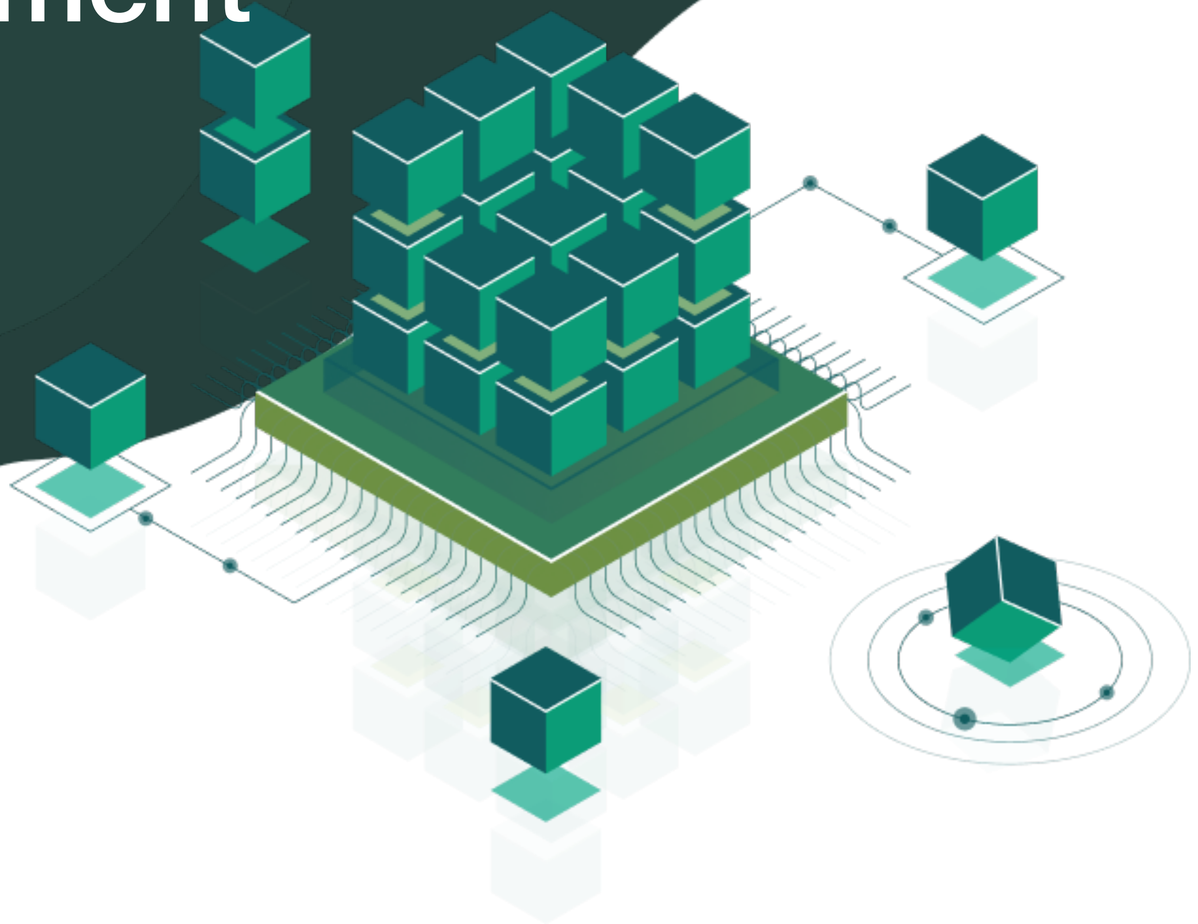
Dependency Management



Common file types

- MyProject/**MyProject**.
  - **.m** and **.h** source files of Objective-C
  - **.swift** source files of Swift
  - **Info.plist**, a file with some overall settings of the project/app.
  - **Strings Files** (**.strings**) localized strings for a particular language.
  - **Nib files**, **Xib files**, or **Storyboard files**, these set up user interfaces.
  - **.xcassets** manages the varying sizes of icons, launch screens, and other images required to support multiple iOS devices
- *MyProject* **Tests**: Contains sources for automated tests to be run on the app.
- *MyProject***.xcodeproj**: Xcode project file itself, actually a subdirectory.

Ways to make dependency

- Manual import of binaries
- Create workspace and combine projects
- Use CocoaPods
- Use Carthage

CocoaPods

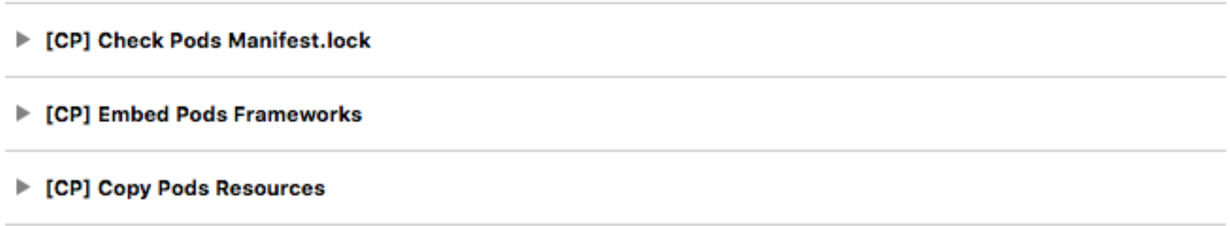
This is mostly used dependency manage so far for the iOS project which is the de facto standard tool.

CocoaPods can be initialized with pod init command which will create template Podfile but we can create our own simple Podfile will look like this:

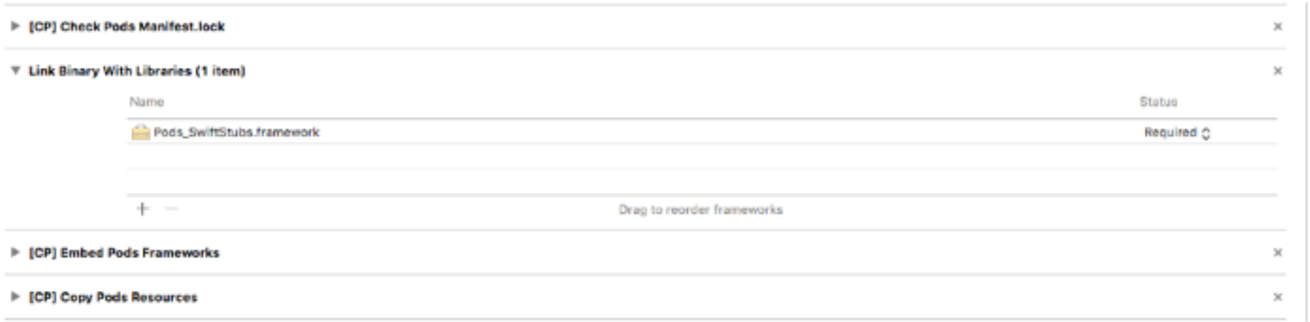
```
platform :ios, '8.0'
use_frameworks!

target 'MyApp' do
  pod 'SwiftyJSON', '~> 2.3'
end
```

What Changes CocoaPods Makes?

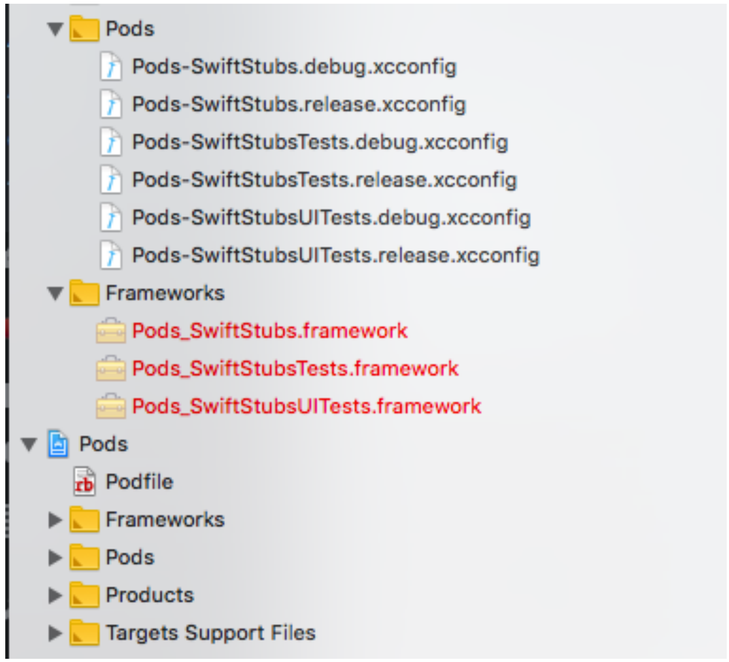
- CocoaPods creates Xcode Workspace directory, which has .xcworkspace extension where it builds all the dependency.
- CocoaPods adds some scripts in the build phases of our target. There are usually three scripts added to the build phases of the target



- Adds Podfile.lock (Locked versions of libraries)
- Links the Pods Frameworks to “Link Binaries with Libraries”



- A Pods directory containing source code of the Pod dependencies, supporting files, xcconfigfiles



- Lots of things inside your Xcode build Settings. It’s hard to cover everything here.

All these things mentioned above any more happens under the hood when you run the magical pod install command.

CocoaPods Build Process

- CocoaPods will build and compile our frameworks every time whenever you are doing the clean build or run pod install or pod update for the project.
- Xcode then checks if Podfile.lock has changed if that changed then Xcode will build the dependency framework again otherwise uses the pre-built frameworks.
- You might have seen that project taking a long time when you do the clean build or delete derived data when using CocoaPods.
- CocoaPods will build all the libraries mentioned in the Podfile for that target, there might be some cases where you don’t want to build some libraries all the time but CocoaPods won’t have that option.

Benefits of Cocoapods

- CocoaPods is easy to setup
- CocoaPods automate the entire process of building, linking the dependency to targets.
- The CocoaPods community has done an impressive job to fix Apple’s shortcomings
- CocoaPods supports libraries having subspec.
- Lots of contributors with well-grown community
- Works well for the small projects with less code to get something working very quickly.

Downsides of CocoaPods

- It requires knowledge of another programming language i.e Ruby on which CocoaPods is built.
- Hard to integrate when already using Xcode workspace.
- Hard to remove once integrated.
- CocoaPods takes controls of entire Xcode project and if something fails entire project stops building. The fixing the errors thrown by CocoaPods are the hard and time-consuming task and requires an understanding of what CocoaPods changed inside the iOS project.
- Pods force your project into a specific structure. CocoaPods updating Xcode Projects and Files is like magic without understanding what’s changed. It works in a black box way.

Downsides of CocoaPods

- Integrating with Continuous Integration Server is hard as we have to install and manage Ruby libraries. All the dependency needs to be installed and built for every build. Or we have to check in the entire third-party dependencies inside the project. Caching mechanism doesn’t always give clean results.
- The building of iOS app became an intransparent and slow process.
- Building libraries and frameworks that support CocoaPods became such a pain for iOS developers who are not skilled in Ruby. The developer needs to write .podspec file and follow many unrelated Ruby conventions.
- Centralized
- Can’t work with framework and project at the same time because of the two-step process for working on dependencies.

Carthage

Carthage is another simple dependency manager for the Cocoa application. Carthage has been written purely in Swift to manage iOS dependencies without changing anything inside your Xcode projects

For using Carthage we have to create file called Cartfile with the following example content:

```
github "SwiftyJSON/SwiftyJSON"
```

What Changes Carthage Makes?

Carthage won't touch Xcode settings or Project files.

Carthage is very simple and just check out and build the dependencies and leave it to you to add the binaries to Xcode. It gives you full control of what we are adding to Xcode.

Carthage Build Process

- Carthage retrieves the libraries and framework using carthage update command which happens only once.
- Xcode will not rebuild any framework when building the project. This speeds up the build process.
- The framework needs to rebuild when we get the new version of any dependency.

Benefits of Carthage

- Carthage is decentralized, simple and ruthless dependency management for iOS
- Carthage is purely written in Swift so iOS developers can understand the technology behind Carthage and probably write another tool using CarthageKit
- Carthage is easy to integrate and easy to remove from the project if it doesn't suit project needs.
- Carthage won't touch your Xcode settings or project files. It just downloads and builds the dependencies so you have proper control on what you are doing.
- Carthage works great for the large or eclectic codebases because of its flexibility
- Building and updating lib(s) are easier with Carthage.
- Carthage can be easily integrated with Continuous Integration server.
- Making Swift libraries Carthage compatible is easy.
- Decentralized
- Supports submodules

Downsides of Carthage

- Carthage has too many manual steps that need to be performed while setup.
- Carthage is still new framework and not many contributors as CocoaPods
- Carthage only works with dynamic frameworks. It doesn’t work with static libraries.
- Carthage has no clean way to support subspec within libraries.
- Traditional iOS developers who came from Objective-C backgrounds feel reluctant to use Carthage.
- Carthage checkout and builds the frameworks, we might need to check-in those frameworks for faster builds which adds repository size.
- Some users reported Carthage is slow while building frameworks.
- Carthage installed with homebrew is not backward compatible.
- Not all the frameworks are available for Carthage especially older libraries that need to be added using git submodules.