Table of Contents

The iOS is the operating system created by Apple Inc. for mobile devices. The iOS is used in many of the mobile devices for apple such as iPhone, iPod, iPad etc.

iOS Architecture

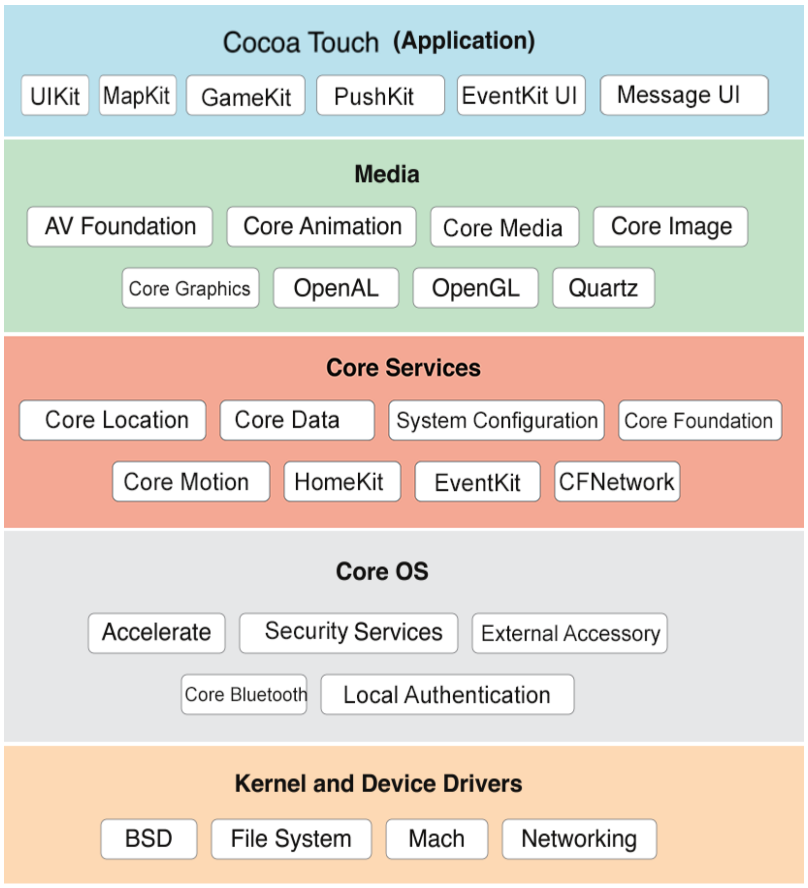
Architecture of IOS is a layered architecture. At the uppermost level iOS works as an intermediary between the underlying hardware and the apps you make. Apps do not communicate to the underlying hardware directly.

Apps talk with the hardware through a collection of well-defined system interfaces. These interfaces make it simple to write apps that work constantly on devices having various hardware abilities.

Lower layers gives the basic services which all application relies on and higher-level layer gives sophisticated graphics and interface related services.

Apple provides most of its system interfaces in special packages called frameworks.

A framework is a directory that holds a dynamic shared library that is .a files, related resources like as header files, images, and helper apps required to support that library. Every layer have a set of Framework which the developer use to construct the applications.

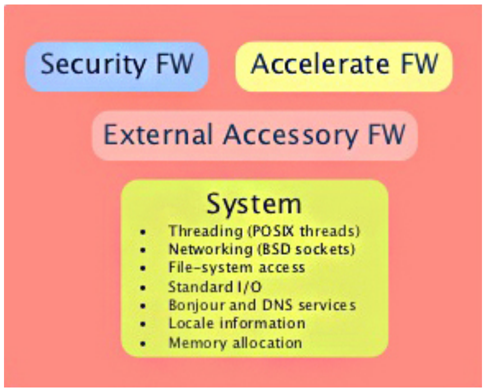


Kernel and Device Drivers

This is the lowest layer of iOS which mainly includes the kernel and device drivers. macOS and iOS share the same Unix-based XNU (hybrid) core called Darwin[1]. The kernel of Darwin is XNU[2], a hybrid kernel which uses OSFMK 7.3 (Open Software Foundation Mach[3] Kernel), various elements of FreeBSD[4] (including the process model, network stack, and virtual file system), and an object-oriented device driver API called I/O Kit. That provides high-performance networking facilities and support for multiple, integrated file systems.

Core OS

Most of the functionality provided by the three higher level layers is built on the Core OS layer and its low level features. The Core OS layer provides a handful of frameworks that your application can use directly, such as the Accelerate and Security frameworks.



The Core OS layer also encapsulates the kernel environment and low level UNIX interfaces to which your application has no access, for obvious security reasons. However, through the C-based **libSystem** library many low level features can be accessed directly, such as BSD sockets, POSIX threads, and DNS services. This important layer responsible to managing memory—allocating and

[1] **Darwin** is an open-source Unix-like operating system first released by Apple Inc. in 2000. It is composed of code developed by Apple, as well as code derived from NeXTSTEP, BSD, Mach, and other free software projects.

[2] **XNU** is the OS kernel developed at Apple Inc (Originally developed by NeXT for the NeXTSTEP operating system). since December 1996 for use in the macOS operating system and released as free and open-source software as part of the Darwin OS, which is the basis for the Apple TV Software, iOS, iPadOS, watchOS, and tvOS OSes.

[3] **Mach** (/mɑːk/) is a kernel developed to support operating system research, primarily distributed and parallel computing. Mach is often mentioned as one of the earliest examples of a microkernel. However, not all versions of Mach are microkernels. Mach's derivatives are the basis of the operating system kernel in GNU Hurd and of Apple's XNU kernel used in macOS, iOS, iPadOS, tvOS, and watchOS. The project ending with Mach 3.0, which is a true microkernel. Mach was developed as a replacement for the kernel in the BSD version of Unix, so no new operating system would have to be designed around it.

[4] The **Berkeley Software Distribution** (**BSD**) was an operating system based on Research Unix, Today, "BSD" often refers to its descendants, such as FreeBSD, OpenBSD, NetBSD, or DragonFly BSD, and systems based on those descendants. BSD was initially called Berkeley Unix because it was based on the source code of the original Unix.

releasing memory once the application has finished with it, taking care of file system tasks, handling networking, and other operating system tasks. It also interacts directly with the hardware.

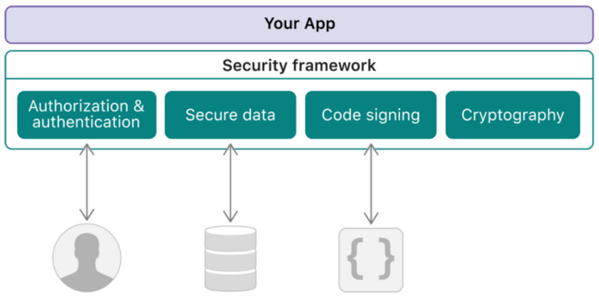
Frameworks

- **Core Bluetooth Framework -** The Core Bluetooth framework allows developers to interact specifically with Bluetooth low energy (LE) accessories.

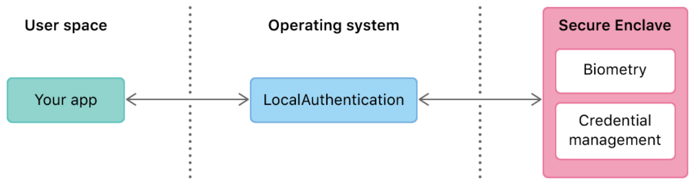
- **Accelerate Framework -** This framework contains a variety of C APIs for vector and matrix math, digital signal processing, large number handling, and image processing.

- **External Accessory Framework -** The External Accessory framework enables your app to communicate with external hardware that is connected to an iOS-based device through the Apple Lightning or 30-pin connector, or wirelessly through Bluetooth. You can configuring your app's Info.plist file and add the UISupportedExternalAccessoryProtocols key, which declares the specific hardware protocols your app supports.

- **Security Services framework -** Use the Security framework to protect information, establish trust, and control access to software. Broadly, security services support these goals:
    - Establish a user’s identity (authentication) and then selectively grant access to resources (authorization).
    - Secure data, both on disk and in motion across a network connection.
    - Ensure the validity of code to be executed for a particular purpose.

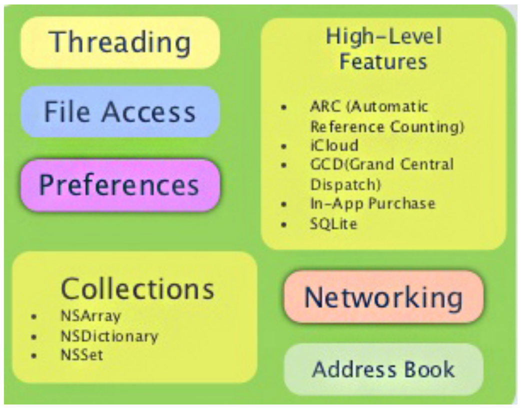


- **Local Authentication framework -** Many users rely on biometric authentication like Face ID or Touch ID to enable secure, effortless access to their devices. As a fallback option, and for devices without biometry, a passcode or password serves a similar purpose. Use the LocalAuthentication framework to leverage these mechanisms in your app



Core Services

The Core Services layer is in charge of managing the fundamental system services that native iOS applications use. The Cocoa Touch layer depends on the Core Services layer for some of its functionality. The Core Services layer also provides a number of indispensable features, such as block objects, Grand Central Dispatch, In-App Purchase, and iCloud Storage.



The iPhone Core Services layer provides much of the foundation on which the above layers are built. It provide Peer-to-Peer Services, iCloud Storage, Block Objects, Data Protection, File-Sharing Support, GCD[1], ARC[2], In-App Purchase, SQLite, XML Support features.

Frameworks

- **EventKit Framework -** The EventKit framework provides access to calendar and reminders data so you and your users can create, retrieve, and edit calendar items in your app.
- **Accounts Framework -** The Accounts framework provides access to user accounts stored in the Accounts database, which is managed by the system. An account stores the login credentials of a particular service, such as Twitter
- **Address book framework –** Gives programmatic access to a contacts database of user.
- **Cloud Kit framework –** Gives a medium for moving data between your app and iCloud.
- **Core data Framework –** Core Data is a framework that you use to manage the model layer objects in your application. It provides generalized and automated solutions to common tasks associated with object life cycle and object graph management, including persistence. As Apple states, it is not a database, but instead a persistence framework that commonly uses SQLite to store and retrieve structured data.

[1] **Grand Central Dispatch (GCD),** is a technology developed by Apple Inc. to optimize application support for systems with multi-core processors. It is an implementation of task parallelism based on the thread pool pattern. The fundamental idea is to move the management of the thread pool out of the hands of the developer, and closer to the operating system

[2] **Automatic Reference Counting (ARC)** is a memory management feature of the LLVM compiler providing automatic reference counting for the Objective-C and Swift programming languages. At compile time, it inserts into the object code messages retain and release which increase and decrease the reference count at run time, marking for deallocation those objects when the number of references to them reaches zero.

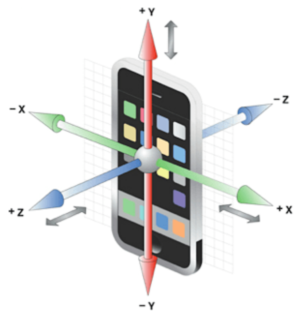
- **Core Foundation framework –** Interfaces that gives fundamental data management and service features for iOS apps. Core Foundation is the C-level API, which provides CFString, CFDictionary and the like.

- **Foundation Framework –** Objective C covering too many of the features found in the Core Foundation framework. Foundation is Objective-C, which provides NSString, NSDictionary, etc.

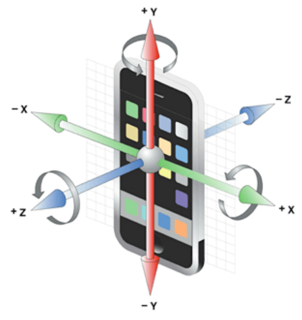
- **Core Location framework –** Gives location and heading information to apps.

- **Core Motion Framework –** Core Motion reports motion- and environment-related data from the onboard hardware of iOS devices, including:

  - **Accelerometers -** An accelerometer measures changes in velocity along one axis. All iOS devices have a three-axis accelerometer, which delivers acceleration values in each of the three axes.
  - **Gyroscopes -** A gyroscope measures the rate at which a device rotates around a spatial axis. Many iOS devices have a three-axis gyroscope
  - **Pedometer -** A pedometer counts each step a person takes by detecting the motion of the person's hands or hips
  - **Magnetometer -** The magnetic sensor, better known as magnetometer, measures earth’s magnetic field. This in addition with software algorithms determines direction your phone is facing.
  - **Barometer -** A barometric pressure sensor measures air pressure and can, therefore, play a key role in weather forecasting



Accelerometer



Gyroscope

Pedometer

Magenetometer

station pressure 1013.92 hPa
station altitude (GPS) 23.97 ft
pressure at sea level 1014.80 hPa
Barometer

- **HealthKit framework –** For handling health-related information of user

- **HomeKit framework –** For talking with and controlling connected devices in a user’s home.

- **Social framework –** Simple interface for accessing the user’s social media accounts.

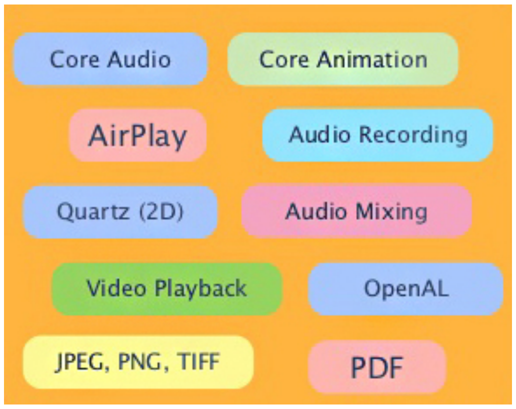
- **StoreKit framework –** Gives support for the buying of content and services from inside your iOS apps, a feature known as In-App Purchase.

- **CFNetwork -** Access network services and handle changes in network configurations. Build on abstractions of network protocols to simplify tasks such as working with BSD sockets, administering HTTP and FTP servers, and managing Bonjour services.

- **System Configuration Framework -** Allow applications to access a device's network configuration settings. Determine the reachability of the device, such as whether Wi-Fi or cell connectivity are active.

Media

Graphics, audio, and video are handled by the Media layer. This layer contains a number of key technologies, such as Core Graphics, OpenGL ES and OpenAL, AV Foundation, and Core Media.



The Media layer contains a large number of frameworks including the Assets Library framework to access the photos and videos stored on the device, the Core Image framework for image manipulation through filters, and the Core Graphics framework for 2D drawing.

Frameworks

- **Core Graphics framework –** The Core Graphics framework is based on the Quartz[1] advanced drawing engine. It provides low-level, lightweight 2D rendering with unmatched output fidelity. You use this framework to handle path-based drawing, transformations, color management, offscreen rendering, patterns, gradients and shadings, image data management, image creation, and image masking, as well as PDF document creation, display, and parsing.

- **Core Animation framework –** Core Animation provides high frame rates and smooth animations without burdening the CPU and slowing down your app. You configure animation parameters such as the start and end points, and Core Animation does the rest.

- **Core Images framework –** Core Image is an image processing and analysis technology that provides high-performance processing for still and video images

[1] **Quartz** is at the heart of the iOS graphics. It provides rendering support for 2D content and combines a rich imaging model with on-the-fly rendering, compositing, and antialiasing of content.

- **OpenGl ES and GLKit framework –** Create 3D and 2D graphics effects with this compact, efficient subset of OpenGL. OpenGL ES provides a C-based interface for hardware-accelerated 2D and 3D graphics rendering.

  GLKit Speed up OpenGL ES or OpenGL app development. Use math libraries, background texture loading, pre-created shader effects, and a standard view and view controller to implement your rendering loop.

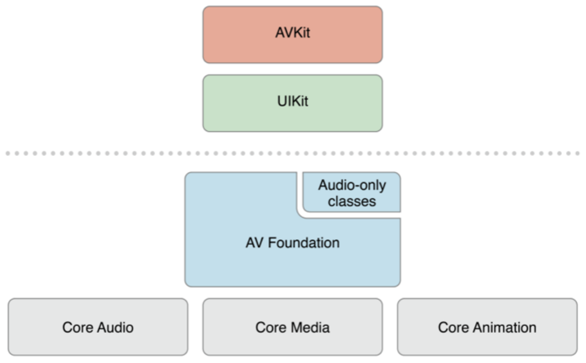
- **Metal framework –** Render advanced 3D graphics and perform data-parallel computations using graphics processors. Graphics processors (GPUs) are designed to quickly render graphics and perform data-parallel calculations. Use the Metal framework when you need to communicate directly with the GPUs available on a device

- **Media Player Framework –** It is a high level framework which gives simple use to a user's iTunes library and support for playing playlists.

- **AV Foundation Framework–** AVFoundation framework provides an Objective-C interface that work on a detailed level with time-based audiovisual data. For example, use can use it to examine, create, edit, or re encode media files. It can be get input streams from devices and manipulate video during real time capture and playback.

- **AV Kit Framework–** framework gives a collection of easy to use interfaces for presenting video. It create view-level services for media playback, complete with user controls, chapter navigation, and support for subtitles and closed captioning.

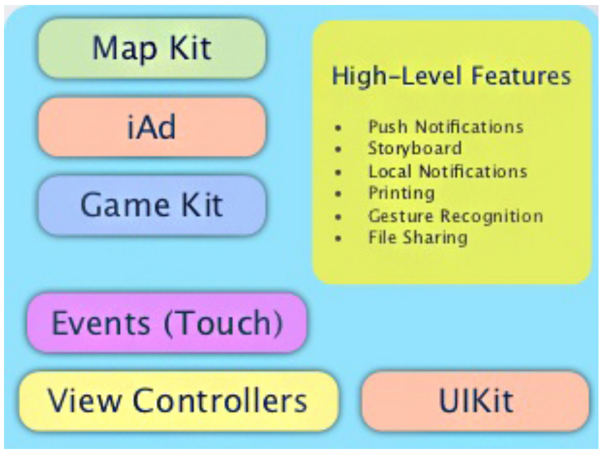
- **Core Media –** The Core Media framework defines the media pipeline used by AVFoundation and other high-level media frameworks found on Apple platforms. Use Core Media's low-level data types and interfaces to efficiently process media samples and manage queues of media data.



Cocoa Touch

The Cocoa Touch layer is the topmost layer of the iOS architecture. It contains some of the key frameworks native iOS applications rely on, with the most prominent being the UIKit framework.

UIKit provides the infrastructure for graphical, event-driven iOS applications. It also takes care of other core aspects that are specific to the iOS platform, such as multitasking, push notifications, and accessibility.



The Cocoa Touch layer provides developers with a large number of high level features, such as Auto Layout, printing, gesture recognizers, and document support. In addition to UIKit, it contains the Map Kit, Event Kit, and Message UI frameworks, among others.

Frameworks

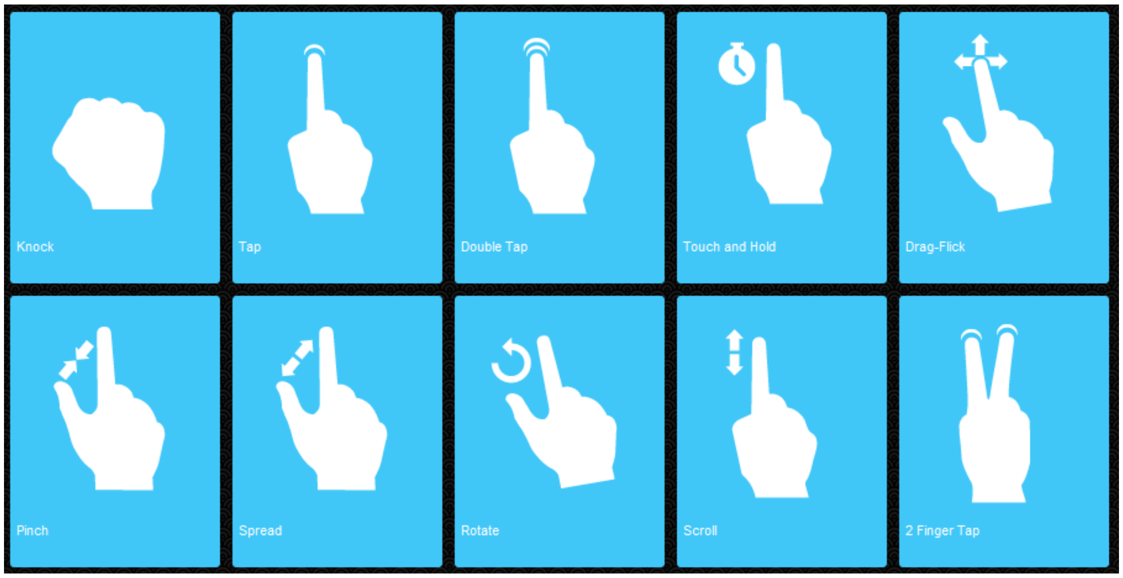
- **EventKitUI framework –** EventKitUI provides view controllers for viewing and editing calendar and reminder information, choosing which calendar to view, and for determining whether to present calendars as read-only or readable and writeable
- **GameKit Framework –** Implements support for Game Center which allows users share their game related information online
- **iAd Framework –** Allows you deliver banner-based advertisements from your app.
- **MapKit Framework –** Gives a scrollable map that you can include into your user interface of app.
- **PushKit Framework –** The PushKit framework supports specialized notifications for updating your watchOS complications, responding to file provider changes, and receiving incoming Voice-over-IP (VoIP) calls.
- **Tweeter Framework -** Supports a UI for generating tweets and support for access the Twitter Services
- **Message UI Framework -** Provides specialized view controllers for presenting standard composition interfaces for email and SMS text messages. Use these interfaces to add message delivery capabilities, without requiring the user to leave your app.
- **AddressBook UI Framework -** provides controllers that facilitate displaying, editing, selecting, and creating records in the Address Book database.

- **Notification center Framework -** Helps you create and manage app extensions that implement Today widgets

- **UIKit Framework –** The UIKit framework provides the required infrastructure for your iOS apps. It provides the window and view architecture for implementing your interface, the event handling infrastructure for delivering Multi-Touch and other types of input to your app.

  Some of the Important functions of UI Kit framework:

  - Basic app management and infrastructure.
  - Multitasking support.
  - User interface management
  - Support for using custom input views that behave like the system keyboard
  - Support for sharing content through email, Twitter, Facebook, and other services
  - Support for Touch and Motion event.
  - Cut, copy and paste support and many more.



Cocoa vs Cocoa Touch

Cocoa and Cocoa Touch are the application development environments for OS X and iOS, respectively. Both Cocoa and Cocoa Touch include the Objective-C runtime and two core frameworks:

- *Cocoa*, which includes the Foundation and AppKit frameworks, is used for developing applications that run on OS X.
- *Cocoa Touch*, which includes Foundation and UIKit frameworks, is used for developing applications that run on iOS.

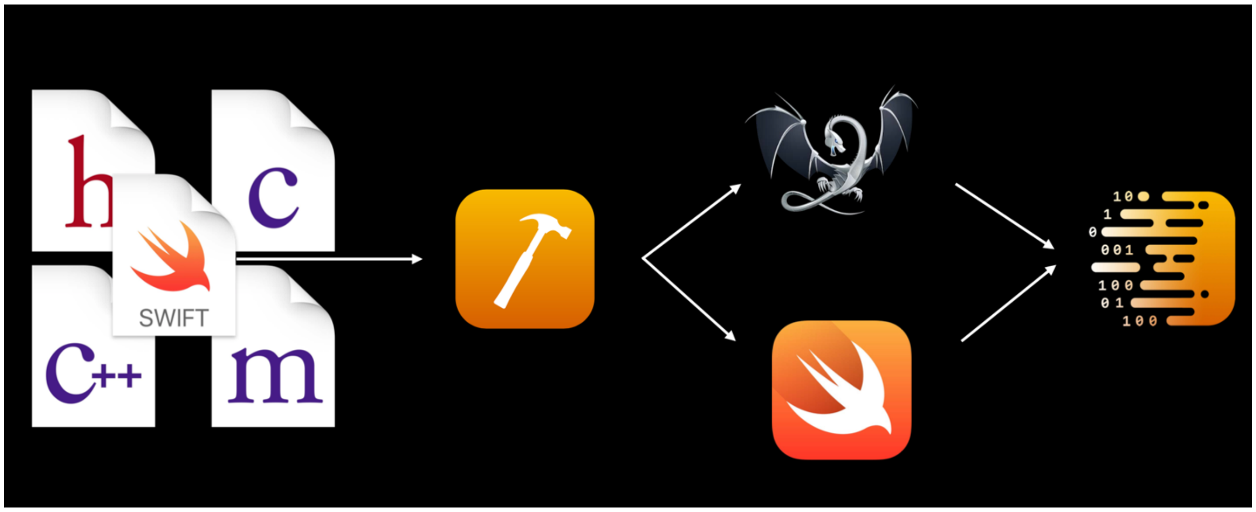
Build System

The Build Process is a collection of work. From the execution order of the build task .The main goal of *Xcode Build System* is to orchestrate execution of various tasks that will eventually produce an executable program.

Xcode runs a number of tools and passes dozens of arguments between them, handles their order of execution, parallelism and much much more.

Xcode uses two different compilers: one for *Swift* and the other for *Objective-C*, *Objective-C++* and *C/C++* files.

- **clang** is Apple's official compiler for the *C* languages family.
- **swiftc** is a *Swift* compiler executable which is used by *Xcode* to compile and run *Swift* source code. Also Swiftc Uses Clang as a Library to Import Objective-C



Here are several ways to build an iOS/macOS project: using XCode, XCodebuild from the command line, fastlane (which uses XCodebuild under the hood), manual scripts or something else. Anyway you need to do the same set of actions to make the app package which can be installed on a device.

```
$ swiftc -module-name YourModule -target arm64-apple-ios12.0 -swift-version 4.2 ...
$ clang -x objective-c -arch arm64 ... YourFile.m -o YourFile.o
$ ld -o YourProject -framework YourFile.o ...
$ actool --app-icon AppIcon ... Assets.xcassets
$ ...
```

Or

```
$ xcodebuild -target YourProject -xcconfig your_configuration_file.xcconfig
```

Code compilation is one of the steps you need to do (together with linking, bundling the resources, code signing and so on).

Originally we have our source code written in some high level programming language: ObjC, Swift, C++ (you also have some resources like images, strings, fonts etc.). Compiler's job is to translate our high level source code into low level machine code to create an executable program. There are different types of compilers but in our iOS/macOS world we deal with ones which can be divided into a frontend and a backend. The frontend takes our source code and converts it into some intermediate language understandable for the backend. The backend takes this intermediate representation and turns it into machine code with the regard of the specific operation system and processor architecture. That's the essence of this two entities.

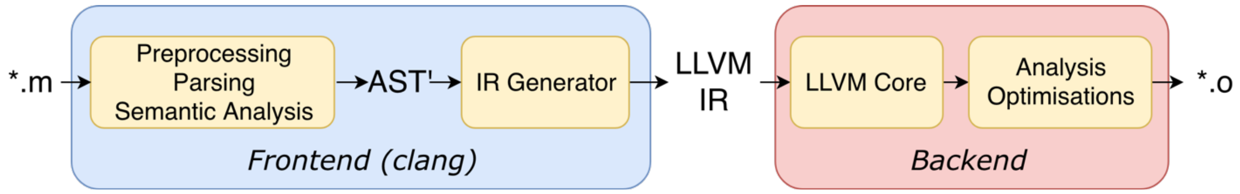
There are different pipelines for Swift and ObjC but both of them are backed by LLVM.

LLVM is a big project including LLVM Core as a backend, Clang as "LLVM native" frontend for C language family (C/ObjC/C++/ObjC++) and interaction protocol between the frontend and the backend. The major part of this protocol is LLVM IR (Intermediate Representation) - the intermediate language understandable for both the front and the back. It doesn't matter for LLVM Core exactly what frontend has produced the IR-code out of the source code. The Core just takes this representation works with it and generates an executable or dynamic library.

Although Clang is sometimes considered as a part of LLVM, the entire idea of the LLVM project is that the frontend is decoupled from the backend. Swift development team (which includes several members of LLVM community) managed to leverage it and use the same LLVM Core as a backend for the swift compiler.

Objective-C

ObjC code the pipeline looks like this:



You can see that the first conversion Clang makes over your code is creation of AST' (Abstract Syntax Tree) - the representation where all the functions, operators, variables, declarations.. are nodes of the huge semantic tree

Then AST is being converted into LLVM IR which is more low level (less human readable) but some part's of the source code still can be understood

LLVM IR is being passed from Clang to LLVM Core where the code is optimized (if applicable) and converted to a target specific machine code. As a result we have a bunch of Mach-O object files(*.o) which are linked together later on and merged into an executable or dynamic library. The output of this last stage is typically called an “a.out”, “.dylib” or “.so” file.

As you could already see LLVM Core is the place where the code is optimized, and Intermediate Representation is specifically the source of these optimizations.

Here are **LLVM optimization levels** with brief corresponding descriptions:

- **None [-O0]** The compiler does not attempt to optimize code. With this setting, the compiler’s goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code. Use this option during development when you are focused on solving logic errors and need a fast compile time. Do not use this option for shipping your executable.
- **Fast [-O, O1]** The compiler performs simple optimizations to boost code performance while minimizing the impact to compile time. This option also uses more memory during compilation.
- **Faster [-O2]** The compiler performs nearly all supported optimizations that do not require a space-time tradeoff. The compiler does not perform loop unrolling or function inlining with this option. This option increases both compilation time and the performance of generated code.
- **Fastest [-O3]** The compiler performs all optimizations in an attempt to improve the speed of the generated code. This option can increase the size of generated code as the compiler performs aggressive inlining of functions. (This option is generally not recommended.)
- **Fastest, Smallest [-Os]** The compiler performs all optimizations that do not typically increase code size. This is the preferred option for shipping code because it gives your executable a smaller memory footprint.
- **Fastest, Aggressive optimization [-Ofast]** This setting enables ‘Fastest’ but also enables aggressive optimizations that may break strict standards compliance but should work well on well-behaved code.

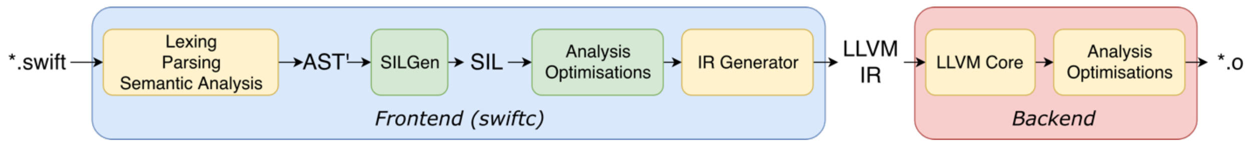
Depending on the level LLVM makes several passes through given IR analysing and changing the code. When creating a new project XCode sets up 2 configurations with following default values for the LLVM optimization levels:

- Debug: **-O0 (none)** - the fastest compile time, the easiest debugging
- Release: **-Os (fastest, smallest)** - the best combination of small binary size and fast runtime execution

Swift

Swift is young and rapidly developing language. So as it's compiler. In terms of code optimization some features are being changed and new features appear almost every year.

Let's take a look at the compilation pipeline for Swift:



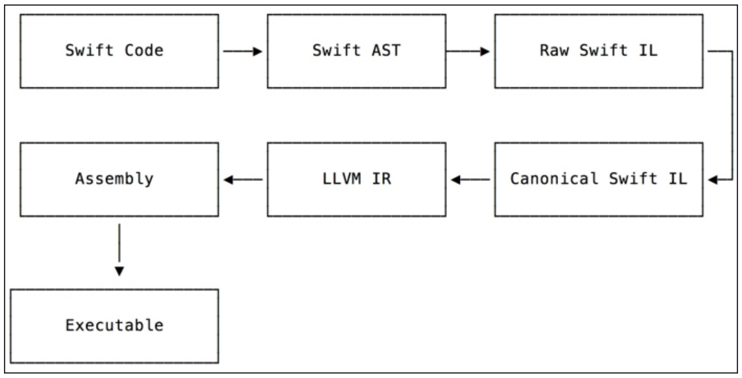
You can see lots of similarities with the one for ObjC: not only the same backend but also the frontend essentially works similarly. Lexical analysis, tokenization (separating some lexical items from the raw string), building AST, type checking. The main difference is the presence of SIL (Swift Intermediate Language) - another intermediate code representation between AST and LLVM IR.

The Swift development team took Clang as an example for a frontend, tried to use all Clang's advantages and compensate some flaws. One of that flaws was inability to implement some high level analysis, reliable diagnostics and optimizations for which neither ATS nor LLVM IR were a proper material. So SIL was the solution for this problem.

So.. what's about optimizations? You might already got it that for swift code we have the same low level LLVM optimizations made over LLVM IR, plus we have some set of additional optimizations made on a higher level by swift compiler over SIL.

At different steps the parts of the Swift compiler produce different kinds of SIL. At the beginning SILGen produces so called “raw SIL”. Raw SIL is just another representation of your code: no diagnostics, no changes, potential data flow errors. A small set of optimisations is being made on the raw SIL

If all diagnostic passes succeed, the final result is the “canonical SIL” for the program. Canonical SIL means that guaranteed optimizations and diagnostics were done, so all the data flow errors should be eliminated and certain instructions must be canonicalized to simpler forms.



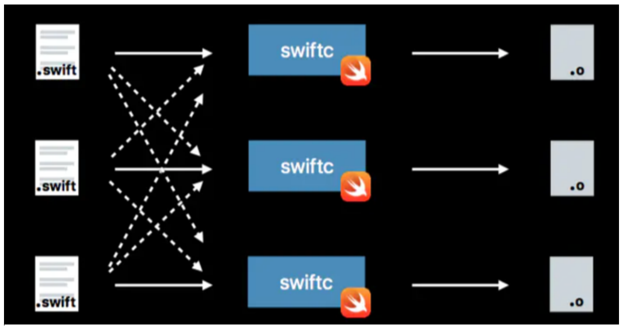
Until Swift 4.1 swift compiler had only two optimization levels: [-Onone] and [-O]. But now there are already 3 different levels:

- **None [-Onone]**
- **Optimize for speed [-O]**
- **Optimize for size [-Osize]**

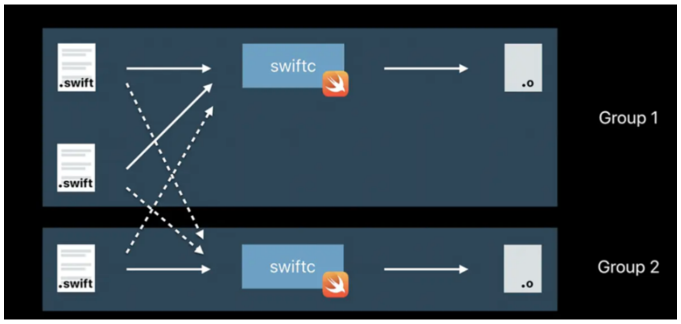
Optimization for speed includes all the General Optimization Passes listed before. Optimization for size (appeared in Swift 4.1) is claimed to have the same optimizations but with the additional parameter - size limitation. Basically the compiler tries to reduce code duplication when optimizing the code. Specifically it tries to be a bit smarter in function inlining (not doing the job if it doesn't decrease a code size). Also the compiler is able to extract some code into a helper functions which also decrease the resulting code size. Optimization for size is an analog of **Fastest, Smallest** optimization made by LLVM. The trade-off between the two optimization level is as following: size optimization can save you 5%-30% of the code size, performance optimization can improve the speed of the code up to 5%. Of course we also shouldn't forget about the Whole-Module Optimization (appeared in Swift 3). Now this property sits in the project settings as Swift compilation mode and has two options:

- Incremental (Single file in XCode 9.3)
- Whole module

The essence of the optimisation is intelligibly described in the official swift blog. But briefly it's not related to how the compiler process the source code, but what exactly it takes as an input for its work. In incremental (single file) mode the compiler works separately with each swift file doing all the analysis and optimisations. In a whole-module mode it "combines" all the swift source files in one module all together and takes the entire module as a source for all the processing. Of course every mode has its trade-offs. Whole-module optimisation has several significant benefits based on the bigger context for the optimizations (whole module instead of just one file) - when the compiler knows more about the code it can do much more for you in terms of optimisation. But it's clear that in this case the compiler needs to recompile the entire module every time you run it, even if you made just a small change in one file (as there are know separate files for the compiler). That's why by default in the new XCode project the debug configuration has an incremental compilation mode (almost no optimizations, the fastest incremental compilation) and the release configuration has whole-module optimisation switched on (no need in benefits of incremental compilation, better optimisations).



Compilation of Swift before Xcode10



Xcode10 compiles Swift

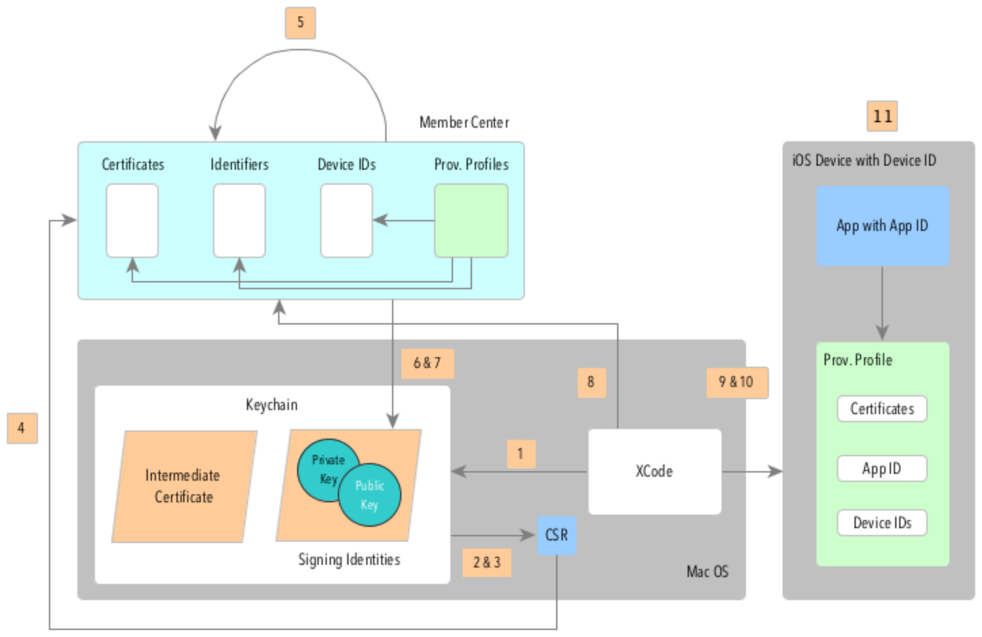
Signing Application

Genuinely, iOS devices only run apps that have been signed by trusted developers unless you have jailbroken iOS devices. If you are an iOS engineer, you must have battled with code signing at some point. As code signing is one of the painful and cumbersome tasks for iOS developers but they can not run away from code signing activities. What are Signing Identities? Why do I need to create Provisioning Profiles and care about such things as a developer?

What is code signing in general. In real life, we sign different contracts, agreements for various things. Why we sign those contracts? what does signature means to us and why it's important? It's simple because signing contracts protect us legally and we have trust that whatever terms and conditions in the contracts cannot be altered whilst contract is in place. The signature also ensures security that contract comes from trusted authorities and not from scammers. It means signing contracts gives us security, safety and trust.

Similarly, code signing is the process of digitally signing any form of the code to confirm that who wrote the code and guarantee that code has not been changed or corrupted when it was signed. The code signing uses a cryptographic hash to verify authenticity and integrity of the software code. In the software world, code signing ensures the identity of the author, integrity of the code, build system and versioning so that users of the software feel secure and safe while using them. Code signing uses various security terms like a public key, private key, certificates, digital signatures etc.

In follow graph you can see a big picture of what happening.



1. **Xcode** will be installed and the **Intermediate Certificate** will be pushed into the Keychain

2. **Certificate Signing Request (CSR)** will be created.
3. **Private Key** will be generated along the **CSR** creation and stored in the Keychain
4. **CSR** will be uploaded to the Member Center
5. Apple will proof everything and issue the ***Certificate***
6. **Certificate** will be downloaded to your Computer
7. The **Certificate** will be pushed into the Keychain and paired with the private key to form the **Code Signing Identity**
8. The **Provisioning Profile** will be created using a *Certificate, App ID and Device Identifiers* and downloaded by **Xcode**
9. **Xcode** will sign the App and push **Provisioning Profiles** onto the Device
10. **iOS** will proof if everything is correctly configured.That means that the **Provisioning Profile** should include the **Certificate** you used to sign the App, your **Device UDID** and the correct **App ID**.
11. Your App should be running now!

In the following, we will explain the details of each section:

Certificate Signing Requests

One thing we need to clear up is the term *Signing*. *Signing* your app allows *iOS* to identify who signed your app and to verify that your app hasn't been modified since you signed it. The **Signing Identity** consists of a **public-private key pair** that Apple creates for you. Think about the public-key as a lock-only mechanism, so you need to know the private key to unwrap, unlock or decode data again.

All this magic happens when you create a **Certificate Signing Request (CSR)** through the Keychain Access Application. If you do so, the Keychain Application will create a **private key** and a **certSigningRequest** file which you'll then upload to Apple. Apple will proof the request and issue a certificate for you. The Certificate will contain the public key that can be downloaded to your system. After you downloaded it you need to put it into your **Keychain Access Application** by double clicking it. It is used by cryptographic functions to generate a unique signature for your application, which is basically your **Code Signing Identity**.

The certificate will also be available through the **Member Center**, but it will only contain the public key, so keep that private key safe. An **intermediate certificate** is also required to be in your keychain to ensure that your *developer* or *distribution certificate* is issued by another certificate authority.

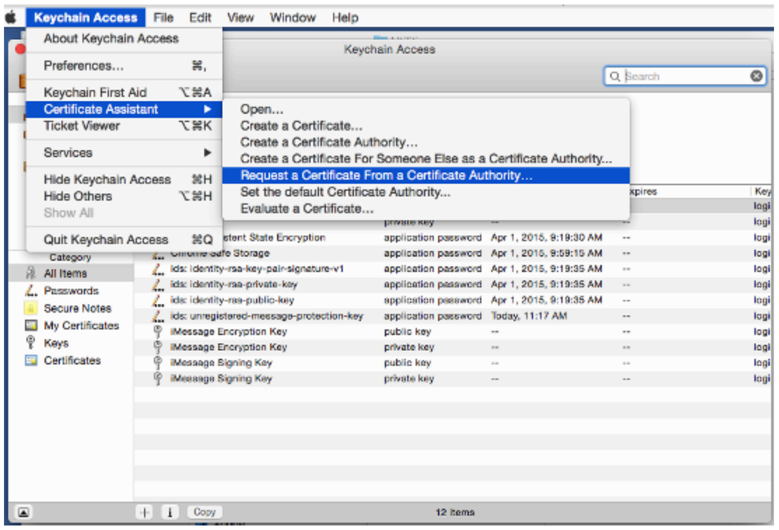
How to create CRS

The Certificate Signing Request (CSR) which is the process of requesting the certificate from Certificate Authority (CA) which is Apple so that Apple can verify the details who is requesting an issue the developer certificate if the details are correct. The requests have to be created from local macOS machine.

CSR from GUI

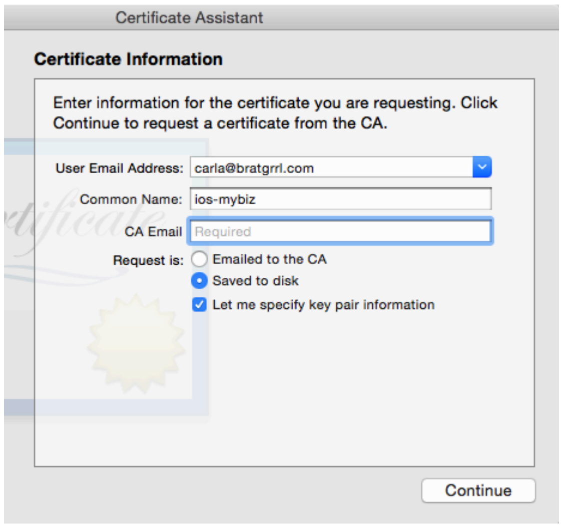
The CSR can be generated using Keychain Access and Apple has some documentation here, but in a summary here is what you have to do

- From Spotlight, Search Keychain Access
- Choose Keychain Access > Certificate Assistant > Request a Certificate From a Certificate Authority.



			xpires	Key
				logi
	private key	--	--	logi
stent State Encryption	application password	Apr 1, 2015, 9:19:30 AM	--	logi
Chrome Safe Storage	application password	Apr 1, 2015, 9:59:15 AM	--	logi
ids: identity-rsa-key-pair-signature-v1	application password	Apr 1, 2015, 9:19:35 AM	--	logi
ids: identity-rsa-private-key	application password	Apr 1, 2015, 9:19:35 AM	--	logi
ids: identity-rsa-public-key	application password	Apr 1, 2015, 9:19:35 AM	--	logi
ids: unregistered-message-protection-key	application password	Today, 11:17 AM	--	logi
iMessage Encryption Key	public key	--	--	logi
iMessage Encryption Key	private key	--	--	logi
iMessage Signing Key	public key	--	--	logi
iMessage Signing Key	private key	--	--	logi

- Fill in your details like email, name and country. Basically, follow all the instructions on the screen.


Certificate Assistant

Certificate Information

Enter information for the certificate you are requesting. Click Continue to request a certificate from the CA.

User Email Address: carla@bratgrrl.com

Common Name: ios-mybiz

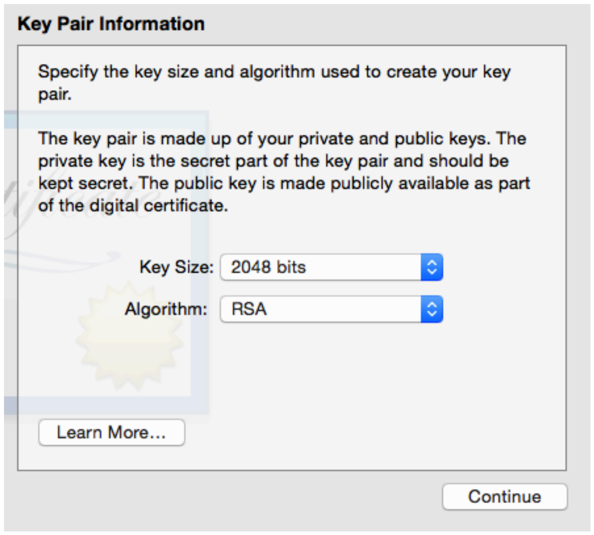
CA Email: Required

Request is: ○ Emailed to the CA [✓ Saved to disk]

[✓] Let me specify key pair information

Continue

- Select the options “Saved to disk”
- Specify a filename and click Save.
- Click Continue and the Certificate Assistant creates a CSR and saves the file to your computer.



Key Pair Information

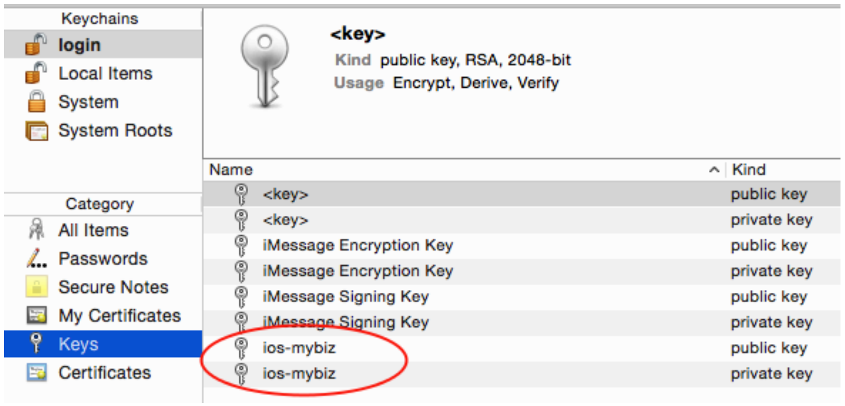
Specify the key size and algorithm used to create your key pair.

The key pair is made up of your private and public keys. The private key is the secret part of the key pair and should be kept secret. The public key is made publicly available as part of the digital certificate.

Key Size: 2048 bits

Algorithm: RSA

- You also get a corresponding public and private key pair, which you can see in the **Login > Keys** section of Keychain.



<key>
Kind public key, RSA, 2048-bit
Usage Encrypt, Derive, Verify

Name	Kind
<key>	public key
<key>	private key
iMessage Encryption Key	public key
iMessage Encryption Key	private key
iMessage Signing Key	public key
iMessage Signing Key	private key
ios-mybiz	public key
ios-mybiz	private key

CSR from Command Line

There is a way to create a CSR from the command line if you want to avoid the hassle of going through GUI. You can get your CSR by running a couple of quick commands. Just fill your name, email address and country.

```
$ openssl genrsa -out mykey.key 2048
$ openssl req -new -key mykey.key -out CertificateSigningRequest.certSigningRequest -su
```

At the end of this process, you should see the file with weird name as *CertificateSigningRequest.certSigningRequest* on your local mac. This is the file which we need to upload to Apple Developer portal while generating certificates for development and distribution. We will cover certificates later in this article.

Now that, we have our CSR on our local machine. Let's see what had happened under the hood when we created CSR. There are few things.

- While creating CSR, the public/private key pair is generated under the hood.
- The public key is attached to your CSR
- The private key is kept inside your local machine.

If you are interested to know more about Public/Private key and how it works in general then there is a term called Asymmetric Cryptography that you can read but in general, the public key is for sharing in public and the private key is private to you. You shouldn't share it with anyone.

Let's find out whats inside the CSR. The key pair generated by has RSA(20148) bit and the public key is attached with CSR. The private key is being used for actual signing so we have to keep it secret. It's not a good idea to generating CSR from multiple macOS as they key pair generated on one mac cannot be present on other mac. Good to use one mac to generate CSR until we create the certificate in P12 format.

Certificates

The certificate is at the core at code signing in iOS apps. Let's forget about Apple Certificates and get into real life. When you finish university or some professional course then you get Certificate. You then attach that certificate to your CV or LinkedIn profile to show that you are trusted by some authority (University) that you have completed that programme and you have knowledge. The employer then gets impressed and offer you the job. Certificates are more powerful when they are offered by popular trusted authorities. If you get the certificate from London Business School, Oxford or Cambridge University then people trust you more. Basically, it's the same concept in iOS or Apple world.

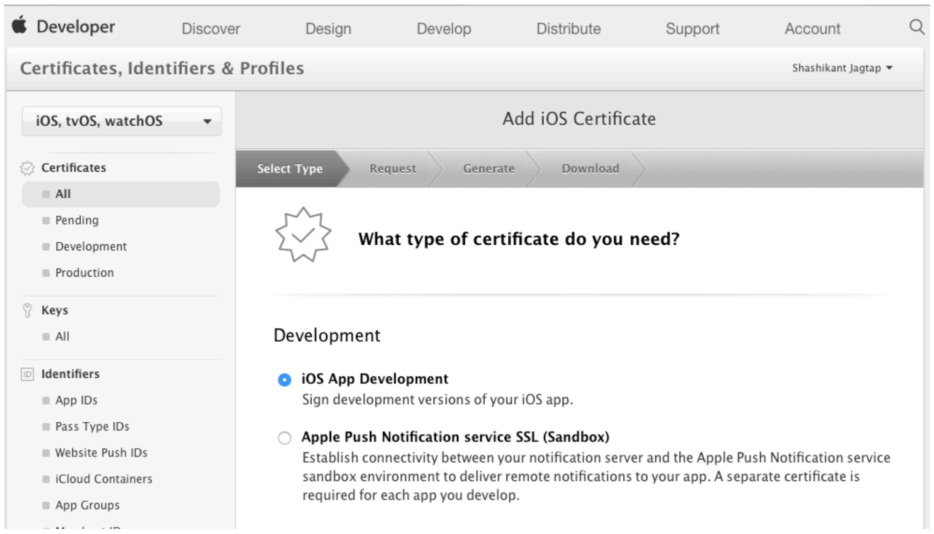
In the iOS world, you get the certificate from trusted authority a.k.a Certificate Authority (CA) which is Apple who certifies that

- You are legitimate (not necessarily good) developer of the app.
- You also paid Apple to get membership Apple Developer or another similar programme.
- Apple has verified that all the information provided by is correct and issued you a certificate to build an awesome iOS apps.

Getting the certificate from Apple isn't a big thing, you just have to pay them enough to get those required certificates. However, understanding the details of certificates and why we need those certificates are definitely the big thing. Apple has ten different types of the certificates as mentioned in the official documentation here. Each certificate has a different purpose but there are two main certificates i.e Development Certificate and Distribution Certificate.

Create Certificates

Let’s go through the process of generating the certificates. In order to generate a new certificate, we have to log in to our developer portal and select the **Certificates and Profile** section. If you click on Certificates section, you will see an option in the form of a + button to create a new certificate. Click on the add new certificate which prompt you to select the type of certificate that you want to create as shown below.



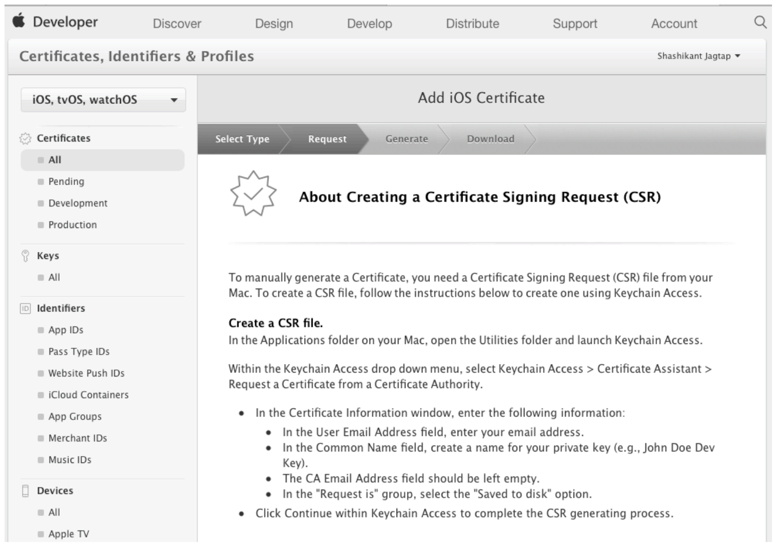
What type of certificate do you need?

Development

iOS App Development [✓]
Sign development versions of your iOS app.

Apple Push Notification service SSL (Sandbox)
Establish connectivity between your notification server and the Apple Push Notification service sandbox environment to deliver remote notifications to your app. A separate certificate is required for each app you develop.

After selecting certificate in this case “iOS App Development”, the next step will be to create CSR which we have seen in the above session. Apple will give all the instructions to generate CSR file locally



About Creating a Certificate Signing Request (CSR)

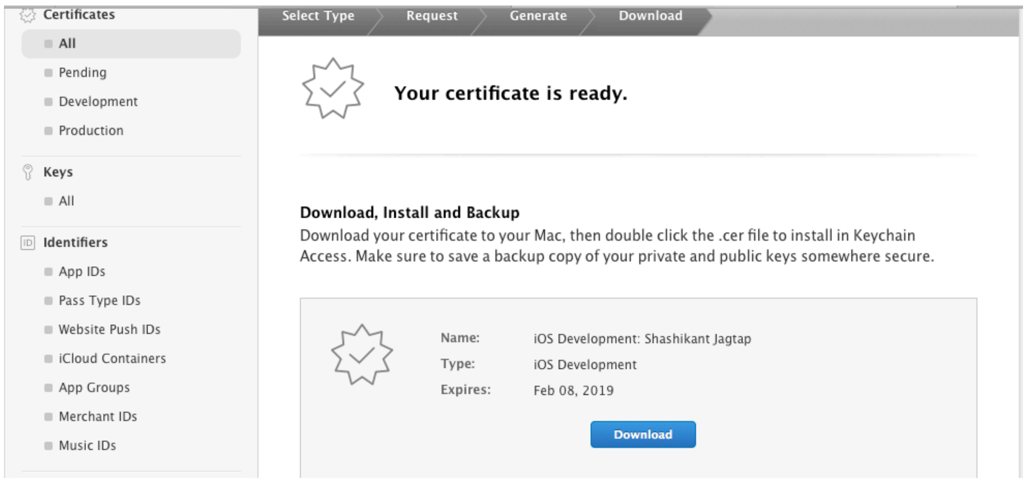
To manually generate a Certificate, you need a Certificate Signing Request (CSR) file from your Mac. To create a CSR file, follow the instructions below to create one using Keychain Access.

Create a CSR file.
In the Applications folder on your Mac, open the Utilities folder and launch Keychain Access.

Within the Keychain Access drop down menu, select Keychain Access > Certificate Assistant > Request a Certificate from a Certificate Authority.

- In the Certificate Information window, enter the following information:
  - In the User Email Address field, enter your email address.
  - In the Common Name field, create a name for your private key (e.g., John Doe Dev Key).
  - The CA Email Address field should be left empty.
  - In the "Request is" group, select the "Saved to disk" option.
- Click Continue within Keychain Access to complete the CSR generating process.

This will ask you save the file CertificateSigningRequest.certSigningRequest on your local machine. The next screen we need to upload this file to developer portal to get the certificate. We can then download the certificate to the local machine.



Certificates
- All
- Pending
- Development
- Production

Keys
- All

Identifiers
- App IDs
- Pass Type IDs
- Website Push IDs
- iCloud Containers
- App Groups
- Merchant IDs
- Music IDs

Select Type › Request › Generate › Download

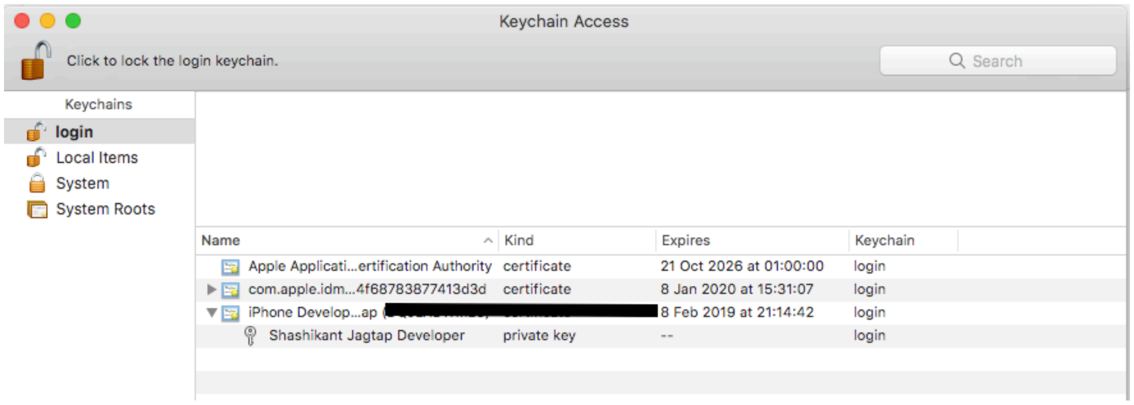
Your certificate is ready.

Download, Install and Backup
Download your certificate to your Mac, then double click the .cer file to install in Keychain Access. Make sure to save a backup copy of your private and public keys somewhere secure.

Name:	iOS Development: Shashikant Jagtap
Type:	iOS Development
Expires:	Feb 08, 2019

Download

At this stage, you will have the file with ***.cer*** extension downloaded to your local machine. You can then double click on the certificate to add it to the keychain. Verify the Keychain has newly generated certificate with private key attached as shown below



Keychain Access

Click to lock the login keychain.

Search

Keychains
- login
- Local Items
- System
- System Roots

Name	Kind	Expires	Keychain
Apple Applicati...ertification Authority	certificate	21 Oct 2026 at 01:00:00	login
▶ com.apple.idm...4f68783877413d3d	certificate	8 Jan 2020 at 15:31:07	login
▼ iPhone Develop...ap [redacted]	[redacted]	8 Feb 2019 at 21:14:42	login
Shashikant Jagtap Developer	private key	--	login

If you see the certificate with the private key in the keychain then you can confirm that you have successfully generated iOS developer certificate. You can also confirm that using the following command.

```
$ security find-identity -v -p codesigning
```

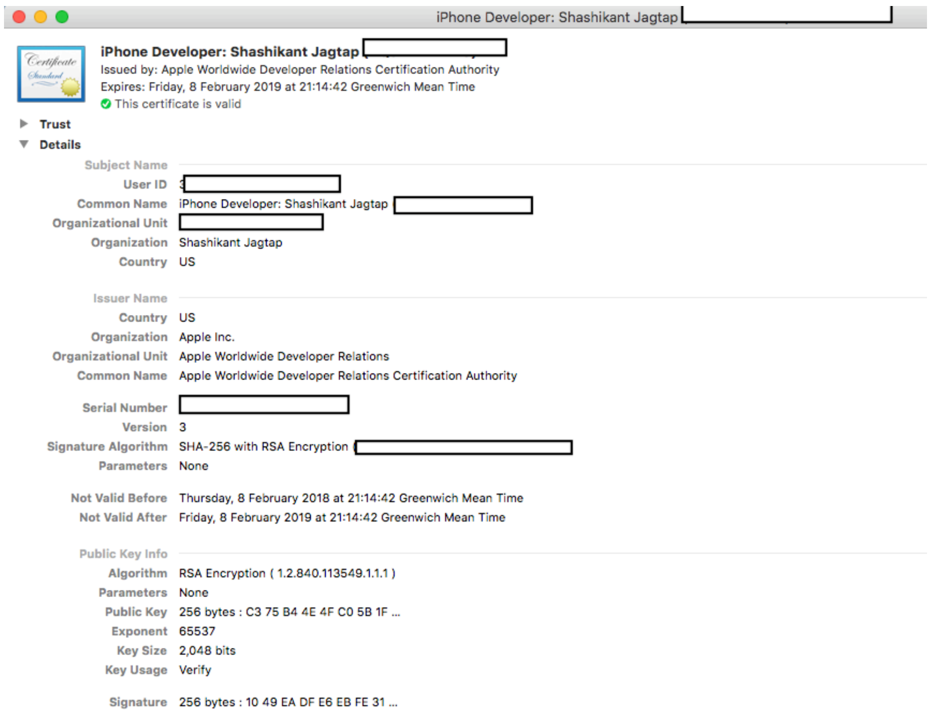
You can generate the same for the distribution as well using a similar process but selecting iOS distribution instead of iOS App development in the certificate type.

Explore Certificate

Now that we have successfully generated the certificate for iOS Development, let's look at the certificate and see whats inside to get more information.

Basically, the certificate has all the data that we have provided during creating certificate signing request, Apple then adds some signer data like Authority, expiry date etc. The whole certificate has the signature attached to it. In the Keychain application if you right click on the certificate and "Get Info" then you will see all this information. It should have following things

- **Subject Name:** This contains UserID, name, organisation details and the country name of the developer.
- **Issuer name:** This contains Apple details like authority, organisation unit that has issued the certificate and expiry date of the certificate.
- **Public Key Info**: This contains details of the public key like which algorithm and signature are used.
- **Fingerprints:** Finally it contains fingerprint details like SHA-256 and SHA-1 etc. The certificate also shows the purpose of what this certificate will be used for like codesigning or any other activity.


iPhone Developer: Shashikant Jagtap [redacted]

iPhone Developer: Shashikant Jagtap [redacted]
Issued by: Apple Worldwide Developer Relations Certification Authority
Expires: Friday, 8 February 2019 at 21:14:42 Greenwich Mean Time
This certificate is valid

Trust
Details

Subject Name	
User ID	[redacted]
Common Name	iPhone Developer: Shashikant Jagtap [redacted]
Organizational Unit	[redacted]
Organization	Shashikant Jagtap
Country	US
Issuer Name	
Country	US
Organization	Apple Inc.
Organizational Unit	Apple Worldwide Developer Relations
Common Name	Apple Worldwide Developer Relations Certification Authority
Serial Number	[redacted]
Version	3
Signature Algorithm	SHA-256 with RSA Encryption [redacted]
Parameters	None
Not Valid Before	Thursday, 8 February 2018 at 21:14:42 Greenwich Mean Time
Not Valid After	Friday, 8 February 2019 at 21:14:42 Greenwich Mean Time
Public Key Info	
Algorithm	RSA Encryption (1.2.840.113549.1.1.1)
Parameters	None
Public Key	256 bytes : C3 75 B4 4E 4F C0 5B 1F ...
Exponent	65537
Key Size	2,048 bits
Key Usage	Verify
Signature	256 bytes : 10 49 EA DF E6 EB FE 31 ...

You can see the certificate details from the command line as well.

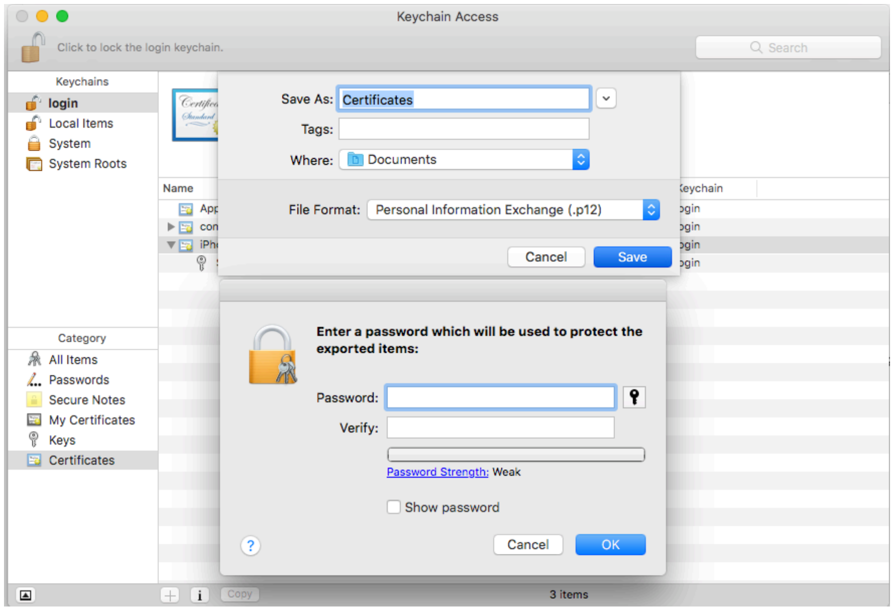
```
$ openssl x509 -in ios_development.cer -inform DER -text -noout
```

Note that, these details are exposing all the information of the certificate itself but we also need private key attached to the certificate to code signing the apps. This is where we need the certificate in the P12 format to back it up with the private key.

P12

In Cryptography, [PKCS#12](#) a.k.a [Public Key Cryptography Standard](#) is the standard used to store private keys with public key certificates protected with the password. This format defines multiple cryptographic objects in the single file.

In our case in the iOS Development certificate, we can export the certificate from keychain into the Personal Information Exchange a.k.a p12 format. Just right-click on the certificate in the Keychain Access and select Export.



Save As: Certificates

Where: Documents

File Format: Personal Information Exchange (.p12)

Enter a password which will be used to protect the exported items:

Password:

Verify:

Password Strength: Weak

You will see that option to export your certificate in the .p12 format also you can encrypt this certificate with a strong password.

You can also create the certificate in the .p12format using the command line. If you have generated the private key using command line while creating CSR as mention above then we can use this key.

```
$ openssl x509 -in XXXXX.cer -inform DER -out mycert.pem -outform PEM
$ openssl pkcs12 -export -inkey my.key -in mycert.pem -out mycert.p12
```

If you provided the passphrase then this command will prompt you to enter the passphrase and you will see your certificate generated in the p12 format. Once, we have generated the certificate in .p12 format, it's easy to backup. There is no risk of losing the private key while using on multiple Macs. The certificate in the p12 format has both your certificate combined with the private key. If you are interested to know whats inside the P12 certificate, you can find the details using the following command.

```
$ openssl pkcs12 -info -in mycert.p12
```

This will prompt you for the password if the p12 is encrypted. The storage container for PKCS#12 called safe bags which stores information as bags with each bag has its own attributes and actual content. In our case, we have two bags one for certificate and other for the private key. You will see bag attribute followed by the actual content of certificate or private keys.

Provisioning Profiles

Historically, provisioning profiles are considered as most painful things in the iOS code signing process.. Basically, provisioning profiles consists of everything from certificates, app ID, device ID and entitlements. The provisioning profiles define the rule for running the app inside the device. Its role is confirm that

- Specific app with App ID
- App with that app ID can run on certain devices that included in provisioning profile. The development provisioning profiles have the list of devices included whilst distribution provisioning profiles don't have the list of devices.
- The app should only have those entitlements defined in the provisioning profile.
- The app can only run trust based on the certificate embedded in the provisioning profile.
-

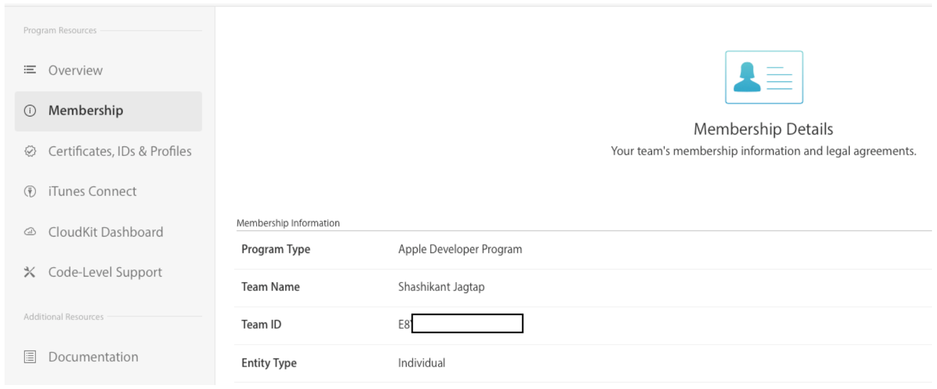
let's understand some terminologies and concepts that we need to know to understand the provisioning profile

- Team ID
- Bundle ID
- App ID
- Device ID
- Entitlements

We will see each of this one by one before diving into the provisioning profile.

Team ID

The team ID is a unique identifier for each development team. You can find this team ID in the Apple Developer portal in the membership session.


Program Resources

- Overview
- Membership
- Certificates, IDs & Profiles
- iTunes Connect
- CloudKit Dashboard
- Code-Level Support

Additional Resources

- Documentation

Membership Details

Your team's membership information and legal agreements.

Membership Information

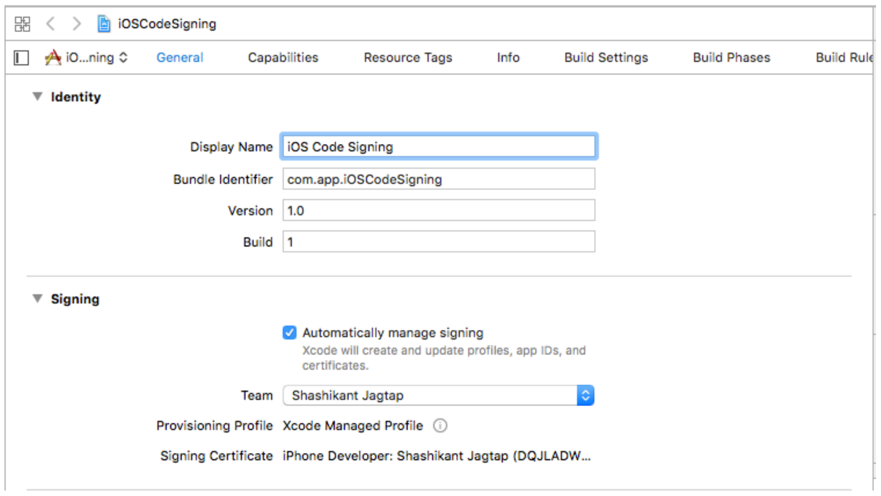
Program Type	Apple Developer Program
Team Name	Shashikant Jagtap
Team ID	E8[redacted]
Entity Type	Individual

You can also find the team ID from your development or distribution certification in the keychain.

```
$ security find-identity -v -p codesigning
```

Bundle Identifier

Every iOS app requires a unique identifier so that app can be identified without ambiguity. Every iOS app must set unique bundle identifier in the reverse domain name of the company or individual. The bundle ID can be set up while creating an app in the iTunes Connect which need to be used in the property list file of the iOS apps. Each target in the iOS app should have a unique identifier. You might have seen the bundle identifier in the format *com.company_name.com* or something like this. In Xcode, we can see the Bundle Identifier in the General tab of the target.



Identity

Display Name	iOS Code Signing
Bundle Identifier	com.app.iOSCodeSigning
Version	1.0
Build	1

Signing

[✓] Automatically manage signing
Xcode will create and update profiles, app IDs, and certificates.

Team	Shashikant Jagtap
Provisioning Profile	Xcode Managed Profile
Signing Certificate	iPhone Developer: Shashikant Jagtap (DQJLADW...

We can also find the bundle identifier of the apps by looking at the ***PRODUCT_BUNDLE_IDENTIFIER*** in the target **Build Settings**.

App Identifier

Generally, App ID is the combination of team ID and the bundle ID. It's very common these days that a team can develop more that one app with different bundle identifiers. The App ID can be used to identify the one or more apps. he Team ID is supplied by Apple, while the bundle ID search string is supplied by you to match either the bundle ID of a single app or a set of bundle IDs for a group of your apps. App ID has two types single App ID for the single app and Wildcard App ID for the set of apps. You can read more about the App ID on the Apple official documentation here.

Device ID

Every iOS has a unique identifier also called UDID. With UDIDs, each iOS device can be found uniquely. Basically, UDID is 40 characters long number made up of the combination of numbers and letters. Like devices, each simulator also has unique UDID. We can find device UDID by attaching the device to Mac and with iTunes connect. There is a great guide here to find out device UDID using iTunes. You can simply get the UDID of the all attached devices and simulators using the following command

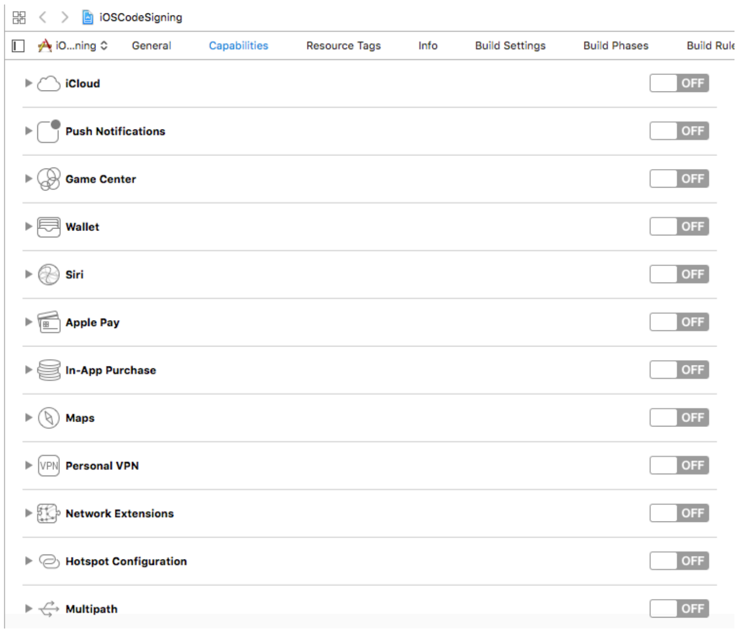
```
$ instruments –s devices
```

You can see the devices with its name, iOS versions and UDID in the square brackets. It looks something like this

```
$ iPhone 6 Plus (11.2) [22B35E0D–C505–4713–8126–80D39A15B34C] (Simulator)
```

You can see the device UDID *22B35E0D-C505-4713-8126-80D39A15B34C* in the brackets for the iPhone 6 Plus simulator. In the same way, you can get it for the attached real devices.

Entitlements and App Sandbox



iCloud OFF
Push Notifications OFF
Game Center OFF
Wallet OFF
Siri OFF
Apple Pay OFF
In-App Purchase OFF
Maps OFF
Personal VPN OFF
Network Extensions OFF
Hotspot Configuration OFF
Multipath OFF

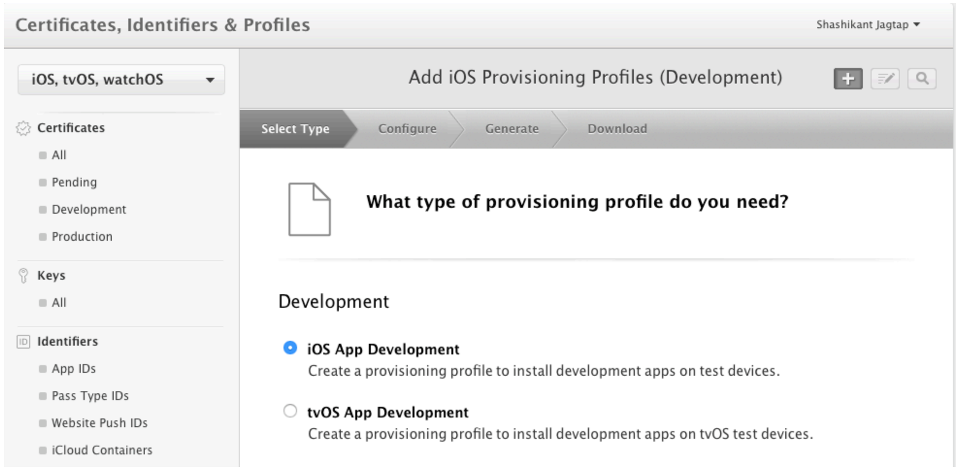
iOS apps can't do everything on its own, we have to explicitly tell the app what app can do or can not do in the form of the entitlements. The app has entitled to do certain things that we need to define in the app sandbox. The app restrictions are managed by the sandbox. The app sandbox is the different infrastructure from the code signing infrastructure, code signing is responsible for running whats inside the sandbox. The app entitlements are configured to specify which resources of the system that app is allowed to use and under what situation. Entitlements also confer capabilities and security of an iOS app. The resources that app allowed to use usually have some default values but they are always disabled. The app developers have to enable them explicitly. The entitlements values that are commonly used are iCloud, push notifications, Apple Pay, and App Sandbox and many more. We can enable the entitlements in the Xcode capabilities tab for a specific target. You should only enable those entitlements that you need.

let's see how to check the apps entitlements and internal representation of the entitlements. Basically, entitlements are the property list generated by Xcode when you enable some capabilities. Xcode will generate .entitlements file depending on the what capabilities you selected. The typical file looks like this

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-
1.0.dtd">
<plist version="1.0">
<dict>
         <key>aps-environment</key>
         <string>development</string>
         <key>com.apple.developer.icloud-container-identifiers</key>
         <array/>
         <key>com.apple.developer.ubiquity-kvstore-identifier</key>
         <string>$(TeamIdentifierPrefix)$(CFBundleIdentifier)</string>
</dict>
</plist>
```

Creating Provisioning Profile

The profile can be easily created from, Certificates and Profile tab of developer portal. Let's add the development profile for the demo app.



Add iOS Provisioning Profiles (Development)

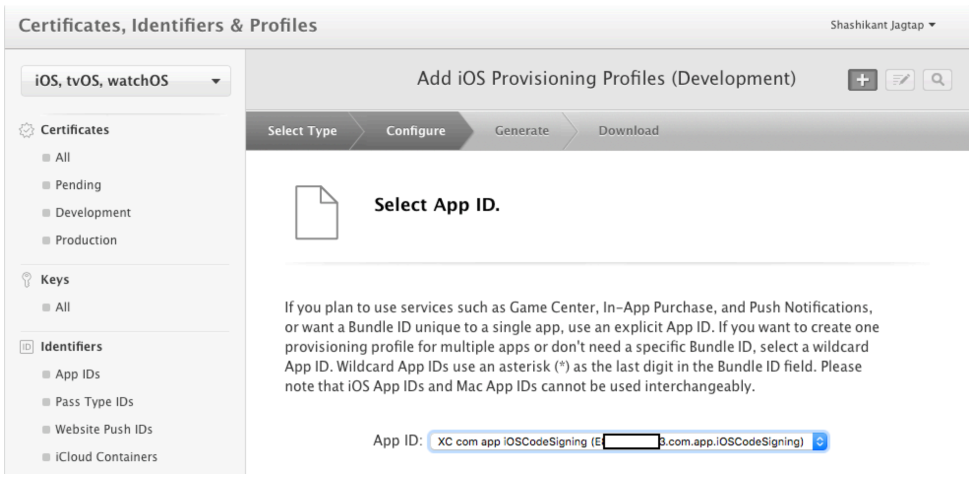
What type of provisioning profile do you need?

Development

● iOS App Development
Create a provisioning profile to install development apps on test devices.

○ tvOS App Development
Create a provisioning profile to install development apps on tvOS test devices.

Note that, we are creating the profile for the development purpose. The next step portal will ask you which app you want to create the profile. We need to select the App ID, in this case, we are using demo iOSCodeSigning app.



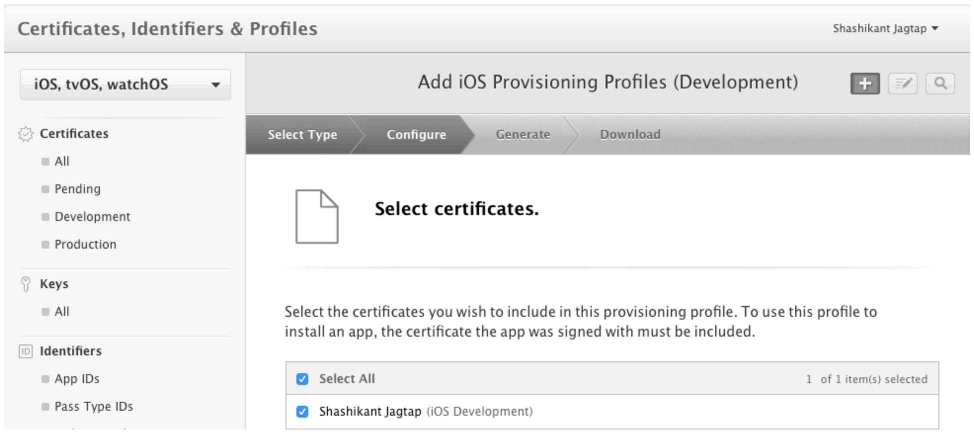
Add iOS Provisioning Profiles (Development)

Select App ID.

If you plan to use services such as Game Center, In-App Purchase, and Push Notifications, or want a Bundle ID unique to a single app, use an explicit App ID. If you want to create one provisioning profile for multiple apps or don't need a specific Bundle ID, select a wildcard App ID. Wildcard App IDs use an asterisk (*) as the last digit in the Bundle ID field. Please note that iOS App IDs and Mac App IDs cannot be used interchangeably.

App ID: XC com app iOSCodeSigning (E[redacted]3.com.app.iOSCodeSigning)

As we are creating the profile for development, we need to select the certificate for the profile as development certificate.



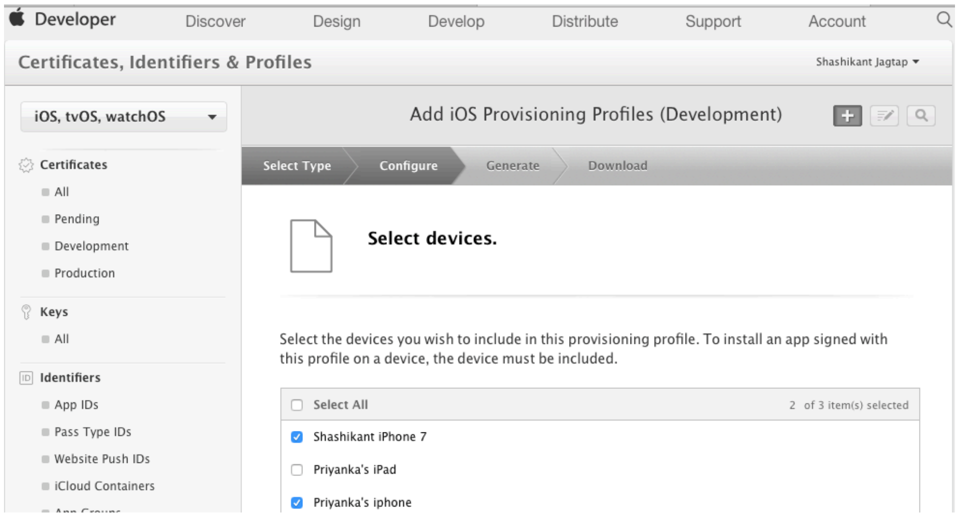
Add iOS Provisioning Profiles (Development)

Select certificates.

Select the certificates you wish to include in this provisioning profile. To use this profile to install an app, the certificate the app was signed with must be included.

✓ Select All	1 of 1 item(s) selected
✓ Shashikant Jagtap (iOS Development)	

Note that, we have provided the information about app ID and certificate for creating the profile. In the step, we need to specify devices for this profile.



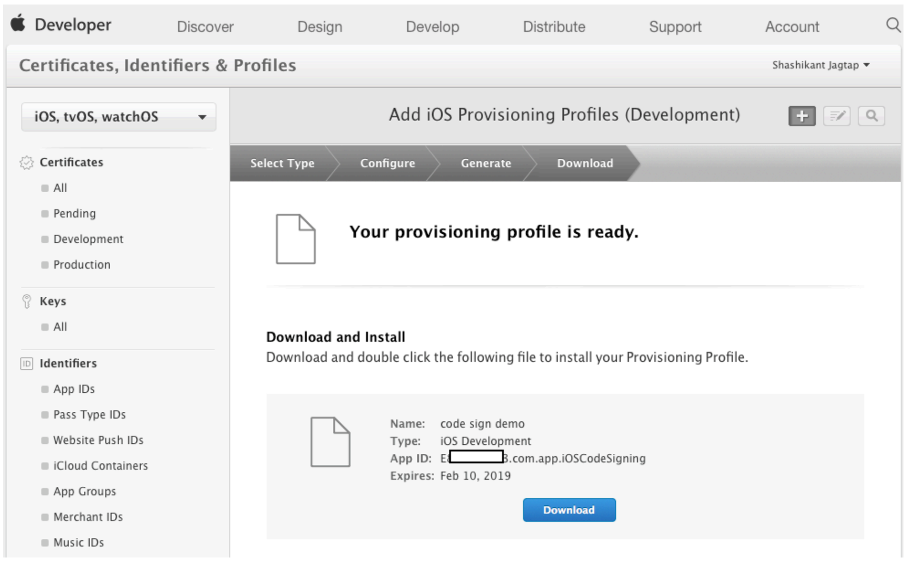
Add iOS Provisioning Profiles (Development)

Select devices.

Select the devices you wish to include in this provisioning profile. To install an app signed with this profile on a device, the device must be included.

☐ Select All	2 of 3 item(s) selected
✓ Shashikant iPhone 7	
☐ Priyanka's iPad	
✓ Priyanka's iphone	

Finally, we have to give a proper name to profile and it will be ready to download.



Add iOS Provisioning Profiles (Development)

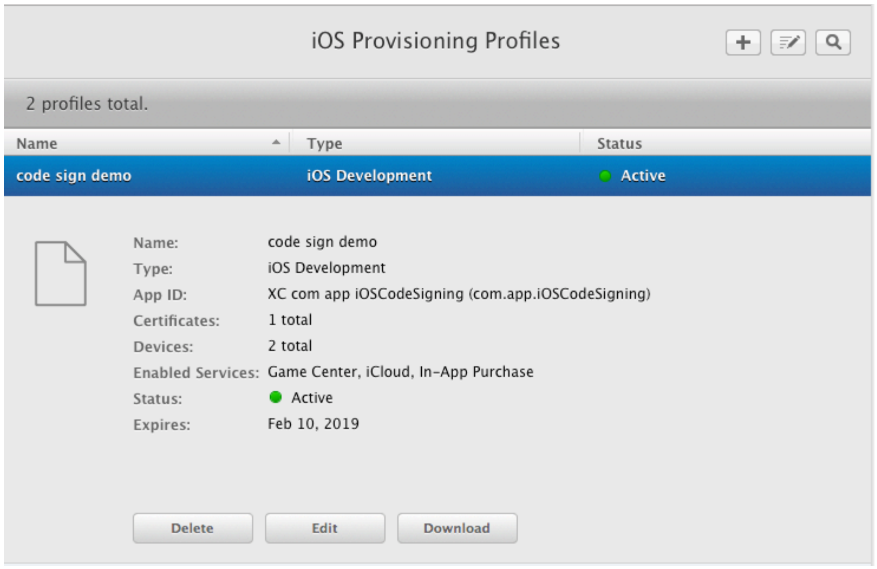
Your provisioning profile is ready.

Download and Install

Download and double click the following file to install your Provisioning Profile.

Name:	code sign demo
Type:	iOS Development
App ID:	E[redacted]3.com.app.iOSCodeSigning
Expires:	Feb 10, 2019

Once clicked on the Download button, it will be downloaded to the ~/Downloads directory. Once doubled clicked on the profile, it will be installed on the local machine. Now that, we have profile downloaded locally which also has the entitlement that we already enabled from Xcode. You can see those entitlements from the enabled services in the Apple developer portal.



iOS Provisioning Profiles

2 profiles total.

Name	Type	Status
code sign demo	iOS Development	● Active

Name:	code sign demo
Type:	iOS Development
App ID:	XC com app iOSCodeSigning (com.app.iOSCodeSigning)
Certificates:	1 total
Devices:	2 total
Enabled Services:	Game Center, iCloud, In-App Purchase
Status:	● Active
Expires:	Feb 10, 2019

Inside Provisioning Profile

The provisioning profile is at the core of the code signing that binds signing, entitlements and app sandbox together. The profiles can be used to enable the app for development, ad-hoc, enterprise and app store distribution.

Once we downloaded the provisioning profile and installed, the profiles are stored in the following directory in the local machine.

```
~/Library/MobileDevices/Provisioning Profiles
```

We can see all the provisioning profiles inside that directory. The profiles have **.mobileprovision** extension but how do we open this file in order to see the contents of the profile. It's not the property list file, its stored in the Cryptographic Message Syntax format. Apple uses this format to ensure that profile shouldn't be changed once it's signed and issued by Apple. We can use security command line tool to explore the contents of the profile. Let's navigate to the profiles directory and look at the contents of the provisioning profile.

```
$ cd ~/Library/MobileDevice/Provisioning\ Profiles/
$ security cms -D -i xxxxxxxx_your_pp_id.mobileprovision
```

This will give all the information about the provisioning profile in the property list format. The property list file contains some main keys including

- App ID Name: This is app ID of the iOS application.
- Creation Date: This is the date when the profile is created.
- Platform: This is usually iOS for the iOS apps.
- Developer Certificates: This is an array of the certificates used to sign the profile.
- Entitlements: This is a dictionary of the array containing all the entitlements of the apps.
- Expiration Date: Expiry date of the provisioning profile.
- Name The name of the provisioning profile.
- Provisioned Devices: This is an array of device UDIDs of the devices registered with the profile.
- Team Identifier: This is team ID of the organisation or development.
- Team Name: This is real name of the team, not the ID
- Version: This is the version of the profile, usual 1 if a profile isn't updated.

As you can see that provisioning profiles have all these details to sign our apps if anything changes we need to modify the profile accordingly.

Signing iOS App

With the latest version of the Xcode, all the code signing activities are handled by Xcode under the hood. We just need to specify the code signing identity and Xcode will manage everything for us. This approach is alright for development and distribution from the local machine because development team can rely on Xcode to handle all these task and apps can be released from the local Xcode machine. Xcode uses codesign command line tool to sign the app one it's built. We can use codesign tool to sign app without Xcode as well. The codesign tool is used to create and verify the code signature

Now let's sign the example app using the codesign tool using the following command:

```
$ codesign -s "iPhone Developer: Shashikant Jagtap (MY_TEAMID)"
~/Library/Developer/Xcode/DerivedData/iOSCodeSigning-
gakpslthucwluoakkipsycrwhnze/Build/Products/Debug-iphonesimulator/iOSCodeSigning.app/
```

This command will sign the components of the demo app. In this case, the app is already signed by Xcode so doesn't need re-signing. Now we will see the what happened under the hood when the app is signed using the code signing identity.

Code signing is macOS security technology and has following main stages while signing iOS apps.

- Seal
- Digital Signature
- Code Requirement

We will see each of these stages in details with practical example whenever applicable.

The Seal

In real life, we seal various objects in order to lock it so that it shouldn't change. The good example is a letter in the sealed envelope guarantee that contents of the letter haven't been changed since sender seals the envelope. A receiver is also sure that letter sent by sender hasn't been modified or altered unless the seal is broken. It means seal ensure the seal ensure an integrity of the content. In the same way, seal in the code signing adds collections of hashes of the various part of the code so that verifier can detect if there are any changes in the code since the code has signed.

The code signing software, in our case, codesign generates seal by running a different part of the code in the app. It applies hashes to each an every part of the code using some sort of hashing algorithm. The hashes applied to the input blocks are unique. A verifier then uses the same hashing algorithm to compare the hashes, if all are same then it satisfies the verification criteria. The small change can result in the corruption of the hash. The hashes are applied to frameworks and all sort of the code. In the property list format. The typical hashing looks like this

```
<key>Frameworks/libswiftos.dylib</key>
<dict>
        <key>hash</key>
        <data>
          dyKltMCMbq+pYDVJBtY78y7BuP0=
        </data>
        <key>hash2</key>
        <data>
          6DxNIVZgqWfOeWfedGQ1+wOnIuA7vQlU+gVA0WhCiRw=
        </data>
</dict>
```

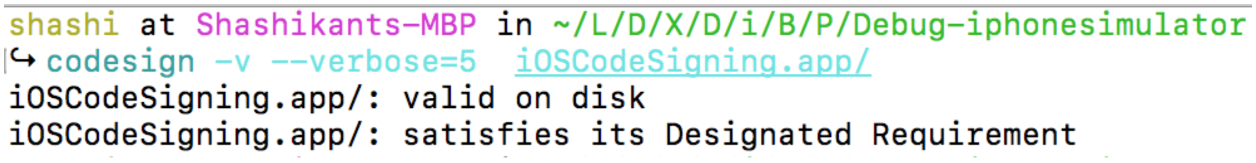
The verifier then uses the same hashing algorithm to verify that data hasn't altered. However, this verification is only as reliable as the reliability of the stored hash. The digital signature guarantees the signing verification.

Digital Signature

The process of digital signature validates the authenticity and integrity of the message. Its based on public key cryptography, also known as asymmetric cryptography. In the code signing process, the software creates a digital signature by encrypting the seal's hashes using signer's private key. The signed hashes stored in the app along with the signer's certificate represent the digital signature. We can then verify the code signature using the codesign tool by running following command

```
$ codesign -v --verbose=5 ~/Library/Developer/Xcode/DerivedData/iOSCodeSigning-
gakpslthucwluoakkipsycrwhnze/Build/Products/Debug-iphonesimulator/iOSCodeSigning.app/
```

This should show an output something like this:


```
shashi at Shashikants-MBP in ~/L/D/X/D/i/B/P/Debug-iphonesimulator
↪ codesign -v --verbose=5  iOSCodeSigning.app/
iOSCodeSigning.app/: valid on disk
iOSCodeSigning.app/: satisfies its Designated Requirement
```

Even the small change in the code will invalidate the hash and report that signature is invalid. The software uses the same set if hashes and signer's public key embedded in the certificate to decrypt the hashes. The verifier then compares the hashes and output the result.

The digital signature of a universal code is stored in the app binary itself while the signature of the frameworks, bundles, tools and other code is stored in the _CodeSignature/CodeResources within the bundle.

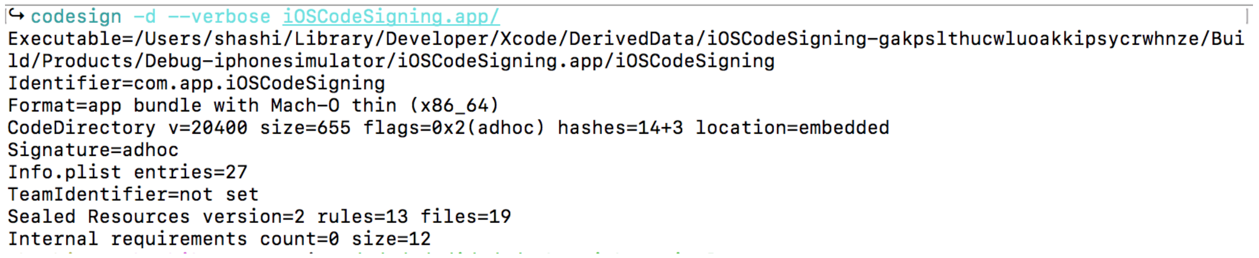
Code Requirements

There are some rules to evaluate the code signature. These rules are known as code requirements. Any app can enforce the code requirement that all the plug-ins used by App should be signed by the Apple. This prevents having the unauthorised third party software installed in the main app. The signer can also specify the code requirement as part of the code signature, these requirements are called as Internal Requirements. The Designated Requirements or DR tells the evaluating system how to identify the particular piece of code. There is detailed documentation of the Code requirement languages on Apple's website here.

Now that, we have briefly covered the three stages of the code signing. Let's print out the detailed information of our code signed demo app using the command.

```
$ codesign -d --verbose iOSCodeSigning.app/
```

This should print the all the codesigning details of our demo app. It looks like this:

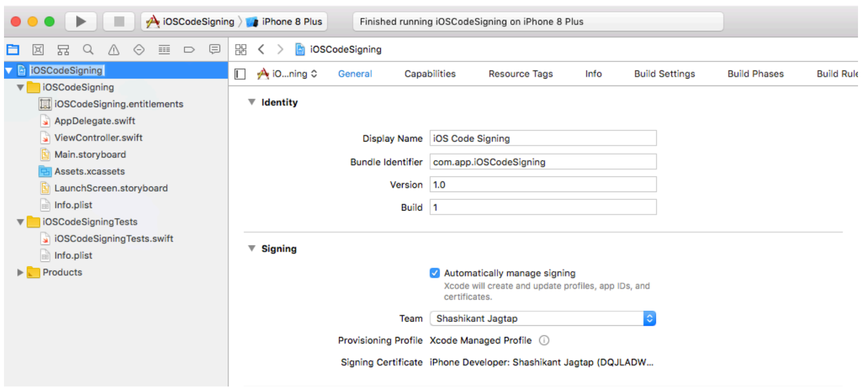


```
↪ codesign -d --verbose iOSCodeSigning.app/
Executable=/Users/shashi/Library/Developer/Xcode/DerivedData/iOSCodeSigning-gakpslthucwluoakkipsycrwhnze/Bui
ld/Products/Debug-iphonesimulator/iOSCodeSigning.app/iOSCodeSigning
Identifier=com.app.iOSCodeSigning
Format=app bundle with Mach-O thin (x86_64)
CodeDirectory v=20400 size=655 flags=0x2(adhoc) hashes=14+3 location=embedded
Signature=adhoc
Info.plist entries=27
TeamIdentifier=not set
Sealed Resources version=2 rules=13 files=19
Internal requirements count=0 size=12
```

As we can see there is a lot of information about the code signed the app. This shows the Executable path, identifier of the app and Signature of the app. As we have used development certificate and debug build, it's showing Format of Mach-O thin binary. There are 14+7 hashes and 13 rules applied. As of now, there are no Internal Requirements specified with code signature.

Automatic Code Signing

Apple announced automatic code signing in Xcode to reduce the code signing pain of iOS developers in Xcode 8 and onwards. In Xcode, we have the new setting in the General tab of build target to enable the automatic code signing. We just need to let Xcode know our development team and Xcode will handle all the code signing tasks like certificates, provisioning profiles etc automatically.



Identity

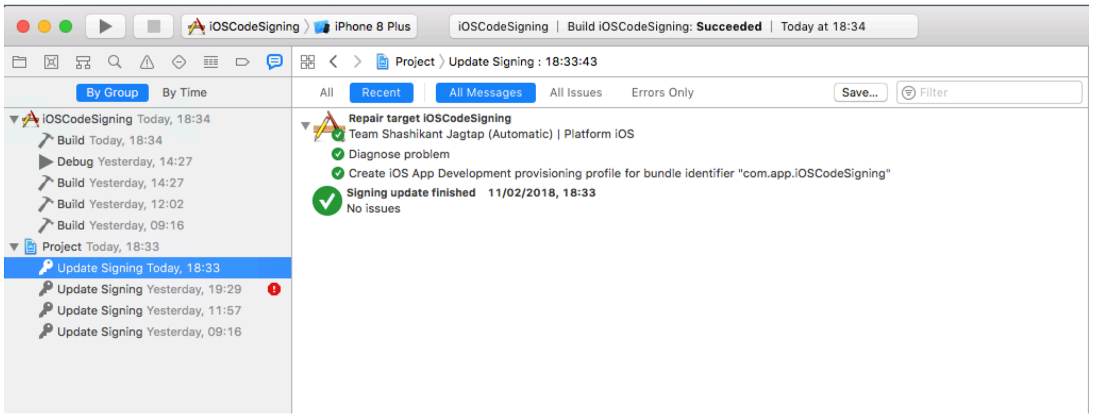
Display Name: iOS Code Signing
Bundle Identifier: com.app.iOSCodeSigning
Version: 1.0
Build: 1

Signing

[✓] Automatically manage signing
Xcode will create and update profiles, app IDs, and certificates.

Team: Shashikant Jagtap
Provisioning Profile: Xcode Managed Profile
Signing Certificate: iPhone Developer: Shashikant Jagtap (DQJLADW...

With the automatic signing, Xcode keeps an eye of any changes in the app setting like entitlement, the new device needs to register with profile etc. Once we give permission to register the device or new capabilities added Xcode creates a new provisioning profile and shows the code signing updates in the reports navigator section.



iOSCodeSigning | Build iOSCodeSigning: **Succeeded** | Today at 18:34

Project 〉 Update Signing : 18:33:43

- iOSCodeSigning Today, 18:34
  - Build Today, 18:34
  - Debug Yesterday, 14:27
  - Build Yesterday, 14:27
  - Build Yesterday, 12:02
  - Build Yesterday, 09:16
- Project Today, 18:33
  - Update Signing Today, 18:33
  - Update Signing Yesterday, 19:29
  - Update Signing Yesterday, 11:57
  - Update Signing Yesterday, 09:16

Repair target iOSCodeSigning
Team Shashikant Jagtap (Automatic) | Platform iOS
Diagnose problem
Create iOS App Development provisioning profile for bundle identifier "com.app.iOSCodeSigning"
Signing update finished 11/02/2018, 18:33
No issues

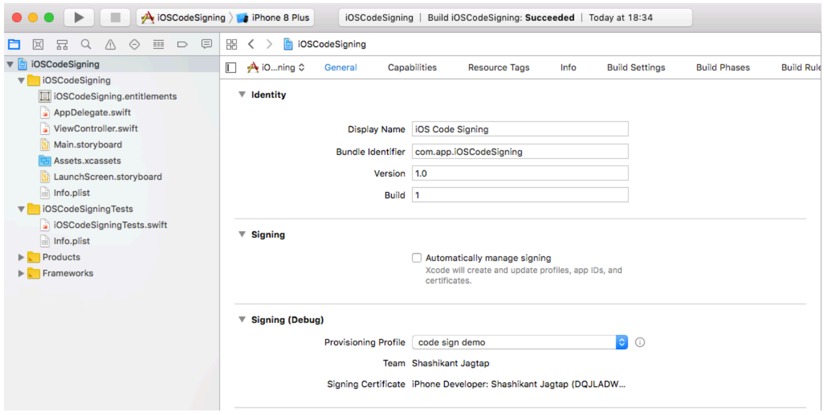
With Xcode 8, there ate two new build settings introduced to manage the code signing process. The build setting *DEVELOPMENT_TEAM* is used to provide greater control over the signing identity, It is mandatory and can not be left blank. The second build setting is *PROVISINING_PROFILE_SPECIFIER* which indicate the type of the signing method. This build setting only applicable for the automatic signing, in the manual signing *PROVISIONING_PROFILE_SPECIFIER* will not be used. We have to use *CODE_SIGN_IDENTITY* as generic entry such as "iPhone Developer"

Other important things to keep in mind about Automatic Code Signing are

- Archive created in Xcode through xcodebuild command line tool is signed with development certificate at first.
- When we specify the distribution method then archive is resigned with distribution code signing.

This means if you are using automatic signing on Continuous Integration servers then you must have development as well as distribution certificate on CI server.

Manual Code Signing



iOSCodeSigning | Build iOSCodeSigning: **Succeeded** | Today at 18:34

- iOSCodeSigning
  - iOSCodeSigning
    - iOSCodeSigning.entitlements
    - AppDelegate.swift
    - ViewController.swift
    - Main.storyboard
    - Assets.xcassets
    - LaunchScreen.storyboard
    - Info.plist
  - iOSCodeSigningTests
    - iOSCodeSigningTests.swift
    - Info.plist
  - Products
  - Frameworks

General | Capabilities | Resource Tags | Info | Build Settings | Build Phases | Build Rules

Identity

Display Name	iOS Code Signing
Bundle Identifier	com.app.iOSCodeSigning
Version	1.0
Build	1

Signing

☐ Automatically manage signing
Xcode will create and update profiles, app IDs, and certificates.

Signing (Debug)

Provisioning Profile	code sign demo
Team	Shashikant Jagtap
Signing Certificate	iPhone Developer: Shashikant Jagtap (DQJLADW...

In the process of manual code signing, we have to explicitly specify the code signing identity and provisioning profile in the build setting. The manual code signing can be enabled in Xcode 8+ by unchecking the "Automatically manage signing" checkbox from the General tab of the target setting. This will give the option to set up the code signing identity and provisioning profile on the General tab rather than going into the build setting.

There is another change in the manual code signing that provisioning profiles are no longer set in the form of UDID, we can set using the name. Again When using manual code signing method, you will have to specify the values for the *CODE_SIGN_IDENTITY* and *PROVISIONING_PROFILE* build settings. Also, we need to have required certificates and provisioning profiles on the local machine from where we are building an app.There is an awesome guide on code signing on Xcode 8, you can read here if you need more information about the manual and automatic signing.

Re-Signing iOS Apps

It's very rare but in some situation, we need to remove the signature of the signed app and re-sign with the new profiles and certificates. This is absolutely possible with codesign tool. In order to re-sign iOS apps, we need to perform following steps.

- Find a signed .ipa file of app that needs resigning
- unzip it and remove code signature

```
$ unzip -q old.ipa
$ rm -rf Payload/*.app/_CodeSignature
```

- Replace old provisioning profile with the new provisioning profile that has been generated for resigning.

```
$ cp new_provisining_profile.mobileprovision Payload/*.app/embedded.mobileprovision
```

- Generate entitlements for the current app using the old entitlements

```
$ cd Payload/
$ codesign -d --entitlements - *.app > entitlements.plist
$ cd ..
$ mv Payload/entitlements.plist entitlements.plist
```

- Force sign the app with new certificate and entitlements

```
$ /usr/bin/codesign -f -s "Your_certificate_in Keychain" '--entitlements'
'entitlements.plist'  Payload/*.app
```

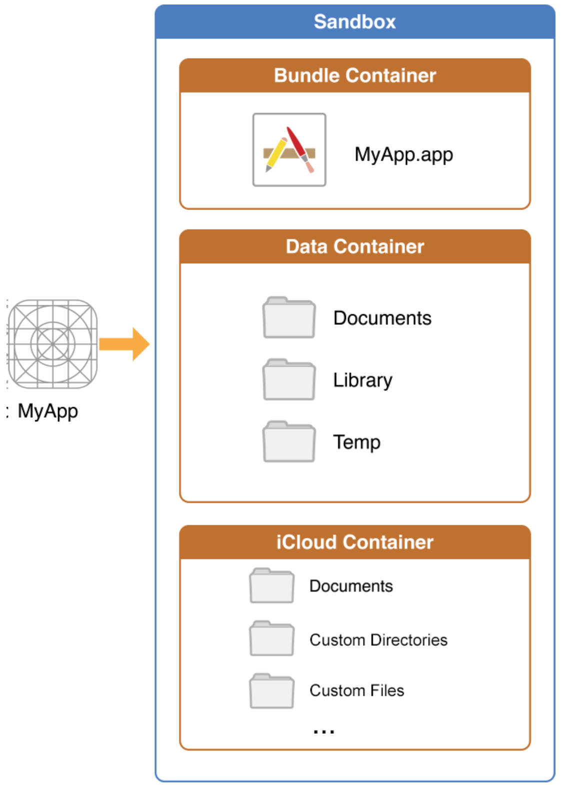
- Zip your resigned ipa file

```
$ zip -qr resigned.ipa Payload
```

At this stage, you will have a binary signed with the new certificate and provisioning profile.

Structure of IPA File

An **.ipa** (iOS **A**pp Store **P**ackage) file is an iOS application archive file which stores an iOS app. They function as containers (like ZIP) for holding the various pieces of data that make up an iPhone, iPad, or iPod touch app.Files with the .ipa extension can be uncompressed by changing the extension to .zip and unzipping. Let us analyse what an IPA file consists of.



The Bundle Container

The Bundle directory or the IPA container consists of all the files that come along with the application when installed from the Apple's App Store or any other source. So, the files in this directory will remain same throughout in a particular version of an application.

All the possible components of the Bundle Directory of a native iOS application are explained below:

Resource Name	Resource Type	Description
iTunesMetadata.plist	File	This file is used to provide information to iTunes about an iOS application using Ad Hoc distribution for either testing or Enterprise deployment.
iTunesArtWork	File	This file contains the image that is used to represent the application in iTunes
META-INF	Directory	It contains two files which hold the metadata about the IPA file.
.app Directory	Directory	This directory holds all the components of the IPA file.
Application Binary	File	This file contains the application's executable code. It's name is same as that of the name of .app Directory excluding the extension ".app".
Info.plist	File	This file is the manifest of the iOS application. It contains information about - Supported Devices, Bundle ID, Display Name, Application Transport Security etc.
Application Icon	File	These are the icon files of the application. There are multiple icon files (Icon.png, Icon@2x.png) for

		representation of the application on devices with different resolution like iPhone, iPad etc.
Launch Images	File	These files (Default.png, Default-portrait.png) are used as launch screen images before the application launches. They are removed as soon as the application is ready to display the UI.
Storyboard/nib files	File / Directory	These files contain the encrypted information about the storyboard of an application.
Settings.bundle	File	This file contains the information that can be tweaked for the application by using the menu in the settings of the device.
Subdirectories for localized resources	Directory	These are the language-specific project directories (en.lproj, fr.lproj etc.) that help in switching the language of the application from the available languages as desired by the user of the application.
Non Localized resource files	File	These files include things like images, sound files, movies, and custom data files that the application uses. All these files must be placed within the .app directory.
_CodeSignature	Directory	This directory contains the file CodeResources which is used to store the signature of all files in the bundle that are signed.
embedded.mobileprov ision	File	This plist file contains the provisioning profile for an application. In simple words, provisioning profile acts as a link between the device and the developer account. This file contains information such as application creation date, keychain access groups etc.
Frameworks	Directory	This directory holds the information about the frameworks that an application uses.
SC_Info	Directory	This directory contains keys which are used for decrypting the app executables. The files in this directory help in integrity checks of the application
Assets.car	File	This file contains images which the app might use to display once opened. The images are stored in an optimized format, and be only taken out using specialized tools such as the Asset Catalog Tinkerer.
PkgInfo	File	This file is an alternate way to specify the type and creator codes of the application or bundle. This file is not required, but is sometimes used to improve performance for code that accesses this information.

The Data Container

The “Data” directory or the Local Data Storage container consists of the files that the developer wishes to store for the application during the time which the application is installed on the device.

The files may be used for caching information for quick access or storing offline information as a backup for resuming the application from the point at intended by the developer. So, the files in this directory and also the information in the files will keep on changing while the application is in use as coded by the developers.

Some of the possible components of the “Data” directory of an iOS application are explained below:

Parent Directory	Component Name	Description
Documents	NA	This directory is used to store the data that the user needs to access from the application and the files such as pdfs downloaded at the runtime. This may also be used to store crash logs that the user gets access to at the time of giving a feedback
Library	Application Support	This directory is used to store all app data files except those associated with the user's documents. Sometimes it may also be used to store a modifiable copy of resources contained initially in the app's bundle. The contents of this directory are backed up by iTunes and iCloud
	Caches	This directory is used to write any app-specific support files that the application can recreate easily. The data in this directory is mostly the cache for the analytics that can be sent when required and also the server's responses for delivering quick responses to the user's queries. This directory also stores the screenshot of the application in the Snapshots directory when it moves to the background in order to improve user experience.
	Preferences	This directory contains app-specific preference files. The main file in this directory is the file named <Bundle_ID>.plist which is used by the developers to store information using NSUserDefaults class.
tmp		This directory usually contains the backup data for an application or some other data that is intended to be deleted after a process gets over. The NSTemporaryDirectory function helps the developers write temporary data to this directory.

Storekit		This directory is important only for business perspective. It provides the access to the following: ● **In-App Purchase -** Offers and promotes in-app purchases for content and services. ● **Apple Music -** Checks a user's Apple Music apabilities and offers a subscription. ● **Recommendations and reviews -** Provide recommendations for third-party content and enable users to rate and review your app.

The iCloud Container

This directory contains data that iCloud enabled iOS applications use. The files in this directory are meant to stored and updated by the sources where a user decides to update the file from.

It consists of usually two parts

- **Documents-** The files in this directory are meant to be read and updated directly by the user. These files are backed up to iCloud regularly to keep in sync.
- **Data-** These files are not meant to be edited or added directly by the user. Data may be kept in different directories as desired by the developer.