The Missing Manual for Swift Development

Written by Bart Jacobs

Cocoacasts

The Missing Manual for Swift Development

Bart Jacobs

This book is for sale at
http://leanpub.com/the-missing-manual-for-swift-development

This version was published on 2017-09-30

Contents

Welcome

The title of my book, **The Missing Manual for Swift Development**, accurately describes what I have in store for you. It's the guide I wish I had when I started out as a software developer years and years ago.

The Missing Manual for Swift Development is a summary of what I've learned over the years building software for Apple's ecosystem. Many of the lessons in my book I learned from experts in their field and, unfortunately, just as many I learned the hard way. I hope that some of the topics in this book can help you on your way to become remarkable in what you do. That's your goal. Is it not?

Writing a few lines of Swift is surprisingly easy. Once you start to dig deeper, though, you discover that building an application for Apple's ecosystem is more challenging than it seems. **The Missing Manual for Swift Development** outlines the challenges you face along your journey and how to overcome them.

Some of the more obvious topics I cover include dependency management, source control, code reviews, continuous integration, style guides, working in a team, tooling, project organization and documentation, and release strategies.

The topics I found most interesting to write about, however, are more meta, such as when to break the rules, freelancing and subcontracting, staying productive as a developer, shipping projects, leaving your comfort zone, and dealing with challenging problems.

My book doesn't include many code snippets or sample projects. The goal of this book is to provide insights and answers to questions that are often overlooked or ignored.

The Missing Manual for Swift Development is for every type of developer, but it primarily focuses on Swift and Cocoa development. If

you're developing for Apple's ecosystem, then you'll find a lot of useful information, regardless of your experience.

There are very few shortcuts in software development. You pay a price for most of the shortcuts you read about, one way or another. But there are a handful of shortcuts that can speed up your learning and your career. A proper education is one of them.

I hope that my book helps you in some way, big or small. If it does, then let me know. I'd love to hear from you.

Enjoy,

Bart

5 Security

Security is a fundamental aspect of software development and it's important to know about best practices and common patterns that can help strengthen the security of the projects you work on. I want to emphasize that I'm not a security expert. The recommendations I provide in this chapter are based on my experience and what I've learned from fellow developers.

Make It Hard

I once read that, if someone wants to access your data, then they will succeed. How badly they want to access your data determines whether they'll succeed. I don't know whether this is true, but I tend to err on the side of safety.

Why is this important? It changed my perspective on security. It's naive to think that you can outsmart people that are trained to find and extract the information they need. That doesn't mean you need to be complacent or ignore the advice you read. It simply means that your actions and motivation change slightly.

An effective approach to security is to have the mindset to make it hard for the other party to access the data you're trying to protect. In other words, you add several layers of security to protect the data of the user. Let's start with the basics.

Plain Text

If you have some experience developing software, you most likely know that you shouldn't store sensitive information in plain text. Ever. Don't

store the user's username and password in the user defaults database, for example. Use the keychain to protect this type of sensitive information.

The same applies to networking. Apple and Google are actively forcing developers to move away from HTTP and use SSL by default. Apple's **App Transport Security** encourages developers to be aware of the security risks of their applications. Make sure that your application communicates with remote services over a secure connection. This isn't always possible if you aren't in control of the remote service. In such a scenario, it's up to you to decide what the next best option is.

But SSL may not always be sufficient. Your application is still susceptible to, for example, man-in-the-middle attacks[1]. You can remedy this by adopting certificate pinning, adding an extra layer of security.

Obfuscating Information

A common question I receive is how to best hide or obfuscate sensitive information that's bundled with your application. That's a good question. The answer may disappoint you, though. As I mentioned earlier in this chapter, there's always a way for people with bad intentions to get a hold of the information they need. You need to consider the sensitivity of the information you're trying to protect.

The same advice applies, though. Make it as hard as possible. But, at the same time, consider the sensitivity of the information you're protecting. Don't store sensitive information, such as API keys, in your application's **Info.plist**. It's easy to dissect an application you downloaded from the App Store and inspect the contents of the **Info.plist**.

I usually store sensitive information as private constants in a configuration file, which means it's compiled alongside the application. This doesn't make it impossible to extract the sensitive information, but it makes it less trivial.

[1] https://en.wikipedia.org/wiki/Man-in-the-middle_attack

Fetching Sensitive Information

You can go one step further and avoid storing keys or credentials in the application itself. Instead, your application contacts a remote service and asks for credentials every time it needs to communicate with that service. This requires a dedicated infrastructure and a lot more work up front, but it adds a powerful layer of security.

Encryption

Encryption is an effective solution to protect the user's data. Realm, for example, has built-in support for encrypting the data stored in its database. For Core Data, however, this is less trivial. I hope Apple will make this less cumbersome in a future release of the framework.

The data the user stores on their device is automatically encrypted if the device is protected with a password or Touch ID. Only you, the user, can unlock the data stored on your device because you hold the key to decrypt it, not Apple. It's great to see that Apple continues to invest in the privacy and security of its customers. Apple's motivation[2] is a bit more nuanced, though.

Privacy

A lot has been written about privacy and protecting the user's privacy. Unfortunately, many developers don't realize that this also means protecting the user's privacy from companies that offer services they use day in day out. If your application uses analytics or displays ads, then you're exposing the user's personal information to the companies behind these services.

I used to use Fabric for crash reporting and analytics, but I no longer do for personal projects. As a developer, it's my responsibility to protect

[2] https://www.apple.com/privacy/approach-to-privacy/

the user's privacy and they expect that from me. I understand that many developers don't have this luxury, but I still believe that you should, at a minimum, consider the option and be aware of the information you may be exposing to third parties.

If you include a third party SDK in your application and you don't have access to the source, then how do you know what information you're sharing with this third party? You don't. That's important to keep in mind.

Logging and Debugging

Logging information to the console is my favorite technique to debug issues because it's simple and to the point. It's a technique many of us use, but it's also a potential security problem. Many developers forget that print or log statements also log information to the console in production. This can be useful and intentional, but it can also be a security issue.

I hope you're not logging credentials or other sensitive information. Even fragments of the user's data shouldn't be logged in production. If you need to generate logs, then I recommend looking into remote logging in combination with data encryption. Avoid that a third party, any third party, can access the logs you generate.

Educating Your Client

The role of a developer is often reduced to writing code and solving a problem. Not only is this incorrect, I strongly believe any developer, regardless of their experience, should also provide a technical service to the parties they work with. What does that mean? If you're told to implement a solution, then it's your responsibility to inform your client or project manager about any security risks or problems.

I believe it's the task of the developer to educate the client. The client still decides what happens and what needs to be implemented, but they

should at a minimum be aware of the risks involved. I've implemented several solutions I didn't agree with, but I tried to educate the client about alternative solutions that were safer.

At one point I inherited a project in which the user's credentials were stored in the user defaults database. Even though there was no room to refactor this glaring security hole, I informed the client about the problem. For a developer, it can be frustrating not having final say in such arguments, but that's how it is. This is very different if you build a product business in which you make the calls.

1 Learn Swift With an Open Mind

Developers making the switch from Objective-C to Swift tend to translate Objective-C into Swift. They take what they know about Cocoa development and apply it to Swift. Unfortunately, this often results in frustration and sometimes even an aversion towards the Swift language.

It's easy enough to understand why developers take this approach. People learning a new spoken language tend to look for the patterns and constructs they're already familiar with. While this is a common phase in the learning process, it emphasizes the importance of a carefully chosen learning trajectory. The same applies to picking up a new programming language.

> Developers new to Swift often take these commonalities to think that Swift and Objective-C are very similar while they're not.

Because Swift is so different from Objective-C, you need to take a step backward. You need to unlearn what you know about Objective-C and start with a clean slate and an open mind. Swift and Objective-C differ more than they're alike. It's true that a Swift application runs in the Objective-C runtime and that the languages are interoperable, they understand one another. But developers new to Swift often take these commonalities to think that Swift and Objective-C are very similar while they're not.

Reference Types and Value Types

While there's nothing inherently wrong with classes and inheritance, Swift implicitly encourages the use of value types (tuples, structures, and

enumerations) instead of reference types (classes). Several months ago, I came across a talk from Andy Matuschak[3] about this topic. It's a great introduction to the advantages value types have over reference types.

The Swift Standard Library[4] also embraces value types. Strings, arrays, dictionaries, and sets, for example, are value types in Swift. This is very different from their Foundation counterparts, `NSString`, `NSArray`, `NSDictionary`, and `NSSet`.

Protocol-Oriented Programming

Even though structures and enumerations in Swift are more powerful than their Objective-C counterparts, they don't support inheritance. This is often seen as a missing feature and scares many developers away from value types. Developers new to Swift and accustomed to object-oriented programming look for inheritance and find it in classes, that is, reference types.

Earlier in the book, I emphasize that the lack of inheritance isn't a problem. Protocols and value types are a potent combination, maybe even more so than reference types and inheritance.

Apple emphasizes that protocol-oriented programming is a pattern developers should embrace in Swift. I encourage you to watch Protocol-Oriented Programming in Swift[5] to understand what it is and how you can apply it in Swift. Dave Abrahams[6] does a tremendous job explaining why protocol-oriented programming is a natural fit for Swift.

Classes and inheritance remain important in Swift. They're not the enemy. The Cocoa frameworks are for the most part powered by Objective-C and that means classes and inheritance are here to stay, at least for the foreseeable future.

[3] https://realm.io/news/andy-matuschak-controlling-complexity/

[4] https://developer.apple.com/library/ios/documentation/General/Reference/SwiftStandardLibraryReference/

[5] https://developer.apple.com/videos/play/wwdc2015/408/

[6] https://twitter.com/daveabrahams

As a Cocoa developer, you continue to use classes and create subclasses. But it's equally important to understand the ideas that power Swift and how you can use them in your projects. In his presentation, Dave repeatedly draws attention to Swift being a protocol-oriented language.

Type Safety

Type safety is a cornerstone of the Swift programming language. As a developer, Swift expects you to be clear about the types of the variables and constants you use. The advantage is that common mistakes, such as passing a value of the wrong type to a method, can be caught by the compiler.

Developers coming from Objective-C often struggle with Swift's strict type safety rules. This ties in with another concept of Swift, **optionals**. Optionals are seen as an obstacle, an unavoidable side effect of Swift's type safety. What happened to sending messages to `nil` in Objective-C? What's wrong with that?

Optionals may indeed feel as obstacles when you first start experimenting with Swift. As time passes and you become more familiar with the language in its entirety, the pieces of the puzzle start to come together and you come to appreciate optionals for what they represent and the problem they solve. In combination with optional binding and optional chaining, optionals start to fit into that proverbial puzzle.

I agree that optionals can feel clunky if you're interacting with an Objective-C API that wasn't built with Swift or optionals in mind. But I wouldn't want to use Swift without them. Optionals are such a nice feature of Swift and I've come to appreciate them. In fact, every time I take a trip down memory lane when I need to fix a bug in Objective-C, I miss them. Optionals clearly express what's happening.

If you're still not sure about optionals, then I suggest you read the chapter about optionals in this book. Give them the benefit of the doubt for now. Use them for a few weeks, avoid the exclamation mark, and see how you feel about them in a month.

Best Practices

The more you read about Swift and the longer you spend time in the Swift community, the more you realize that Swift is still a very young language, especially if you compare the language with C and Objective-C. A common problem developers face is the lack of best practices. They look for strategies to solve common problems and struggle to find them. Best practices are slowly taking form as more people use the language in real-world scenarios.

This isn't surprising. The Swift community is still exploring the language, finding out what's possible and how things can or should be done. The language is still developing at a rapid pace and the open sourcing of the language is another component that drives this *swift* evolution.

Several best practices are slowly taking shape, such as the adoption of protocol-oriented programming, the value of immutability, and the use of value types over reference types.

Forget What You Know

My advice is to approach Swift with an open mind. Try to forget what you know. What you know has value, but it may slow you down or it may lead to frustration. Explore the language, become familiar with the foundations it's built on, and learn from others.

5 Exclamation Marks and Fatal Errors

Fatal errors have a negative connotation and with reason. You should use them sparingly if you want to avoid having your application crash and burn at the slightest hiccup. Despite their negative undertone, fatal errors are an integral part of my workflow as I write elsewhere in this book.

Whenever I write or speak about my use of fatal errors, I usually see two types of responses. Developers unfamiliar with fatal errors and how they can be used safely are surprised and excited. They spot the benefits fatal errors can bring to a project. Can you guess what the second type of response sounds like?

> Why don't you use an exclamation mark instead?

The suggestion to use an exclamation mark instead of throwing a fatal error is understandable. From a user's perspective, the result is identical. But I'm not using fatal errors with the user in mind. I don't throw a fatal error to crash the application when the user is using it. In an ideal scenario, a fatal error should only be thrown in development or when the application is being tested.

I agree that the user won't appreciate my use of fatal errors if the application crashes the moment they're about to best their previous high score. The thing is that I'm a developer and I look at code most of my working hours. And that's exactly the reason I choose for fatal errors more frequently than I choose for the exclamation mark. Let me explain what I mean by that.

Clarity Over Subtleness

My biggest complaint with the exclamation mark is its subtleness. Ironically, plenty of developers use the exclamation mark for exactly that reason. It's so easy to append an exclamation mark to a variable or a constant. It's almost too easy. I understand why the Swift team has chosen to support forced unwrapping and forced conversion using the `as!` operator, but I wouldn't shed a tear if both were removed from the Swift language tomorrow.

I agree that it can be frustrating to interact with an ancient Objective-C API that doesn't care about `nil` and optionals. Interacting with the file system, for example, can often lead you down a rabbit hole of optionals, indentation, and conditionals. But that's what it takes if you decide to write software in Swift.

Don't be lazy by appending an exclamation mark to a variable or a constant you're pretty sure will always contain a value. It will contain a value … most of the time … almost always. As the documentation explains, you should only force unwrap an optional if you're certain that it contains a value. I turn it around when I use fatal errors. A fatal error should be thrown if the application enters a state it didn't anticipate.

Choosing for Clarity

A common trait among developers is an obsession with simplicity and minimalism. Clean code is but one manifestation of this trait. By using fatal errors I choose for clarity. If the application throws a fatal error, I want to know about it. It's true that the exclamation mark will also do that for me. But I also want to know about it when I'm simply reading through my code.

An exclamation mark doesn't jump out, but a `guard` statement with a `fatalError()` call does. It immediately shows you that you know that a certain scenario should never happen and you guard against that.

Take a look at the following implementation of the `prepare(for:sender:)` method. This is a common pattern I use. If the user triggers the `Segue.SelectProfile` segue, the application expects the destination view controller to be of type `SelectProfileViewController`. It simply doesn't know how to respond if that isn't true hence the fatal error in the `else` clause of the `guard` statement. While it may look a bit verbose, it's clear and explicit.

```
1  // MARK: - Navigation
2
3  override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
4      guard let identifier = segue.identifier else { return }
5
6      switch identifier {
7      case Segue.SelectProfile:
8          guard let destination = segue.destination as? SelectProfileV\
9  iewController else {
10             fatalError("Unexpected Destination View Controller for S\
11 egue")
12         }
13
14         ...
15     default: break
16     }
17 }
```

The alternative is to use the `as!` operator to forcefully convert the destination view controller to the type the application expects.

```
1  // MARK: - Navigation
2
3  override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
4      guard let identifier = segue.identifier else { return }
5
6      switch identifier {
7      case Segue.SelectProfile:
8          let destination = segue.destination as! SelectProfileViewCon\
9  troller
10
11         ...
12     default: break
13     }
14 }
```

I understand that this practice is a bit controversial, but I've seen its effectiveness. It's why I'm a big fan of this pattern. As I mentioned earlier in this chapter, the issue I have with the exclamation mark is that it isn't explicit enough, it's too subtle. It's easy to overlook it while browsing a codebase.

The difference between the use of fatal errors and the use of the exclamation mark is subtle. You could also say that the difference is easy to miss, which is exactly why I bring it up. Give it a try and let me know what you think.

9 Speed, Quality, and Technical Debt

The first version of a product can never be released soon enough. That makes sense. As long as designers and developers are working on a product that isn't making money, it's costing money.

But speed can come at a cost. To gain speed, you need to make sacrifices. And very often speed is traded for quality, resulting in technical debt[7].

Speed and Quality

In product development, speed and quality are almost always irreconcilable. Increasing the velocity of a project means compromising on quality. From a development perspective, this usually results in the creation or accumulation of technical debt.

Assume for a moment that you're the product owner of a software project. The first version of the product needs to include five major features. Each of these features takes a week to develop. The problem is that the client wants to ship the first version of the product in three weeks. What do you do?

You have several options. The most obvious one is dropping two features. Unfortunately, that's rarely something the client agrees to. Another option is cutting down development time of each feature by one or two days. This usually translates to removing anything that doesn't involve the implementation of the feature, such as code reviews, quality assurance, and testing.

[7] https://en.wikipedia.org/wiki/Technical_debt

Technical Debt

Technical debt can creep into a project without the product owner knowing about it. The development team usually knows, though. If the team is led by a senior developer and code reviews are baked into the company's culture, technical debt is easy to spot.

A fast approaching deadline can cause even the best to ignore technical debt. The symptoms start to appear when new features take longer to build than expected and regressions make their way into the product. These are the early symptoms of technical debt.

In advanced stages, technical debt takes a project hostage. Nobody wants to touch the project anymore. Features take ages to complete and are often compromised by technical limitations caused by technical debt. Days, weeks, or months of bug fixing are needed to stabilize the project. I'm not exaggerating.

In the meantime, the product itself evolves at a snail's pace. The speedy start that was once so important for the project's success has long been forgotten and, looking back, wasn't that important after all. It rarely is.

Focus

If the client wants to ship in three weeks, the correct answer is removing features. For projects with a long shelf life, you want to avoid technical debt at any cost. Technical debt is very much like a virus. It's hard to eradicate and, if it isn't treated in its early stages, it spreads out rapidly across the project's codebase.

Focus is essential to avoid technical debt. Instead of rushing out a feature, you take the time to craft something that stands out. This doesn't only relate to the final product; it also involves the development aspect of the feature.

You don't need to overengineer the solution, but you need to make sure the feature can grow beyond its current scope. You plan and anticipate how it can or could evolve.

How to Get Rid of Technical Debt

There's no miracle cure to rid a project of technical debt. You need to take action, though, if you want to avoid worse.

Refactoring

The least radical approach is refactoring the project, focusing on one problem at a time. If you're in luck, one round of refactoring is sufficient. However, most projects suffering from technical debt have many problems that need fixing.

As I mentioned earlier, technical debt spreads like a virus and that means many areas of the codebase are infected. I strongly advise against a single round of refactoring. Map the problems you plan to attack and spread the refactoring over several releases. This ensures the release cycle isn't blocked as long as the refactoring is ongoing.

In the meantime, make sure you don't introduce new problems. Hold off on new features if possible. Don't make the same mistake twice.

If you're working on a large project with years of history, be prepared to spend weeks or months refactoring. That's the price you pay for technical debt.

Clean Slate

The most radical approach is starting anew. This means you don't need to spend months refactoring. This isn't an option for every project. If you have the opportunity to start with a clean slate, though, you're in luck.

While it means that you need to implement every feature from scratch, it's an opportunity many developers would grab with both hands. Learn from your mistakes, or those of your colleagues, and create an amazing product.

I've worked on several projects that would have cost less if the project was rebooted. But that's often a very hard sell. The client doesn't see

the problems the developer sees. They only start to see the symptoms of technical debt when deadlines are missed or features become very expensive.

Taking Shortcuts

Taking shortcuts rarely pays off in software development and most developers know this. But deadlines are often more important and developers rarely have the authority to make decisions about the feature set of the product.

1 Choose Your Teacher Wisely

The vast number of tutorials and courses about software development is a blessing for anyone interested in building software. And this is no less true[8] for anyone interested in Swift development. Getting started with Swift development is easy and it doesn't need to cost you a fortune. But, as you've probably discovered, there's a downside to this wealth of information. In this chapter, I'd like to highlight three problems that I frequently face and hear about from my students and readers.

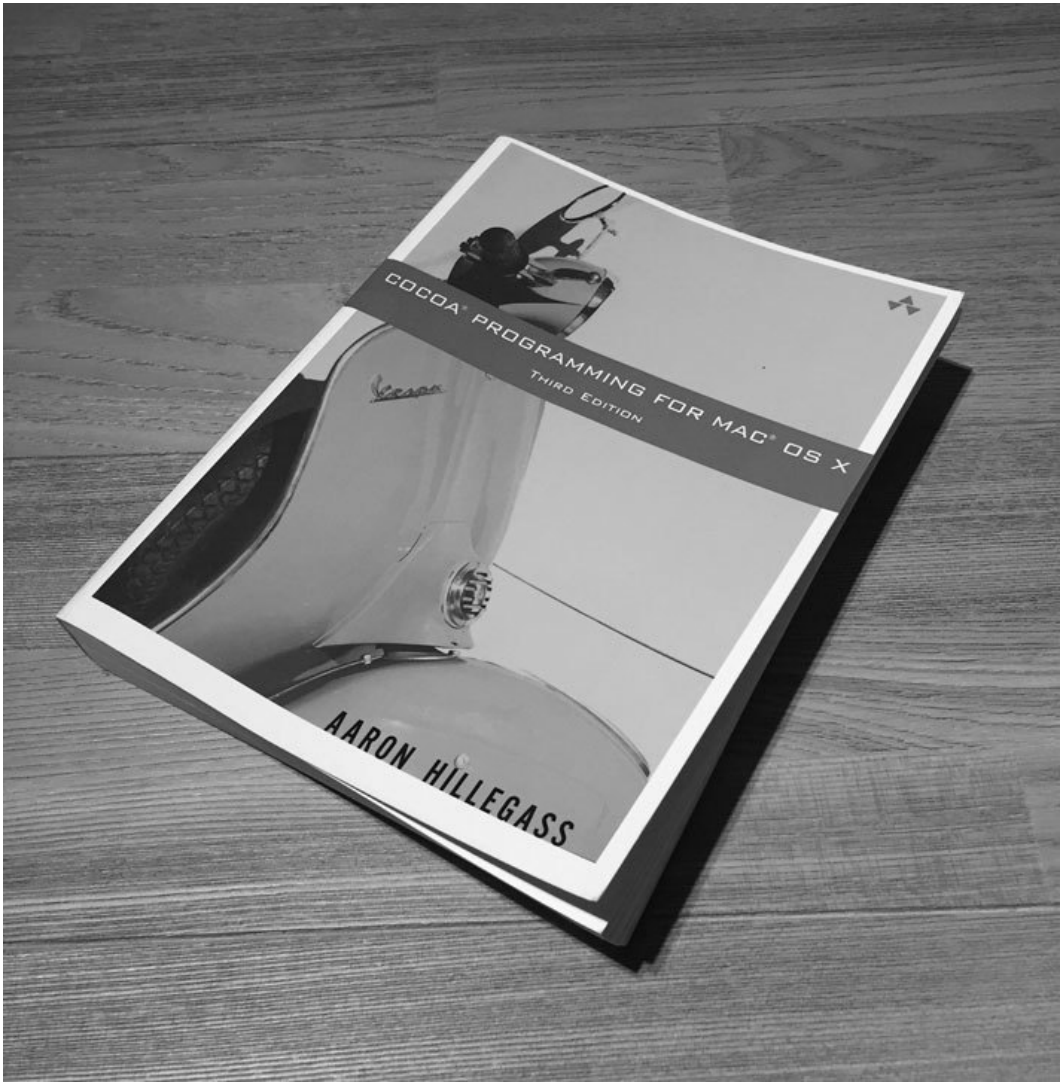
Information Overload

When I started learning Cocoa development in 2006, there were only a handful of books available. Aaron Hillegass[9]' book Cocoa Programming for (Mac) OS X[10] was the unofficial golden standard.

[8] https://www.wired.com/2015/01/redmonk-swift/

[9] https://twitter.com/aaronhillegass

[10] https://www.amazon.com/Cocoa-Programming-OS-Ranch-Guides/dp/0134076958/



Cocoa Programming for (Mac) OS X

But, with the introduction of the iPhone in 2007 and the release of the official SDK in 2008, that number increased substantially. There are many books available from traditional publishers, such as Apress and O'Reilly, but many developers, including yours truly, have chosen the path of self-publishing, bypassing traditional publishers.

It has never been easier to publish a book or course. Books and courses are available at any price point, including free. Unfortunately, this has made it more challenging for developers to decide which path to choose to learn the topic they're interested in. Each day new tutorials, videos, and courses are published. It's challenging. That's for sure.

The blessing of having so much free information at your fingertips is more often than not a curse for developers that want to learn Cocoa and Swift development. Not only is the amount of information overwhelming, there doesn't seem to be a clear path to follow. It's difficult to see the forest for the trees. Don't stop here, though. There's hope.

Who to Trust

Every developer that writes about Cocoa and Swift development does this with the best of intentions. They want to help people learn something new or solve a problem they're having. It's fantastic to see how many of us take the time to write a tutorial or produce a video to help others. It's one of the key ingredients of the thriving Cocoa and Swift communities.

Despite the author's good intentions, though, what they're teaching may be incorrect or ignore good practices. If you're new to a subject, then you may not be able to spot these mistakes. The title of this chapter is very telling. It's become more important than ever to choose your teacher wisely. While I don't believe that people are intentionally putting out bad content or incorrect information, it's up to you, the student, to filter the good from the bad.

A few weeks ago, I was looking for a solution to customize the color of the clear button of a `UITextField` instance. It turns out that the tint color of a `UITextField` doesn't affect the clear button. Bummer. I was having an issue where the clear button was nearly invisible against a dark background.

During my search for an answer, I stumbled on several Stack Overflow entries that recommended digging into the view hierarchy of the text

field, looking for the clear button, and modifying its tint color. As I mention elsewhere in this book, this is a bad practice. If the API doesn't allow for this type of customization, you file a bug with Apple and implement a custom solution. That's the only correct solution. Your clever workaround will inevitably break when Apple makes changes to the internals of the `UITextField` class. Respect the SDK. Always.

The answers that suggested digging into the view hierarchy of the text field were upvoted because, at that time, it solved the problem. Unfortunately, this creates a bigger problem. Every inexperienced developer that reads one of these answers is made to believe that this is a viable solution, that this is fine. Instead of spotting the risk of the solution, they wrongly believe that they've picked up a good practice they can adopt in other, similar situations.

If I need to pick up a new framework or API, I rely on the people I have come to trust over time. These are very often people that have built up authority in the community over several years or people like Aaron Hillegass[11], Marcus Zarra[12], and Jeff LaMarche[13] that have been around since the very early days of the platform.

Focus, Focus, Focus

I used to follow a slew of developers on social media. I wanted to know what they were learning, what they had to share, and which techniques and lessons I could adopt in my own projects. About a year ago, I stopped doing this because it was too overwhelming. There's so much to learn. The platforms and the Swift language evolve so quickly that it's hard to keep your focus if you don't protect it ferociously.

There are countless talks about almost any topic you can imagine, but it can leave you with more questions than you started with. I don't know about you, but focus has been the cure for me. Attention has become so

[11] https://twitter.com/aaronhillegass
[12] https://twitter.com/mzarra
[13] https://twitter.com/jeff_lamarche

important in today's busy world that having the skill to focus obsessively is a skill every developer should learn to master.

I don't believe in the concept of a genius. It's true that some people are more gifted than others, but, in the end, developers that excel in what they do are those who can focus and commit to something. You don't master Swift by reading a few chapters in Apple's language guide. That's a good start, but it's a process that continues and never stops. That's the beauty of it. No? Didn't you become a developer because the sky's the limit?

That's also why I often ask developers what goals they have for the coming months or years. What I'm actually asking them is what their focus is. What are they trying to accomplish? Ambitious people have a clear focus, very often a singular one. Let me ask you then, "What is your focus?"

Never Stop Learning

The mobile space is still very young, relatively speaking, and it evolves at an incredible pace. With Apple and Google heavily investing in their platforms, the speed with which mobile platforms evolve requires developers to focus and learn non-stop.

In 2015, Apple introduced no less than two new platforms, tvOS and watchOS. While developers familiar with iOS won't have a hard time getting up to speed with Apple's brand new SDKs, there are many, many APIs and paradigms to become familiar with. It's understandable if you feel a little overwhelmed as a mobile developer. That's fine and it's fine to admit that.

If you decide to become a developer, regardless of the platform you write software for, you need to accept and become comfortable with the fact that learning on a daily basis is part of the job.

For the past few years, Apple has released a new version of its operating systems every year, introducing new technologies we need to become

familiar with. The Swift project continues to evolve at a fast pace. At the time of writing, Swift 4 is just around the corner and with it come new features, bug fixes, and numerous improvements.

Are you ready to dive in head first? If Apple announces a slew of new APIs next year, will that scare you or will it excite you? Do you have the motivation to not only continue learning but also push the envelope. Using an API is one thing, pushing it to its boundaries and beyond is where it's at.

If you like programming, but prefer to stick with what you know, then mobile development may not be the best choice. If learning is in your blood and the mere thought of WWDC or Google I/O gives you goose-bumps, then being a mobile developer is the best job in the world.