the user and the application's codebase.

5.2 Expanded Developer's Perspective

5.2.1 DHZESupplier API

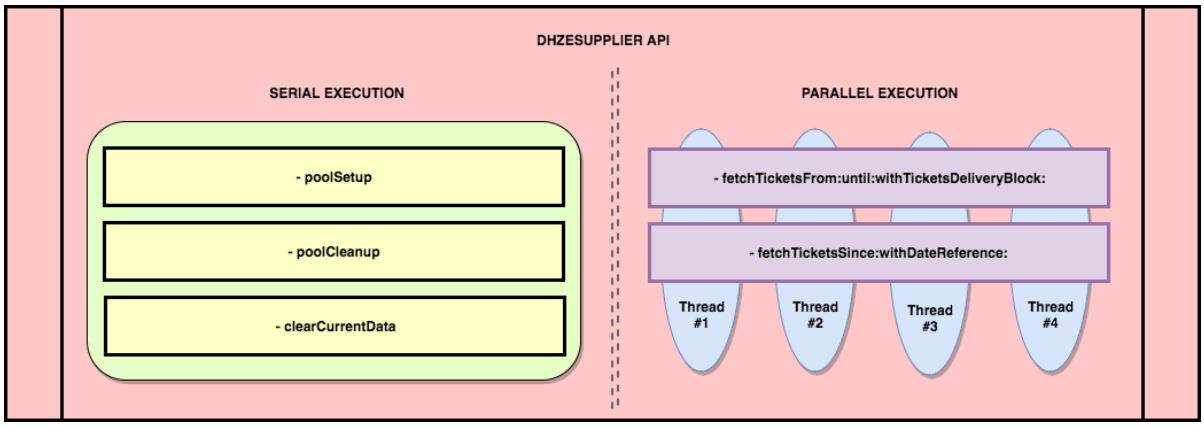


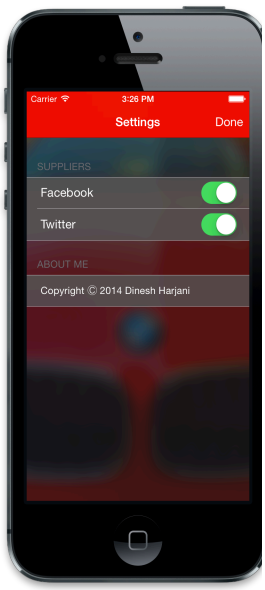
Figure 43: Expanded DHZESupplier API.

vii. This allowed us to update one-line comments for specific Twitter accounts, which are visible to the user, whenever we wanted, without having to release an app update to change them.

Because we realise, and admit, that writing a Supplier is the most complex part of making use of the Standard Engine API, we will add a little more detail, so that you may better understand how the Supplier callbacks can be used to fully implement an application like **Racing Tweets**. To begin with, it's very important to understand that **not all callbacks within the Supplier API are called the same way**. The three helper methods, `-poolSetup;`, `-poolCleanup;` and `-clearCurrentData;`, are called, as implemented in the `DHSEModularControlUnit` class the Engine provides, **serially**. This means **a Supplier can block the caller's thread** and make sure it has everything it needs before the next Supplier receives the same call, or, a call to return new Tickets in the Supplier's `-fetchTicketsSince:withDateReference:` function is made. Specifically, the `DHTdF1TwitterSupplier` class uses `-poolSetup;` to perform OAuth 2.0 Twitter authentication, including a pop-up dialog asking the user permission to verify the app, either through the official Twitter client, or via an embedded web browser. Once the OAuth 2.0 token is received, the Supplier stores it, and unblocks the caller thread, allowing the remaining Suppliers to go through the same process, and in the case of **Racing Tweets**, reuse the same token. If, on the other hand, an OAuth 2.0 token isn't stored, the Supplier can simply elect to always return an empty set of Tickets in order to **fail gracefully**. But a much better engineering solution would be to make the Control Unit not return the unauthorized Supplier to the Standard Engine when asked.

There are two more things we'd like to note about what happens in `-poolSetup;`. One is that if the authentication process takes far too long, the Standard Engine will halt the update operation and the user won't see any new content, because we will hit our user protection time limit. A possible solution to this would be to place user authentication outside the Standard Engine Track update loop, if there are multiple authentications to be performed, or if we just want a cleaner solution. The second thing is that, for the purposes of **Racing Tweets**, all Suppliers are essentially a subclass of `DHTdF1TwitterSupplier` with different Twitter lists from which they each pull content. This means our scheme works pretty well, given that we don't care which Twitter Supplier is the first one to be called, and that once the first one gets authorized or rejected, the remaining ones will follow suit, without having to worry about any kind of multi-threading induced woes.

The `-poolCleanup;` call happens after the Engine has finished updating the Track, and is primarily intended for cleaning up any cached information that won't be necessary until the next Track update cycle. In **Racing Tweets**, `-poolCleanup;` is used to clear the cache of filtered Tickets within the current Pool, because we've already stored them in an array. And to finish with the Supplier API, `-clearCurrentData;` is called when the user leaves the app, to clear out any data that might be needed when the user is actively looking at the app, but can be discarded once we're backgrounded or closed. Specifically, this API was added with the intention of clearing out the image cache for the presentation layer, allowing Suppliers to handle this individually, if needed so. In practice, **Racing Tweets** handles image caching in a different way, but the API remains to allow for future use cases.



Lastly, we'd like to mention an implementation detail that does not affect any Supplier, but that might make sense to add in this paper. Supplier callbacks (API function calls) are performed by two separate kinds of objects: Control Units and Standard Engine instances. Specifically, **the serial execution callbacks are made by the Control Unit, whereas all *fetch ticket* calls are performed by the Standard Engine**, which aims to extract maximum performance by parallelizing any calls for new Tickets, which are assumed to be expensive by the API (and rightly so). This is a minor technicality, since in reality, the Engine does not have direct access to the Suppliers, but instead, asks the Control Unit for a list of Suppliers it may query to obtain new Tickets. In our opinion, this was a better solution than including the Supplier-query logic inside the Control Unit, which is meant to be an element for the API user to control how the Engine works.

5.2.2 Ticket Presentation

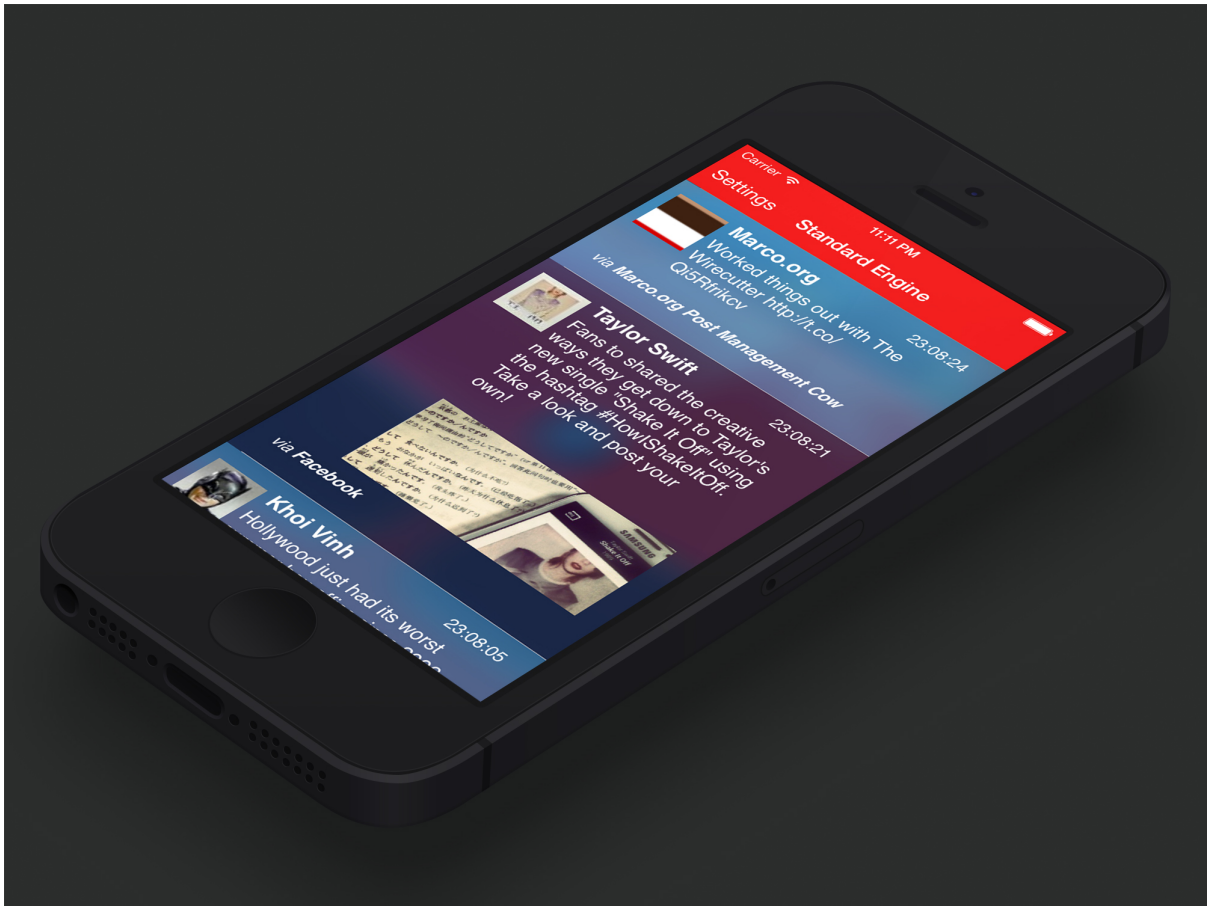


Figure 44: Standard Engine prototype, showing both Twitter (top and bottom Tickets) and Facebook (middle Ticket) content, circa 2014.

Here's the deal: we were not expecting to talk about how presentation of Tickets is handled within the Standard Engine. Then again, what's the point of building such a complex data engine if not for aiding us in showing information to the user? So we're going to go through it briefly, without giving it the copious amounts of details we've given to other areas, like the Supplier API.

To begin with, we'll start with the `DHZETicketPresenter` API, which defines the following protocol:

Language: Objective-C

```
//
// Created by Dinesh Harjani on 2/7/15.
// Copyright (c) 2015 Dinesh Harjani. All rights reserved.
//

@protocol DHZETicketPresenter <NSObject>

- (UITableViewCell *)cellForTicket:(id<DHZETicket>)ticket
forTableView:(UITableView*)tableView atIndexPath:
```

```
(NSIndexPath*)indexPath;

- (NSAttributedString *)baseAttributedTextForText:(NSString
*)text;

- (void)refreshCell:(UITableViewCell *)cell forTicket:
(id<DHZETicket>)ticket;

- (void)willDisplayCell:(UITableViewCell *)cell;

- (void)clearData;

@end
```

Figure 45: DHZETicketPresenter API.

As you can see, the header dates this file back to 2015, a year after the Standard Engine was deemed finished, and months before **Racing Tweets** was released. Most of these methods existed before, but they were bundled within a base Supplier superclass and hidden from my day-to-day view. Then, when our basic presentation design began to heavily differ from what we wanted for the first **Racing Tweets** release, we defined the Presentation API, integrated it within the Supplier API, and wrapped our existing basic behavior inside the `DHSETicketPresenter` class. The picture above actually shows the code that would eventually be packaged into the `DHSETicketPresenter` class in action, with a minor alteration in the Facebook and Twitter Supplier subclasses defining a different background color for their `UITableViewCell`(s). What does this mean for you, if you were to license the Standard Engine? That you can use the `DHSETicketPresenter` directly, as long as you're sticking with `DHSETicket` objects to represent your data, and get a great amount of functionality for free.

To shed some light into the API, we'll begin with the most important callback, `–cellForTicket:forTableView:atIndexPath:`, which is called from the User Interface layer, and mirrors a similar callback from the `UITableViewDataSource` protocol. This callback **does not set a `UITableViewCell` instance for a specific Ticket**, it merely returns one, because `UITableViewCell`(s) are recycled to reduce memory fragmentation. Furthermore, this first callback not only looks to recycle a `UITableViewCell` using the `UITableView`'s API, if needed it should also build a new `UITableViewCell` and add it to the `UITableView`'s pool of recyclable cells. Once we have a valid `UITableViewCell` for our Ticket, we can call `–refreshCell:forTicket:` from the UI layer to actually set up the cell for a specific Ticket. And lastly, if we need it, we can call `–willDisplayCell:` if we want something to happen just before the user sees a `UITableViewCell`, like an animation.

`-clearData;` simply mirrors the `DHZESupplier` API, and isn't being used. But, the last callback we haven't gone through in the Presenter API, is a little bit more complicated to explain. `-baseAttributedTextForText:` has its roots on what we need to do, from the developer's perspective, to alter how a String is shown. Simple elements like changing a text's font, font size and font color can be done for an entire String. But if we want to do it in a much more granular way, like a social media app requires, we need to use a different class called `NSAttributedString`, and its mutable variant, `NSMutableAttributedString`. `NSMutableAttributedString` basically works by assigning text properties to ranges within the given String, so for example, to highlight **@dinesharjani** in a tweet's contents, we need to add a bold attribute to our `NSMutableAttributedString` instance in the range corresponding to my username.

But there's more: **attributes are additive**. Meaning that, to set a base font, font size, font style and font color the entire String, we need to set all of these first, and then modify the parts that change in respect to the base style. Because if we do it the other way around, as I discovered the hard way, **the last attribute you set overrides the previous one**, no matter what it was. Following our previous example, if we add the bold attribute to **@dinesharjani** first and then apply the common style to a `NSMutableAttributedString`, which would set the font's style to regular, our bold attribute for @dinesharjani would be lost. It makes sense once you understand how these attributes work, but it doesn't so much when you're only trying to reuse existing code by *injecting* the new parts and allowing the non-modified parts to flow around your changes. Instead, this API forced us to turn around the callback structure for styling `UITableViewCell`(s), and in order to do so, given that we didn't know how many times the base style would need to be applied, we added that call to the basic API. In practice, that API is not used from the Presenter's subclasses; it's called from other ancilliary classes that have access to the Presenter, like the Ticket class, or in the case of **Racing Tweets**, the `DHSETweetEntity` class, which represents usernames, hyperlinks and hashtags in a tweet, and transforms a given `NSMutableAttributedString` to add its entity's attributes.

6.Bibliography

6.1 References

1. Amadeo, Ron. “The (Updated) History of Android.” Ars Technica, October 31, 2016. https://arstechnica.com/gadgets/2016/10/building-android-a-40000-word-history-of-googles-mobile-os/.

2. “Intel Tick-Tock Model.” Intel. Accessed January 19, 2018. https://www.intel.com/content/www/us/en/silicon-innovations/intel-tick-tock-model-general.html.

3. Wu, Qiang. “Making Facebook’s Software Infrastructure More Energy Efficient with Autoscale | Engineering Blog | Facebook Code,” August 8, 2014. https://code.facebook.com/posts/816473015039157/making-facebook-s-software-infrastructure-more-energy-efficient-with-autoscale/.

4. Rossi, Chuck. “Rapid Release at Massive Scale.” Facebook Code, August 31, 2017. https://code.facebook.com/posts/270314900139291/rapid-release-at-massive-scale/.

5. Zhao, Haiping. “HipHop for PHP: Move Fast,” February 2, 2010. https://www.facebook.com/notes/facebook-engineering/hiphop-for-php-move-fast/280583813919/.

6. Apache. “Apache Thrift - Home.” Accessed January 16, 2018. http://thrift.apache.org/.

7. Figuière, Michaël. “What Is Facebook’s Architecture? - Quora,” December 26, 2014. https://www.quora.com/What-is-Facebooks-architecture-6.

8. Vajgel, Peter. “Needle in a Haystack: Efficient Storage of Billions of Photos.” Facebook Code, April 30, 2009. https://code.facebook.com/posts/685565858139515/needle-in-a-haystack-efficient-storage-of-billions-of-photos/.

9. McCormick, Rich. “Facebook’s Prototype Cold Storage System Uses 10,000 Blu-Ray Discs to Hold a Petabyte of Data.” The Verge, January 29, 2014. https://www.theverge.com/2014/1/29/5359628/facebook-blu-ray-storage-system-uses-10000-discs-for-petabyte-data.

10. Luckerson, Victor. “How Twitter Slayed the Fail Whale.” Time, November 6, 2013. http://business.time.com/2013/11/06/how-twitter-slayed-the-fail-whale/.

11. “Google I/O.” Wikipedia, October 21, 2017. https://en.wikipedia.org/w/index.php?title=Google_I/O&oldid=806320060.

12. Asana. “Use Asana to Track Your Team’s Work & Manage Projects · Asana.” Asana. Accessed January 16, 2018. https://asana.com/.

13. Atlassian. “Jira | Issue & Project Tracking Software.” Atlassian. Accessed January 16, 2018. https://www.atlassian.com/software/jira.

14. Ari, Grant, and Zhang Kang. “Airlock - Facebook’s Mobile A/B Testing Framework.” Facebook Code, June 9, 2014. https://code.facebook.com/posts/520580318041111/airlock-facebook-s-mobile-a-b-testing-framework/.

15. “[OT] Hi All, Could You Help Me Beta-Test My F1-Related IOS App? • r/Formula1.” reddit. Accessed January 19, 2018. https://www.reddit.com/r/formula1/comments/7fwa2h/ot_hi_all_could_you_help_me_betatest_my_f1related/.

16. “Obama: Not Every Problem Has Military Solution.” USA TODAY, May 28, 2014. https://www.usatoday.com/story/news/2014/05/28/obama-foreign-policy-west-point-commencement-address/9661593/.

17. “Obama at West Point: ‘Becase We Have the Best Hammer Does Not Mean That Every Probleim Is a Nail’ - YouTube,” May 28, 2014. https://www.youtube.com/watch?v=f7d1BBDHa7s.

18. “Tweetbot: Fix These Things and I Will Love You More.,” April 27, 2012. https://www.adendavies.com/tweetbot-fix-these-things-and-i-will-love-you-more/.

19. “Rate Limiting — Twitter Developers.” Accessed January 16, 2018. https://developer.twitter.com/en/docs/basics/rate-limiting.

20. “3D Touch - IOS - Apple Developer.” Accessed January 20, 2018. https://developer.apple.com/ios/3d-touch/.

21. “Mastodon.” Mastodon hosted on mastodon.social. Accessed January 18, 2018. https://mastodon.social/about.

22. “Micro.Blog.” Accessed January 18, 2018. https://micro.blog/.

23. S Jackson. Apple Steve Jobs The Crazy Ones - NEVER BEFORE AIRED 1997 - (Original Post). Accessed January 16, 2018. https://www.youtube.com/watch?v=8rwsuXHA7RA&feature=youtu.be.

24. Siracusa, John. “Hypercritical: Archive: All Articles.” Accessed January 16, 2018. http://hypercritical.co/archive/all/.

25. “Exponential Backoff.” Wikipedia, November 28, 2017. https://en.wikipedia.org/w/index.php?title=Exponential_backoff&oldid=812544442.

26. Bohn, Dieter. “Twitter Dictates Third-Party App Form and Function in New API, Gives Six Months to Comply.” The Verge, August 16, 2012. https://www.theverge.com/2012/8/16/3248079/twitter-limits-app-developers-control.

27. Viticci, Federico. “Tweetbot for IPhone Review,” April 4, 2011. https://www.macstories.net/news/tweetbot-for-iphone-review/.

28. Harjani, Dinesh. “Week 3 - Gap Technology (Part 1).” Dinesh Harjani’s thoughts. Accessed January 16, 2018. http://dinesharjani.com/post/48561548010/week-3-gap-technology-part-1.

29. “RSS.” Wikipedia, December 15, 2017. https://en.wikipedia.org/w/index.php?title=RSS&oldid=815481176.

30. EverySteveJobsVideo. Steve Jobs Previews OS X Leopard & Mac Pro - WWDC (2006). Accessed January 16, 2018. https://www.youtube.com/watch?v=fnWcmtCJtOc&feature=youtu.be&t=28m43s.

31. “Shia Labeouf Snl GIF - Find & Share on GIPHY.” Accessed January 20, 2018. https://giphy.com/gifs/reactiongifs-ujUdrdpX7Ok5W.

32. “Using Text Kit to Draw and Manage Text.” Accessed January 18, 2018. https://developer.apple.com/library/content/documentation/StringsTextFonts/Conceptual/TextAndWebiPhoneOS/CustomTextProcessing/CustomTextProcessing.html.

33. Harjani, Dinesh. “iOS - How to Properly Resize Textview with ExclusionPaths inside of Table Header View - Stack Overflow,” July 24, 2017. https://stackoverflow.com/questions/41845443/how-to-properly-resize-textview-with-exclusionpaths-inside-of-table-header-view/45280820.

34. Harjani, Dinesh. “iOS - UITextView: Disable Selection, Allow Links - Stack Overflow,” August 3, 2017. https://stackoverflow.com/questions/36198299/uitextview-disable-selection-allow-links/45480781.

35. “PaintCode - Turn Your Drawings into Objective-C or Swift Drawing Code.” Accessed January 16, 2018. https://www.paintcodeapp.com/.

36. Harjani, Dinesh. “Update #17: What’s in Version 1.0.4?” Dinesh Harjani’s thoughts. Accessed January 16, 2018. http://dinesharjani.com/post/134996262175/update-17-whats-in-version-1-0-4.

37. “Tweetshot.” Product Hunt, December 29, 2014. https://www.producthunt.com/posts/tweetshot.

38. “Air France Flight 447.” Wikipedia, January 16, 2018. https://en.wikipedia.org/wiki/Air_France_Flight_447#Final_report.

39. “Social Media Marketing & Management Dashboard - Hootsuite.” Accessed January 20, 2018. https://hootsuite.com/.

40. Warren, Tom. “Windows’ Best Twitter Client Is about to Die.” The Verge, March 5, 2014. https://www.theverge.com/2014/3/5/5473110/metrotwit-for-windows-end-of-support.

41. Muzaffar, Ali. “Using Concurrency to Improve Speed and Performance in Android.” Medium (blog), January 2, 2016. https://medium.com/@ali.muzaffar/using-concurrency-and-speed-and-performance-on-android-d00ab4c5c8e3.

42. dotnet-bot. “Threading Model.” Accessed January 20, 2018. https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/threading-model.

43. “Ray Wenderlich | Tutorials for iPhone / iOS Developers and Gamers.” Accessed January 16, 2018. https://www.raywenderlich.com/.

44. Timmer, John. “Finding the Higgs? Good News. Finding Its Mass? Not so Good.” Ars Technica, February 19, 2013. https://arstechnica.com/science/2013/02/finding-the-higgs-good-news-finding-its-mass-not-so-good/.

45. raulalicante. “TdV6, tweets de F1 y del deporte del motor en tu iPhone.” TodoiPhone.net (blog), November 10, 2015. https://www.todoiphone.net/tdv6-f1-motor-iphone-ipad/.

46. Brodkin, Jon. “FBI Security Expert: Apple Are ‘Jerks’ about Unlocking Encrypted Phones.” Ars Technica, January 11, 2018. https://arstechnica.com/tech-policy/2018/01/fbi-security-expert-apple-are-jerks-about-unlocking-encrypted-phones/.

47. Google Developers. Google I/O 2010 - Writing Zippy Android Apps, 2010. https://www.youtube.com/watch?v=c4znvD-7VDA&feature=youtu.be.