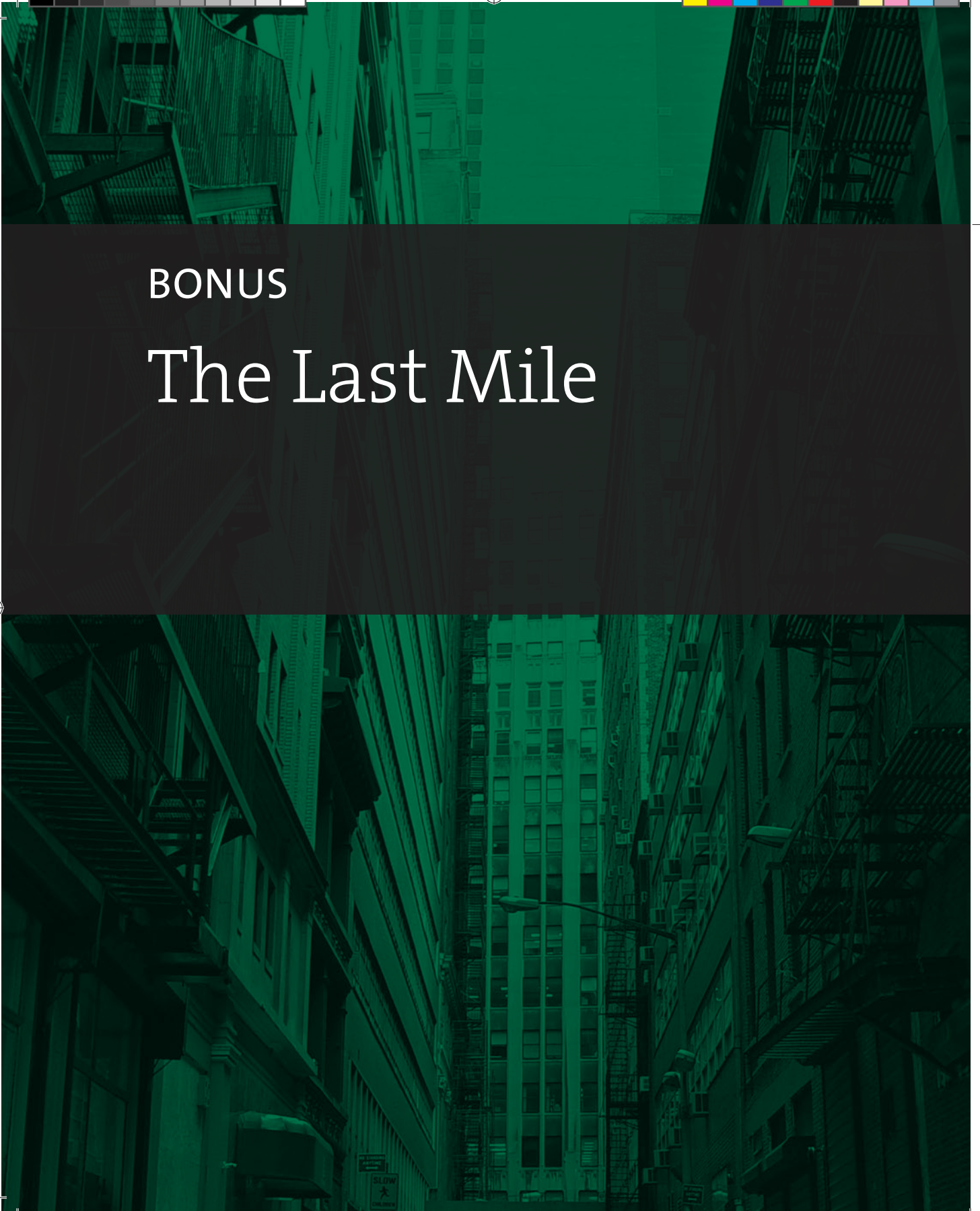




BONUS

The Last Mile





This chapter covers many of the details behind working with code. This includes managing our source code as well as testing, profiling and debugging our applications. While this does not add any new features to our applications, these are vital skills that all developers need to master if you want to produce production-quality code.



MANAGING SOURCE CODE

As our project grows, we create and modify an ever-increasing number of files. Simply managing these files soon becomes a huge job in its own right. For a single developer working on their own computer, you might be able to just get by without any real organization or plan. Just keep the files backed up and safe, and everything should be fine.

But, what happens if we start making big changes, and then decide that our new approach simply does not work? How do we get back to the previous version of our code? What do we do if our team grows? How do we add new artists or developers? How do we integrate all of the individual edits and additions? How do we make sure each of us is working on the same, canonical version of the project?

Fortunately, this is a problem that software engineering has solved years ago. We just need to use some sort of source control software (also called version control, revision control or a software configuration management system). Some source control systems, like Subversion, provide a single, centralized repository. Everyone commits their changes to this repository, where they are merged into a single, canonical version. The team can then update their local copy from this centralized version. Other systems, like Git, use a distributed source control system—each user has their own copy of the entire repository. Changes are copied from one repository to another. Both approaches have their own advantages and disadvantages. People can—and will—argue endlessly over the details. Still, most of the basic operations remain the same.

Xcode supports both Subversion and Git repositories. As we saw in Chapter 2, “Our Application’s Architecture,” Xcode can even create local Git repositories for us. In my opinion, all projects should at least use a local Git repository, and most serious projects should also use an off-site repository as well.

WHAT SOURCE CONTROL DOES

Source control systems record a running history of all changes made to the project. They coordinate changes from multiple sources, merging the changes where possible, and alerting us to conflicts when they arise.

Let’s say we’re working on a project together. I modify line 42 in the `mainView.m` file, and commit my change. Meanwhile, you delete line 42 entirely, and commit your change. Obviously these changes contradict each other. Since I committed my change first, you will receive an error when you try to commit yours. You then need to review the conflict and decide whether to use my change, to keep your change, or to otherwise fix the problem manually.

All committed changes are also annotated showing both who made the change along with comments that hopefully explain the reasons behind those changes. We can also compare different versions of the file from anywhere along the project’s history. We can even revert the file to any of its previous states.

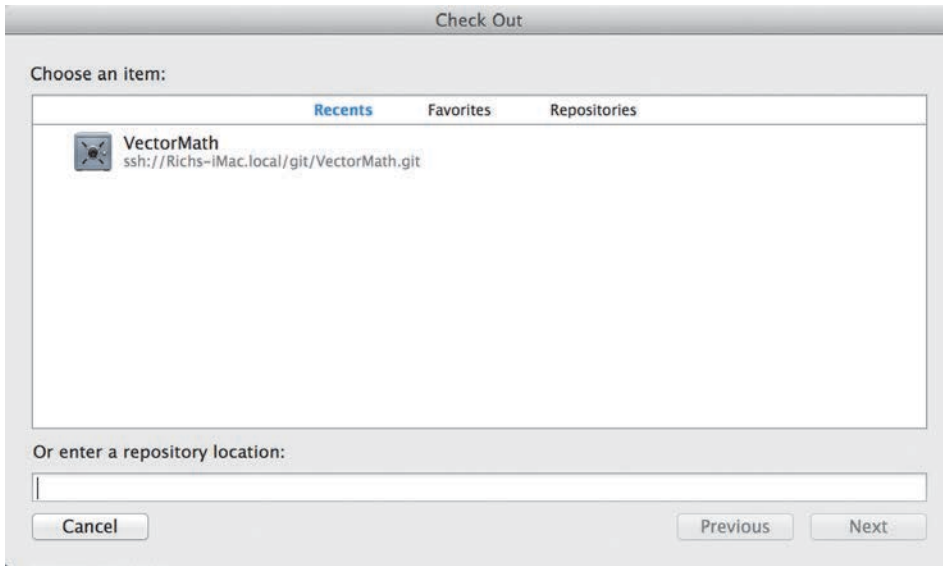


FIGURE 1 Checking projects out of a repository

By doing all this, the source control system gives us several advantages. It provides a single, canonical version of our code. Anyone on our team can easily find, download and use the most up-to-date version. Remote repositories also act as a backup for our projects. Even if something unfortunate happens to our development machine, we can always re-download the most recent version. Finally, source control provides a safe sandbox for experimentation. We can alter our code without actually modifying the main project. This is particularly important when debugging.

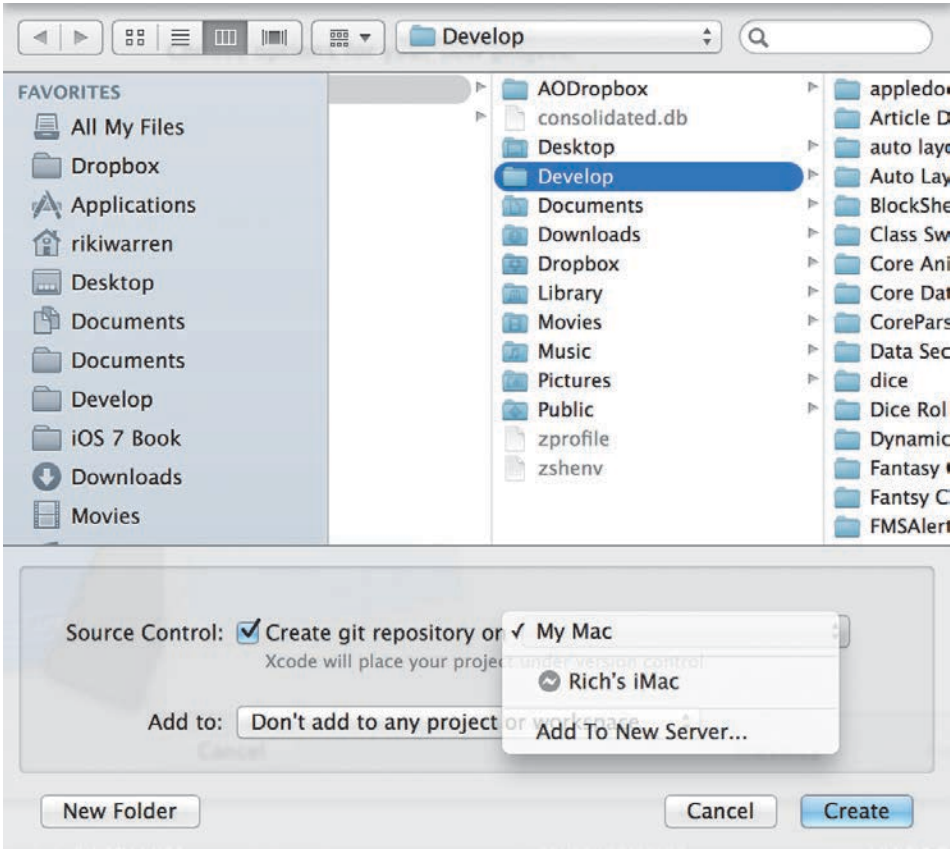
Often, when trying to fix a bug, we need to write some exploratory code—we need to play around with the project and figure out what exactly is going wrong. Then, once we understand the problem, we can revert back to the original version and fix it.

MANAGING THE CODE HISTORY

We typically start by creating a new repository, or checking the project out of an existing repository. To check projects out of a repository in Xcode, simply select the Source Code > Check Out... menu item. Then, from the Check Out sheet, you can either select a repository, or provide the URL to a new repository (**Figure 1**).

We can also create a new Git repository when creating our application. This can either be a local repository on our development machine or a remote repository on a Git server. To create a remote repository, click on the “Create git repository on” chooser. From here, you can either select a known server, or add a new server (**Figure 2**). If you create a new remote repository, Xcode will clone the repository for you, providing you with a local copy to work on.

FIGURE 2 Creating a remote repository



Once we have the repository in Xcode, we can open the project as normal. We can also manage the source code using the Source Control menu. The exact content of this menu may change depending on the type of repository we're using. With a Git repository, *Commit* is used to save your changes to your local repository. *Push* and *Pull* are then used to upload and download these changes to the remote repository. In Subversion, *commit* uploads your changes to the centralized repository, while *update* downloads all the new changes from the repository.

When a file's under source control, a letter may appear next to its name indicating its status. **Table 1** has more information on these status labels. We can right click on these files to commit or update individual files, add files to the repository, discard all existing changes, or mark a file to be ignored. We can also manually mark a file as resolved when we run into conflicts.

The Working Copies section of the Source Control menu also lets us manage our projects branches. Our project's menu item shows the current branch. We can also create new branches, switch to a different branch or merge branches (**Figure 3**).

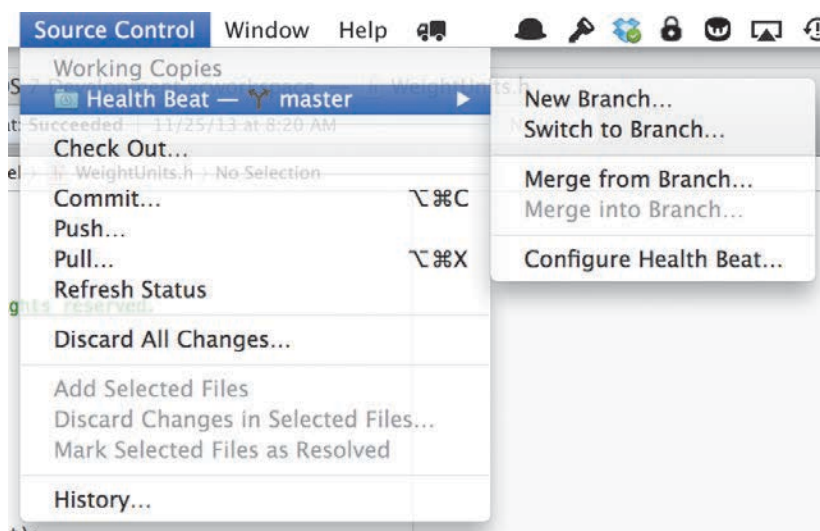


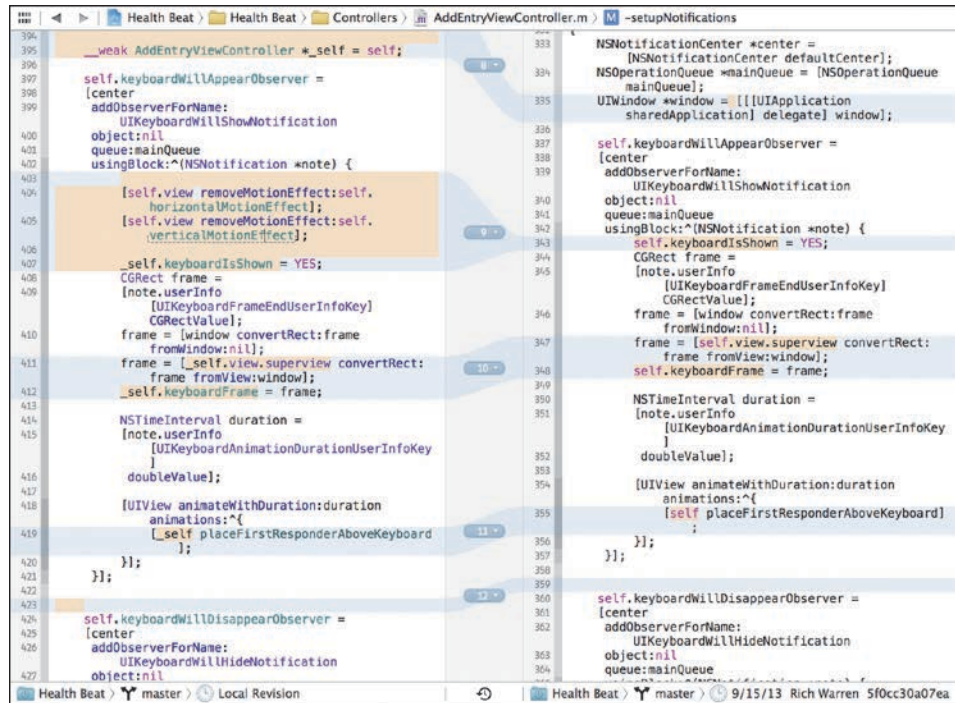
FIGURE 3 Managing branches

TABLE 1 Source Control Status Labels


LETTER	MEANING
M	The local copy has been modified
U	The copy in the repository has been updated
A	File will be added to the repository on the next commit
D	File will be deleted from the repository on the next commit
I	Local file is ignored
R	The file has been replaced in the repository
-	Typically used for a group or folder when the contents have a mixture of statuses
?	The local file is not under source control

Branches let us work on multiple versions of our project at the same time. Often this is done to let developers modify one version of the project, while still keeping another version intact. Typically, the main branch (often called the master or trunk) is where we do our day-to-day development. We typically create a new branch for each release. This lets us maintain the released applications while simultaneously building the next. We may also create branches for experimentation, letting us try out new ideas without touching the trunk code.

FIGURE 4 Highlighting changes between versions



COMPARING VERSIONS

So far, Xcode's built-in source control support provides functional access to all our source control system's major features. However, it really starts to shine when we begin comparing different versions of a given file. Start by opening a file as normal, and then click on the Version editor button () in the toolbar.

Like the Assistant, this displays two files side-by-side in the editor area. Unlike the assistant editor, this shows two versions of the same file. By default, the left file is our current local copy. The right file is the base file in the repository (the most recent version committed to the repository). If you've just updated your file from the repository, and you haven't made any local changes yet, these files will be identical.

Xcode will highlight the differences between the files for us. **Figure 4** shows how local additions are highlighted. In addition, as you scroll, red tick marks appear in the scroll bar, indicating the location of our changes.

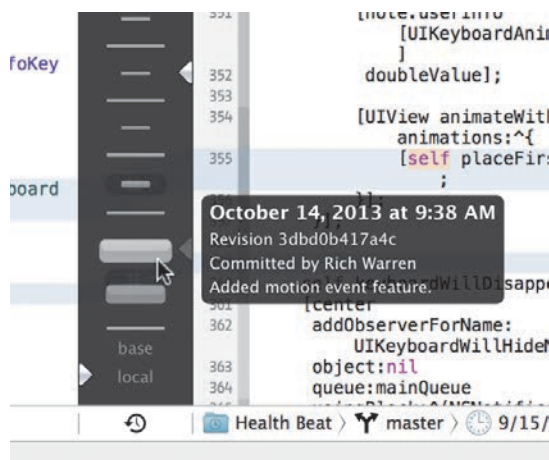



FIGURE 5 Exploring the repository's history

Xcode also makes it easy to navigate back through the file's history. Simply click the Timeline button (the  icon at the bottom of the editor window) to bring up our repository's history. This appears as a black bar between our versions, with a track of grey lines running up the middle. Longer lines indicate time periods (separating different days). Shorter lines indicate the different versions. Faded lines are not being used. As you mouse over each version, Xcode displays a bubble with information about that commit—including the revision number, the person who committed the changes, and the commit's comment (**Figure 5**).

The two grey arrows along the side of the timeline show us which versions are being displayed. You can click on any of the revisions in the timeline to open them. Click the right side of the timeline to open that version in the right editor. Click the left side to open on the left. Alternatively, you can drag either arrow (left or right) to change the corresponding view.

While we've gone through the source control basics, there's a lot more to learn. Xcode's built-in support makes common tasks easy—but I still find myself retreating back to third-party tools or even the command line when things go awry. This is particularly true when dealing with distributed teams and remote repositories—anytime you add a network connection or multiple developers, you're bound to have unexpected issues sooner or later.

A number of great books cover different source control systems and practices in detail. I highly recommend finding and reading through some of these, especially if you're setting up a large or distributed team.



TESTING

Testing—a short word that covers so many different topics. When we talk about testing, we’re really talking about a large group of loosely related tasks. At a minimum, this includes unit testing, performance testing, usability testing, and beta testing.

- **Unit testing:** Here, we are testing individual functions or methods, making sure they perform as expected. Unit tests need to be both automated and repeatable. The goal is to run these tests as frequently as possible, and to have them produce consistent results. We don’t want random results. We don’t want to manually set up or tear down our test suites.

Ideally, we want to run all our unit tests after any significant change to our code. We typically don’t want to run all of the tests every time the app is built—that would quickly bog down our development, making it difficult to perform quick iterations. Therefore, we typically run a more focused set of tests while working with the code, and only run the entire suite just before committing our changes. Many teams also automatically run nightly tests on their repository’s trunk. Having your commits red-flagged in the next morning’s build report really discourages developers from committing broken or incomplete code.

Ideally, unit tests should cover every possible branch throughout your entire application. In the real world, 100% coverage is rarely practical. In particular, it is often difficult (if not impossible) to test the user interface. Still, at a bare minimum most of our model should be covered. Many developers also prefer to focus their unit tests on complex, bug-prone code, and avoid writing extensive test cases for accessors and other common, low-risk methods.

Unit tests are also the basis of Test Driven Development (TDD). TDD is an approach to writing code that emphasizes tests as part of the design process. It’s often described as writing tests before writing code—but that’s a bit of an over simplification. For example, if we wanted to make a new class, we would first write a few tests to begin sketching out the class’s interface. With those tests were in place, we begin implementing the class itself, writing just enough code to get our tests to pass. Once all our tests pass, we add more tests, and then write more code. We proceed iteratively, switching between writing tests and writing code until our class is complete.

While TDD is an interesting concept, it often feels more idealistic than practical. Still, it can greatly help organize and focus your thoughts—especially when you’re struggling with a piece of code. Often, it’s easier to define a few tests, and start working from there.

For me, however, the real benefit of unit testing comes when we start modifying our code. This happens whenever we add new features to our project, refactor old code, or begin fixing bugs. With a strong set of tests in place, it’s easy to make changes, confident that we’re not simply replacing old bugs with new ones. This is particularly important in larger teams, where the developer fixing the bug may not be the person who wrote the code in the first place.

- **Performance testing:** Performance tests focus on using software tools to monitor the application while it runs. Typically, these tests look for performance bottlenecks and



memory usage. We can use performance testing either proactively or reactively. In proactive performance testing, we are looking for places where we might be able to optimize our application. Our app can never be too fast, and its memory footprint can never be too small. Reactive performance testing, on the other hand, is used to troubleshoot performance or memory problems.

In both cases, we will want to run our tests both before and after making any changes. By comparing our new code's performance with our old performance, we can quantify how our changes affect our application. Obviously, we don't want to accidentally make changes that reduce our application's performance.

Unfortunately, much of the time the true costs and benefits of our changes are not completely clear. We often need to decide if the change in performance justifies the additional complications introduced by our optimizations. Similarly, we may need to sacrifice memory usage for increased performance (or performance for memory usage). Sadly, once the obvious mistakes are removed, software engineering often grinds down to a game of compromise and balance between different, contradictory goals.

- **Usability testing:** Usability testing involves getting the app into the hands of a new user and observing how they interact with it. Typically the user is given a list of tasks to perform, but are not told how these tasks should be performed. Engineers then observe the users and see how they actually interact with the application, making notes on how to improve the experience.

Some types of usability testing may also occur after an app has been released. Many developers use third-party analytic toolkits to remotely monitor their users' interactions. These tools can provide vital information about the app. Which features do the users spend the most time using? Are there any features that are not being used very often? How can those features be improved? Unfortunately, these monitoring tools can be somewhat controversial. There are definite privacy concerns, since many users may object to apps that, essentially, spy on them and report back to the mother ship.

- **Beta testing:** Here we distribute our application to a large number of users. The goal is to get the app into the hands of "typical customer." Often, this means finding a wide range of people with different backgrounds. It is hoped that these testers will uncover bugs and problems that were not found during the earlier stages of testing. Beta testers are given pre-release access to the application. In return they are supposed to report any bugs they find. It's usually easy to find people willing to try out your product for free—getting good feedback from them can be more difficult.

Beta testing typically involves making and distributing ad hoc builds to your testers. This can be a somewhat tedious process—since you need to gather and register all the UUIDs for all the test devices. Then you need to distribute the application back to the users. Finally, you will need to gather crash reports and logs from the users.

Fortunately, there are a number of third-party libraries that can help facilitate beta testing. The two most popular are TestFlight and HockeyApp. While there are some differences between the features that they offer, both of these help automate many of the tasks involved in getting builds to your beta testers and getting information back from them.



These aren't the only types of software testing. Many teams incorporate formalized integration, regression, verification and validation, alpha and other testing in their development cycles. They may have formalized test requirements and a dedicated testing team. Other times, developers implicitly perform these tests as part of their routine, daily work.

There are a number of books that cover software testing and software project management in great detail. For the rest of this section; however, we will focus on unit testing and performance testing. Unlike many of the other testing topics, both of these have a strong technical side. We must either write custom code, or use specialized tools to perform these tests.

UNIT TESTING

Xcode supports integrated unit testing through the XCTest framework. Tests can be run on both the simulator and on a physical test device. We can also automate unit tests using bots.

The XCTest framework also divides our tests into four parts: test targets, test cases, tests and asserts.

TEST TARGETS

Test targets determine how our tests are built and run. By default, all new projects begin with two build targets, the application target and the test target. We've used the application target to modify both our application's configuration and its build settings. The test target provides similar control over our unit tests.

In particular, the test target will include all the test cases in our project by default. However, we can modify which test cases are included in the build. This lets us control which test cases are built and run during unit testing.

The test target also determines the build settings for our unit tests. This means our unit tests could use a different set of build settings than our application. For example, if we customize the warnings and errors for our application, we may want to duplicate those settings for our test build as well.

However, note that our unit tests always run within the context of our full iOS application. This means we have access to all the classes in our application target—we don't need to add them to the test target as well. Additionally, Xcode will always build our application classes using our application's build settings. The test target's build settings only affect the unit tests themselves.

CREATING TEST CASES

Test cases are classes that contain one or more unit tests. Each test case is a subclass of XCTestCase. When we run our unit tests, the system will look for all the subclasses of XCTestCase, and run all of their test methods.

Xcode automatically creates an initial test case for us when we create our project. We can add other test suites by selecting New File..., and then selecting the iOS > Cocoa Touch > Objective-C test case class template. In the options panel, we can set the class's name and its superclass—though we almost always want to leave the superclass as XCTestCase.



Test case files are organized a little differently than most class files. First, we place both the `@interface` and the `@implementation` blocks within the same `.m` file. This simplifies our testing code, and helps keep everything in one file. We can get away with this, because we will never be importing our test cases into other files—so there's no need for a separate `.h` file.

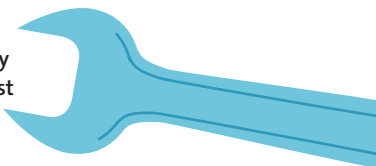
Next, the test case template provides three method stubs: `setUp`, `tearDown` and `testExample`. `setUp` and `tearDown` are special methods. As the names suggest, `setUp` is run before every test, while `tearDown` is run after each test. We should use these methods to instantiate and configure any resources our tests need, then dispose of them cleanly once the test has ended. By bracketing each test case, we ensure that our tests are run in a clean environment, and that all our resources are in a known good state. This helps prevent side effects, interactions or dependencies between test cases.

The `testExample` provides an example test. The default implementation simply fails. While this may not seem particularly useful, we can use `testExample` as a quick sanity check to ensure that our test case has been properly added to the test target. Run the unit tests. If everything is wired up properly, the `testExample` will fail. Typically, however, we will delete the `testExample` method and replace it with our own tests.

TESTS

Each test is simply a method containing one or more asserts. To create a new test, we simply add a new method to our suite. This method must begin with the word “test.” It cannot take any arguments and it shouldn't return any values. When we run our tests, the system uses reflection to search for these methods at runtime, and then executes them. As long as we follow the naming convention, the system handles everything for us automatically.

NOTE: Both our test suites and our individual test cases may be run in any order. Therefore, we cannot have any dependencies between individual test cases. Each test case should be independent and isolated from all others.



ASSERTS

Finally, we get to the asserts themselves. Asserts are simply XCTest macros that evaluate or compare expressions. Each assert is an atomic operation that either passes or fails. The system then records and reports the results of all the asserts in all of our tests across all our test cases.

The complete list of asserts can be found in **Table 2**.



TABLE 2 XCTest Asserts

TEST NAME	DESCRIPTION
XCTFail(format...)	This test always fails.
XCTAssertNil(a1, format...)	Fails if the provided expression does not evaluate to nil.
XCTAssertNotNil(a1, format...)	Fails if the provided expression evaluates to nil.
XCTAssert(expression, format...)	Fails if the given expression evaluates to NO.
XCTAssertTrue(expression, format...)	Fails if the given expression evaluates to NO.
XCTAssertFalse(expression, format...)	Fails if the given expression evaluates to YES.
XCTAssertEqualObjects(a1, a2, format...)	Fails when [a1 isEqual:a2] is NO or when one of the objects is nil and the other is not.
XCTAssertNotEqualObjects(a1, a2, format...)	Fails when [a1 isEqual: a2] is YES or when both of the objects are nil.
XCTAssertEqual(a1, a2, format...)	Fails when a1 == a2 is NO. This macro should be used when comparing C scalar values, structures or unions.
XCTAssertNotEqual(a1, a2, format...)	Fails when a1 == a2 is YES. This macro should be used when comparing C scalar values, structures or unions.
XCTAssertEqualWithAccuracy (a1, a2, accuracy, format...)	Compares two C scalars. It fails if the difference between the objects is greater than the specified accuracy. This is designed for use with floating point values, where small differences could be introduced due to rounding errors—however, it should work for any C scalars.
XCTAssertNotEqualWithAccuracy (a1, a2, accuracy, format...)	Compares two C scalars. It fails if the difference between the objects is less than or equal to the specified accuracy. This is designed for use with floating point values, where small differences could be introduced due to rounding errors—however, it should work for any C scalars.
XCTAssertThrows(expression, format...)	Fails if the given expression does not throw an exception.
XCTAssertThrowsSpecific (expression, specificException, format...)	Fails if the given expression does not throw an exception of the specified class.
XCTAssertThrowsSpecificNamed (expression, specificException, exception_name, format...)	Fails if the given expression does not throw an exception matching both the specified class and name.
XCTAssertNoThrow(expression, format...)	Fails if the given expression throws an exception.
XCTAssertNoThrowSpecific (expression, specificException, format...)	Fails if the given expression throws an exception of the specified class
XCTAssertNoThrowSpecificNamed (expression, specificException, exception_name, format...)	Fails if the given expression throws an exception matching both the specified class and name.

SAMPLE TEST SUITE

Here's a simple test suite. We're just running a few sanity checks to determine if our conversion class methods return the correct answers. Since we're comparing floating-point values, it's best to compare against an accuracy threshold. We're not trying to get a specific answer—we're just trying to see if our answers are close enough. In this case, we define "close enough" to mean within 0.01 of the value that we calculated using an external unit converter.

Then we check to see if our `stringForUnit:` method returns the correct unit label. Notice that we also check failure cases—specifically we make sure an invalid unit value causes our method to throw an exception.

Finally, we have an unimplemented test case. Here, we use `XCTFail()` as a `TODO:` label. It will fail any time we run the tests, reminding us to come back and implement this test.

Modify `Health_BeatTests.m` as shown:

```
#import <XCTest/XCTest.h>
#import "WeightEntry.h"
#import "WeightEntry+addons.h"
#import "WeightUnits.h"

static const CGFloat accuracy = 0.01;

@interface Health_BeatTests : XCTestCase

@end

@implementation Health_BeatTests

-(void)testLbsToKg {

    // correct values according to Wolfram Alpha
    XCTAssertEqualWithAccuracy([WeightEntry convertLbsToKg:0.0f],
                               0.0f,
                               accuracy,
                               @"Incorrect weight for 0 lbs");

    XCTAssertEqualWithAccuracy([WeightEntry convertLbsToKg:10.0f],
                               4.5359f,
                               accuracy,
                               @"Incorrect weight for 10 lbs");
}
```



```
XCTAssertEqualWithAccuracy(  
    [WeightEntry convertLbsToKg:145.6f],  
    66.043f,  
    accuracy,  
    @"Incorrect weight for 145.6 lbs");  
}  
  
- (void)testKgToLbs {  
  
    // correct values according to Wolfram Alpha  
    XCTAssertEqualWithAccuracy([WeightEntry convertKgToLbs:0.0f],  
                                0.0f,  
                                accuracy,  
                                @"Incorrect weight for 0 kg");  
  
    XCTAssertEqualWithAccuracy([WeightEntry convertKgToLbs:10.0f],  
                                22.0462f,  
                                accuracy,  
                                @"Incorrect weight for 10 kg");  
  
    XCTAssertEqualWithAccuracy([WeightEntry  
                                convertKgToLbs:145.6f],  
                                320.9931f,  
                                accuracy,  
                                @"Incorrect weight for 145.6 kg");  
}  
  
- (void)testStringForUnit {  
  
    XCTAssertEqualObjects([WeightEntry stringForUnit:LBS],  
                           @"lbs",  
                           @"Invalid string returned for LBS");  
  
    XCTAssertEqualObjects([WeightEntry stringForUnit:KG],  
                           @"kg",  
                           @"Invalid string returned for KG");  
}
```





```
XCTAssertThrows([WeightEntry stringForUnit:2],
                @"Any invalid value should throw "
                @"an exception");

}

- (void)testStringForWeightOfUnit {
    XCTFail(@"TODO: implement the testStringForWeightOfUnit "
            @"test case");
}

@end
```

Notice that we're only testing the class methods. Ideally, we would also want to test `WeightEntry` objects as well. Unfortunately, we need to set up an entire Core Data stack before we can instantiate new `WeightEntry` objects.

This is a very common problem, and we have a few possible solutions. We could set up a `UIManagedDocument` instance, and use that to build our test objects. This has a few problems. First, our document is created asynchronously. We'll need to make sure the document is completely initialized before we begin running our tests. Second, the tests will modify the document's data. We'll need to make sure to remove any objects we add—otherwise they may accumulate over time, eventually creating problems. On the plus side, this would let us stress test our document and see how it handles adding 100,000 new weight entries.

Alternatively, we could create a Core Data stack specifically for testing. In particular, we could build a stack with an in-memory persistent store. This would both help our tests run faster, as well as preventing any lasting side effects. However, building a mock object from scratch requires a lot of work.

Mock objects are a common feature in unit testing. We often use them to isolate sections of our application, avoiding the need to pull in a large graph of objects just to test a single class. We can also use them to replace resources like web services or databases. This both lets us test our code without altering live data, and lets us simulate a wide range of responses—including both invalid responses and errors.

In fact, mock objects are so common that third-party developers have created a number of mock libraries. These typically let us instantiate mock objects quickly, further simplifying our tests. However, they aren't without their own problems. Whenever we test against mock objects, we are building a set of assumptions into our tests, and our tests are only as good as those assumptions.

This issue often crops up when working with complex web services. We may not know the entire range of possible errors that we might encounter. The best solution often involves writing tests for the obvious errors, and then logging any unexpected errors. We then need to periodically review the logs, and write new unit tests for any new errors that we find. Ideally, we would like to continue this even after the app goes into production—however, that involves setting up some sort of remote logging.

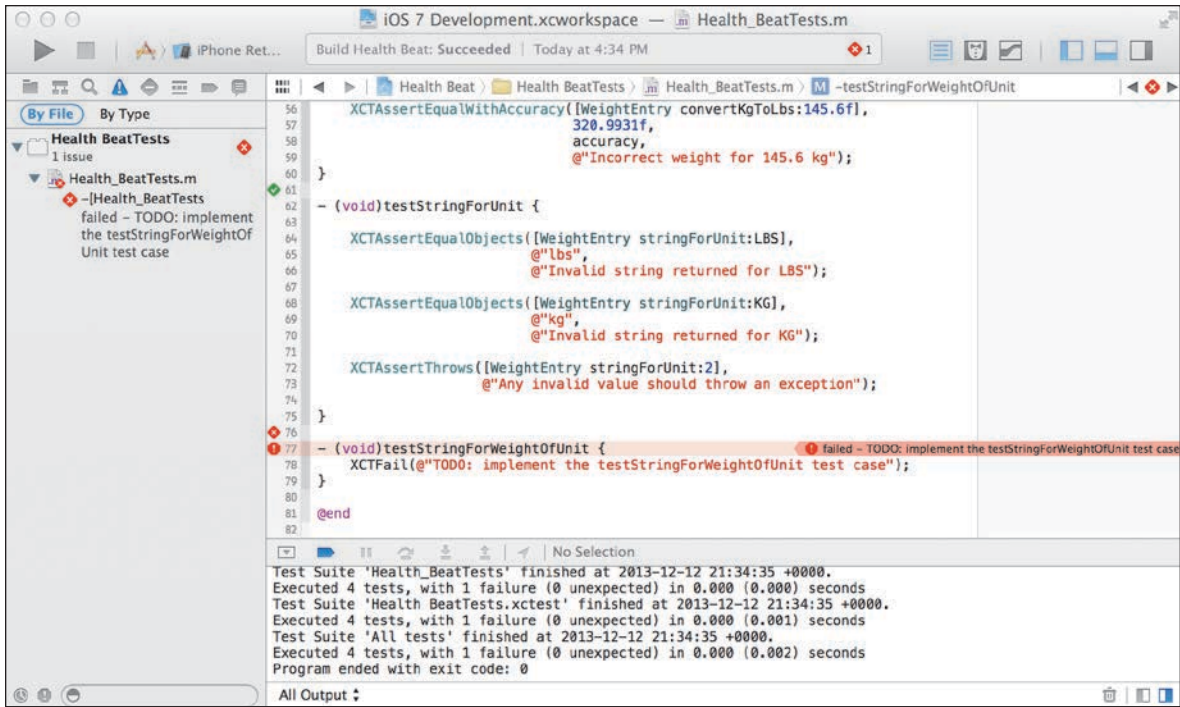


FIGURE 6 Xcode displaying a failed test case

RUNNING UNIT TESTS

We can run our tests by simply selecting the Product > Test menu. Alternatively, if you click and hold on the run button, it will bring up a list of options: Run, Test, Profile or Analyze. Select Test.

Xcode will compile and run our application, then execute all of our test cases. Once it is finished, it closes the application. If all the tests succeed, our Issue navigator will be empty, and we should see summary information at the bottom of the console log.

```
Test Suite 'All tests' finished at 2013-12-12 21:33:19 +0000.
Executed 4 tests, with 0 failures (0 unexpected) in 0.000 (0.002) seconds
Program ended with exit code: 0
```

You can scroll up through the console log to see more information about the individual test suites and test cases.

If any tests fail, they will appear on the Issue navigator, and the failing test case will be flagged within Xcode's Editor area (Figure 6).

Notice that in each test case, the class and each method has an icon beside it. This is a green diamond with a check mark for tests that have passed, a red diamond with an X for tests that failed. Clicking on an icon will rerun that particular test, or in the case of the class icon, it will rerun the entire test case.



FIGURE 7 Launching OS X Server

AUTOMATING TESTING WITH BOTS

With OS X Mavericks and Xcode 5, Apple has provided a convenient way to automatically run unit tests, called “bots.” Bots let us periodically build, test and archive our projects. Bots have one significant advantage over other continuous integration technologies—they are heavily integrated into Xcode. Once we have a server set up, we can create, manage and monitor our bots directly from Xcode. We can also monitor our bots using a web browser.

There are three main steps to using a bot. First, we need to set up our server. Second, we need to set up our remote repository. Finally, we need to set up the bot itself.

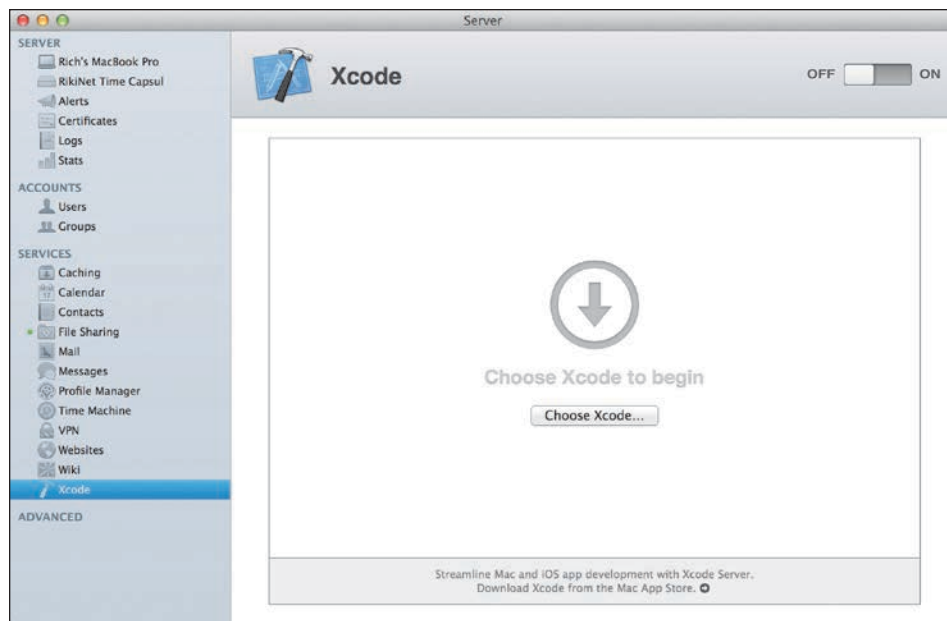
Start by making sure both OS X Server and Xcode are installed on the test machine. OS X Server is an application that can run on any Macintosh running Mavericks. You can buy it from the Mac App Store for \$19.99 in the US; however, members of the developer program can download a copy for free.

When you run the Server app, it asks to set up OS X Server on this Mac (Figure 7). Click Continue to install. You will have to agree to the terms of service and log in with administrator access. It will then take a few minutes to set everything up.

Once complete, you should see two separate windows. One is the Server app itself. This lets you start, stop, configure and monitor any of the provided services. The other window is the Server Tutorial.

OS X Server provides a wide range of possible services, including caching downloads from iTunes or the App Store; running your own calendar, contacts, mail, message, Time Machine or web server; or setting up a Profile Manager, VPN, or wiki. However, for our purposes, the most important service is the Xcode service. This lets us both host remote Git repositories, and setup Bots for automatic testing.

FIGURE 8 Setting up the Xcode service.



Close the tutorial window (you can always open it again by selecting the Help > Server Tutorials menu). And, in the Server window, select the Xcode service (**Figure 8**).

Click on the Choose Xcode... button and select the Xcode app you wish to use, and then tap on the switch in the upper right corner to turn the service on (**Figure 9**). You will need to add a developer team and the test devices if you want to run the unit tests on actual, tethered devices. A development team is also required to archive your project as part of the tests, and to remove some of the warnings that will appear when the tests are built. Still, if you haven't joined the developer program yet, the default settings will at least let us run tests in the simulator.

Next, we need to create a remote repository that the Xcode service can access. This could be a repository on the test machine itself, or a remotely hosted repository—for example a repository on GitHub.

There are a couple of ways we can create the remote repository from within Xcode. We've already seen how to set up repositories when creating a new project. In the Save sheet, click on the “Create git repository on” chooser. If you've already used the desired server, it will automatically appear on the list. If not, select “Add To New Server...” (**Figure 10**).

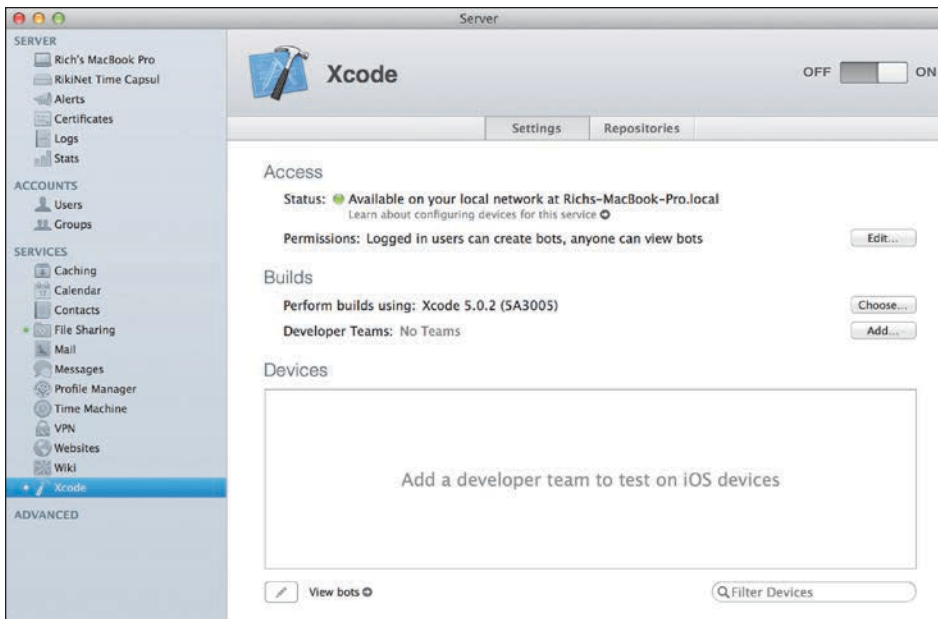


FIGURE 9 Running the Xcode service

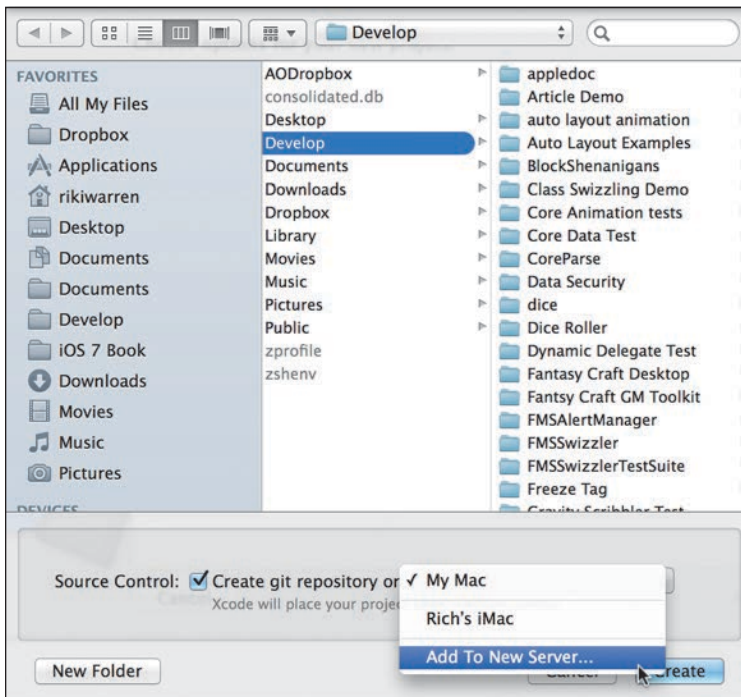


FIGURE 10 Creating a repository on a remote server

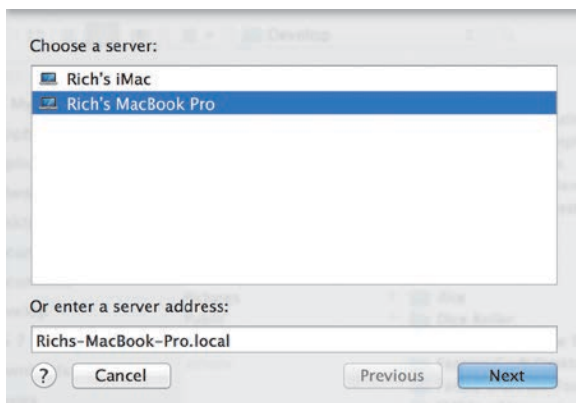


FIGURE 11 Adding a new server

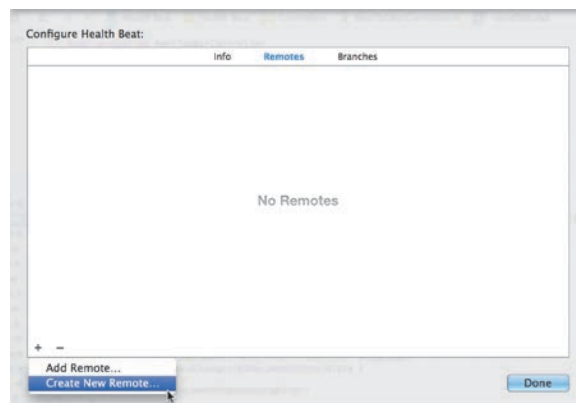


FIGURE 12 Creating a new remote repository

All local machines running the Xcode service will appear in the server list. Alternatively, we can provide the URL for a different remote server (Figure 11). You will then be prompted to log into the remote server.

Now, when you create the project, it will create both the local and remote repository and sync the two.

For an existing app, if it already has a local Git repository, we simply need to add the remote server, and push our project to that server. Let's do that for our Health Beat app.

Unfortunately, as I'm writing this, Xcode seems to have a problem combining Git repositories and workspaces. This is a shame, since the 2013 WWDC Video "Understanding Source Control in Xcode" shows some of the cool things you can do with multiple working copies inside a workspace. However, until it is working properly, it is easier to work with single projects.

Open the Health Beat project directly by clicking on Health Beat.xcodeproj, and then select Source Control > Health Beat – Master > Configure Health Beat.... In the configuration sheet, select the Remotes tab and then click the "+" button and select "Create New Remote..." (Figure 12).

Xcode will then prompt you for the server and the remote's name. Select the server, just as we did in the previous example, and leave the repository's name set to **origin**. This is the default name that Git uses when cloning a repository from a remote server. That means, when this is done, it will look like the remote repository is the original source code and our local repository is the copy. That's exactly what we want.

Click Create. Once you've created the repository, you can use Source Control > Push... to push our project out to the remote repository.

NOTE: If possible, your test server and your development computer should be different machines. Among other things, this helps protect your source code. If anything bad happens to your development machine, you can always check out the most recent version of the project from the test server.

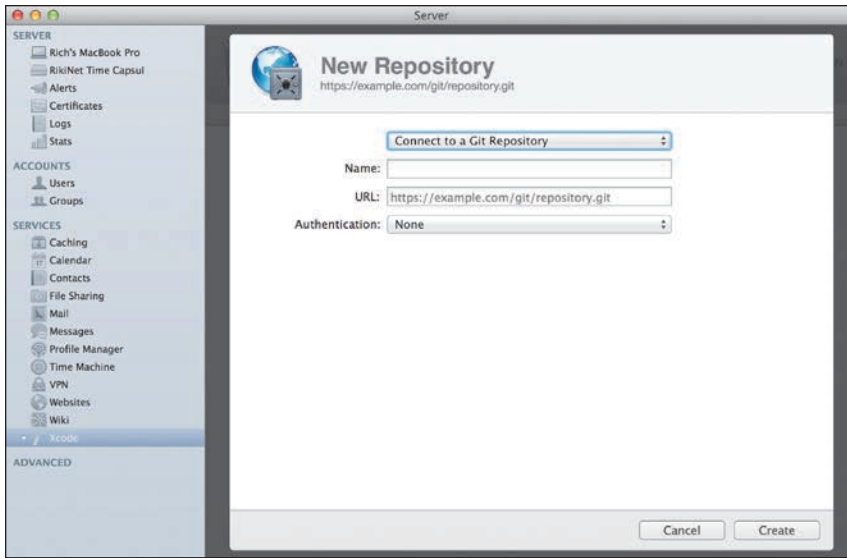


FIGURE 13 Connecting to a third-party repository

If your project doesn't have a local Git repository—you will have to find a way to either set one up remotely or locally. Unfortunately, there is no way to do this from within Xcode—so you will probably need to either use raw Git commands from the command line, or look into a third-party Git app. The “Xcode Continuous Integration Guide” has instructions for doing this from the command line. Look for the “Enable Access to Your Source Code Repositories, Use Git to Manage an Unmanaged Workspace Directory on a Development Mac” section.

If your remote repository is not hosted by the Xcode service, you will need to give it access to the repository. In the Server app, make sure the Xcode Service is selected, and then switch to the Repositories tab. Click the “+” button. In the New Repository form, select “Connect to a Git Repository” and then fill in the repository's name, URL and authentication information (**Figure 13**).

Once we have our remote server set up, we can create our bot. Select Project > Create Bot.

In the “Create a new bot” sheet, we need to select the scheme we wish to use to build and run our tests. This needs to be a shared scheme. If it's not shared already, make sure the Share scheme box is checked. That will add the scheme to our repository and commit it automatically.

We also need to give the bot a name and select the server. You also probably want to leave “Commit changes and integrate immediately” checked. This will force the server to immediately run our unit tests—giving us a chance to see the tests in action (**Figure 14**).

Click Next. We may be prompted to log into the server. If so, use your user name and password on the test computer.

Once we've logged in, we get to configure when and how our tests are run (**Figure 15**). Tests can be run either periodically (weekly, daily or hourly), they can be performed after each commit, or they can be performed manually. By default, the bot will analyze our code, run our unit tests and build an archive of our app. We can turn any of these steps off.

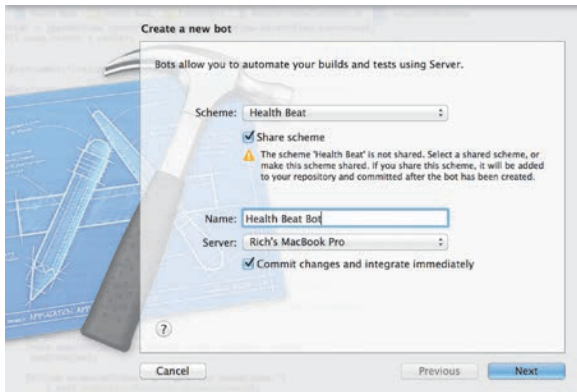


FIGURE 14 Creating a new bot

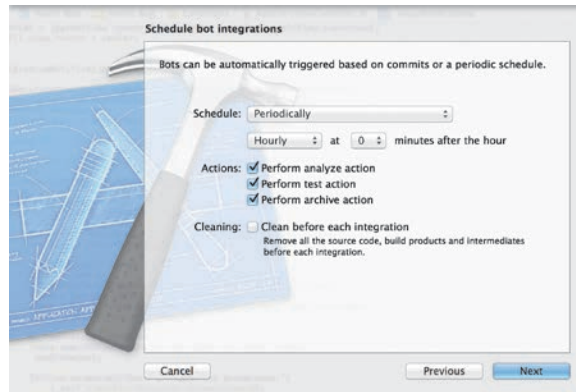


FIGURE 15 Configuring the bot

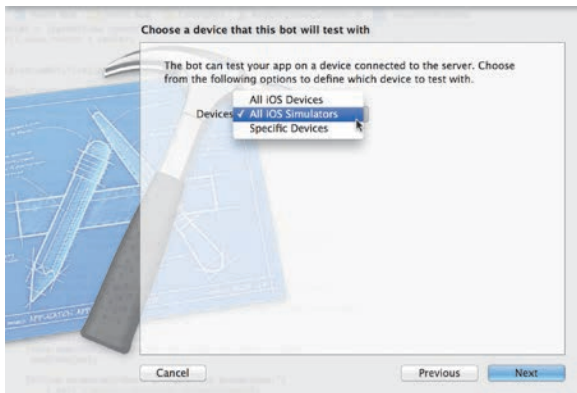


FIGURE 16 setting the test devices

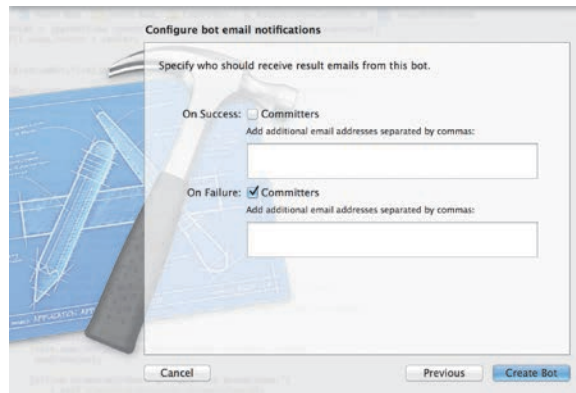


FIGURE 17 Setting email notifications

NOTE: You must set up your team on the Xcode service before you can perform any archive actions. If you haven't joined the developer program yet, be sure to uncheck the archive action before proceeding.

We can also force the bot to do a complete clean and build before running the tests. When I say complete, I mean complete. It will delete all the build products, intermediate products and even the source code. It will then re-download everything from the remote repository. Obviously, this can use up a lot of bandwidth, particularly if you are performing hourly tests on a large project. So, enable this feature with care.

On the next sheet, we get to set the test devices (Figure 16). We can either select all connected iOS devices, all simulators or select a particular mixture of devices and simulators. For Health Beat, let's test on all the simulators.

Finally, on the last sheet, we get to specify who gets notified both on success and on failure (Figure 17). By default, just the committer is notified when there is a failure. However, we can add any email addresses we wish to either list.

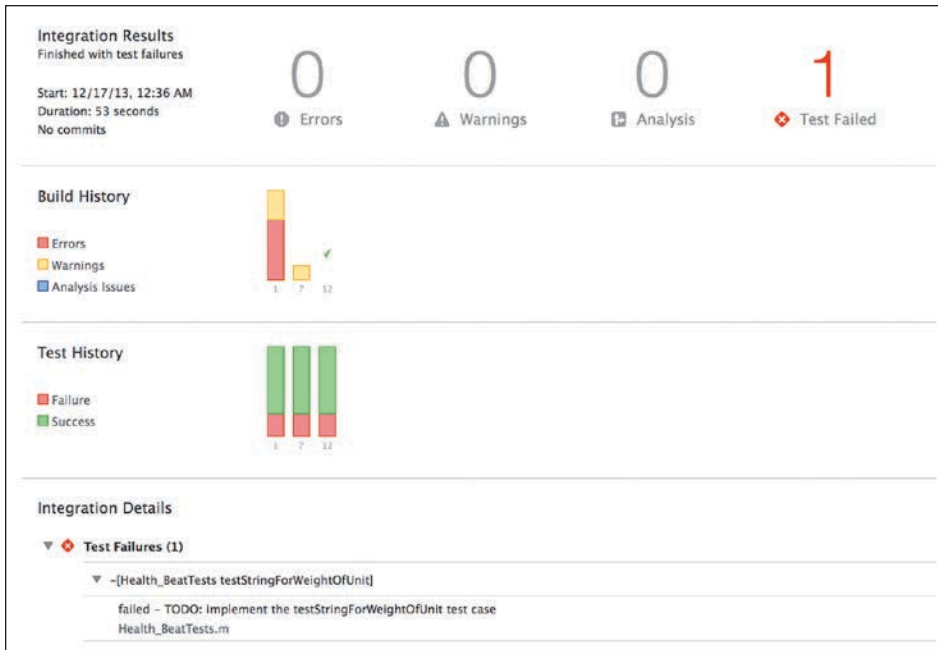


FIGURE 18 Viewing our bots history

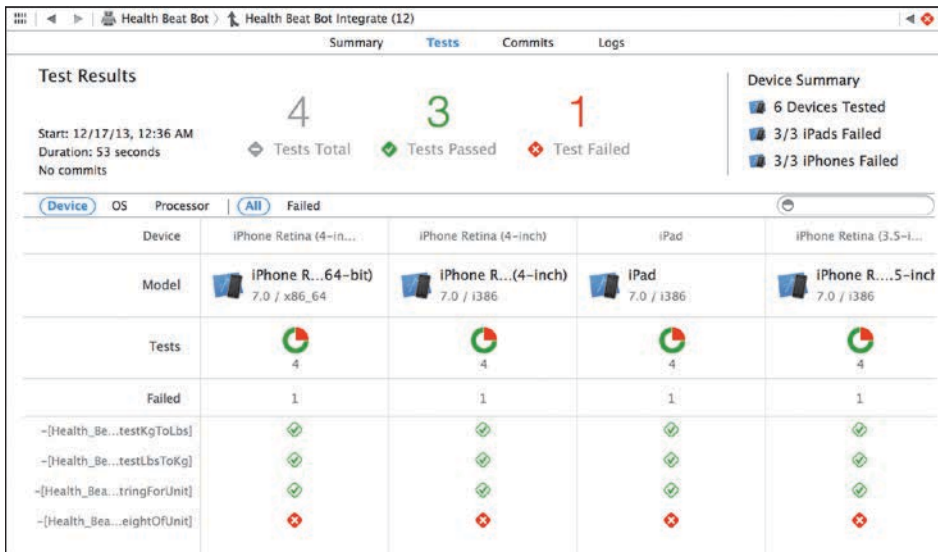
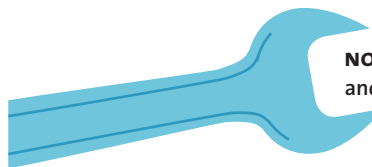


FIGURE 19 Viewing the test details

Once the bot is created, we will be asked to commit and push our changes to the repository. Then our bot will run; however, it can take several minutes before the results to show up.

We can monitor our bots using the Log navigator. Clicking on the Bot icon in the navigator will display both our build and test history (**Figure 18**). We can also click on the individual integrations, and dig into the details (**Figure 19**).



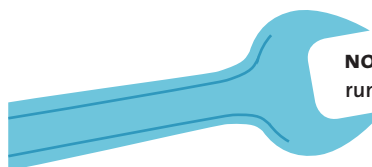
NOTE: The fact that we can create multiple bots, using different schemes and build schedules gives us a lot of flexibility when it comes to designing our automated tests. We might have one bot that automatically analyzes and tests the code whenever there's a commit. We might have another that does nightly tests and creates an archive of our nightly build. Still other bots may use specially-created schemes to run different test targets, letting us run a different set of tests on different devices. The possible combinations are nearly endless.

As you can see, there were some linking errors (our unit tests were not being built with 64-bit support, so they could not link with the application on the 64-bit simulators). Once those were fixed, we still had a single warning, caused by the fact that we don't have a valid signing identity. That went away as soon as we set up our team and updated Xcode's provisioning profiles.

That just leaves our four tests. Three of them pass. However, since we never finished writing the fourth test, it continues to fail.

As you can see, bots are a powerful addition to our developer toolkit. This is a 1.0 developer product, so we should expect a few rough patches. For example, we are somewhat limited in the amount of data we receive. The bots tell us which tests passed and which ones failed (and on which devices). The information is easy to find and easy to read. Still, many useful statistics, like code coverage, are simply not available.

The bottom line, between Xcode server and bots, there's no reason that even small one- and two-man development teams cannot set up remote repositories and automatic testing. More testing means fewer bugs, and that means better programs for everyone.



NOTE: With previous versions of Xcode, it was difficult, if not impossible, to run unit tests from the command line. Fortunately, the command line tools in Xcode 5 are considerably better than those provided with previous versions. We can now use `xcodebuild` to build and test our apps, opening up the possibility of rolling our own continuous integration server. It won't be as simple as Bots—but it may be necessary, either if we want services that Bots do not provide, or if we need to mesh with an existing CI server.

PERFORMANCE TESTING

For iOS, performance testing really means running Instruments. Instruments provides tools for dynamically tracing and profiling code running either in the simulator or on an iOS device. It comes with a number of different profiling tools (somewhat confusingly also called instruments), that let us track and record data on everything from object allocations and timer-based sampling to file access and energy use.

Given Instrument's breadth, we're only going to cover the basics here, focusing on analyzing both the application's CPU performance and its memory usage. More information can be found in Apple's Instruments User Guide.



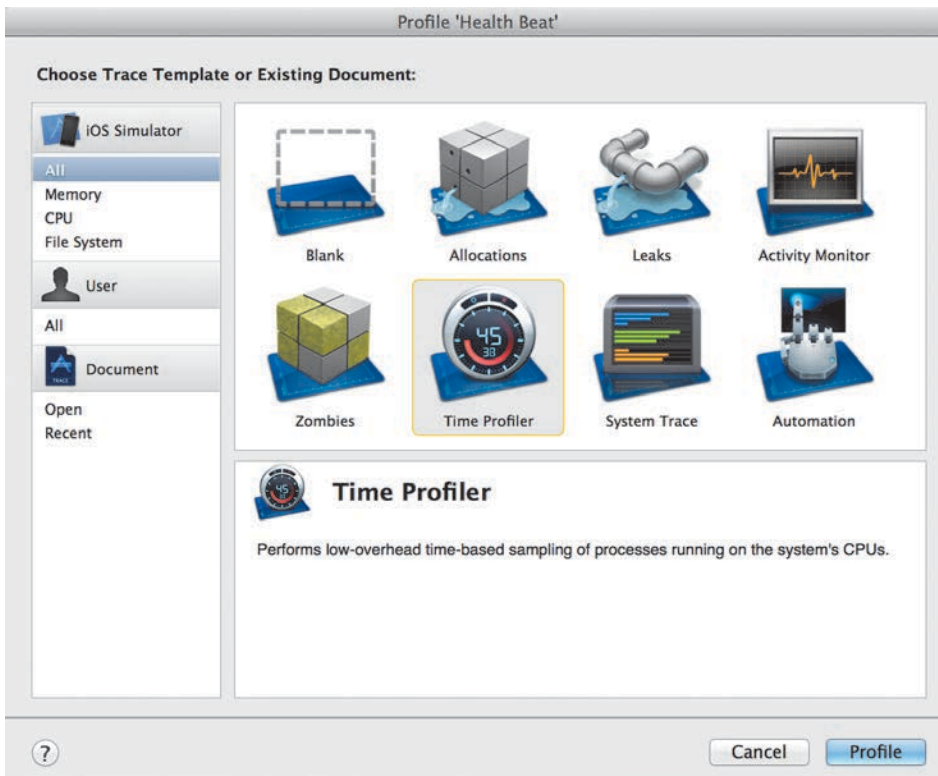


FIGURE 20 Selecting a trace template

NOTE: Not all instruments are available on both the simulator and the device. For example, most of the instruments that deal with file access or Core Data performance are only available on the simulator. On the other hand, most hardware-based instruments are only available on the device itself. This includes a number of instruments to test graphics performance, since these depend heavily the device's graphics accelerator.

The easiest way to start using instruments is to simply profile our application. Either select Product > Profile from the main menu, or click and hold on the Run button, and then select Profile from the drop-down list.

This will build our application for profiling, and launch Instruments. A window will pop up listing all the available trace templates (Figure 20). Select the Timer Profile template and click the Profile button.

NOTE: The templates do not cover all the possible tools in our Instruments library. Rather, they offer a quick way to jump into many common profiling tasks. Most of the templates load multiple tools at once, and we can further add or remove tools once Instruments is running. We can even save our own collection of tools as a custom template, making it easier to reuse those tools in the future.

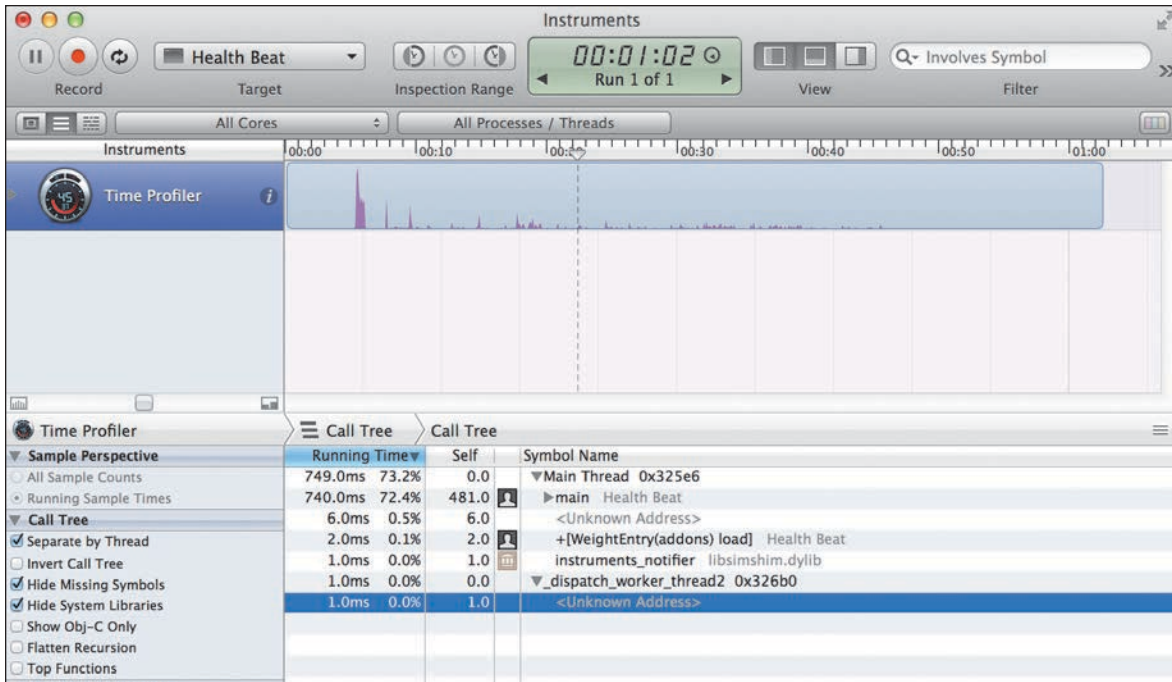


FIGURE 21 Analyzing the Time Profiler

Instruments starts and automatically launches our app. Depending on the template we chose, it will have one or more tool tracks open, each of these recording data. We should then use the app for a bit, performing any actions we wish to test. Then press the Stop button (Figure 21).

As you can see, we are running a single instrument, the Time Profiler. At the top, we have a graphical representation of our application's CPU usage. There is brief spike when the app first launches. Then we have a number of smaller spikes as we're actually using the device.

On the bottom half of the screen, we have detailed information about the selected instrument. When I am using the Time Profiler, I always start by checking Separate by Thread, Hide Missing Symbols and Hide System Libraries. This helps focus my attention on just my own code. We can always add the other information back later, once we've started digging into the data.

I also prefer to drill down through my method calls from the top down. However, if you prefer a bottom-up view, you might want to select Invert Call Tree. In fact, moving between the regular and inverted call trees can sometimes reveal problems that wouldn't be obvious when working in just one view.

As you can see, we have a lot of information about our call tree. As a first pass, I tend to look at the percentages. We're looking for bottlenecks—any method that is taking up an unexpectedly large percentage of our time. We can then try to optimize that code to improve our performance.

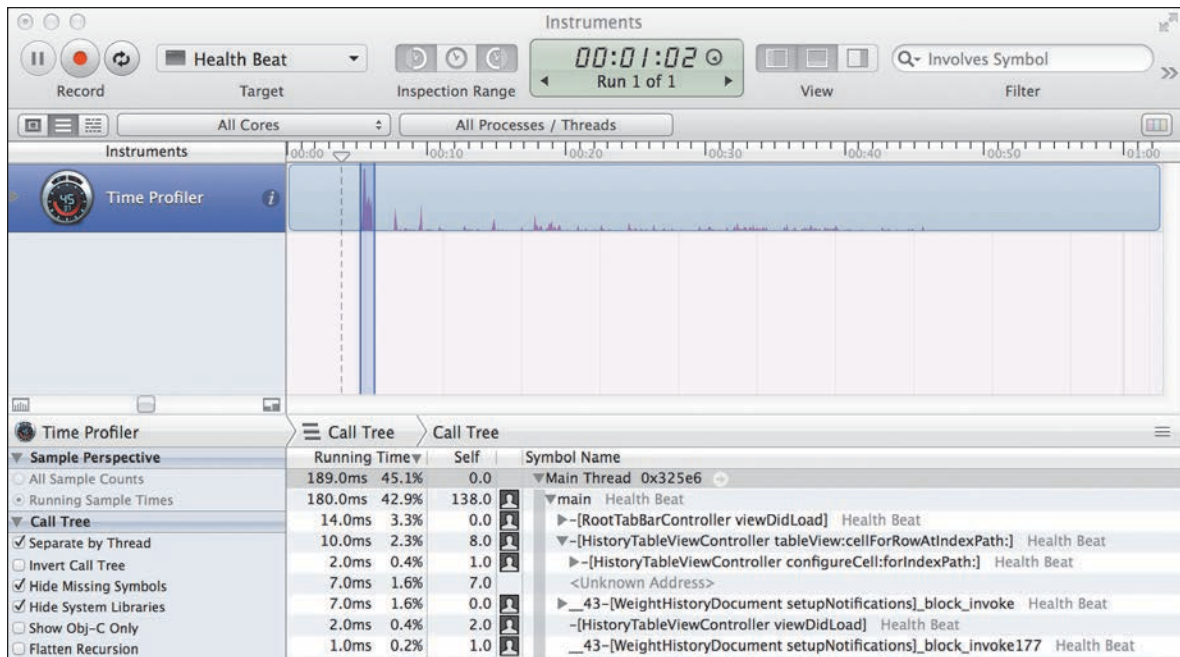


FIGURE 22 Examining data over a specific range

NOTE: The Time Profiler uses timer-based sampling to calculate the usage values. It periodically examines the application and records the stack trace at that point in time. As a result, methods that execute very quickly may occasionally not show up on the list. Normally, this isn't a problem since we're only interested in large bottlenecks.

We can also zoom in on a particular part of the trace. We can use the Three clock-icon buttons in the toolbar to set a specific range. For example, to examine the startup performance, click the mouse over the timeline to position the cursor just before our initial peak. Click the left clock icon. This will filter out everything before the cursor. Now, move the cursor to the other side of the startup peak. Click the right clock icon. This will filter out everything after the cursor. Clicking the center clock icon then clears the selected range.

NOTE: You could do the same thing by option-clicking and dragging on the instrument graph (the graph itself, not the timeline). This will select the highlighted time range.

The detail view now only shows information about the selected range (Figure 22). As you can see, we spent 8.0% of our startup time in our `[HistoryTableViewController tableView:cellForRowAtIndexPath:]` method. That's actually pretty good. Still, if we need to optimize our startup code that might be a good place to look.

FIGURE 23 Side-by-side comparisons.

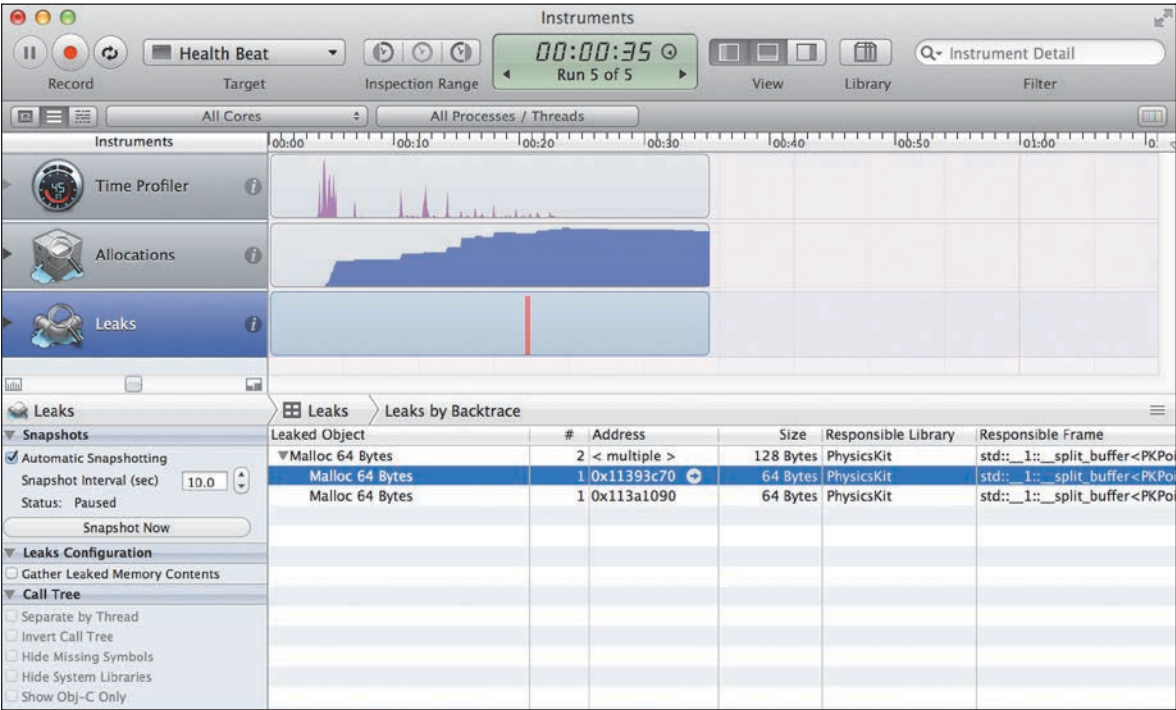
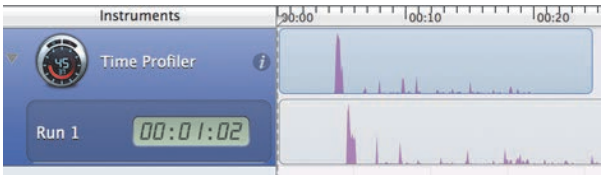


FIGURE 24 Profiling with multiple instruments

After changing our code, we can execute the test again. Simply click the Record button in Instruments. This will re-launch our application. After performing the second test, we can click our Time Profiler’s disclosure arrow to view our two tests side-by-side (Figure 23).

As you can see, performance can vary from run to run. Sometimes it can very considerably—especially if one of the frameworks ends up doing extra cleanup or maintenance in the background.

Typically, we will also want to monitor our device’s memory usage and look for potential problems. We could simply close Instruments, and profile our app again, selecting the Allocations template. However, it’s nice to run these tests together. Let’s add a few more instruments to our existing test.

Click on the Library button in Instrument’s toolbar. Scroll through the list of instruments until you find Allocations and Leaks. Double click on each of these to add them. Now click the record button and run our test again (Figure 24).

Statistics Allocation Summary						
Graph	Category	Live Bytes	# Living	# Transient	Overall Bytes	# Overall # Allocations (Net / ...)
<input checked="" type="checkbox"/>	All Heap & Anonymous VM	7.40 MB	41,675	92,755	22.48 MB	134,430 +++
<input type="checkbox"/>	All Heap Allocations	2.86 MB	41,599	92,604	13.07 MB	134,203
<input type="checkbox"/>	All Anonymous VM	4.54 MB	76	151	9.41 MB	227
<input type="checkbox"/>	VM: CoreAnimation	2.39 MB	36	10	2.78 MB	46
<input type="checkbox"/>	VM: CG raster data	1.14 MB	22	37	1.97 MB	59
<input type="checkbox"/>	VM: CoreServices	432.00 KB	1	1	864.00 KB	2
<input type="checkbox"/>	Malloc 2.00 KB	288.00 KB	144	98	484.00 KB	242
<input type="checkbox"/>	VM: SQLite page cache	256.00 KB	2	2	512.00 KB	4
<input type="checkbox"/>	Malloc 32 Bytes	248.53 KB	7,953	13,310	664.47 KB	21,263
<input type="checkbox"/>	CFString (immutable)	216.48 KB	5,705	10,851	785.47 KB	16,556
<input type="checkbox"/>	Malloc 1.00 KB	181.00 KB	181	354	535.00 KB	535
<input type="checkbox"/>	Malloc 16 Bytes	128.41 KB	8,218	16,174	381.12 KB	24,392
<input type="checkbox"/>	VM: Allocation 128.00 KB	128.00 KB	1	0	128.00 KB	1
<input type="checkbox"/>	Malloc 62.50 KB	125.00 KB	2	0	125.00 KB	2
<input type="checkbox"/>	Malloc 4.00 KB	104.00 KB	26	362	1.52 MB	388
<input type="checkbox"/>	CFString (store)	99.47 KB	754	1,249	267.64 KB	2,003
<input type="checkbox"/>	Malloc 64 Bytes	98.25 KB	1,572	3,268	302.50 KB	4,840
<input type="checkbox"/>	Malloc 8.00 KB	88.00 KB	11	5	128.00 KB	16
<input type="checkbox"/>	sfnt_name_t	85.16 KB	703	0	85.16 KB	703
<input type="checkbox"/>	VM: CoreUI image data	84.00 KB	7	4	100.00 KB	11

FIGURE 25 The allocation instrument's detail view

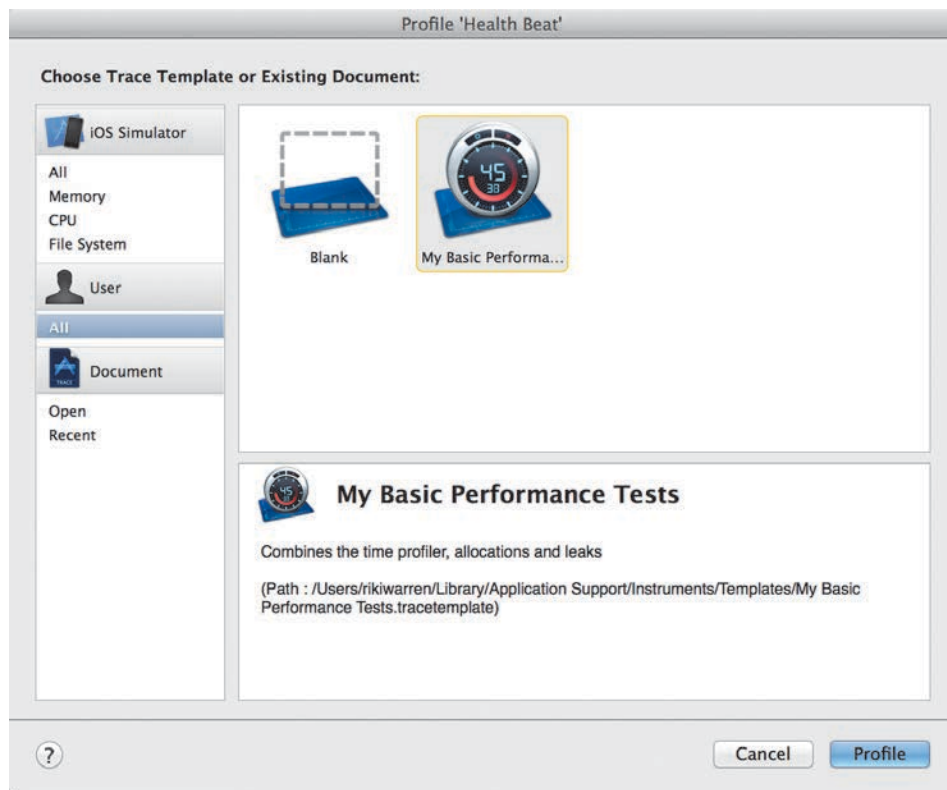
We shouldn't get any memory leaks, since we're using ARC. Still, a single leak shows up. Examining this further, we see that the system is mallocing 64 bytes of memory somewhere deep in the PhysicsKit framework. This is presumably a private framework used by UIKit Dynamics. Since it's not in code that we can control, there's really nothing that we can do about that. It's either a bug in the system or a false positive. Either way, it's small. We can safely ignore it.

The Allocations track is much more interesting. By default the graph shows all allocations. As you can see, our memory usage is, in general, increasing over time, using a total of 7.4 MB.

In the detail view, we have a large number of statistics about our memory usage (**Figure 25**). We see a list of all the different objects and C-code mallocs that our application has performed. We can list the number of bytes currently used by that data type, the number of instances that are currently live, or the overall number of instances that we have created (including instances that have since been deallocated). We can also sort by all these values. For example, sort by Live Bytes to see which objects are taking up the most memory. Or sort by the overall number of allocations, and look for any places where we are allocating a large number of objects but never deallocating them. It's also good to sort by category and search for our own classes, just to make sure nothing's misbehaving.

Let's save our tests. File > Save and File > Save As... will let us save all the data from our current session. This lets us reload the data and compare it against future tests. File > Save As Template... will save our current set of instruments as a template, letting us run the same set of tests again in the future. For example, I have saved the session data as HealthBeat1, and saved the tests as a template named My Basic Performance Tests.

FIGURE 26 Opening a custom test template.



Now, if I quit Instruments, and then profile again from Xcode, my template shows up under User > All (**Figure 26**). I can also reopen the old session under Document > Recent.

In many ways, profiling is more an art than a science. I highly recommend that you spend some time playing with the tools. The more familiar you become with them, and the more you explore their features and options, the better you will be at finding and fixing potential problems.

USING THE DEBUGGER

The debugger lets us examine running programs, halt execution, check values and step through the code one operation at a time. We typically interact with the debugger by setting breakpoints.

We can set a breakpoint at a line of code by simply clicking in the margin beside our code (just to the left of the line numbers, if you have line numbering turned on). A blue arrow will appear where we clicked (**Figure 27**).

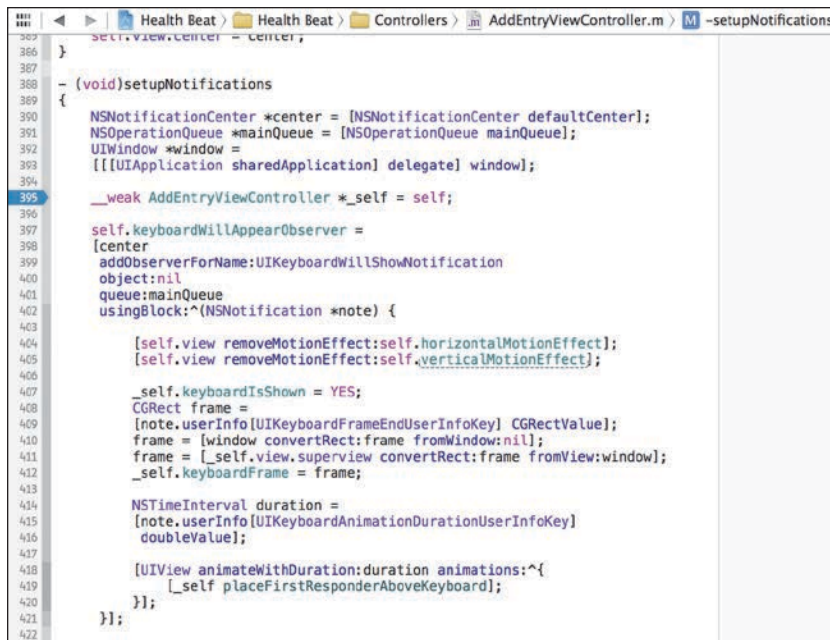


FIGURE 27 Setting a breakpoint

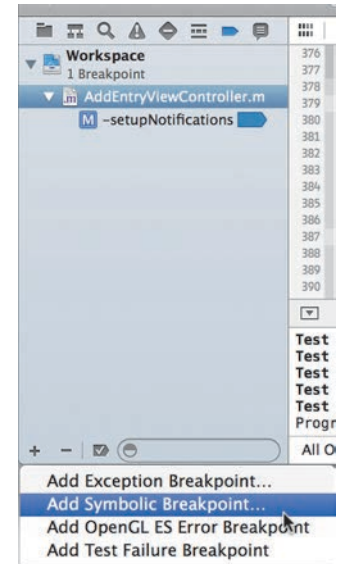


FIGURE 28 Adding Special Breakpoints

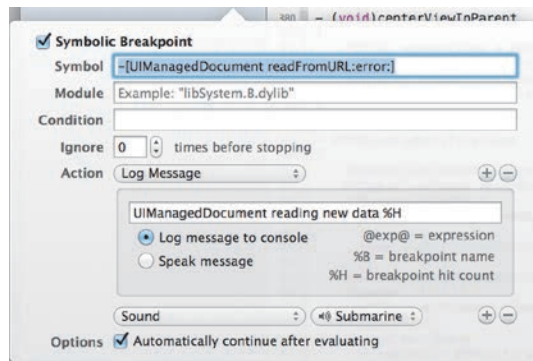
Additionally, we can add a few special breakpoints. Switch to the Breakpoint navigator. Now press the “+” icon in the bottom right corner. This gives us the option to add an exception breakpoint, a symbolic breakpoint, an OpenGL ES breakpoint or a unit test failure breakpoint (Figure 28).

If we add an exception breakpoint, we can configure it to break on either Objective-C exceptions, on C++ exceptions or on all exceptions. We can also select whether we break when the exception is thrown or when it is caught. By default they break whenever any type of exception is thrown.

Adding a breakpoint when exceptions are thrown can be very helpful when developing code. If an exception occurs, this will let us explore the stack trace and figure out exactly what went wrong. Unfortunately, it can also be misleading—especially if you’re using third-party libraries. The breakpoint will be triggered, even if the code eventually catches and handles the exception. Apple tends to only use exceptions in truly exceptional circumstances, so these breakpoints generally work fine. Unfortunately, a few third party developers use exceptions where they really should be using conditional statements—causing a few false positives.

Symbolic breakpoints let us specify a method name, the method from a particular class, or a function name. Whenever the system encounters that symbol, it triggers our breakpoint. This is great for placing breakpoints when we don’t have access to the source code. For example, we could place a breakpoint on `-[UIManagedDocument readFromURL:error:]`. This breakpoint will be triggered whenever our `UIManagedDocument` tries to read new data.

FIGURE 29 Breakpoint actions



DEBUG VS. RELEASE BUILDS

By default Xcode defines two different build types for our application: debug and release. The default scheme always runs our code in debug mode, but profiles it in release mode. Xcode will trigger breakpoints regardless of the mode; however, debug mode provides us with a little more information. In release mode, all debugging symbols are stripped from the application, and the final code is optimized. In some cases, this can make it harder to analyze our application's state or to step through our code.

The other two types of breakpoints are used less frequently. The OpenGL ES breakpoints are typically only used by developers working directly with OpenGL ES. While the Test Failure Breakpoints will halt our unit tests whenever a test fails.

By default, breakpoints cause our application to halt. We can then examine the current state, explore the stack trace, and step through our application. However, we can change this default behavior. Right click on the breakpoint (either in the margin or in the Breakpoint navigator) and select Edit Breakpoint....

This lets us modify our breakpoints in a number of ways. We can add a condition, which must be true for our breakpoint to be triggered. We can also set the number of times our breakpoint will be ignored before it's finally triggered. The most interesting option, however, is the ability to add actions to our breakpoints.

Actions include running an Applescript, capturing an OpenGL frame, executing a debugger command, logging a message to the console, running a shell script or playing a sound. We can also decide whether or not execution halts once the breakpoint is triggered.

For example, say we just want to be notified whenever our application loads new data. We don't actually want the application to stop—we just need some sort of notification. Let's edit the UIManagedDocument breakpoint we set earlier. Add two actions. In the first, log the following message to the console: **UIManagedDocument reading new data %H**. In the second, have it play a sound. Now, check the Automatically continue after evaluating actions option (Figure 29). This will cause Xcode to beep and print out the message (including the number of times the breakpoint has been triggered), but it won't stop the application.

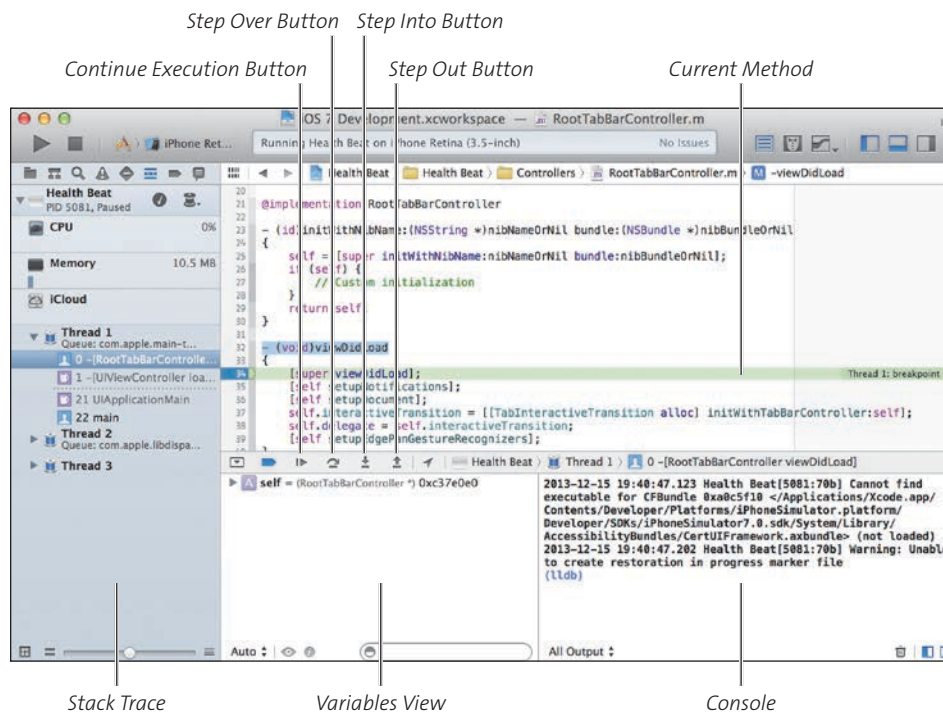


FIGURE 30
The debugger in action

If our application halts, Xcode should automatically move us to the Debug navigator and open the Debug area. This places a lot of information at our fingertips (**Figure 30**). In the Debug navigator, we can see the current state of our stack traces. The thread that triggered the breakpoint should be open, with the current method highlighted. We can click on other methods in the stack trace to move back through our program's history. This can be more or less useful depending on the depth of the stack trace, and whether the stack trace is filled with our own methods, or with calls to private frameworks. Typically, we can't get many useful details from calls to Apple's frameworks or third-party libraries.

The debug area is divided into two sections. On the left, we have our variables view. It lists the current state of all the variables in scope. On the right we have the console. We can also hide either of these sections, letting the other use the entire space.

On the mini-bar above the debug area, we have a number of buttons. We can use these to navigate through our code. In order, the buttons let us continue execution, step over the next method, step into the next method, or step out of the current method (returning back to the calling method).

The console contains all the output from our `NSLog()` calls, as well as any system log messages. It is also an active control. We can type debugger commands directly into the console. For example, `po self` will print out the current `self` object. We can even make Objective-C method calls—though the console sometimes doesn't understand the dot-syntax. Still, to view our current document we could type `po [self document]`.



I often find it easier to print information in the console than to search for it in the variables window. This is particularly true when it comes to digging through multiple layers of instance variables, exploring the contents of collections, or any dynamically calculated values.

Debugging is another broad topic, with a number of fascinating nooks and crannies. The information here will get you started, but you will probably need to do some additional reading and some experimentation before you really master it.

WRAPPING UP

This chapter covered three major topics in software engineering—source control, testing and debugging. Each of these topics represents a vital skill for the professional developer. They won't help you build your app directly—but they will help you manage your projects and produce bug-free, high-performance code.

OTHER RESOURCES

For more information, check out the following resources:

1. **Understanding Source Control in Xcode**

WWDC 2013 Videos

An in-depth exploration of the source control features found in Xcode 5.

2. **Xcode Continuous Integration Guide**

iOS Developer's Library

This contains a wide range of information on setting up an OS X Server and Bots for continuous integration.

3. **Instruments Users Guide**

iOS Developer's Library

This guide provides a detailed overview on using Instruments to profile and analyze our applications. This includes a complete description of all the tools included in Instruments, and a list of the different settings we can use to configure those tools.

4. **iOS Debugging Magic**

iOS Developer's Library

While this is somewhat out of date, it still contains a wealth of outstanding hints, tricks and tips for effective debugging. This includes a number of more-advanced debugging techniques.