

Implement continuous integration and delivery in your iOS projects



# Pro iOS Continuous Integration

Romain Pouclet

Apress®

*For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.*



**Apress®**

---

# Contents at a Glance

<b>About the Author .....</b>	<b>xiii</b>
<b>About the Technical Reviewers .....</b>	<b>xv</b>
<b>Acknowledgments .....</b>	<b>xvii</b>
<b>Introduction .....</b>	<b>xix</b>
<b>■ Chapter 1: Introduction to Continuous Integration .....</b>	<b>1</b>
<b>■ Chapter 2: Continuous integration features in iOS and Xcode.....</b>	<b>5</b>
<b>■ Chapter 3: Using Xcode to Release an Application Outside the App Store .....</b>	<b>29</b>
<b>■ Chapter 4: Invoking the Power of the Command Line .....</b>	<b>41</b>
<b>■ Chapter 5: Automatic Builds with Jenkins .....</b>	<b>67</b>
<b>■ Chapter 6: Automated builds with Bamboo.....</b>	<b>93</b>
<b>■ Chapter 7: Over-The-Air distribution .....</b>	<b>117</b>
<b>■ Chapter 8: Day-to-day use of Xcode bots .....</b>	<b>141</b>
<b>■ Chapter 9: Adding Unit testing to the mix .....</b>	<b>161</b>
<b>■ Chapter 10: Quality assurance .....</b>	<b>183</b>
<b>Index.....</b>	<b>197</b>

---

# Introduction

There are as many definitions of Continuous integration as there is programming language out there, probably more. All of these opinionated definitions have been discussed all over the Internet: what it is, how it can help, which tools to use... Only a few of them are complete, exhaustive guides about where to start and where to go, let alone in the iOS ecosystem. In all fairness, the iOS platform is still relatively new.

Because you have to learn how to walk before you can run, this book will give you a tour of the iOS ecosystem from a developer point of view. If you're interested in reading this book, you probably already have spent a few hours in front of Xcode, but do you know that it will make your life a lot easier with automated testing and versioning? Did you know that you could easily use multiple versions of Xcode? Even more importantly, do you know how it works under the hood? That's the kind of thing this book will teach you.

Continuous integration is a matter of deciding of the best workflow and choosing the right tools. We didn't want to sound as if you should take our opinion as a work of gospel, so we chose to give you all the knowledge you need before covering two different continuous integration platforms, Jenkins and Bamboo. We'll show you how to get started as quickly as possible and how to take a sample application, build it, test it, and finally release it to your testers. Once we'll have covered how these third party tools, we'll cover the OS X Server and the Xcode, which are official tools provided by Apple. Far from being perfect, they provide enough neat advantages to give all those non-official alternatives a scare.

For all those solutions, we'll show you how to integrate tools to run your automated tests and analyze your code, making sure you're spending all this time to automate the workflow of an application that is well coded and works has expected.

In the end, it will be up to you to decide which tool suits you and your company best.

# Introduction to Continuous Integration

In the world of software engineering, the more time that passes the bigger your software usually becomes. It may be because your application is a huge, full-featured tool that does a lot of things or it may be simply because your code has become more and more complex. At this point, you have to start making decisions and choosing the tools you will use and the principles you will follow to make your life easier. The sole purpose of continuous integration is exactly that - making your life easier.

You may be reading this book because you're an independent software engineer or leading a team of iOS developers. We will help you find the tools you need and teach you how to make them fit in your day-to-day workflow.

## What is Continuous Integration?

Continuous integration is a software engineering principle with a very self-explanatory name. During the life cycle of a project, you will be continuously integrating small pieces of software into your project. Then, as the project grows, you'll need to make sure these pieces actually fit the way they are supposed to. It may sound like a very metaphoric way of saying you will merge branches using your favorite version control system, but that's actually what happens - no matter which tool you will be using or how often you will be integrating code.

The principle of continuous integration is never far from the concept of "automated testing" and "Software Quality Assurance" (SQA). Indeed, integrating pieces of software is one thing, but making sure they fit - meaning they don't break your software, respect your coding conventions, or hurt the final product in any way - is a whole other thing.

## Apple History 101

At the time of writing this book, Apple has just released the final version of Xcode 5.1. This 5.0 version was a pretty big release for Apple. It arrived at the same time that iOS 7 did, which was a whole new operating system with a completely new paradigm. What is even more interesting for us is that Apple finally stepped up in the world of continuous integration by starting to promote new tools. We will try to cover as many of these tools as we can in the following chapters.

Of course, the community of iOS developers didn't wait for Apple to finally make a move, and the principle of continuous integration had been advocated for way before the first public release of the iOS SDK. In the case of Cruise Control, Jenkins, Apache Continuum, etc., it was mostly a matter of adapting existing tools to new technologies.

The first version of Xcode was released in 2003. More than ten years later, Apple has finally started giving us officially supported tools such as the Xcode Bots. Needless to say, we have come a long way!

## Pros & Cons

The advantages of continuous integration are numerous. The most important is the immediate feedback you get when working on a part of your application. It may be because you broke your test suite or you did not follow your team's coding conventions. It may even be simply because one of the generated reports told you that you were writing code that was too complex. In the end, that feedback results in considerable time saved and not spent - or even wasted - debugging.

Equally important is the peace of mind you and your team will get from being able to diagnose broken code, bugs, or poor quality code early.

Finally, integrating a new, significant, portion of code usually means deploying a new version of your application that a member of your QA team will need to test. When working as part of a team, there is always someone coming to your desk, asking you to install the latest version of the application you are working on on his device; don't be that guy! Be the smart one who automated everything. Building, automated testing, code coverage... all these concepts are simply a shell command execution away; why not let them work for you? As Douglas McIlroy, a well-known UNIX contributor said, "What you do today can be automated tomorrow." That could not be more true in the world of iOS development.

Making a build of your application available for the member of your team, your clients, or your boss to download, is complicated. For mobile, it's actually a multi-step process requiring building, code-signing, and finally deploying.

While we will show in the next chapters that it is totally achievable to do this by yourself, some services were born with the purpose of easing that process. Testflight, Hockey, or even Cloudbees, to name a few are some of the numerous services out there filling the gap. Some are free, some come with a price, some are in the cloud, and some are self-hosted, but all of them share the same common goal.

This book does not aim at comparing these services and telling you which one is the best. It will not give you an exhaustive list, and it will certainly not turn you into an expert in the utilization of the one we chose to talk about.

*“There is no single development, in either technology or management technique, which by itself promises even one order of magnitude [tenfold] improvement within a decade in productivity, in reliability, in simplicity.”*

“No Silver Bullet — Essence and Accidents of Software Engineering” - Fred Brooks (1986)

Of course, all of this is not magic. Being able to receive immediate feedback, while a neat advantage, comes at a cost. Setting up a continuous integration system takes time. This time is not lost - don't get us wrong - but that's something you have to take into account. You may not need a full-featured, continuous integration platform that builds your app every five minutes when you are working alone on a two-screen application. Sometimes, you have to be pragmatic.

Plus, your tools are only as good as the way you use them. You may not be able to take the feedback from your CI platform for granted if you set it up incorrectly.

## The Road Ahead

Each chapter will try to cover a very specific part of the principle of continuous integration for iOS. Reading this book is kind of like taking a journey. You will start by learning how to build an iOS application and release it to your team using the simplest approach, and end with a fully set up continuous integration installation. Here is an overview of the chapters we will cover together.

**Chapter 2 - Continuous Integration Tools and Features in iOS and Xcode** - We will create a simple iOS application that we will use as an example for our continuous integration process, and use it as an example for the rest of the book. In this chapter, we'll see what tools Xcode provides to help you in your day-to-day workflow.

**Chapter 3 - Using Xcode to Release an Application Outside the App Store** - With the application created in Chapter 2, we will see how we can create a build that you will be able to send to your testers. This will be the simplest approach of all, but you have to start somewhere, right?

**Chapter 4 - Invoking the Power of the Command Line** - Pressing a bunch of buttons is one thing, but knowing what they mean is another. In this chapter, you will learn how to leverage the power of the command line and what Xcode actually means by “build,” among other things.

**Chapter 5 - Automating Builds with Jenkins** - Jenkins is probably the de-facto solution for continuous integration. You will learn how to get it started and how it can build your iOS applications.

**Chapter 6 - Automating Builds with Bamboo** - Atlassian is an Australian enterprise software company that develops products for software developers and project managers. One of their products is a commercial alternative to Jenkins. You will learn how to get it started and how you can use it to build your iOS applications.

**Chapter 7 - Over The Air (OTA) Distribution** - Continuous shipping is a bit different than continuous integration and usually the final step of a successful build process. You will learn how you can make your life and the life of your testers easier by sending them new builds of your application as soon as they are available.

**Chapter 8 - Day-to-Day Use of Xcode Server and Xcode Bots** - As we have said, Apple finally released its own continuous integration tools. We will teach you how to use them and how you can fit them into your day-to-day workflow.

**Chapter 9 - Adding Unit Testing to the mix** - Whether you've decided to go for the free solution that is Jenkins, the enterprise one that is Bamboo, or a totally different solution we haven't included, you will learn how to automate the execution of your unit and functional testing as part of your build process.

**Chapter 10 - Quality Assurance** - As the cherry on top, we will show you which tool can be used to maintain high-quality code.

During this journey, don't hesitate to jump directly to a chapter. You may after all want to learn how to use Jenkins, but don't care about how Bamboo works, and that would be totally fine with us!

## Sample Application: Github Jobs

This book's goal is not to teach you how to create iPhone or iPad applications using Objective-C and Xcode. To get the most out of this book, make sure you're comfortable with Objective-C and Xcode first.

As an example, we will create an iOS application communicating with Github jobs' API. This application will be a very simple master/detail application. Coding the application won't take long, but be prepared to spend a lot of time configuring your project.

## Prerequisites

Make sure you have Xcode 5.1 installed, as all screenshots in this book will use this version specifically. You wouldn't want to get lost because of an outdated Xcode, would you?

Without any further ado, let's dive in!

## Summary

Continuous integration is a vast notion that can't be explained in a single chapter. It is a complex topic that impacts the technology you choose and the methodology you work with. This chapter merely introduced that notion of complexity. Now that you know more about the approach we took, let's start this journey by taking a tour of the tools and features available, while we work on the sample application.



# Continuous integration features in iOS and Xcode

Continuous integration is a matter of choosing the right tools, and the iOS community didn't wait for Apple before setting up their environments. How could they have? It has been known that Apple works in secret and releases all their new tools once a year, usually at WWDC. First, let's have a look at the things that have always been around, in Xcode specifically or in the community abroad.

If Xcode 5 came with a couple of shiny new tools for continuous integration, like the Xcode bots we will talk about later in this book, the previous versions already came with useful integrations to help you get some work done. Even if alternatives are getting more and more popular nowadays (for example, AppCode), and even if there are still people using their favorite text editor and a terminal to build the app, Xcode remains the de-facto IDE.

Of course it has its flaws, like the famous website "Text from Xcode," available at [textfromxcode.com](http://textfromxcode.com), has funnily shown in the past few years. No software is perfect, and we as developers tend to take everything for granted, but Xcode actually handles a lot for you. Let's see what it can do and how it will help us in setting up our continuous integration environment.

In this book we will be spending a lot of time in Xcode. As it wouldn't make sense to give you random screenshots to adapt to your existing application yourself, we will create a very simple application using the GitHub Jobs API.


We will start by creating the sample application and taking a tour of what the default project template gives us: what the application's info file is, where the unit tests are, and which framework do they rely on. Then, we'll move on to some more advanced stuff like versioning the project using Git and managing our dependencies using CocoaPods, a well-known dependency-manager for Xcode projects. Finally we will set up everything we need to be ready to release the application first to our beta testers and finally to the rest of the world. That will give us a great opportunity to cover in depth how Xcode will help us to manage multiple environments.

## Sample application: Github jobs

GitHub is a hosting service, available at [github.com](https://github.com), for software development projects that uses the Git revision control system. It comes with free, paying, and enterprise plans and is very well known in the community. Among other services, it comes with a dedicated job offers section for both job hunters and companies. If you want to know more about this service, visit <https://jobs.github.com>.

As we said earlier, this book does not aim at teaching you how to code in Objective-C. Chances are you know enough of it if you're reading this book. That's why we are keeping the application very simple: it will call the GitHub API, retrieve a bunch of job offers, and display them in a list using a UITableView. Nothing fancy.

## Creating the Application

Start by setting up a new iOS project using Apple's template. Open Xcode and hit the "Create a new Xcode project" button or, with Xcode open, choose "New  Project..." from the "File" menu. Make sure the iOS application templates are selected from the sidebar and choose "Single view application," as seen in Figure 2-1.

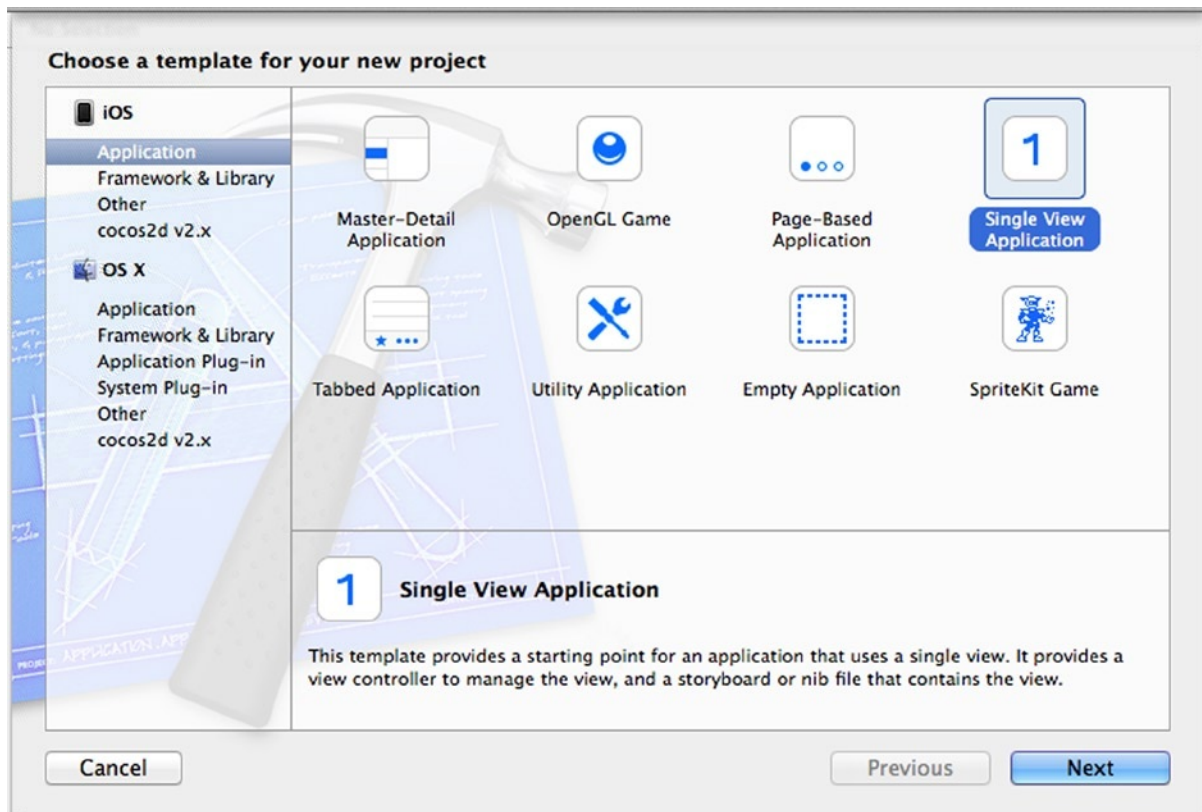


Figure 2-1. Selecting the template we want to start working on the application

Using “Github Jobs” as the product name, “com.perfectly-cooked” as the company identifier and “PCS” as the class prefix. It’s the one we’ll use and it will make it easier for you to understand the screenshots. The other options don’t matter much, just make sure to check the “Create Git repository on...” and select “My Mac” for now. Xcode allows you to use OS X server to host your code, among other things, but we will talk about that later.

## What Xcode Gives Us

We are using a standard template given by Xcode. It comes with a simple Storyboard file, an application delegate, and a UIViewController subclass. Let’s take a look at the things we can use in our day-to-day workflow and how they’ll fit in our continuous integration process.

## Application’s Info file

Every application comes with an Info file written in the property list format, you will find it under the name “MyApplicationInfo.plist” or in our case, “Github Jobs-Info.plist.” When you open that file, you will basically find all the public information about the application, among other things: The name of the application and the product identifier you set earlier when you created the project.

**Note About property list format:** Property list files are files that store serialized objects using the filename extension “.plist,” and thus are often referred to as Plist files. Because there are all the APIs you need in the iOS and OSX SDKs, property list files usually are the de-facto file format to store information about an application and other various settings and data.

First, open the file from your favorite text editor either by going manually to the directory where you created the application and navigating to the “Github Jobs” directory that corresponds to your application’s main target (the other one is the testing target, more on that later) or by right-clicking on the file in Xcode file explorer and selecting “Open with external editor.” The first lines should look like something similar to the following:

```
<?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>CFBundleDevelopmentRegion</key>
    <string>en</string>
    <key>CFBundleDisplayName</key>
    <string>${PRODUCT_NAME}</string>
    <key>CFBundleExecutable</key>
    <string>${EXECUTABLE_NAME}</string>
    <key>CFBundleIdentifier</key>
    <string>com.perfectly-cooked.${PRODUCT_NAME:rfc1034identifier}</string>
    ...

```

This is not really human friendly, to say the least. Don't worry though: Xcode doesn't expect you to edit this file manually. In fact, Xcode comes with a property list editor that will make it easier to view this file. It also turns the keys into comprehensible sentences. This way, "LSRequiresiPhoneOS" turns into "Application requires the iPhone environment."

Note that you can still view the name of the keys underneath. This can be helpful for debugging or simply because tutorials on the Internet often refers to these raw keys. From the editor, right-click on and select "Show Raw Keys/Values."

What interests us most are the two versions properties, by default, their values should be 1.0. The first one, the "Bundle versions string, short" a.k.a. `CFBundleShortVersionString` is the main version of the application. If you have already shipped an application on the app store, you've probably already changed its value. The second one, "Bundle version" a.k.a. `CFBundleVersion`, is the full version number and doesn't have to be human-readable. Most people tend to keep both in sync, but that would be a waste of configuration in our case, as the long version will be proven useful in the future.

Later in the book we will start talking about automated builds. These builds always come with an incrementing number and this is when the `CFBundleVersion` key comes in handy. According to Apple documentation, the `CFBundleVersion` specifies a version of the application, released or not, which fits perfectly. Every time our application is built, we will use the build's number to change the `CFBundleVersion`. To keep things clear, it is important to have coherence between those two versions.

This Info property list file contains information about the application that is made available from the `NSBundle` class. Its content is not limited to the keys available now, there are in fact a lot of other keys that will be added if, for example, you want your application to be opened when a certain URL scheme is called from a webpage. Xcode manages most of these keys for you, so you don't need to know them all.

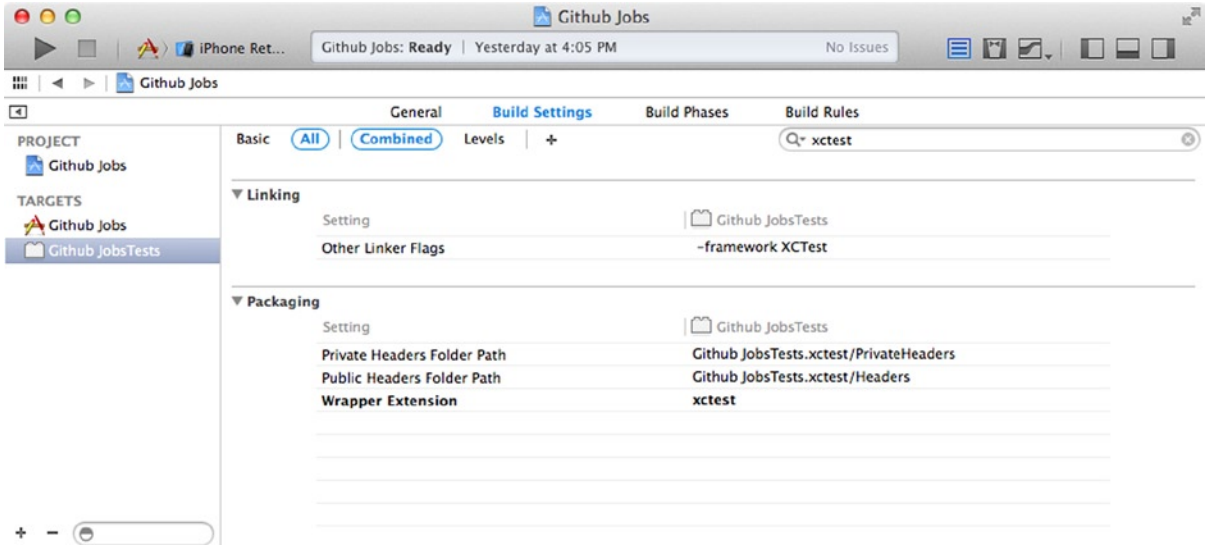
## Tests

An Xcode project comes with a testing target, a test class, and your very first test case. This last one is a failing one, as it basically just calls `XCTFail`, a C macro used to force-fail a case. Still, that's something.

Everything has been configured so you can get to work right away and start writing tests at the beginning of your project. That's something you'll be very happy about when your project becomes big and complex.

When we started writing the book, we talked a bit about how Apple took its time before actually giving you tools for continuous integration. That's not entirely true. Since Xcode 2.1, Xcode came with a version of the testing tools `SenTestingKit` and `OCUnit`, which wasn't an official Apple-labelled tool back then, but a framework created by a company called "Sen:te" (<http://sente.ch>).

Open the sample application and select "Github Jobs," the first element in the file explorer. Select your testing target, named "Github JobsTests" and select the "Build Settings" tab. You should see all the settings of this target. Filter these settings using the search field at the top right and look for "xctest." As the Figure 2-2 shows, you should see that a framework called `XCTest` is automatically linked with your application. `XCTest` is a brand new testing framework heavily based on the old one but way better and way more deeply integrated in the IDE and much more powerful.



**Figure 2-2.** The filtered list of build settings shows the XCTest framework linked to the Github Jobs testing target

In the 5.1.1 release, OCUit and the SenTestingKit framework were marked as deprecated and will be removed from a future release of Xcode. In fact, source code using OCUit now generates warnings at compilation time.

For starters, Xcode 5 comes with a test navigator available in your sidebar or by pressing “⌘ + 5.” This navigator allows you to see the recently run test cases and to re-run them by pressing a single button, making every refactoring a breeze. That’s not all there is. Xcode 5 comes with new assistant editors that will help you unit-test your code. The assistants, called “Test Callers” and “Test Classes,” provide access to unit tests related to the current source code in the primary editor. In our example shown in Figure 2-3, we have created a fake PCSGithubJob class (on the left) with a method whose purpose is to analyze a job offer and tell you if it will make you happy. The “Test Callers” editor (on the right) automatically jumped to the related testing method.

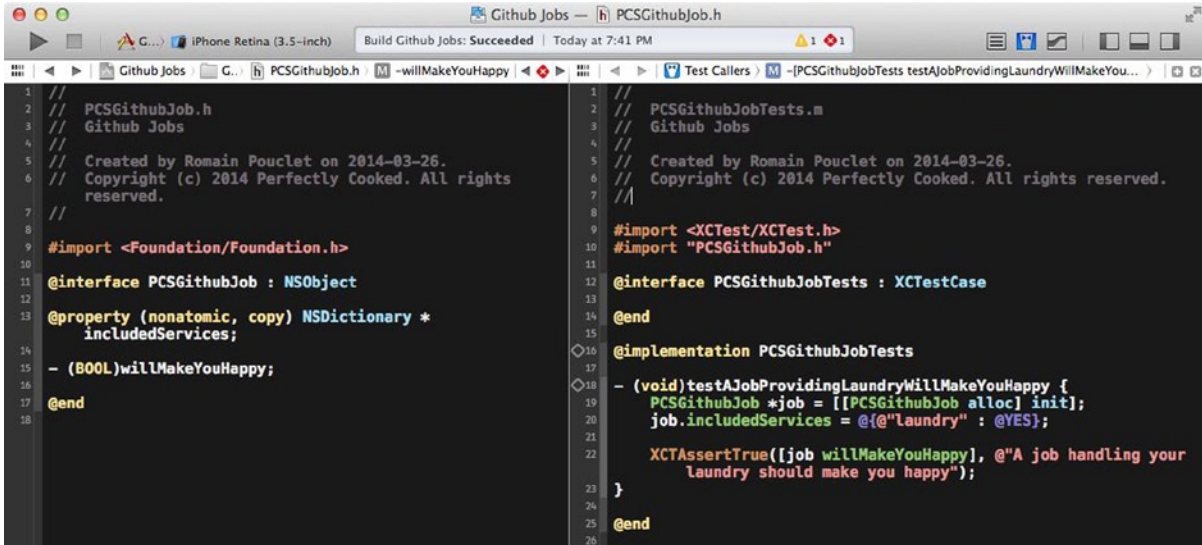


Figure 2-3. Xcode with the primary and the assistant editors opened showing the unit-testing code related to our class

Last but not least, the xcodebuild command line tool now supports the test action for iOS projects. Before this release it was not possible directly to run your unit-tests from the command line. We will cover more about the command line equivalent of these tools in Chapter 3.

Back in Xcode 4, when you created an iOS or OSX application, you had to check an option to add a testing target to your project and now this is the default behavior. This is a nice touch from Apple since the community of the iOS developers has always had this image of one that mostly focuses on the design and the user experience. It shows a clear will of helping the iOS developers to become more professional and start using the proper tools.

Finally, the new testing framework was actually announced at the same time as the “Xcode Bots.” They are meant to help you embrace the concept of continuous integration and they fit perfectly with XCTest. Make sure to read Chapter 8 to learn more about them!

A good test suite is key in a continuous integration environment. We talked earlier about how a good environment will provide you with maximum feedback. Automated testing is one of the tools for this purpose. Whether you are an independent iOS developer working alone or a developer wondering if you should integrate the feature your coworker has been working on, making sure all the tests are green first sure will help you in making a decision.

## Git Integration

Integrating pieces of codes in your project as we said in the introduction usually means merging branches using your favorite version control system. A successfully continuous integration usually means a well-established workflow. When you are working on an application, there are a couple of things that will happen. Once the first version has been released, you will start working on new features, some of them will take time, and that’s why you will be working on separate branches of your repository. During this process, your users will probably send you bug reports and feedback about various glitches in the application. These will have to be fixed as soon as possible and be

shipped as new minor versions of your application. Once again, because you don't want to mess with the application, you will work on separate branches just as you work on new features.

The main thing to keep in mind is that at any and every time you work you will want feedback about the tasks you are working on. You will want to know if you are currently breaking something while implementing your new feature, or if the bug that has been reported to you is being fixed. Plus, your QA team, your boss, or even your client may want that feedback as well, and probably a fresh build of your application. Because, "I'm sorry, I broke the app for now, please call me back in an hour or two" is not an acceptable answer, you will have to keep everything separated.

The good news is that branches are cheap in Git. You can create as much as you want without any trouble for you or your coworkers. The great news is Xcode comes with Git support right out of the box, as you may have noticed when you created the Github Jobs project or by looking at the big "Source Control" menu in Xcode.

Back in the old days, Xcode only supported Subversion and CVS. While the first one still exists, even if it's grown a bit out of date due to the arrival of Decentralized Version Control System (DVCS), such as Git and Mercurial, the second one was clearly a legacy from the sad old days of versioning control.

Git is a really popular Version Control Systems that works in a decentralized way. Meaning the opposite way of how SVN works, every single working copy of a project can be used as a remote and even better, can work independently.

To show you how useful it is to have Git deeply integrated in your Xcode project, let's break it. That's right, let's create a very simple conflict and see what happens. In the world of Version Control Systems, conflicts are what happen when merging two versions of a file is not possible. If you changed the same part of the same file differently in the two branches you're merging together, it won't be able to merge them and a manual intervention will be required.

In Xcode, from the Source Control menu, select "Github Jobs – master" and then, select "New Branch...". In the field that appears, fill the branch name field with "saying-hi" and press "Create." At the end of this process, you will be working on a dedicated branch for your "Saying Hi" feature. This means that your application will evolve while a stable version, the master branch, is still available somewhere.

Now, open the "AppDelegate.m" file and add a simple NSLog instruction that greets the user when the application finishes launching:

```
#import "PCSAppDelegate.h"

@implementation PCSAppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSLog(@"Hi, I'm a sample application for a book.");

    return YES;
}

@end
```



With this greeting implemented, open the Source control menu and select “Commit...”. In the form that appears, enter a simple commit message such as “Saying hi when the application starts” and press the “Commit 1 File” button. You’ve now made your very first commit in this feature branch.

To create a conflict, a file must have been modified at a similar place in the file in a different branch, so we need to go back to the master branch and create a commit similar to the one we’ve just made.

From the Source Control menu, select “Github Jobs – saying-hi” and then “Switch to Branch...”. In the list that appears, select the master branch and press “Switch.” We are now back to the master branch, and the NSLog instruction we’ve added is no longer here.

Let’s repeat the process with a slightly different greeting message while still on the master branch. We chose to go with “Hi, I’m a book created for a sample application,” which indeed doesn’t make much sense. Commit your modification like we did before. We now have 2 branches, both with a slightly different greeting instruction at the same position in the same file.

Finally, from the Source Control menu, select “Github Jobs – master” and then “Merge from Branch...”. In the list that appears, select “saying-hi” and press “Merge”: a window similar to the one shown in Figure 2-4 will appear. Select the third button in the menu at the bottom of the screen to integrate the modifications on the right and press the “Merge.”

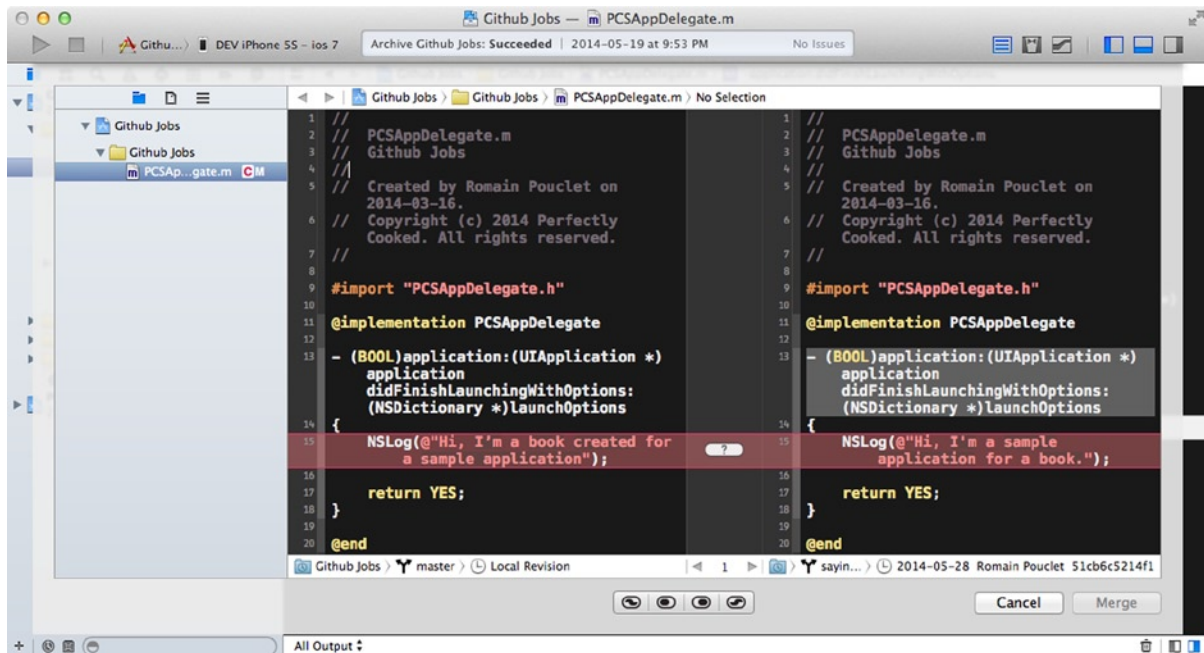
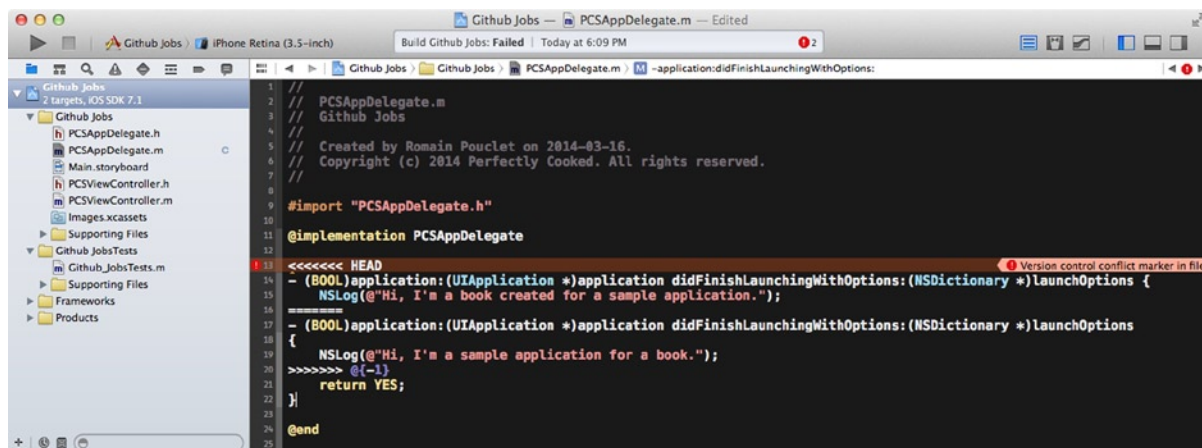


Figure 2-4. Xcode has detected a conflict and will help you to fix it

That’s basically how conflicts work and that’s how Xcode will assist you to fix them and keep your project clean. However, sometimes conflict will happen outside of Xcode, because you decided to merge a branch using the command line or because you started using a dedicated Git client.



That’s the kind of thing Xcode will understand. When that happens and some of your files become disfigured with weird characters all over the place, building your application could simply throw you some random syntax error and let you find out why your build is failing. Xcode is much more clever than that. Indeed, you won’t even be able to build your application unless the conflict is marked as resolved manually using your shell or your favorite Git client, or directly in Xcode, as shown in Figure 2-5.



**Figure 2-5.** Xcode has detected a conflicted file and marked it as such in the sidebar and the primary editor

To fix this conflict and mark it as resolved, edit the “AppDelegate.m” file until it looks good to you. In our case, that would be keeping only one “application:didFinishLaunchingWithOptions:” method and choosing the second message. You must remove all the conflict markers, which are the lines with the multiple chevrons and equal signs, for your application to build again. After all, in this context, they are nothing more than syntax errors. Once this is done, simply right-click on the conflicted file and select “Mark Selected File as Resolved” in the “Source Control” menu item.

Versioning control in Xcode is not all sparkles and rainbows. There are edge cases that come with Apple using file formats such as the one for the “xcodeproj” file or even “xib” and “storyboards.” In Xcode 5, Apple came with a much simpler file format for those Interface Builder files, making merge operations and conflict resolutions a lot easier. From NIB to bloated XIB, to almost-readable, ten-times shorter XIB, Apple is indeed working at easing this process. Once again, we’ve come a long way, even if we are still waiting for a simpler file format for those “xcodeproj,” which still uses an old-style brace based format to delimit the configuration hierarchy.

If there is one tip this book can give you, it’s to close Xcode during merge operations involving an Xcode project or an Xcode workspace file. Usually, this happens when someone on your team is changing a setting in the “provisioning profiles” and/or “code signing” sections. As this format is hardly human-readable at all, the results from a merge can get ugly and Xcode can get a little bit “crashy.”

There are still a couple of downsides to this integration directly in the IDE. For starters, it is pretty limited. You'll only be able to do the basic Git operations and will probably end up using your shell or a dedicated Git client. Also, Xcode ships its own version of Git. At the time of writing this book, the current stable version of Git is the 2.0.1, as shown in a shell after running `git --version`:

```
$ git --version
git version 1.9.1
```

This version can be easily installed using a package manager such as Homebrew or even compiled and installed manually, if you have some spare time on your hands. However, if you open a terminal and run a simple command, you should see that the version of Git used by Xcode is Apple's own version, based on 1.8.5.2:

```
bash-3.2$ $(xcode-select -print-path)/usr/bin/git --version
git version 1.8.5.2 (Apple Git-48)
```

In most cases, it won't cause any trouble, but incompatibilities between Xcode's Git version and the one you're using may arise. This was the case when Xcode 4 was the current version and the latest Git was a couple of versions ahead of the one that was shipped with it.

**Note About xcode-select:** `xcode-select` is one of those very useful tools shipped with a bunch of others provided by Apple's official command line tools and is very helpful if you are playing with multiple installations of Xcode, during betas for example. We'll talk about this more in Chapter 4.

## CocoaPods

Someone once said, "with great projects come huge dependencies" and managing dependencies with external libraries in an iOS project has pretty much always been a pain. This is a pain you don't need in your iOS project, but Apple doesn't seem to be willing to provide an official tool besides workspaces, subprojects, and *manually* managing the search paths of your dependencies. This was before CocoaPods.

CocoaPods is a dependency manager written in Ruby, unsurprisingly inspired from RubyGems (<https://rubygems.org/>). If you don't know about it but have been writing Ruby in the past, its syntax used in the PodFile, in which you'll declare all your dependencies, is really easy to understand and should not give you too much of a hard time.

The use of CocoaPods really is up for debate in the iOS and Mac community. Whether you hate it or love it, in this book we definitely stand with the people who love it. In the process of continuously integrating pieces of code, letting a well-designed tool handling the management of your dependencies makes something less to worry about and gives you more time to focus on other things.

As a bonus, you can actually use CocoaPods for a specific application's dependencies, meaning you can use it to split your big application in multiple modules and let CocoaPods handle the merge.

Once again, there is no silver bullet. As it is written on the website, “CocoaPods is not ready for prime-time yet.” At the time of writing this book, 0.33.1 is the latest version. It’s far from being perfect but it’s a lot better than no tools at all. We will be using CocoaPods in the sample application.

Speaking of which, let’s code.

## Coding the Sample Application

Open the “Github Jobs” project we created earlier in Xcode, select the only view controller header class present in the file explorer, and make it an **UITableViewController** instead of a simple **UIViewController**. Go to the associated implementation file, declare a “jobs” property of type `NSArray`, and implement the required methods from `UITableViewDataSource`: “tableView:numberOfRowsInSection:” and “tableView:cellForRowAtIndexPath:.” At the end of this process, your implementation file should look like this:

```
#import "PCSVViewController.h"

@interface PCSViewController ()

@property (nonatomic, strong) NSArray *jobs;

@end

@implementation PCSViewController

#pragma mark - Table View
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    return self.jobs.count;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"Cell"
forIndexPath:indexPath];

    return cell;
}

@end
```

Then, open the storyboard file and remove the view controller already created by Xcode. Instead, drop a new navigation controller. It should automatically come with a table view controller. Select it and use “PCSVController” as its class. Select the navigation item and change the title to

“iOS jobs.” In the UITableViewController, select the first prototype cell and give it the “Cell” reuse identifier to avoid getting an exception on run later on. That’s pretty much all there will be in this application. We are trying to keep it very simple.

The Github Jobs API is very simple, all you need is calling a specific URL to retrieve a JSON-encoded list of iOS job offers in New-York. Using the iOS 7.0 new NSURLSession API, go back to the implementation file of your view controller and fetch content from that famous URL in the viewWillAppear: method as follows:

```
- (void)viewWillAppear:(BOOL)animated {
    NSURL *url = [NSURL URLWithString:
        @"https://jobs.github.com/positions.json?description=ios&location=NY"];
    NSURLSessionDataTask *jobTask = [[NSURLSession sharedSession] dataTaskWithURL:
        url completionHandler:^(NSData *data, NSURLResponse *response, NSError *error) {
        dispatch_async(dispatch_get_main_queue(), ^{

            if (error) {
                UIAlertView *alert = [[UIAlertView alloc] initWithTitle: @"An error occurred"
                    message: error.localizedDescription
                    delegate: nil
                    cancelButtonTitle: @"Dismiss"
                    otherButtonTitles: nil];

                [alert show];
                return;
            }

            NSError *jsonError = nil;
            self.jobs = [NSJSONSerialization JSONObjectWithData: data options: 0 error: &jsonError];
            [self.tableView reloadData];
        }]);
    }

    [jobTask resume];
}
```

Note that we wrapped the whole content of the completion block using a function from Grand Central Dispatch (GCD) so all the UI job is performed on the main thread. Don’t forget to update the tableView:cellForRowAtIndexPath: method to display the “title” property of a job.

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"Cell"
        forIndexPath:indexPath];
    cell.textLabel.text = self.jobs[indexPath.row][@"title"];

    return cell;
}
```

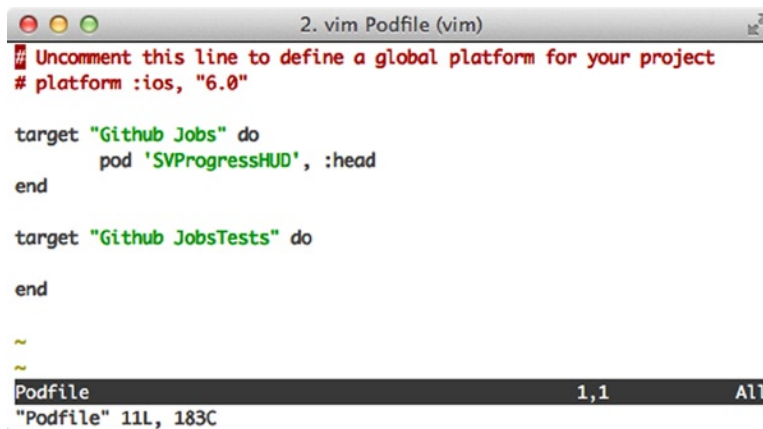
The application could have been a master-detail application, and selecting a row could have opened a detailed page showing the application logo, detailed information about the offer, and a direct button helping the user postulate. Instead, because a complete application is not the goal of this

book, selecting a cell with simply... will open Safari mobile. To do that, implement the delegate method that will be called when the user taps on a cell in the “PCSVController.m,” as follows:

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    NSURL *jobUrl = [NSURL URLWithString: self.jobs[indexPath.row][@"url"]];
    [[UIApplication sharedApplication] openURL: jobUrl];
}
```

The only missing part of this application is a way to show the user that work is being done and content is not available yet. We will use a very simple library called SVProgressHUD for this very purpose and we will install it using – that’s right, CocoaPods.

Close your Xcode project and open a terminal. Navigate to the location of your iOS project and install CocoaPods if you haven’t already, using RubyGems (see instructions below). Run the `pod init` command to create an empty-ish Podfile and open it. Add SVProgressHUD as a dependency, as shown in Figure 2-6:



```
2. vim Podfile (vim)
# Uncomment this line to define a global platform for your project
# platform :ios, "6.0"

target "Github Jobs" do
  pod 'SVProgressHUD', :head
end

target "Github JobsTests" do

end

~
~
Podfile 1,1 All
"Podfile" 11L, 183C
```

**Figure 2-6.** Using vim, we are adding the SVProgressHUD library as a dependency

Finally, run `pod install` and open the generated ``Github Jobs.xcworkspace`` instead of the previous ``Github Jobs.xcodeproj``. This is why we asked to close Xcode.

Here is the whole process:

```
$ (sudo) gem install cocoapods
$ cd path/to/Github\ Jobs
$ pod init
$ vim Podfile
// edit Podfile...
$ pod install
$ open Github\ Jobs.xcworkspace
```

**Installing CocoaPods** The installation of CocoaPods may, of course, vary in case you are using a Ruby version management system such as rbenv (<http://rbenv.org>) or RVM (<https://rvm.io>).

SVProgressHUD provides really easy methods to use to display messages. Let's go back to the view controller and use these methods, before and after fetching the content from the Github Jobs API. Before calling the resume method of the NSURLSessionDataTask object that we created, import the SVProgressHUD .h file and call the following class method:

```
#import <SVProgressHUD/SVProgressHUD.h>

// ...

- (void)viewWillAppear:(BOOL)animated {
    NSURL *url = [NSURL URLWithString:
        @"https://jobs.github.com/positions.json?description=ios&location=NY"];

    NSURLSessionDataTask *jobTask = ...
    [SVProgressHUD showWithStatus: @"Fetching jobs..."];
    [jobTask resume];
}
```

The SVProgressHUD library comes with two different methods, “showWithStatus:” and “showWithStatus:maskType.” We are using the second one so we can have a dark background. Otherwise we wouldn't be able to properly see the HUD being animated.

Once the jobs have been fetched and the JSON has been properly decoded and turned into a list of jobs to display, we want to give the user a small indication that the process is over. To do that, add the following snippet at the end of the NSURLSession's “dataTaskWithURL: completionHandler:” completion block, as follows. Calling showWithStatus: will show a temporary confirmation message, there is no need to dismiss it manually.

```
NSURLSessionDataTask *jobTask = [[NSURLSession sharedSession] dataTaskWithURL: url
    completionHandler:^(NSData *data, NSURLResponse *response, NSError *error) {
        // ...

        [SVProgressHUD showSuccessWithStatus: [NSString stringWithFormat: @"%lu jobs fetched",
            (unsigned long)[self.jobs count]]];
    }];
```

When you run your application, you should see the HUD in both situations, similar to Figure 2-7.