futurice / **ios-good-practices**

Good ideas for iOS development, by Futurice developers. http://www.futurice.com

erikjalevik Merge pull request #36 from falsecrypt/bugfix-assets-link … Latest commit 1e2e677 Jan 27, 2016

LICENSE	Add license (CC BY 4.0)	Apr 22, 2015
README.md	Merge pull request #36 from falsecrypt/bugfix-assets-link	Jan 27, 2016

README.md

iOS Good Practices

Just like software, this document will rot unless we take care of it. We encourage everyone to help us on that – just open an issue or send a pull request!

Interested in other mobile platforms? Our Best Practices in Android Development and Windows App Development Best

Practices documents have got you covered.

Why?

Getting on board with iOS can be intimidating. Neither Swift nor Objective-C are widely used elsewhere, the platform has its own names for almost everything, and it's a bumpy road for your code to actually make it onto a physical device. This living document is here to help you, whether you're taking your first steps in Cocoaland or you're curious about doing things "the right way". Everything below is just suggestions, so if you have a good reason to do something differently, by all means go for it!

Contents

If you are looking for something specific, you can jump right into the relevant section from here.

Getting Started

Human Interface Guidelines

If you're coming from another platform, do take some time to familiarize yourself with Apple's Human Interface Guidelines for the platform. There is a strong emphasis on good design in the iOS world, and your app should be no exception. The guidelines also provide a handy overview of native UI elements, technologies such as 3D Touch or Wallet, and icon dimensions for designers.

Xcode

Xcode is the IDE of choice for most iOS developers, and the only one officially supported by Apple. There are some alternatives, of which AppCode is arguably the most famous, but unless you're already a seasoned iOS person, go with Xcode. Despite its shortcomings, it's actually quite usable nowadays!

To install, simply download Xcode on the Mac App Store. It comes with the newest SDK and simulators, and you can install more stuff under *Preferences > Downloads*.

Project Setup

A common question when beginning an iOS project is whether to write all views in code or use Interface Builder with Storyboards or XIB files. Both are known to occasionally result in working software. However, there are a few considerations:

Why code?

- Storyboards are more prone to version conflicts due to their complex XML structure. This makes merging much harder than with code.
- It's easier to structure and reuse views in code, thereby keeping your codebase DRY.
- All information is in one place. In Interface Builder you have to click through all the inspectors to find what you're looking for.

- Storyboards introduce coupling between your code and UI, which can lead to crashes e.g. when an outlet or action is not set up correctly. These issues are not detected by the compiler.

Why Storyboards?

- For the less technically inclined, Storyboards can be a great way to contribute to the project directly, e.g. by tweaking colors or layout constraints. However, this requires a working project setup and some time to learn the basics.
- Iteration is often faster since you can preview certain changes without building the project.
- Custom fonts and UI elements are represented visually in Storyboards, giving you a much better idea of the final appearance while designing.
- For size classes (available from iOS 8), Interface Builder gives you a live layout preview for the devices of your choice, including iPad split-screen multitasking.

Why not both?

To get the best of both worlds, you can also take a hybrid approach: Start off by sketching the initial design with Storyboards, which are great for tinkering and quick changes. You can even invite designers to participate in this process. As the UI matures and reliability becomes more important, you then transition into a code-based setup that's easier to maintain and collaborate on.

Ignores

A good first step when putting a project under version control is to have a decent `.gitignore` file. That way, unwanted files (user settings, temporary files, etc.) will never even make it into your repository. Luckily, GitHub has us covered for both Swift and Objective-C.

Dependency Management

CocoaPods

If you're planning on including external dependencies (e.g. third-party libraries) in your project, CocoaPods offers easy and fast integration. Install it like so:

```
sudo gem install cocoapods
```

To get started, move inside your iOS project folder and run

```
pod init
```

This creates a Podfile, which will hold all your dependencies in one place. After adding your dependencies to the Podfile, you run

```
pod install
```

to install the libraries and include them as part of a workspace which also holds your own project. It is generally recommended to commit the installed dependencies to your own repo, instead of relying on having each developer running `pod install` after a fresh checkout.

Note that from now on, you'll need to open the `.xcworkspace` file instead of `.xcproject`, or your code will not compile. The command

```
pod update
```

will update all pods to the newest versions permitted by the Podfile. You can use a wealth of operators to specify your exact version requirements.

Carthage

Carthage takes the "simple, not easy" approach by building your dependencies into binary frameworks, without magically integrating them with your project in any way. This also greatly reduces build times, because your dependencies have already been compiled by the time you start building.

There is no centralized repository of projects, which means any library that can be compiled into a framework supports Carthage out of the box. Note that dynamic frameworks are only available from iOS 8 onwards.

To get started, follow the instructions in Carthage's documentation.

Project Structure

To keep all those hundreds of source files from ending up in the same directory, it's a good idea to set up some folder structure depending on your architecture. For instance, you can use the following:

```
 ├─ Models
├─ Views
├─ Controllers (or ViewModels, if your architecture is MVVM)
├─ Stores
├─ Helpers
```

First, create them as groups (little yellow "folders") within the group with your project's name in Xcode's Project Navigator. Then, for each of the groups, link them to an actual directory in your project path by opening their File Inspector on the right, hitting the little gray folder icon, and creating a new subfolder with the name of the group in your project directory.

Localization

Keep all user strings in localization files right from the beginning. This is good not only for translations, but also for finding userfacing text quickly. You can add a launch argument to your build scheme to launch the app in a certain language, e.g.

```
-AppleLanguages (Finnish)
```

For more complex translations such as plural forms that depending on a number of items (e.g. "1 person" vs. "3 people"), you should use the `.stringsdict` format instead of a regular `localizable.strings` file. As soon as you've wrapped your head around the crazy syntax, you have a powerful tool that knows how to make plurals for "one", some", "few" and "many" items, as needed e.g. in Russian or Arabic.

Find more information about localization in these presentation slides from the February 2012 HelsinkiOS meetup. Most of the talk is still relevant in October 2014.

Constants

Keep your constants' scope as small as possible. For instance, when you only need it inside a class, it should live in that class. Those constants that need to be truly app-wide should be kept in one place. In Swift, you can use structs defined in a `Constants.swift` file to group, store and access your app-wide constants in a clean way:

```
struct Config {
    static let baseURL: NSURL(string: "http://www.example.org/")!
    static let splineReticulatorName = "foobar"
}

struct Color {
    static let primaryColor = UIColor(red: 0.22, green: 0.58, blue: 0.29, alpha: 1.0)
    static let secondaryColor = UIColor.lightGrayColor()
}
```

When using Objective-C, keep app-wide constants in a `Constants.h` file that is included in the prefix header.

Instead of preprocessor macro definitions (via `#define`), use actual constants:

```
 static CGFloat const XYZBrandingFontSizeSmall = 12.0f;
static NSString * const XYZAwesomenessDeliveredNotificationName = @"foo";
```

Actual constants are type-safe, have more explicit scope (they're not available in all imported/included files until undefined), cannot be redefined or undefined in later parts of the code, and are available in the debugger.

Branching Model

Especially when distributing an app to the public (e.g. through the App Store), it's a good idea to isolate releases to their own branch with proper tags. Also, feature work that involves a lot of commits should be done on its own branch. `git-flow` is a tool that helps you follow these conventions. It is simply a convenience wrapper around Git's branching and tagging commands, but can help maintain a proper branching structure especially for teams. Do all development on feature branches (or on `develop` for smaller work), tag releases with the app version, and commit to master only via

```
git flow release finish <version>
```

Common Libraries

Generally speaking, make it a conscious decision to add an external dependency to your project. Sure, this one neat library solves your problem now, but maybe later gets stuck in maintenance limbo, with the next OS version that breaks everything being just around the corner. Another scenario is that a feature only achievable with external libraries suddenly becomes part of the official APIs. In a well-designed codebase, switching out the implementation is a small effort that pays off quickly. Always consider solving the problem using Apple's extensive (and mostly excellent) frameworks first!

Therefore this section has been deliberately kept rather short. The libraries featured here tend to reduce boilerplate code (e.g. Auto Layout) or solve complex problems that require extensive testing, such as date calculations. As you become more proficient with iOS, be sure to dive into the source here and there, and acquaint yourself with their underlying Apple frameworks. You'll find that those alone can do a lot of the heavy lifting.

AFNetworking

A perceived 99.95 percent of iOS developers use this network library. While `NSURLSession` is surprisingly powerful by itself, AFNetworking remains unbeaten when it comes to actually managing a queue of requests, which is pretty much a requirement in any modern app.

DateTools

As a general rule, don't write your date calculations yourself. Luckily, in DateTools you get an MIT-licensed, thoroughly tested library that covers pretty much all your calendar needs.

Auto Layout Libraries

If you prefer to write your views in code, chances are you've met either of Apple's awkward syntaxes – the regular `NSLayoutConstraint` factory or the so-called Visual Format Language. The former is extremely verbose and the latter based on strings, which effectively prevents compile-time checking. Fortunately, they've addressed the issue in iOS 9, allowing a more concise specification of constraints.

If you're stuck with an earlier iOS version, Masonry/SnapKit remedies the problem by introducing its own DSL to make, update and replace constraints. For Swift, there is also Cartography, which builds on the language's powerful operator overloading features. For the more conservative, FLKAutoLayout offers a clean, but rather non-magical wrapper around the native APIs.

Architecture

- Model-View-Controller-Store (MVCS)
  - This is the default Apple architecture (MVC), extended by a Store layer that vends Model instances and handles the networking, caching etc.
  - Every Store exposes to the view controllers either `RACSignal`s or `void`-returning methods with custom completion blocks.
- Model-View-ViewModel (MVVM)
  - Motivated by "massive view controllers": MVVM considers `UIViewController` subclasses part of the View and keeps them slim by maintaining all state in the ViewModel.
  - To learn more about it, check out Bob Spryn's fantastic introduction.
- View-Interactor-Presenter-Entity-Routing (VIPER)
  - Rather exotic architecture that might be worth looking into in larger projects, where even MVVM feels too cluttered and testability is a major concern.

“Event” Patterns

These are the idiomatic ways for components to notify others about things:

- **Delegation:** *(one-to-one)* Apple uses this a lot (some would say, too much). Use when you want to communicate stuff back e.g. from a modal view.
- **Callback blocks:** *(one-to-one)* Allow for a more loose coupling, while keeping related code sections close to each other. Also scales better than delegation when there are many senders.
- **Notification Center:** *(one-to-many)* Possibly the most common way for objects to emit “events” to multiple observers. Very loose coupling — notifications can even be observed globally without reference to the dispatching object.
- **Key-Value Observing (KVO):** *(one-to-many)* Does not require the observed object to explicitly “emit events” as long as it is *Key-Value Coding (KVC)* compliant for the observed keys (properties). Usually not recommended due to its implicit nature and the cumbersome standard library API.
- **Signals:** *(one-to-many)* The centerpiece of ReactiveCocoa, they allow chaining and combining to your heart's content, thereby offering a way out of callback hell.

Models

Keep your models immutable, and use them to translate the remote API's semantics and types to your app. For Objective-C projects, Github's Mantle is a good choice. In Swift, you can use structs instead of classes to ensure immutability, and use a parsing library such as SwiftyJSON or Argo to do the JSON-to-model mapping.

Views

With today's wealth of screen sizes in the Apple ecosystem and the advent of split-screen multitasking on iPad, the boundaries between devices and form factors become increasingly blurred. Much like today's websites are expected to adapt to different browser window sizes, your app should handle changes in available screen real estate in a graceful way. This can happen e.g. if the user rotates the device or swipes in a secondary iPad app next to your own.

Instead of manipulating view frames directly, you should use size classes and Auto Layout to declare constraints on your views. The system will then calculate the appropriate frames based on these rules, and re-evaluate them when the environment changes.

Apple's recommended approach for setting up your layout constraints is to create and activate them once during initialization. If you need to change your constraints dynamically, hold references to them and then deactivate/activate them as required. The main use case for `UIView`'s `updateConstraints` (or its `UIViewController` counterpart, `updateViewConstraints`) is when you want the system to perform batch updates for better performance. However, this comes at the cost of having to call `setNeedsUpdateConstraints` elsewhere in your code, increasing its complexity.

If you override `updateConstraints` in a custom view, you should explicitly state that your view requires a constraint-based layout:

Swift:

```
override class func requiresConstraintBasedLayout() -> Bool {
    return true
}
```

Objective-C:

```
+ (BOOL)requiresConstraintBasedLayout {
    return YES
}
```

Otherwise, you may encounter strange bugs when the system doesn't call `updateConstraints()` as you would expect it to. This blog post by Edward Huynh offers a more detailed explanation.

Controllers

Use dependency injection, i.e. pass any required objects in as parameters, instead of keeping all state around in singletons. The latter is okay only if the state *really* is global.

Swift:

```
let fooViewController = FooViewController(viewModel: fooViewModel)
```

Objective-C:

```
FooViewController *fooViewController = [[FooViewController alloc] initWithViewModel:fooViewModel];
```

Try to avoid bloating your view controllers with logic that can safely reside in other places. Soroush Khanlou has a good writeup of how to achieve this, and architectures like MVVM treat view controllers as views, thereby greatly reducing their complexity.

Stores

At the "ground level" of a mobile app is usually some kind of model storage, that keeps its data in places such as on disk, in a local database, or on a remote server. This layer is also useful to abstract away any activities related to the vending of model objects, such as caching.

Whether it means kicking off a backend request or deserializing a large file from disk, fetching data is often asynchronous in nature. Your store's API should reflect this by offering some kind of deferral mechanism, as synchronously returning the data would cause the rest of your app to stall.

If you're using ReactiveCocoa, `SignalProducer` is a natural choice for the return type. For instance, fetching gigs for a given artist would yield the following signature:

Swift + RAC 3:

```
func fetchGigsForArtist(artist: Artist) -> SignalProducer<[Gig], NSError> {
    // …
}
```

ObjectiveC + RAC 2:

```
- (RACSignal *)fetchGigsForArtist:(Artist *)artist {
    // …
}
```

Here, the returned `SignalProducer` is merely a "recipe" for getting a list of gigs. Only when started by the subscriber, e.g. a view model, will it perform the actual work of fetching the gigs. Unsubscribing before the data has arrived would then cancel the network request.

If you don't want to use signals, futures or similar mechanisms to represent your future data, you can also use a regular callback block. Keep in mind that chaining or nesting such blocks, e.g. in the case where one network request depends on the outcome of another, can quickly become very unwieldy – a condition generally known as "callback hell".

Assets

Asset catalogs are the best way to manage all your project's visual assets. They can hold both universal and device-specific (iPhone 4-inch, iPhone Retina, iPad, etc.) assets and will automatically serve the correct ones for a given name. Teaching your designer(s) how to add and commit things there (Xcode has its own built-in Git client) can save a lot of time that would otherwise be spent copying stuff from emails or other channels to the codebase. It also allows them to instantly try out their changes and iterate if needed.

Using Bitmap Images

Asset catalogs expose only the names of image sets, abstracting away the actual file names within the set. This nicely prevents asset name conflicts, as files such as `button_large@2x.png` are now namespaced inside their image sets. Appending the modifiers `-568h`, `@2x`, `~iphone` and `~ipad` are not required per se, but having them in the file name when dragging the file to an image set will automatically place them in the right "slot", thereby preventing assignment mistakes that can be hard to hunt down.

Using Vector Images

You can include the original vector graphics (PDFs) produced by designers into the asset catalogs, and have Xcode automatically generate the bitmaps from that. This reduces the complexity of your project (the number of files to manage.)

Coding Style

Naming

Apple pays great attention to keeping naming consistent, if sometimes a bit verbose, throughout their APIs. When developing for Cocoa, you make it much easier for new people to join the project if you follow Apple's naming conventions.

Here are some basic takeaways you can start using right away:

A method beginning with a *verb* indicates that it performs some side effects, but won't return anything: `- (void)loadView;` `- (void)startAnimating;`

Any method starting with a *noun*, however, returns that object and should do so without side effects: `- (UINavigationItem *)navigationItem;` `+ (UILabel *)labelWithText:(NSString *)text;`

It pays off to keep these two as separated as possible, i.e. not perform side effects when you transform data, and vice versa. That will keep your side effects contained to smaller sections of the code, which makes it more understandable and facilitates debugging.

Structure

`MARK:` comments (Swift) and pragma marks (Objective-C) are a great way to group your methods, especially in view controllers. Here is a Swift example for a common structure that works with almost any view controller:

```
import SomeExternalFramework

class FooViewController : UIViewController, FoobarDelegate {

    let foo: Foo

    private let fooStringConstant = "FooConstant"
    private let floatConstant = 1234.5

    // MARK: Lifecycle

    // Custom initializers go here
    // MARK: View Lifecycle

    override func viewDidLoad() {
        super.viewDidLoad()
        // …
    }

    // MARK: Layout

    private func makeViewConstraints() {
        // …
    }

    // MARK: User Interaction

    func foobarButtonTapped() {
        // …
    }

    // MARK: FoobarDelegate

    func foobar(foobar: Foobar didSomethingWithFoo foo: Foo) {
        // …
    }

    // MARK: Additional Helpers

    private func displayNameForFoo(foo: Foo) {
        // …
    }

}
```

The most important point is to keep these consistent across your project's classes.

External Style Guides

Futurice does not have company-level guidelines for coding style. It can however be useful to peruse the style guides of other software companies, even if some bits can be quite company-specific or opinionated.

- GitHub: Swift and Objective-C
- Ray Wenderlich: Swift and Objective-C
- Google: Objective-C
- The New York Times: Objective-C
- Sam Soffes: Objective-C
- Luke Redpath: Objective-C

Security

Even in an age where we trust our portable devices with the most private data, app security remains an often-overlooked subject. Try to find a good trade-off given the nature of your data; following just a few simple rules can go a long way here. A good resource to get started is Apple's own iOS Security Guide.

Data Storage

If your app needs to store sensitive data, such as a username and password, an authentication token or some personal user details, you need to keep these in a location where they cannot be accessed from outside the app. Never use `NSUserDefaults`, other plist files on disk or Core Data for this, as they are not encrypted! In most such cases, the iOS Keychain is your friend. If you're uncomfortable working with the C APIs directly, you can use a wrapper library such as SSKeychain or UICKeyChainStore.

When storing files and passwords, be sure to set the correct protection level, and choose it conservatively. If you need access while the device is locked (e.g. for background tasks), use the "accessible after first unlock" variety. In other cases, you should probably require that the device is unlocked to access the data. Only keep sensitive data around while you need it.

Networking

Keep any HTTP traffic to remote servers encrypted with TLS at all times. To avoid man-in-the-middle attacks that intercept your encrypted traffic, you can set up certificate pinning. Popular networking libraries such as AFNetworking and Alamofire support this out of the box.

Logging

Take extra care to set up proper log levels before releasing your app. Production builds should never log passwords, API tokens and the like, as this can easily cause them to leak to the public. On the other hand, logging the basic control flow can help you pinpoint issues that your users are experiencing.

User Interface

When using `UITextField`s for password entry, remember to set their `secureTextEntry` property to `true` to avoid showing the password in cleartext. You should also disable auto-correction for the password field, and clear the field whenever appropriate, such as when your app enters the background.

When this happens, it's also good practice to clear the Pasteboard to avoid passwords and other sensitive data from leaking. As iOS may take screenshots of your app for display in the app switcher, make sure to clear any sensitive data from the UI *before* returning from `applicationDidEnterBackground`.

Diagnostics

Compiler warnings

It is recommended that you enable as many compiler warnings as possible, and treat warnings as errors. This recommendation is justified in these presentation slides. The slides also contain information on how to suppress certain warnings in specific files, or in specific sections of code.

In short, add at least these values to the *“Other Warning Flags”* build setting:

- `-Wall` *(Enables lots of additional warnings)*
- `-Wextra` *(Enables more additional warnings)*

Also enable the *“Treat warnings as errors”* build setting.

Clang Static Analyzer

The Clang compiler (which Xcode uses) has a *static analyzer* that performs control and data flow analysis on your code and checks for lots of errors that the compiler cannot.

You can manually run the analyzer from the *Product* → *Analyze* menu item in Xcode.

The analyzer can work in either “shallow” or “deep” mode. The latter is much slower but may find more issues due to cross-function control and data flow analysis.

Recommendations:

- Enable *all* of the checks in the analyzer (by enabling all of the options in the “Static Analyzer” build setting sections).
- Enable the *“Analyze during ‘Build’”* build setting for your release build configuration to have the analyzer run automatically during release builds. (Seriously, do this — you’re not going to remember to run it manually.)
- Set the *“Mode of Analysis for ‘Analyze’”* build setting to *Shallow (faster)*.
- Set the *“Mode of Analysis for ‘Build’”* build setting to *Deep*.

Faux Pas

Created by our very own Ali Rantakari, Faux Pas is a fabulous static error detection tool. It analyzes your codebase and finds issues you had no idea even existed. There is no Swift support yet, but the tool also offers plenty of language-agnostic rules. Be sure to run it before shipping any iOS (or Mac) app!

(Note: all Futurice employees get a free license to this — just ask Ali.)

Debugging

When your app crashes, Xcode does not break into the debugger by default. To achieve this, add an exception breakpoint (click the "+" at the bottom of Xcode's Debug Navigator) to halt execution whenever an exception is raised. In many cases, you will then see the line of code responsible for the exception. This catches any exception, even handled ones. If Xcode keeps breaking on benign exceptions in third party libraries e.g., you might be able to mitigate this by choosing *Edit Breakpoint* and setting the *Exception* drop-down to *Objective-C*.

For view debugging, Reveal and Spark Inspector are two powerful visual inspectors that can save you hours of time, especially if you're using Auto Layout and want to locate views that are collapsed or off-screen. Granted, Xcode offers something very similar for free, but it's iOS 8+ only and feels somewhat less polished.

Profiling

Xcode comes with a profiling suite called Instruments. It contains a myriad of tools for profiling memory usage, CPU, network communications, graphics and much more. It's a complex beast, but one of its more straight-forward use cases is tracking down memory leaks with the Allocations instrument. Simply choose *Product* > *Profile* in Xcode, select the Allocations instrument, hit the Record button and filter the Allocation Summary on some useful string, like the prefix of your own app's class names. The count in the Persistent column then tells you how many instances of each object you have. Any class for which the instance count increases indiscriminately indicates a memory leak.

Also good to know is that Instruments has an Automation tool for recording and playing back UI interactions as JavaScript files. UI Auto Monkey is a script that will use Automation to randomly pummel your app with taps, swipes and rotations which can be useful for stress/soak testing.

Pay extra attention to how and where you create expensive classes. `NSDateFormatter`, for instance, is very expensive to create and doing so in rapid succession, e.g. inside a `tableView:cellForRowAtIndexPath:` method, can really slow down your app. Instead, keep a static instance of it around for each date format that you need.

Analytics

Including some analytics framework in your app is strongly recommended, as it allows you to gain insights on how people actually use it. Does feature X add value? Is button Y too hard to find? To answer these, you can send events, timings and other measurable information to a service that aggregates and visualizes them – for instance, Google Tag Manager. The latter is more versatile than Google Analytics in that it inserts a data layer between app and Analytics, so that the data logic can be modified through a web service without having to update the app.

A good practice is to create a slim helper class, e.g. `AnalyticsHelper`, that handles the translation from app-internal models and data formats (`FooModel`, `NSTimeInterval`, …) to the mostly string-based data layer:

```
func pushAddItemEventWithItem(item: Item, editMode: EditMode) {
    let editModeString = nameForEditMode(editMode)

    pushToDataLayer([
        "event": "addItem",
        "itemIdentifier": item.identifier,
        "editMode": editModeString
    ])
}
```

This has the additional advantage of allowing you to swap out the entire Analytics framework behind the scenes if needed, without the rest of the app noticing.

Crash Logs

First you should make your app send crash logs onto a server somewhere so that you can access them. You can implement this manually (using PLCrashReporter and your own backend) but it's recommended that you use an existing service instead — for example one of the following:

- Crashlytics
- HockeyApp
- Crittercism
- Splunk MINTexpress

Once you have this set up, ensure that you *save the Xcode archive (* `.xcarchive` *)* of every build you release. The archive contains the built app binary and the debug symbols (`dSYM`) which you will need to symbolicate crash reports from that particular version of your app.

Building

This section contains an overview of this topic — please refer here for more comprehensive information:

- iOS Developer Library: Xcode Concepts
- Samantha Marshall: Managing Xcode

Build Configurations

Even simple apps can be built in different ways. The most basic separation that Xcode gives you is that between *debug* and *release* builds. For the latter, there is a lot more optimization going on at compile time, at the expense of debugging possibilities. Apple suggests that you use the *debug* build configuration for development, and create your App Store packages using the *release* build configuration. This is codified in the default scheme (the dropdown next to the Play and Stop buttons in Xcode), which commands that *debug* be used for Run and *release* for Archive.

However, this is a bit too simple for real-world applications. You might – no, *should!* – have different environments for testing, staging and other activities related to your service. Each might have its own base URL, log level, bundle identifier (so you can install them side-by-side), provisioning profile and so on. Therefore a simple debug/release distinction won't cut it. You can add more build configurations on the "Info" tab of your project settings in Xcode.

`xcconfig` files for build settings

Typically build settings are specified in the Xcode GUI, but you can also use *configuration settings files* (“ `.xcconfig` files”) for them. The benefits of using these are:

- You can add comments to explain things.
- You can `#include` other build settings files, which helps you avoid repeating yourself:
    - If you have some settings that apply to all build configurations, add a `Common.xcconfig` and `#include` it in all the other files.
    - If you e.g. want to have a “Debug” build configuration that enables compiler optimizations, you can just `#include "MyApp_Debug.xcconfig"` and override one of the settings.
- Conflict resolution and merging becomes easier.

Find more information about this topic in these presentation slides.

Targets

A target resides conceptually below the project level, i.e. a project can have several targets that may override its project settings. Roughly, each target corresponds to "an app" within the context of your codebase. For instance, you could have country-specific apps (built from the same codebase) for different countries' App Stores. Each of these will need development/staging/release builds, so it's better to handle those through build configurations, not targets. It's not uncommon at all for an app to only have a single target.

Schemes

Schemes tell Xcode what should happen when you hit the Run, Test, Profile, Analyze or Archive action. Basically, they map each of these actions to a target and a build configuration. You can also pass launch arguments, such as the language the app should run in (handy for testing your localizations!) or set some diagnostic flags for debugging.

A suggested naming convention for schemes is `MyApp (<Language>) [Environment]`:

```
 MyApp (English) [Development]
MyApp (German) [Development]
MyApp [Testing]
MyApp [Staging]
MyApp [App Store]
```

For most environments the language is not needed, as the app will probably be installed through other means than Xcode, e.g. TestFlight, and the launch argument thus be ignored anyway. In that case, the device language should be set manually to test localization.

Deployment

Deploying software on iOS devices isn't exactly straightforward. That being said, here are some central concepts that, once understood, will help you tremendously with it.

Signing

Whenever you want to run software on an actual device (as opposed to the simulator), you will need to sign your build with a **certificate** issued by Apple. Each certificate is linked to a private/public keypair, the private half of which resides in your Mac's Keychain. There are two types of certificates:

- **Development certificate:** Every developer on a team has their own, and it is generated upon request. Xcode might do this for you, but it's better not to press the magic "Fix issue" button and understand what is actually going on. This certificate is needed to deploy development builds to devices.
- **Distribution certificate:** There can be several, but it's best to keep it to one per organization, and share its associated key through some internal channel. This certificate is needed to ship to the App Store, or your organization's internal "enterprise app store".

Provisioning

Besides certificates, there are also **provisioning profiles**, which are basically the missing link between devices and certificates. Again, there are two types to distinguish between development and distribution purposes:

- **Development provisioning profile:** It contains a list of all devices that are authorized to install and run the software. It is also linked to one or more development certificates, one for each developer that is allowed to use the profile. The profile can be tied to a specific app or use a wildcard App ID (*). The latter is discouraged, because Xcode is notoriously bad at picking the correct files for signing unless guided in the right direction. Also, certain capabilities like Push Notifications or App Groups require an explicit App ID.

- **Distribution provisioning profile:** There are three different ways of distribution, each for a different use case. Each distribution profile is linked to a distribution certificate, and will be invalid when the certificate expires.

    - **Ad-Hoc:** Just like development profiles, it contains a whitelist of devices the app can be installed to. This type of profile can be used for beta testing on 100 devices per year. For a smoother experience and up to 1000 distinct users, you can use Apple's newly acquired TestFlight service. Supertop offers a good summary of its advantages and issues.
    - **App Store:** This profile has no list of allowed devices, as anyone can install it through Apple's official distribution channel. This profile is required for all App Store releases.
    - **Enterprise:** Just like App Store, there is no device whitelist, and the app can be installed by anyone with access to the enterprise's internal "app store", which can be just a website with links. This profile is available only on Enterprise accounts.

To sync all certificates and profiles to your machine, go to Accounts in Xcode's Preferences, add your Apple ID if needed, and double-click your team name. There is a refresh button at the bottom, but sometimes you just need to restart Xcode to make everything show up.

Debugging Provisioning

Sometimes you need to debug a provisioning issue. For instance, Xcode may refuse to install the build to an attached device, because the latter is not on the (development or ad-hoc) profile's device list. In those cases, you can use Craig Hockenberry's excellent Provisioning plugin by browsing to `~/Library/MobileDevice/Provisioning Profiles`, selecting a `.mobileprovision` file and hitting Space to launch Finder's Quick Look feature. It will show you a wealth of information such as devices, entitlements, certificates, and the App ID.

Uploading

iTunes Connect is Apple's portal for managing your apps on the App Store. To upload a build, Xcode 6 requires an Apple ID that is part of the developer account used for signing. This can make things tricky when you are part of several developer accounts and want to upload their apps, as for mysterious reasons *any given Apple ID can only be associated with a single iTunes Connect account*. One workaround is to create a new Apple ID for each iTunes Connect account you need to be part of, and use Application Loader instead of Xcode to upload the builds. That effectively decouples the building and signing process from the upload of the resulting `.app` file.

After uploading the build, be patient as it can take up to an hour for it to show up under the Builds section of your app version. When it appears, you can link it to the app version and submit your app for review.

In-App Purchases (IAP)

When validating in-app purchase receipts, remember to perform the following checks:

- **Authenticity:** That the receipt comes from Apple
- **Integrity:** That the receipt has not been tampered with
- **App match:** That the app bundle ID in the receipt matches your app's bundle identifier
- **Product match:** That the product ID in the receipt matches your expected product identifier
- **Freshness:** That you haven't seen the same receipt ID before.

Whenever possible, design your IAP system to store the content for sale server-side, and provide it to the client only in exchange for a valid receipt that passes all of the above checks. This kind of a design thwarts common piracy mechanisms, and — since the validation is performed on the server — allows you to use Apple's HTTP receipt validation service instead of interpreting the receipt `PKCS #7` / `ASN.1` format yourself.

For more information on this topic, check out the Futurice blog: Validating in-app purchases in your iOS app.

License

More Ideas

- Add list of suggested compiler warnings
- Ask IT about automated Jenkins build machine
- Add section on Testing
- Add "proven don'ts"