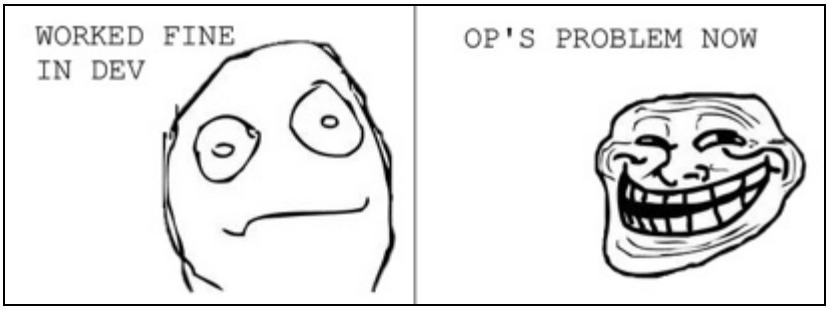
Chapter 23: Beginning Automated Testing with XCode

By Charlie Fulton

All developers have to test their software, and many of the smart ones create **test suites** for this purpose. A test suite is a collection of test cases, also known as **unit tests**, targeting small "units" of code, usually a specific method or class.

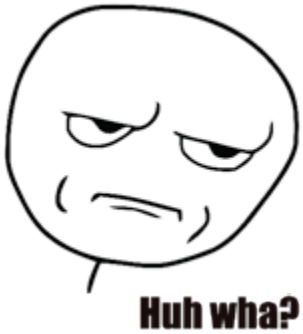
Unit tests allow you to employ test-driven development practices and ensure that your code has fewer bugs. And by creating a unit test for each bug found, you can help make sure the bug never comes back! By unit testing and debugging proactively, you significantly narrow the chances that your app will behave unexpectedly out of the development environment.



But as you know, we're often rushed to get finished – or we get lazy – so automating your unit tests to run automatically each time you change your code is even better. This is especially important when you're working as part of a team.

So if you could put in place a system to automatically build, test, and then submit your app to beta testers, would you be interested? If so, then read on!

This bonus chapter and the next will walk you through setting up an automated building and testing system for your iOS apps. You will take an example project, add some unit tests to it, add a remote GitHub (github.com) repository, and then set up a continuous integration (CI) server with Jenkins. Jenkins will periodically check your GitHub repository for changes and automatically build, test, and submit your app to TestFlight (testflightapp.com)!



If you're new to automated testing, you might be wondering what some (or all) of the above terms are, including GitHub, Jenkins, and TestFlight. We'll go into each of these in detail in this chapter, but for now here are instant micro-introductions to these tools:

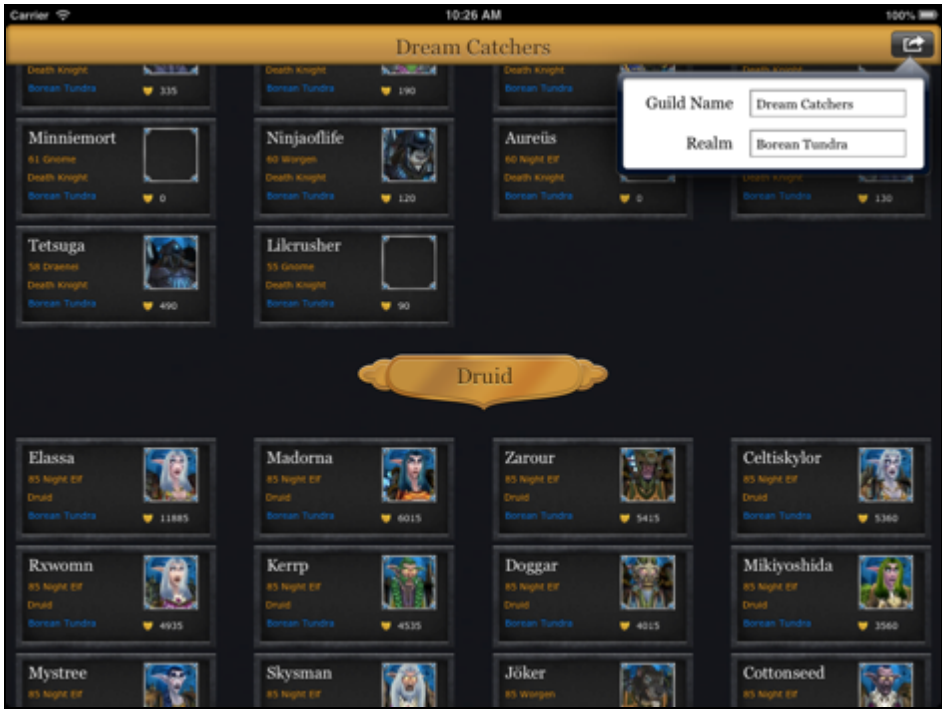
- **GitHub –** Free public Git repositories, collaborator management, issue tracking, wikis, downloads, code review, graphs, and much more!
- **Jenkins –** A continuous integration server and scheduler that can watch for changes to Git repositories, kick off builds and tests automatically, and notify you of results.
- **TestFlight –** A service to automate the sending of iOS apps over the air to testers, and perform crash log analysis.

That ought to tide you over for a short while, giving me time to introduce the geektastic example project that will be your vehicle for this educational odyssey – but be ready to learn more about all of this in the coming pages!

Introducing GuildBrowser

In this chapter, you will be creating an automated build and test system for an app called **GuildBrowser.** GuildBrowser is a simple app that lets you browse the members of a guild from the popular game World of Warcraft. You can enter a guild

and a realm name to see all of the members listed in a `UICollectionViewController`, as you can see below:



The app is driven by the freely-available WOW API from Blizzard. You will be using the AFNetworking framework to make RESTful calls to the WOW API, retrieving JSON data that contains guild and character information.

If you never have time to play games, the next best thing is making apps that help with games!

> **Note:** For more details about the WOW API, check out Blizzard's GitHub project (https://github.com/Blizzard/api-wow-docs), the official WOW API docs (http://blizzard.github.com/api-wow-docs/), and the WOW forum devoted to the community platform API (http://us.battle.net/wow/en/forum/2626217/).

Gitting started

If you're not using some form of version control for your code these days, then you're definitely not doing it right! Arguably the most popular system right now is **Git.** Linus Torvalds (yes, he who created Linux) designed and developed Git as a better system than its predecessor, SVN, to manage the code for the Linux kernel. Not only is git a great source control system, but it's integrated directly into Xcode, which makes it very convenient to use in iOS development.

Git is a "distributed" version control system, meaning that every Git working directory is its own repository, and not dependent upon a central system.

However, when you want to share your local repository with other developers, you can define a connection in your "remotes" setting. Remote repositories are versions of your project that are usually set up on a central remote server. These can even be a shared via a Dropbox folder (although this isn't particularly recommended).

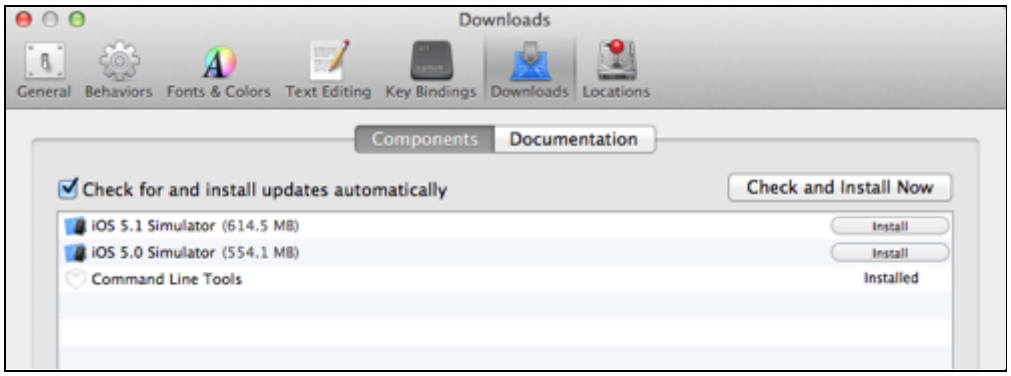
A number of services have emerged in the last few years to make it easier to set up remote repositories for Git, and other distributed version control systems. Github has become one of the best and most popular of these services, and for good reason. It offers free public Git repositories, collaborator management, issue tracking, wikis, downloads, code review, graphs and much more… sorry to sound like an advertisement, but Github is pretty awesome!

> **Note:** For a more detailed description of Git and how you can work with it, check out the tutorial on How To Use Git Source Control with Xcode in iOS 6, coming soon to raywenderlich.com.

Your first action in this chapter will be setting up a remote repository (**repo** for short) on GitHub. It will be your remote "origin" repo. It will allow you to work locally as normal, and then when you are ready to sync with teammates or send a new build to testers, you'll push all of your local commits to this repo.

> **Note:** This chapter assumes you already have a GitHub account with SSH key access setup. If you do not, please consult this guide (https://help.github.com/articles/generating-ssh-keys), which shows you how to get set up by generating and using an SSH key pair. You can also use an https URL with your GitHub username/password, but the guide assumes SSH key access setup.

First, make sure you have the command line tools installed, choose the **Xcode/Preferences/Downloads** tab, and make sure your screen looks similar to this (the important thing is to have **Installed** next to Command Line Tools):



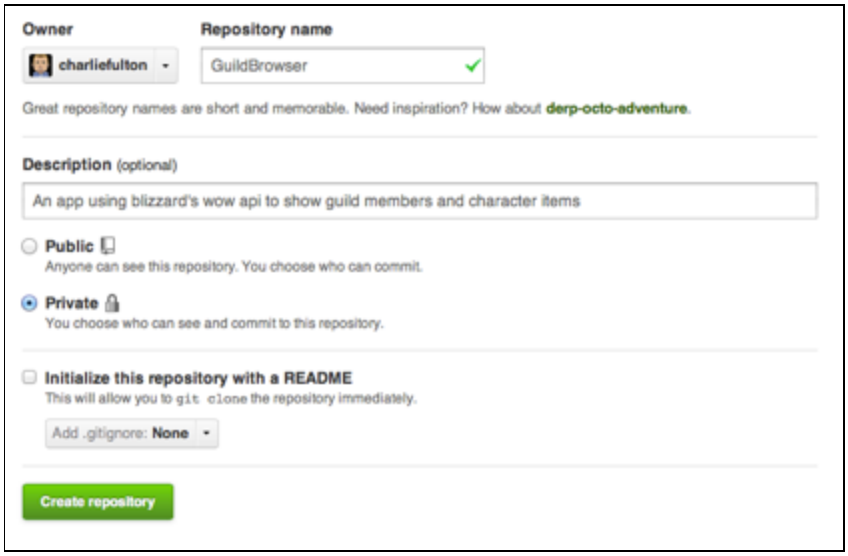
This chapter comes with a starter project, which you can find among the chapter resources. Copy the project to a location of your choice on your hard drive, and open the **GuildBrowser** project in Xcode.

The GuildBrowser app was created with the Single View Application template using ARC, Unit testing, Storyboards, iPad-only, and a local Git repository. Compile and run, and you should see the app showing some characters from the default guild.

Open a browser, go to github.com and login to your account, or create an account if don't have one already. Then, click on the **Create a New Repo** button on the top-right corner:

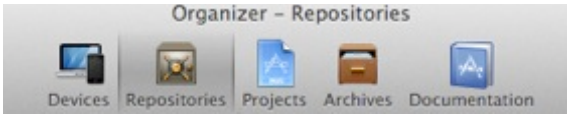
Enter whatever you like for the **Repository name** and **Description** fields – the name does not actually have to correspond with the name of your Xcode project. Then click the **Create repository** button.

> **Note:** Do **NOT** check the **Initialize this repository with a README** box. You want to create an empty repo and then push your Xcode project into it.

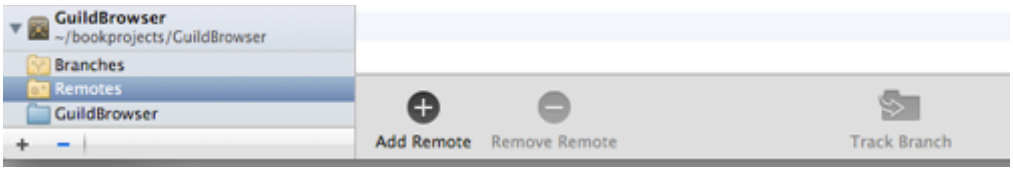


Get a copy of the repository SSH URL by clicking the **copy to clipboard** button or typing ⌘–C (remember to tap the **SSH** button first to switch to the SSH URL).

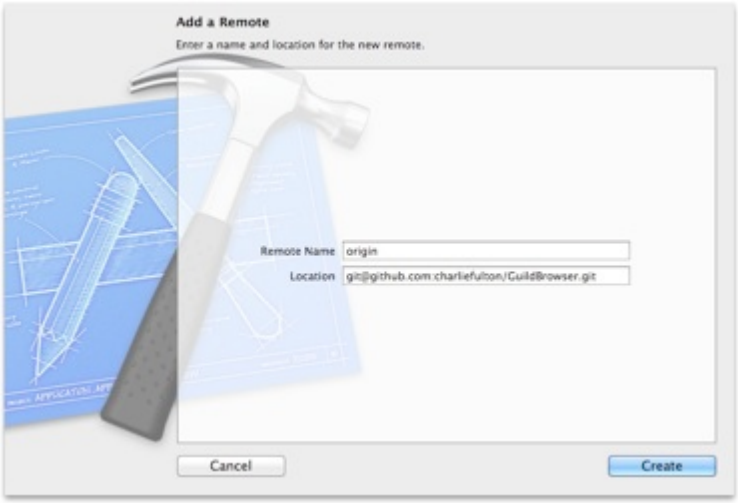
Now open **Organizer (Cmd-shift-2)** and then select the **Repositories** tab.



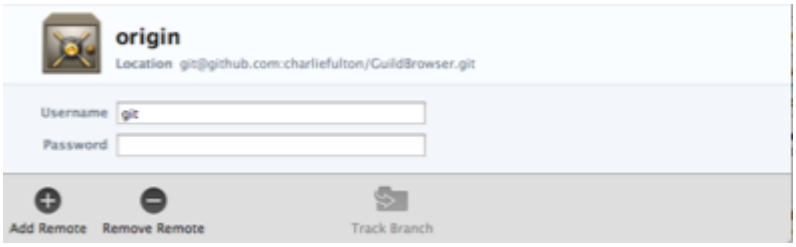
You should see the GuildBrowser project listed at the bottom. Click on the **Remotes** folder and then click the **Add Remote** button.



Enter **origin** for the Remote Name, and paste in your personal GitHub URL that you copied earlier into Location:



Click **Create**, and Organizer should look like the following:

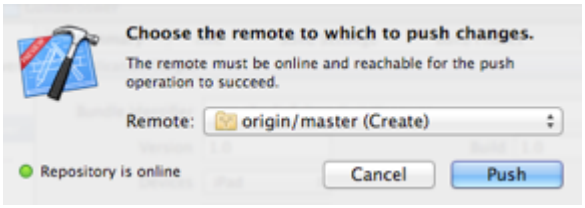


Don't be alarmed that there is no password entered for the origin location. You'll be using Git's SSH access, so no password is needed.

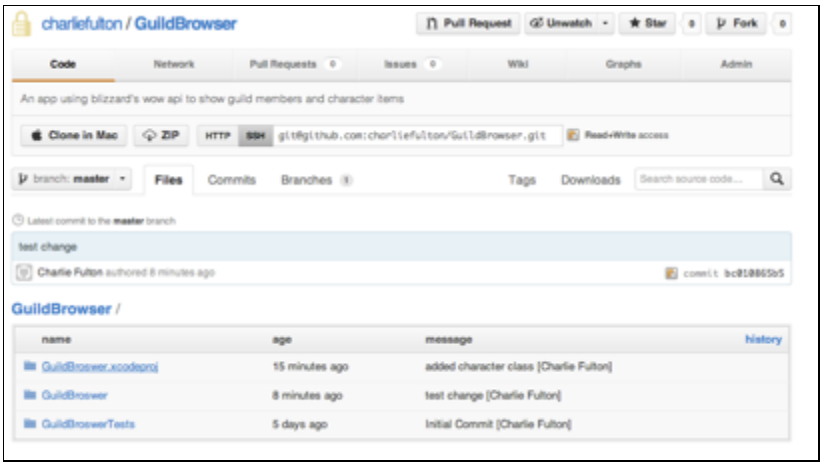
You now have a remote repo set up! You can use these steps for all of your Xcode projects. This is great, but you still have not pushed anything from your local Git repo to the remote Git repo. Let's do that now with Xcode.

Go back to your project and first make sure you have committed all of your local changes. Xcode will not allow you to push to a remote repo until you have done that. Here are the steps:

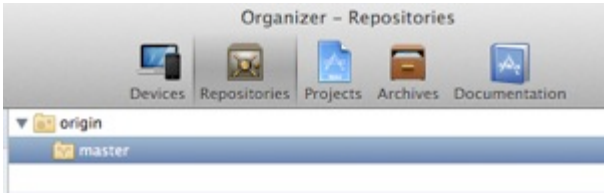
1. Go to **File\Source Control\Commit (⌥⌘–C)** and on the following screen, enter a commit message like "initial commit" and click **Commit.**
2. Now you need to push the local master branch to the remote origin**/**master branch. Go to **File\Source Control\Push** and click **Push**.
3. Since this is the first time you're pushing to this repo, you should see **(Create)** beside the Remote name, **origin/master**. Tap the **Push** button, and Xcode will handle the rest.



Once the push is complete, go back to your browser, refresh the page, and the changes you committed should now show up in your GitHub repo:



You should also now see your remote repo and its master branch listed in **Organizer\Repositories\Guild Browser project\Remotes**:



Congratulations, you know how to connect your local Git repo to a remote repo on GitHub and push changes to it!

Now that you have a GitHub repository set up, you could invite others to clone your repository, accept pull requests, or take a look at your code. When someone else pushes their commits to the repo on GitHub **origin/master** repo you go to **File\Source Control\Pull (⌥⌘–X)**, in the popup Remote should be set to **origin/master**, select **Choose** and those changes will be added to your project.

To see what has changed, go to **Organizer\Repositories\GuildBrowser**, expand the arrow and click on the **View Changes** button.

Continuous integration

Continuous integration (CI) is having your code automatically built and tested (and even deployed!) by a server that is continuously watching your revision control system for changes.

CI becomes tremendously important when you have a lot of developers on a project sending in their changes multiple times per day. If any conflicts or errors arise, everyone is notified immediately. Eventually, someone breaks the build!

Meet Jenkins

Jenkins (http://jenkins-ci.org) is the open source CI server you're going to use in this chapter to automatically build, test, and deploy code that is pushed to the GitHub remote repo.

You'll spend the rest of this section getting Jenkins up and running.

Installing Jenkins

There are a few different ways you can install Jenkins on a Mac:

1. You can use the installer from the Jenkins website, which will set up Jenkins to run as a daemon. This is nice because it is configured to use the Max OS X startup system, but it will result in you, the user, having to go through a few more steps in order to get Xcode building to work properly.
2. You can run the WAR file (a Java archive filetype) from the Terminal, which uses the built-in winstone servlet container to run Jenkins.
3. You can deploy the WAR file to an application container you have already set up, such as Tomcat, JBoss, etc.

In this chapter, you are going to run Jenkins from the Terminal, because it's the quickest and easiest way to do it. Download the Jenkins WAR file from http://mirrors.jenkins-ci.org/war/latest/jenkins.war. Open a Terminal window, and then run this command:

```
nohup java -jar ~/Downloads/jenkins.war --httpPort=8081 --
ajp13Port=8010 > /tmp/jenkins.log 2>&1 &
```

Note: The above assumes that you have the Jenkins WAR file in your Downloads folder. If you downloaded the WAR file to a different location, adjust the path in the command accordingly.

You might find it useful to create an alias for starting up Jenkins. From a Terminal, or text editor showing hidden files, open up **/Users/<username>/.bash_profile** and enter this line:

```
alias jenkins="nohup java -jar ~/jenkins.war --httpPort=8081 --
ajp13Port=8010 > /tmp/jenkins.log 2>&1 &"
```

Save .bash_profile, open a new Terminal, and now type in **jenkins** to start it up.

This starts the Jenkins server on port 8081. The `nohup` command is a UNIX command that allows a process to keep running after you have logged out or the shell has exited – it stands for "no hangup".

The above command also sets up a log file to **/tmp/jenkins.log.** I like to configure the ports to avoid any conflicts with servers I have running. To access Jenkins, open the following URL in a browser:

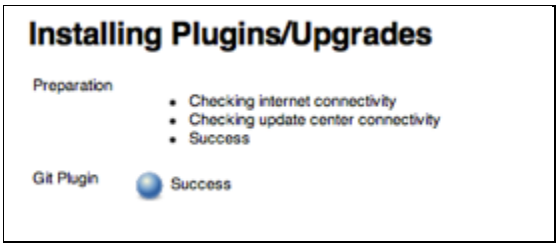
```
http://localhost:8081/
```

Configure Jenkins plugins

You need to add a few things to Jenkins before you can start setting up jobs. Open the Jenkins dashboard (via http://localhost:8081) and click on the **Manage Jenkins\Manage Plugins\Available** tab. There are a lot of plugins available, so filter the choices down by typing **Git** into the filter search box in the top right.

Here's what you want:

Check the box next to the **Git Plugin** and then click the **Install without restart** button**.** You should see the following screen when it's done installing:



Now enter **Chuck Norris** in the filter search box, and look for the Chuck Norris quote plugin. While this one is optional for the chapter, I highly recommend you check it out! If you don't install it, he will know! Beware the wrath of Chuck. ☺

> **Note:** My friends call me Charlie, but my real name is Charles. My Dad's middle name was Norris. So I had legitimate claims to naming one of my sons Chuck Norris Fulton… but unfortunately **she** wouldn't let me. ☺

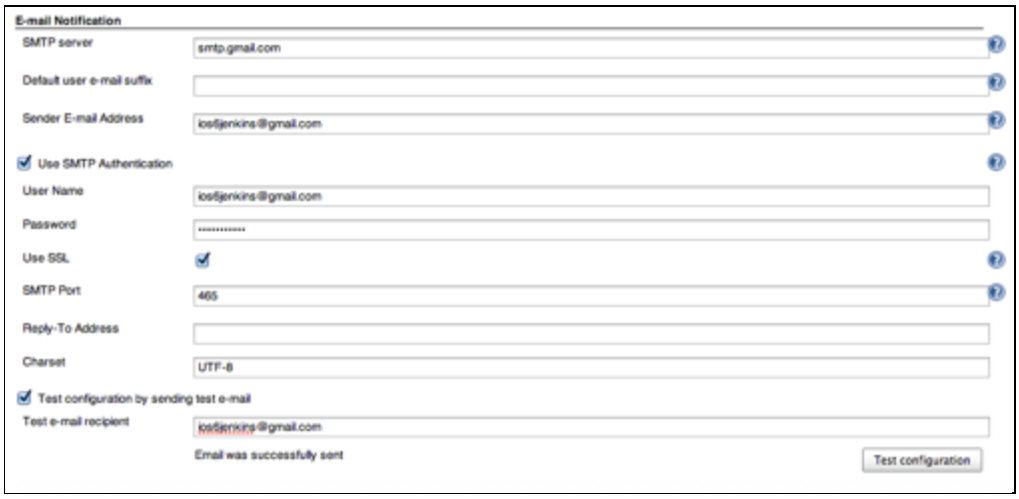
Setting up Jenkins email notification

It would be nice to have an SMTP server so that Jenkins can send you build notification errors. But if you're not running an SMTP server, you can use your Gmail account with Jenkins (or any other SMTP server you prefer).

Go to **Manage Jenkins\Configure System** and in the **Email Notification** section, enter the following settings (use the **Advanced Settings** button to access some of the settings):

- **SMTP Server** : smtp.gmail.com
- **Sender Email address** : <your Gmail address>
- **Use SMTP Authentication**
- **User Name**: <your full Gmail address>
- **Password**: <your Gmail account password>

- **Use SSL**
- **SMTP Port: 465**



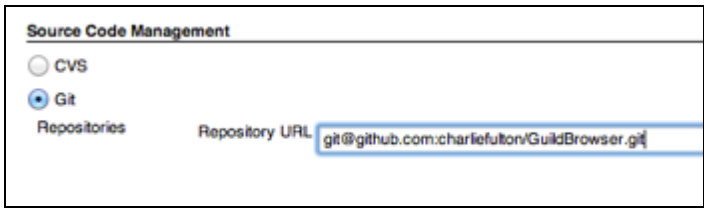
Now test out email notifications by checking the **Test configuration by sending test-email,** entering an email address, and clicking the **Test configuration** button. Once you've confirmed that the test is successful, remember to tap the **Save** button to save your configuration changes.

Now you can receive emails from Jenkins about build failures.

Creating a Jenkins job

You are now ready to create a Jenkins job! Open the Jenkins dashboard, click on **New Job,** enter **GuildBrowser** for the job name, choose the **Build a free-style software project** radio button, and finally click the **OK** button. You should now see the job configuration screen.

In the **Source Code Management** section, click the **Git** radio button, and enter your GitHub remote repo (origin/master) URL into the **Repository URL** text field. Your screen should look something like this:



Go to the **Post-build Actions** section, select **Email Notification** from the **Add post-build action** button, enter the email address(es) for the recipient(s), and finally check **Send email for every unstable build**. You should see the following:

If you installed the Chuck Norris plugin, you can also add a post-build action to **Activate Chuck Norris.** That sounds kinda scary though! Warning, don't break the build!!

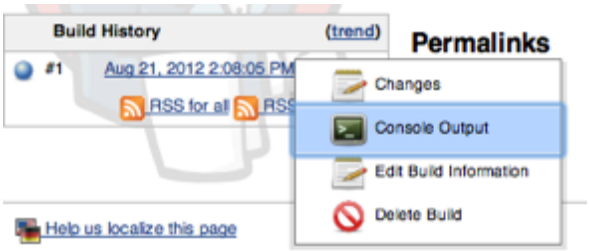
> **Note:** The programs that Chuck Norris writes don't have version numbers because he only writes them once. If a user reports a bug or has a feature request, they don't live to see the sun set. ☺

Don't forget to click the **Save** button.

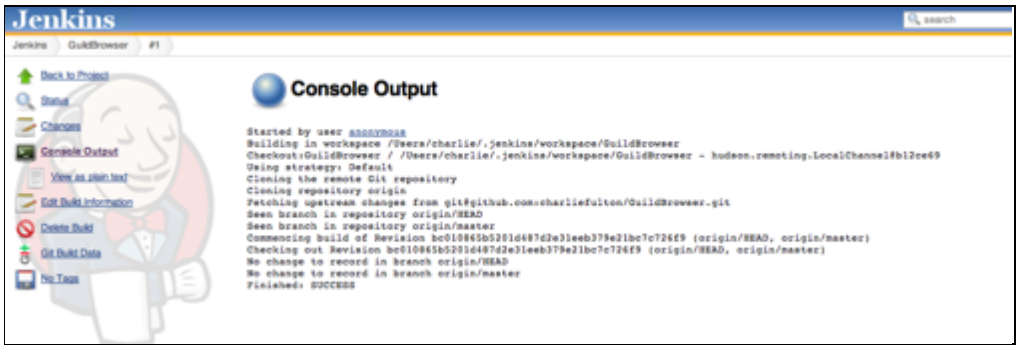
Let's make sure everything is set up correctly with GitHub by clicking the **Build Now** button on the left sidebar. When the job starts, you will see an update in the Build History box:

If everything works correctly, your build #1 should be blue, indicating success:

To see what exactly happened during the build, hover over the build and look at the **Console Output**:



Your console output should look similar to this, showing that the job successfully pulled down the code from the GitHub repo:



So far, you've connected to GitHub with your job and set up email notification. Let's press on and add the interesting bits: testing, building, and uploading your project to TestFlight.

> **Note:** A Jenkins pro tip! To quickly get to the job configure screen, hover over the job name on the breadcrumb trail at the top of the screen and select **Configure** from the menu. This is especially helpful when you're looking at console output.

Command line tools

Before you begin to set up Jenkins to build your iOS app, let's talk a little bit about the command line tools that come with Xcode.

xcodebuild

`xcodebuild` is the command line tool for building, archiving, and querying your project or workspace. You're going to use it to build a project from Jenkins. You will soon see all the ways it can be put to work.

xcode-select

This sets the Xcode.app that is used by tools in **/usr/bin.** Open a Terminal and run the following command, and note the output:

```
xcode-select -print-path
/Applications/Xcode.app/Contents/Developer
```

This shows which Xcode.app will be used when running the tools found in **/usr/bin** like `xcodebuild`, `xcrun`, `opendiff`, `instruments`, `agvtool`, and `git`.

When you have multiple versions of Xcode installed, like a beta release, it's really important for your CI server to have this specifically defined. You do that by setting the `DEVELOPER_DIR` environment variable and running all of your build commands through the `xcrun` command that comes with Xcode.

xcrun

This finds or runs tools inside Xcode.app. Apple recommends running all command line tools (such as the git command line tool) through this command.

Note: It is possible to change the Xcode.app that is configured. At the time of this writing, I had both 4.4.1 and 4.5 beta4 installed.

The following sequence of commands shows switching between Xcode 4.5 beta4 and Xcode version 4.4.1.

```
xcode-select -print-path
```

Output: /Applications/Xcode45-DP4.app/Contents/Developer

```
xcodebuild -version
```

Output: Xcode 4.5

Build version 4G144l

```
sudo xcode-select --switch /Applications/Xcode.app/Contents/Developer/

xcode-select -print-path
```

Output: /Applications/Xcode.app/Contents/Developer

```
xcodebuild -version

Output: Xcode 4.4.1

Build version 4F1003
```

Automating xcodebuild from Jenkins

Now that you know a little more about `xcodebuild` and the other command line tools, let's update your Jenkins job to do the build. Go to the **Jenkins Dashboard\GuildBrowser job\Configure.** In the **Build** section, click on the **Add build step** dropdown select **Execute shell.**

Enter the following code into the Command window:

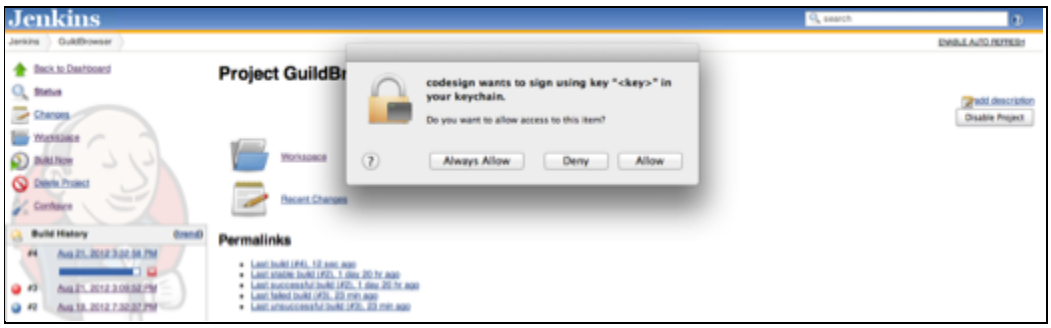
```
export DEVELOPER_DIR=/Applications/Xcode.app/Contents/Developer/

xcrun xcodebuild clean build
```

Your command window should look like this:

Now click **Save.**

Let's test the job again by clicking on **Build Now.** When the job starts executing, pay attention! You might see the following screen, asking you if you want to allow Jenkins access to run codesign so that it can sign your code. Select **Always Allow**.



If you missed the prompt (because you were chasing your two year-old who was running off with your iPhone, for example), you will see this message in your failed job's console output:

```
Command /usr/bin/codesign failed with exit code 1
** BUILD FAILED **


The following build commands failed:
       CodeSign build/Release-iphoneos/GuildBroswer.app
(1 failure)
Build step 'Execute shell' marked build as failure
```

If this happens, simply try running the job again, but don't miss the prompt (or let your two year-old run away with your iPhone) this time. ☺

Assuming you didn't miss the prompt, then in your output you will see a message similar to this:

```
/Users/charlie/.jenkins/workspace/GuildBrowser/build/Release-
iphoneos/GuildBrowser.app
Unable to validate your application. - (null)
```

This means the build was successful! You can safely ignore the "unable to validate" message – it's a known bug that we were able to verify at WWDC this year.

> **Note:** There is a Jenkins plugin for building iOS apps named Xcodeplugin. It works great, but I think it's best to understand exactly what is going on when building your app. You can achieve the same results as the plugin with some simple build scripts of your own.
>
> What's more, becoming dependent on the plugin carries some risks. If Apple changes how things work and that breaks the plugin, and it's no longer updated, you will still be able to adapt if you know how things work.

Unit testing

Unit testing ensures that the output of your code remains unchanged as you are developing it. After establishing expectations of what a class will do, you write unit tests to verify these expectations remain true. Unit testing also allows you to make changes incrementally, seeing results step-by-step, which makes it easier to develop new features and make big changes.

Some other benefits of including unit tests in your projects are:

- **Reliability** – With a good set of tests, you will most likely have fewer bugs. This is especially true with code that was developed with tests from the beginning, verifying every change and new bit of code along the way.
- **Code confidence** – It's a lot less nerve-wracking to go in and refactor a massive amount of code with unit tests in place, knowing that your code should still be producing the same results and passing tests.
- **Stability** – When bugs are found, new tests can be added ensuring the same bugs don't harass you again.

Xcode has built-in support for unit tests via the SenTestingKit framework. When writing unit tests, there are two terms to be aware of:

- **Test case –** The smallest component of a unit test is the test case. It is what verifies the expectations of what your unit of code should produce. For example, a test case might focus on testing a single method, or a small group of methods in your class. In Xcode, all test cases must start with the name **test.** You will see an example of this below.
- **Test Suite –** A group of test cases. This is usually a group of test cases against a particular class. In Xcode, there is a template for creating a test suite named **Objective-C test case class**. It produces a class that is a subclass of **SenTestCase**.

There are two general approaches to setting up unit tests: you can either test from the bottom up (testing each method/class individually), or the top down (testing functionality of the app as a whole). Both approaches are important – it's usually good to have a combination of both.

In this chapter, you are going to test from the bottom up, focusing on a few of the model classes. In the next chapter, you'll try your hand at testing from the top down, from the User Interface to the model code via UI Automation.

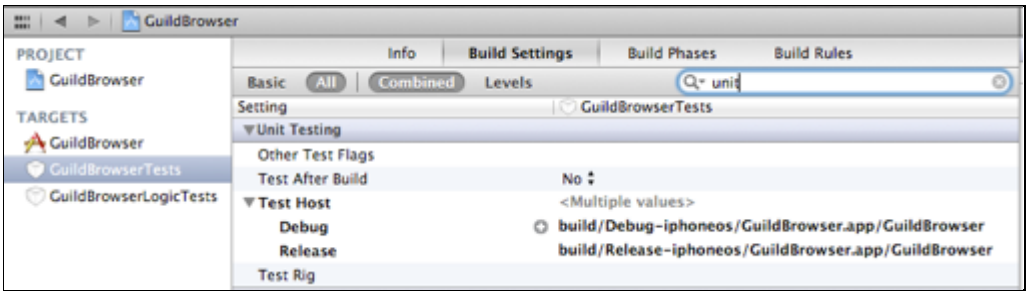
Application vs. logic test targets

Next let's take a look at the two different kinds of unit tests available in Xcode.

Application unit tests

An application unit test basically means any code that needs to use something in UIKit, or really needs to run in the context of a host app. This is the type of unit test target you get by default when creating a new project and including unit tests.

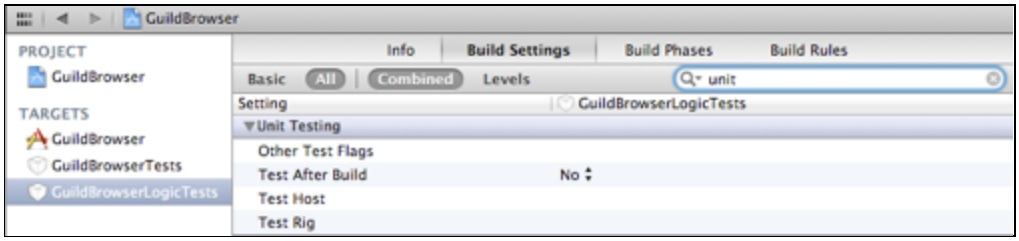
All of the code in this target will be run within your app bundle. You can see the difference by looking in the build settings for each target and checking the settings for **Test Host** and **Bundle Loader**:



Logic unit tests

For now, this is the only type of unit test you can run on your units of code from the command line. This code is tested in isolation from the app.

These are unit tests that test your own code, basically anything that doesn't need to run in the Simulator. It's great for testing custom logic, network access code, or your data model classes.



Running tests from command line notes – no touching!

Unfortunately, as of Xcode 4.5 we are still not "officially" allowed to run application unit tests via the `xcodebuild` command. This is very frustrating, because you obviously can run application unit tests from within Xcode. For this reason, you will create a special logic test target in the next section.

Some creative developers have made patches to the testing scripts inside of the Xcode.app contents. For this chapter and the next, you are going to stick to running your logic tests from Jenkins, and application tests from Xcode. If you're curious about the patches, take a peek at these files:

```
/Applications/Xcode.app/Contents/Developer/Tools/RunUnitTests
```

```
/Applications/Xcode.app/Contents/Developer/Tools/RunPlatformUnitTest
s.include
```

This is the script that runs by default when you run unit tests via ⌘-U:

```
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneSimulat
or.platform/Developer/Tools/RunPlatformUnitTests
```

Here is the error message you'll get, and the reason we can't run application unit tests in the Simulator from the command line (remember that this works fine from within Xcode):

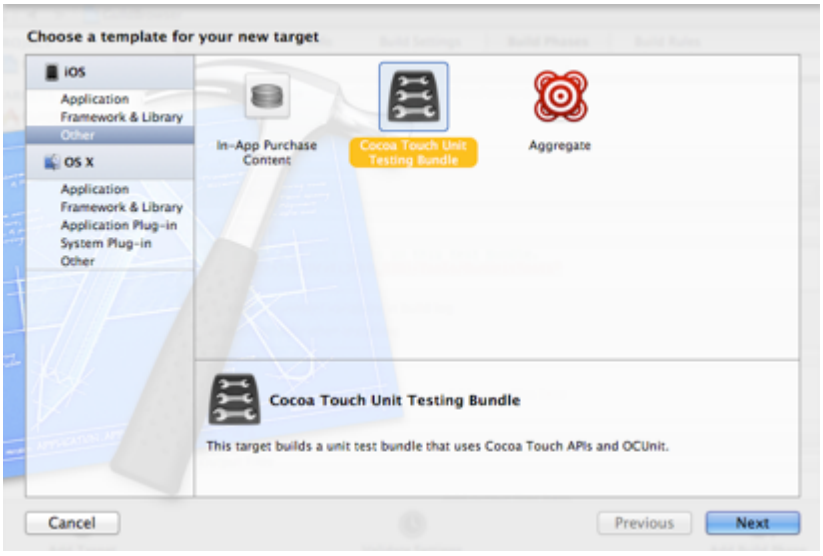
```
RunTestsForApplication() {
  Warning ${LINENO} "Skipping tests; the iPhoneSimulator platform
does not currently support application-hosted tests (TEST_HOST
set)."
}
```

Some clever hacks have gotten application tests to work, but the scripts have changed from Xcode 4.4 to 4.5, so do so at your own risk.

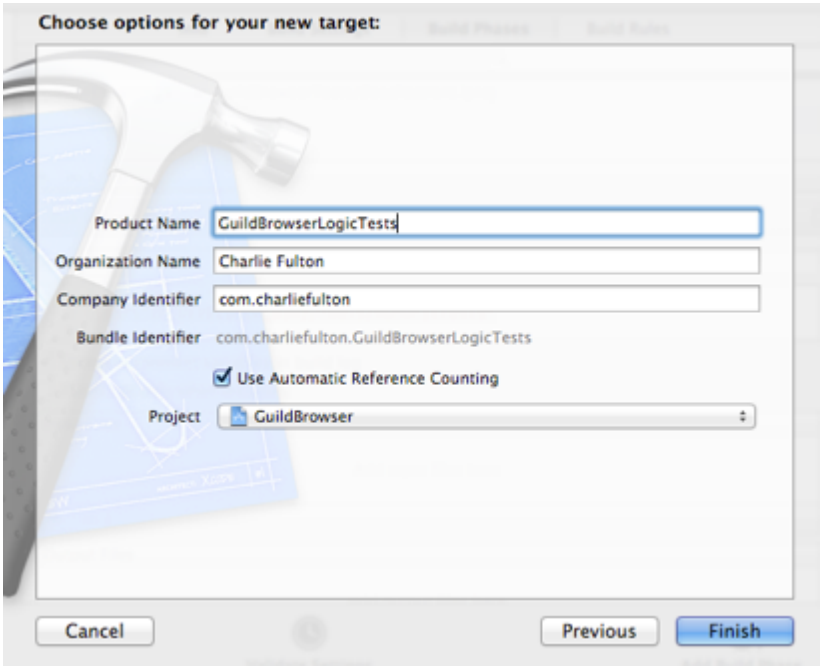
Adding a logic unit test target

You are now going to create a new test target that can be called from your build script.

In Xcode, Go to **File\New\Target...\iOS\Other\Cocoa Touch Unit Testing Bundle** and then click **Next.**

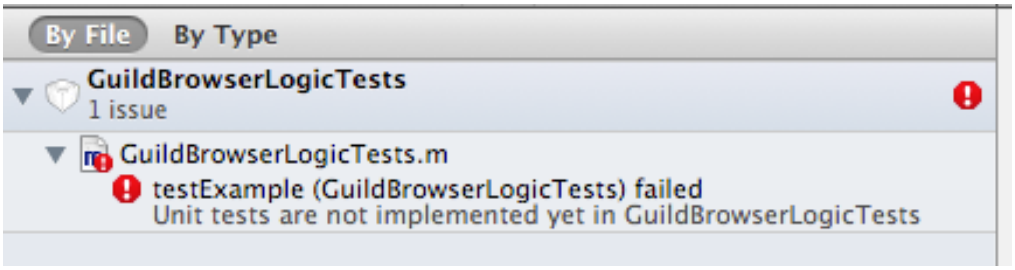


Enter a name that ends with **Tests**, for example **GuildBrowserLogicTests**, make sure that **Use Automatic Reference Counting** is checked, and then click **Finish**.



You will now have a new scheme with the same name. To test that everything is working, switch to the new scheme (using the scheme selector on the Xcode toolbar next to the Run and Stop buttons) and go to **Product\Test** (⌘–U).

You should see that the build succeeded, but the test failed. ☹



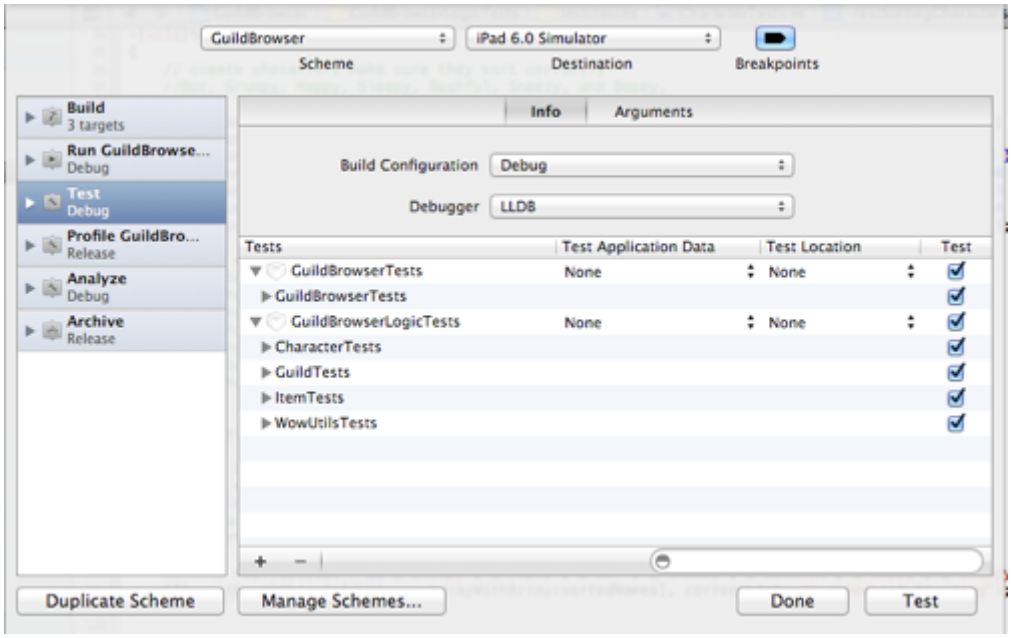
As the error message indicates, this is because the unit tests have not been implemented yet. Let's fix that!

First, delete the default test class by highlighting **GuildBrowserLogicTest.h** and **GuildBrowserLogicTest.m**, tapping **Delete**, and then selecting **Move to Trash.**

So that you don't have to remember to switch schemes, let's modify the main app scheme, GuildBrowser, to include your new target when running Product\Test.

Switch to the **GuildBrowser** scheme and then edit the scheme via **Product\Edit Scheme** (or you can simply Alt-click on the Run button). In the scheme editor, highlight the **Test** configuration, click on the **+** button to add a new target, select **GuildBrowserLogicTests** in the window and then click **Add.**

Your screen should look similar to this:

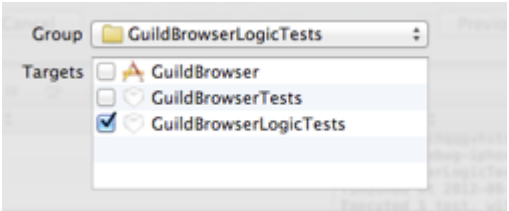


Creating a unit test class

Now let's add a simple unit test class for testing the `WowUtils` class. `WowUtils` looks up the string values for the Character class, Character race, and Item quality from the web service JSON data.

It's best to create a new unit test class for each new class you want to test. Xcode has an **Objective-C test case class** template just for this purpose.

In the Xcode project navigator, right-click on the **GuildBrowserLogicTests** group, choose **New File\Cocoa Touch\Objective-C test case class,** click **Next,** enter **WowUtilsTests** for the class name, click **Next** again, make sure only the **GuildBrowserLogicTests** target is selected, and then click **Create.**



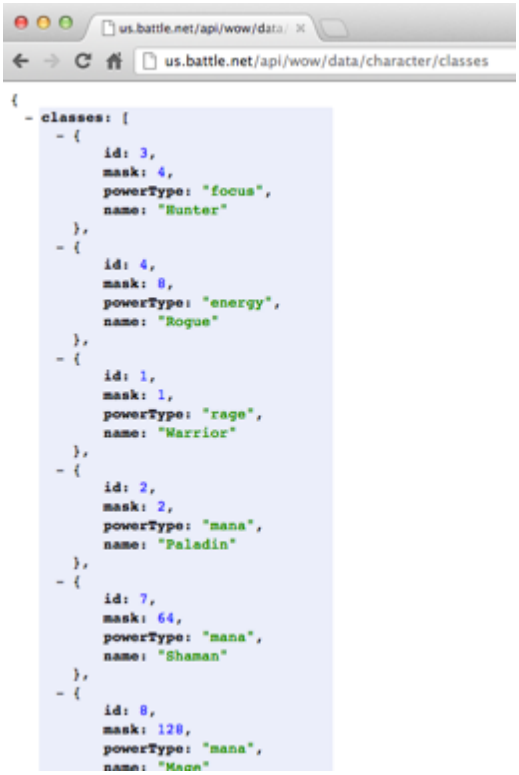
Now you have a **test suite** to which you can add some **test cases**. Let's create your first test case. This test case will make sure that you get the correct response when looking up a character's class.

Here are the methods you are testing from **WowUtils.h**:

```
+(NSString *)classFromCharacterType:(CharacterClassType)type;
+(NSString *)raceFromRaceType:(CharacterRaceType)type;
```

```
+(NSString *)qualityFromQualityType:(ItemQuality)quality;
```

Note: These methods are expected to get the correct name based on the data retrieved from Blizzard's web service. Below you can see the output used by the `WowUtils` class (Chrome is a great tool for inspecting JSON output from a web service):



Replace the contents of **WowUtilsTests.m** with:

```
#import "WowUtilsTests.h"
#import "WowUtils.h"

@implementation WowUtilsTests

// 1
-(void)testCharacterClassNameLookup
{
    // 2
    STAssertEqualObjects(@"Warrior",
                         [WoWUtils classFromCharacterType:1],
                         @"ClassType should be Warrior");
    // 3
```

```
    STAssertFalse([@"Mage" isEqualToString:[WoWUtils
classFromCharacterType:2]],
                  nil);

    // 4
    STAssertTrue([@"Paladin" isEqualToString:[WoWUtils
classFromCharacterType:2]],
                 nil);
    // add the rest as an exercise
}

- (void)testRaceTypeLookup
{
    STAssertEqualObjects(@"Human", [WoWUtils raceFromRaceType:1],
nil);
    STAssertEqualObjects(@"Orc", [WoWUtils raceFromRaceType:2],
nil);
    STAssertFalse([@"Night Elf" isEqualToString:[WoWUtils
raceFromRaceType:45]],nil);
    // add the rest as an exercise
}

- (void)testQualityLookup
{
    STAssertEquals(@"Grey", [WoWUtils qualityFromQualityType:1],
nil);
    STAssertFalse([@"Purple" isEqualToString:[WoWUtils
qualityFromQualityType:10]],nil);
    // add the rest as an exercise
}

@end
```

Let's break this down bit-by-bit.

1. As I stated earlier, all test cases must start with the name **test**.
2. The expectation is that the `WowUtils` class will give you the correct class name, given an ID. The way you verify this expectation is with the `STAssert*` macros. Here you are using the `STAssertEqualObjects` macro.

   The expected result, "**Warrior**", is compared to the result from the `WowUtils` method; if the test fails, you log the message **"ClassType should be Warrior"**.
3. It's always good to include a "failing test" in your test case. This is a test where the result is expected to fail. Again, you are using one of the assertion macros from the SenTestingKit – this time `STAssertFalse`.

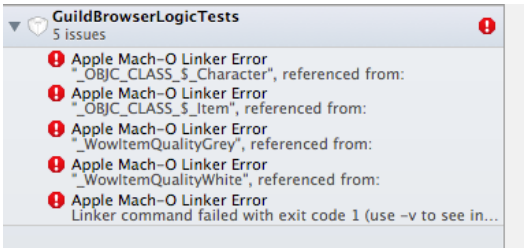
The expected result, **"Mage"**, is compared to the result from the `WowUtils` method; if the test fails, you use the default message, since you passed in `nil` in this example.

4. Finally, you have another example test macro to use.

> **Note:** For a complete list of the testing macros, check out the Unit-Test Result Macro Reference (http://bit.ly/Tsi9ES) from the Apple Developer library.

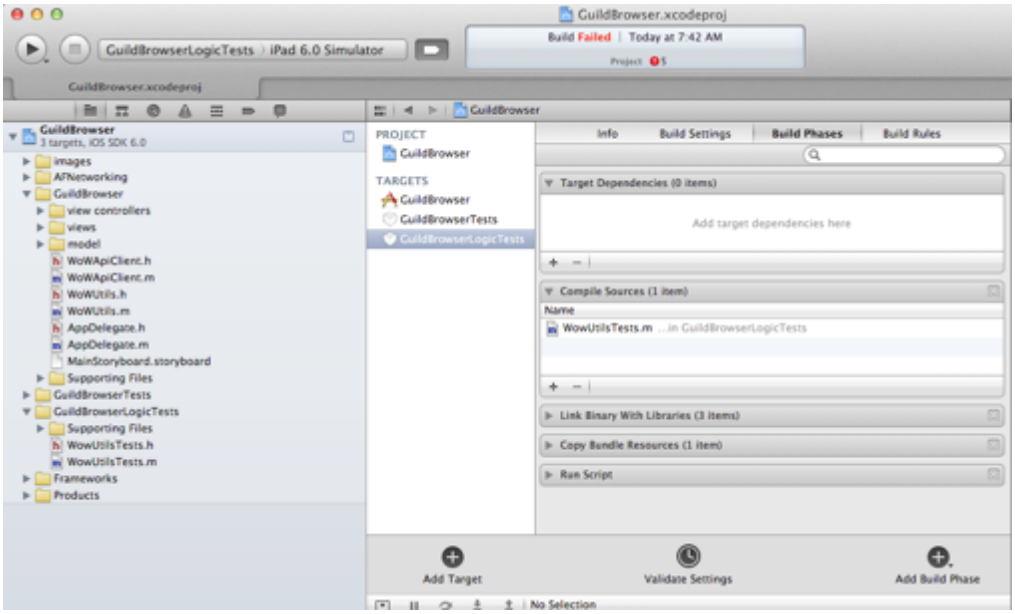
Now you can run your new test suite that presently contains one test case. Go to **Product\Test** (⌘–U).

Doh! You'll see a compile error:

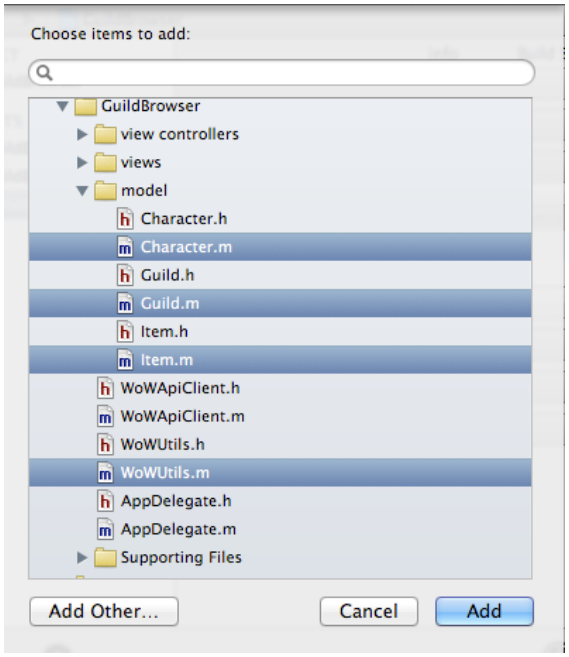


Since you added a new target, you need to let it know about the classes you're trying to test. Every target has its own set of source files available to it. You need to add the source files manually, since you're running a logic test target without the bundle and test host set.

1. Switch to the project navigator – if it's not open, use the **View\Navigators\Show Project Navigator** menu item (⌘-1).
2. Click on the project root to bring up the **Project** and **Targets** editor in the main editor area.
3. Select the **Build Phases** tab along the top of Xcode's main editing pane.
4. Now click on the **GuildBrowserLogicTests** target in the **TARGETS** section. You should see this:

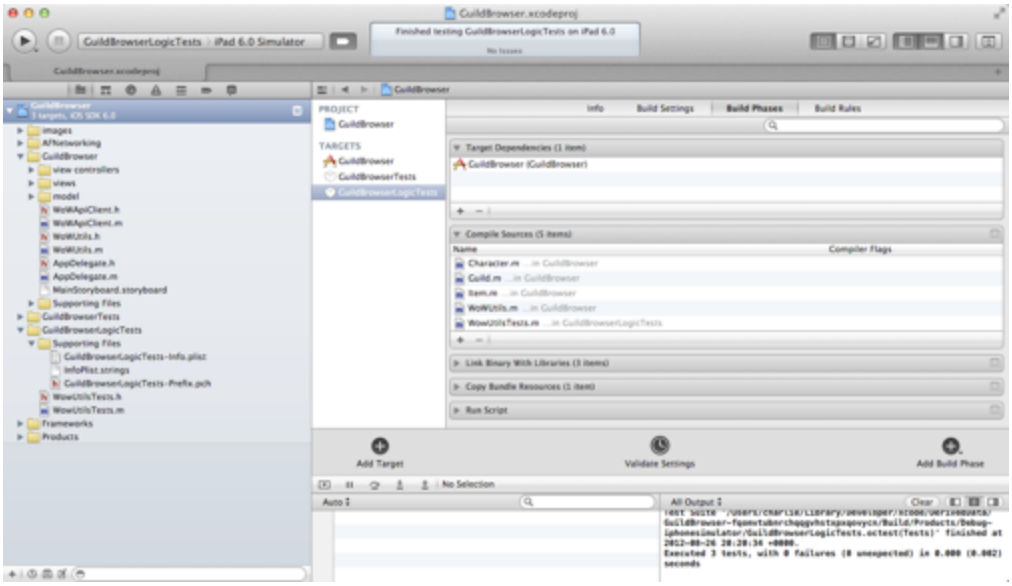


5. Expand the disclosure arrow in the **Compile Sources** section, and click on the **+** button**.** In the popup window, choose the following classes: **Character.m**, **Guild.m**, **Item.m**, and **WowUtils.m**. Then click **Add**.



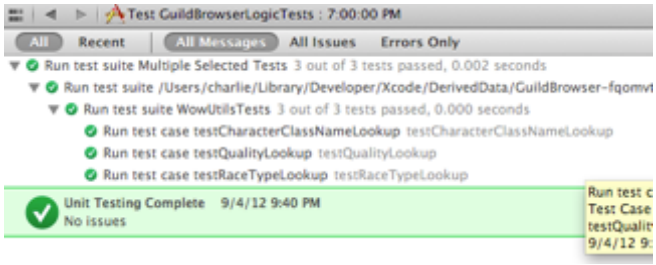
6. Now add the app target as a dependency so that it will get built before your test target runs. Expand the disclosure arrow in the **Target Dependencies** section, click the **+** button**,** choose the **GuildBrowser** app target and click **Add**.

When these steps are complete, you should see the following:



Now run Product\Test (⌘-U) and the test should succeed. You can see the output from all the tests by switching to the Log Navigator (go to View\Navigators\Show Log Navigator (⌘–7)).

You should see this when you click on the last build on the left sidebar:



Make sure to commit and push your latest changes to Github.

Testing a class with local JSON data

Let's take a look at the `Character` class and add a test suite for it. Ideally, you would have added these tests as you were developing the class.

What if you wanted to test creating your `Character` class from local data? How would you do that? What if you wanted to share such data with each test case?

So far, you haven't used two special methods that are available in your unit tests: **setUp** and **tearDown**. Every time you run your unit tests, each test case is invoked independently. Before each test case runs, the `setUp` method is called, and afterwards the `tearDown` method is called. This is how you can share code between each test case.

When building an app that relies on data from web services, I find it very helpful to create tests using data that matches the payload from the services. Quite often in a

project, you will only be working on the client side and waiting for the services from another developer. You can agree on a format and can "stub" out the data in a JSON file. For this app, I downloaded some character and guild data to create tests for the model classes so I could flesh those out before working on the networking code.

First add your test data. Extract the **TestData.zip** file found in the chapter resources and drag the resulting folder into your Xcode project. Make sure that **Copy items into destination group's folder (if needed)** is checked, **Create groups for any added folders** is selected, and that the **GuildBrowserLogicTests** target is checked, and then click **Finish**.

Now add an Objective-C test case class for your `Character` test cases. In the Xcode project navigator, right-click on the **GuildBrowserLogicTests** group, choose **New File\Cocoa Touch\Objective-C test case class,** click **Next,** enter **CharacterTests** for the class name, click **Next** again, make sure only the **GuildBrowserLogicTests** target is selected, and then click **Create.**

Replace the contents of **CharacterTests.m** with:

```
#import "CharacterTests.h"
#import "Character.h"
#import "Item.h"

@implementation CharacterTests
{
    // 1
    NSDictionary *_characterDetailJson;
}

// 2
-(void)setUp
{
    // 3
    NSURL *dataServiceURL = [[NSBundle bundleForClass:self.class]
                             URLForResource:@"character"
withExtension:@"json"];

    // 4
    NSData *sampleData = [NSData
dataWithContentsOfURL:dataServiceURL];
    NSError *error;

    // 5
    id json = [NSJSONSerialization JSONObjectWithData:sampleData
                                              options:kNilOptions
                                                error:&error];
```

```
    STAssertNotNil(json, @"invalid test data");


    _characterDetailJson = json;
}

-(void)tearDown
{
    // 6
    _characterDetailJson = nil;
}

@end
```

Hammer time, break it down! ☺

1. Remember that test classes can have instance variables, just like any other Objective-C class. Here you create the instance variable `_characterDetailJson` to store your sample JSON data.
2. Remember that `setUp` is called before **each** test case. This is useful because you only have to code up the loading once, and can manipulate this data however you wish in each test case.
3. To correctly load the data file, remember this is running as a test bundle. You need to send `self.class` to the `NSBundle` method for finding bundled resources.
4. Create `NSData` from the loaded resource.
5. Now create the JSON data and store it in your instance variable.
6. Remember that `tearDown` is called after **each** test case. This is a great spot to clean up.

Make sure everything is loading up correctly by running Product\Test (⌘-U). OK, now you have your test class set up to load some sample data.

After the `tearDown` method, add the following code:

```
// 1
- (void)testCreateCharacterFromDetailJson
{
    // 2
    Character *testGuy1 = [[Character alloc]
initWithCharacterDetailData:_characterDetailJson];
    STAssertNotNil(testGuy1, @"Could not create character from
detail json");

    // 3
```

```
    Character *testGuy2 = [[Character alloc]
initWithCharacterDetailData:nil];
    STAssertNotNil(testGuy2, @"Could not create character from nil
data");
}
```

Break it down!

1. Here you are creating test cases for the `Character` class designated initializer method, which takes an `NSDictionary` from the JSON data and sets up the properties in the class. This might seem trivial, but remember that when developing the app, it's best to add the tests while you are incrementally developing the class.
2. Here you are just validating that `initWithCharacterDetailData` does indeed return something, using another `STAssert` macro to make sure it's not `nil`.
3. This one is more of a negative test, verifying that you still got a Character back even though you passed in a `nil` `NSDictionary` of data.

Run Product\Test (⌘-U) to make sure your tests are still passing!

After the `testCreateCharacterFromDetailJson` method in **CharacterTests.m**, add the following:

```
// 1
-(void)testCreateCharacterFromDetailJsonProps
{
    STAssertEqualObjects(_testGuy.thumbnail, @"borean-
tundra/171/40508075-avatar.jpg", @"thumbnail url is wrong");
    STAssertEqualObjects(_testGuy.name, @"Hagrel", @"name is
wrong");
    STAssertEqualObjects(_testGuy.battleGroup, @"Emberstorm",
@"battlegroup is wrong");
    STAssertEqualObjects(_testGuy.realm, @"Borean Tundra", @"realm
is wrong");
    STAssertEqualObjects(_testGuy.achievementPoints, @3130,
@"achievement points is wrong");
    STAssertEqualObjects(_testGuy.level,@85, @"level is wrong");

    STAssertEqualObjects(_testGuy.classType, @"Warrior", @"class
type is wrong");
    STAssertEqualObjects(_testGuy.race, @"Human", @"race is
wrong");
    STAssertEqualObjects(_testGuy.gender, @"Male", @"gener is
wrong");
    STAssertEqualObjects(_testGuy.averageItemLevel, @379, @"avg
item level is wrong");
```

```
    STAssertEqualObjects(_testGuy.averageItemLevelEquipped, @355,
@"avg item level is wrong");
}

// 2
-(void)testCreateCharacterFromDetailJsonValidateItems
{
    STAssertEqualObjects(_testGuy.neckItem.name,@"Stoneheart
Choker", @"name is wrong");
    STAssertEqualObjects(_testGuy.wristItem.name,@"Vicious Pyrium
Bracers", @"name is wrong");
    STAssertEqualObjects(_testGuy.waistItem.name,@"Girdle of the
Queen's Champion", @"name is wrong");
    STAssertEqualObjects(_testGuy.handsItem.name,@"Time Strand
Gauntlets", @"name is wrong");
    STAssertEqualObjects(_testGuy.shoulderItem.name,@"Temporal
Pauldrons", @"name is wrong");
    STAssertEqualObjects(_testGuy.chestItem.name,@"Ruthless
Gladiator's Plate Chestpiece", @"name is wrong");
    STAssertEqualObjects(_testGuy.fingerItem1.name,@"Thrall's
Gratitude", @"name is wrong");
    STAssertEqualObjects(_testGuy.fingerItem2.name,@"Breathstealer
Band", @"name is wrong");
    STAssertEqualObjects(_testGuy.shirtItem.name,@"Black
Swashbuckler's Shirt", @"name is wrong");
    STAssertEqualObjects(_testGuy.tabardItem.name,@"Tabard of the
Wildhammer Clan", @"nname is wrong");
    STAssertEqualObjects(_testGuy.headItem.name,@"Vicious Pyrium
Helm", @"neck name is wrong");
    STAssertEqualObjects(_testGuy.backItem.name,@"Cloak of the
Royal Protector", @"neck name is wrong");
    STAssertEqualObjects(_testGuy.legsItem.name,@"Bloodhoof
Legguards", @"neck name is wrong");
    STAssertEqualObjects(_testGuy.feetItem.name,@"Treads of the
Past", @"neck name is wrong");
    STAssertEqualObjects(_testGuy.mainHandItem.name,@"Axe of the
Tauren Chieftains", @"neck name is wrong");
    STAssertEqualObjects(_testGuy.offHandItem.name,nil, @"offhand
should be nil");
    STAssertEqualObjects(_testGuy.trinketItem1.name,@"Rosary of
Light", @"neck name is wrong");
    STAssertEqualObjects(_testGuy.trinketItem2.name,@"Bone-Link
Fetish", @"neck name is wrong");
    STAssertEqualObjects(_testGuy.rangedItem.name,@"Ironfeather
Longbow", @"neck name is wrong");
```

```
}
```

You need to make sure that the properties correctly match up with the JSON data that you loaded initially. Just a few comments on what's going on here:

1. This tests the information shown in the main screen Character cell.
2. This tests the information shown in the `CharacterDetailViewController` when you click on a Character cell from the main screen.

To have a little fun, "forget on purpose" to run Product\Test (⌘-U), and commit and push your changes. You will see what your little friend Jenkins will find, when you update your Jenkins job script to include the tests.

Finalizing your Jenkins job

First off, make sure to commit all changes and push them to your origin/master branch on GitHub.

As a reminder, you will need to do the following:

1. Go to **File\Source Control\Commit (⌥⌘–C)**, then on the following screen, enter a commit message like "added test target" and click **Commit.**
2. Push the local master branch to the remote origin/master branch. Go to **File\Source Control\Push** and click **Push**.

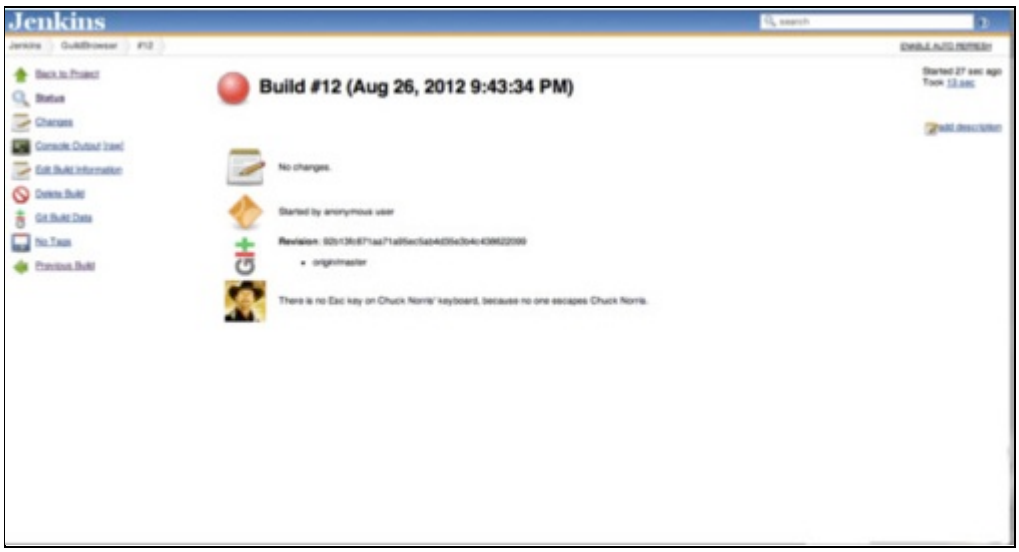
Now edit your Jenkins job again to include the tests you just set up. Go to the **Jenkins Dashboard\GuildBrowser job\Configure**. In the **Build** section, replace the existing code with this:

```
export DEVELOPER_DIR=/Applications/Xcode.app/Contents/Developer/

xcodebuild -target GuildBrowserLogicTests \
-sdk iphonesimulator \
-configuration Debug \
TEST_AFTER_BUILD=YES \
clean build
```

Click **Save** and then **Build Now.**

Ummm, RUN! You broke the build and Chuck Norris is going to find you! The gloves are on! This is serious, man. No one can escape him. ☺



So what happened? You could scour the output logs of the build that put you on Mr. Norris's radar, or… you could set up Jenkins to give you some test results reporting! I don't know about you, but if I'm on that radar I want to be off it FAST! You should have also received an email telling you that you broke the build. Let's get some reporting, stat!

Getting a unit test report

It would be quite tedious to pour through the logs after each build, and it sure would be nice if there were a handy report you could look at to see what passed and what failed.

Well, it turns out there is a script that does exactly that! Christian Hedin has written an awesome Ruby script to turn the OCUnit output into JUnit-style reports. You can find it on GitHub at https://github.com/ciryon/OCUnit2JUnit.

A copy of the Ruby script is included in the resources for this chapter. Work quickly; remember whose radar you are on!

Copy the **ocunit2junit.rb** file to somewhere that Jenkins can access it – I placed mine in **/usr/local/bin**. Make a note of the location, and use it below when updating the build job.

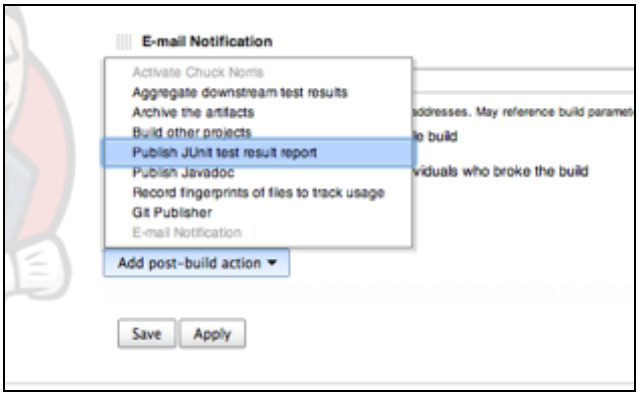
In the **GuildBrowser Jenkins job**, update the shell script to the following:

```
export DEVELOPER_DIR=/Applications/Xcode.app/Contents/Developer/

xcodebuild -target GuildBrowserLogicTests \
-sdk iphonesimulator \
-configuration Debug \
TEST_AFTER_BUILD=YES \
clean build | /usr/local/bin/ocunit2junit.rb
```

The only new bit is on the final line, where the output from the build is piped through to the Ruby script for processing.

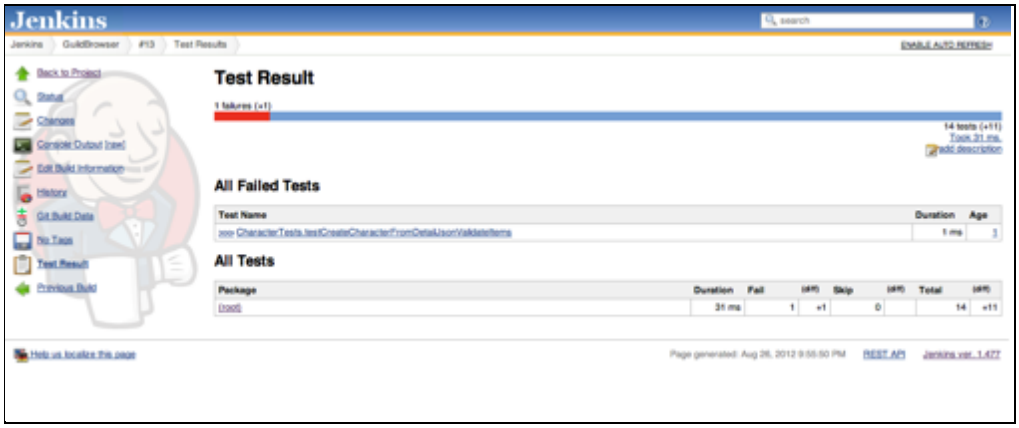
Now you need to add another Post-Build Action to your job to capture these reports. At the bottom of the configure screen, click on the **Add post-build action** button and choose **Publish JUnit test result report**.



Enter **test-reports/*.xml** in the **Test report XMLs** text field.

Click **Save** and then **Build Now.** When the job is finished and you go to the main project area for the GuildBrowser job, you should see a **Latest Test Result** link. Work quickly; get on that link!

Now it's really obvious what has angered "He who must not be named."



It looks like a bug has been found by the `CharacterTests` class. Let's investigate this further by clicking on the link to the test name:

```
Error Message

"Vicious Pyrium Bracers" should be equal to "Girdle of the Queen"s
Champion" name is wrong

Stacktrace
```

```
/Users/charlie/.jenkins/workspace/GuildBrowser/GuildBrowserLogicTe
sts/testclasses/CharacterTests.m:58
```

OK, let's go check out the test code. First look at line 76 from **CharacterTests.m:**

```
STAssertEqualObjects(hagrel.waistItem.name,@"Girdle of the Queen's
Champion", @"name is wrong");
```

Interesting! Open the file **character.json** to see what you have; maybe your data was wrong, but surely not your code!

> **Note:** When working with JSON, I highly recommend verifying your test data or any JSON using the awesome website jsonlint.com. Not only does it validate your JSON, but it will format it too!

Remember character.json represents what's returned from Blizzard's real character web service. You'll find this snippet:

```
{
    "thumbnail": "borean-tundra/171/40508075-avatar.jpg",
    "class": 1,
    "items": {
        …
        "wrist": {
            "icon": "inv_bracer_plate_dungeonplate_c_04",
            "tooltipParams": {
                "extraSocket": true,
                "enchant": 4089
            },
            "name": "Vicious Pyrium Bracers",
            "id": 75124,
            "quality": 3
        },
        "waist": {
            "icon": "inv_belt_plate_dungeonplate_c_06",
            "tooltipParams": {
                "gem0": 52231
            },
            "name": "Girdle of the Queen's Champion",
            "id": 72832,
            "quality": 4
        },
…
```

Hmmm, the plot thickens. Your test found "Vicious Pyrium Bracers", the name of the wrist item, but was looking for "Girdle of the Queen's Champion", the name of the waist item. Now it's looking more like a bug!

You can see in the test in **CharacterTest.m** that you create a `Character` like so:

```
Character *hagrel = [[Character alloc]
initWithCharacterDetailData:characterDetailJson];
```

Open **Character.m** and let's take a look at **initWithCharacterDetailData**:. Can you spot the bug?

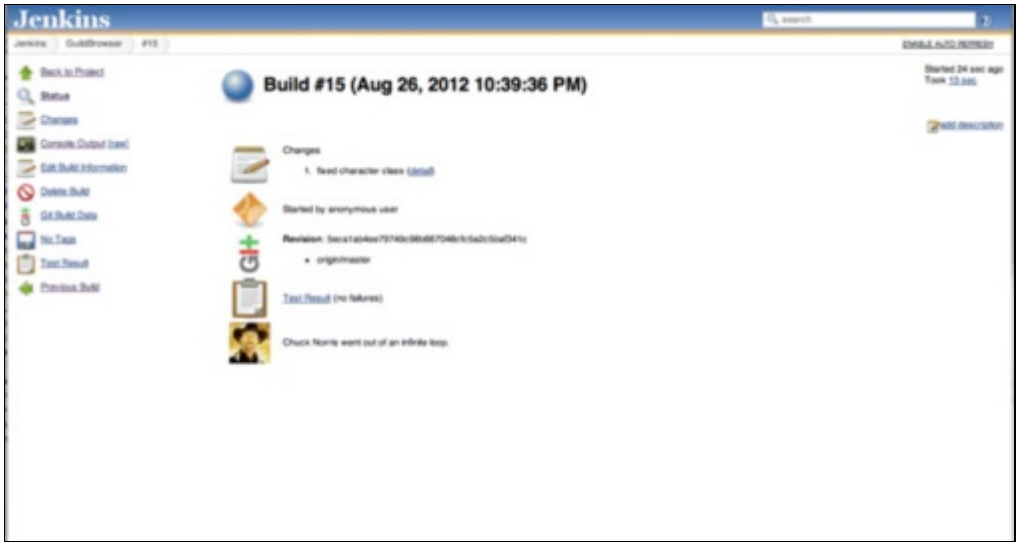
```
_wristItem = [Item initWithData:data[@"items"][@"wrist"]];
_waistItem = [Item initWithData:data[@"items"][@"wrist"]];
```

You were using the wrong key for the `waistItem` property. Fix that by changing the last line to:

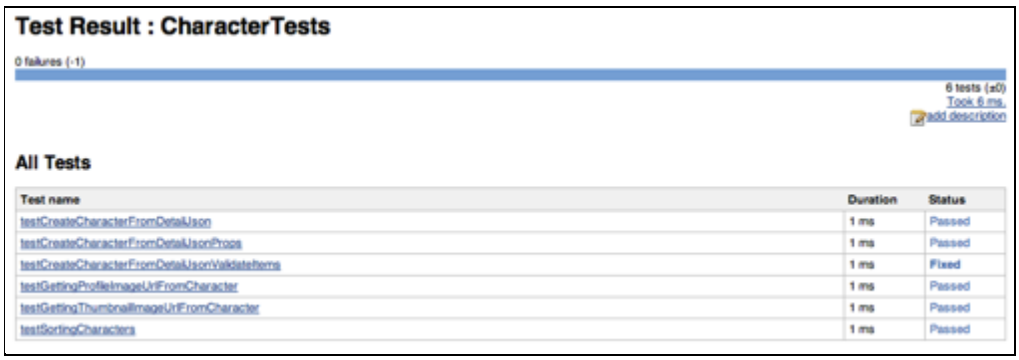
```
_waistItem = [Item initWithData:data[@"items"][@"waist"]];
```

OK, now commit your changes and push them to GitHub. Go to the Jenkins project and click on **Build Now.**

Test result: no failures! Chuck gives that a thumbs up – what a huge relief!



Drill down into the report by clicking on **Latest Test Result\root** and you will see all the tests that were run. By drilling further into the **CharacterTests**, you can see that you fixed the issue (look closely at the status on line three ☺):



Polling for changes

Now let's set up the Jenkins project to look for changes every 10 minutes, and if it finds any, to run the build. Edit your Jenkins job again by going to the **Jenkins Dashboard\GuildBrowser job\Configure.** In the **Build Triggers** section, check the **Poll SCM** box. In the schedule text area, enter:

```
*/10 * * * *
```

Click **Save**. Now builds will happen if there is new code that has been pushed to the origin/master repo. Of course, you can still manually build like you've been doing with the Build Now button.

Automate archiving

Let's update the build script to archive your app after successfully running the test script. Edit your Jenkins job again by, you guessed it, going to the **Jenkins Dashboard\GuildBrowser job\Configure.**

You are now going to add another shell step, which will happen *after* the test script step. In the **Build** section, click on the **Add build step** dropdown and select **Execute shell.**

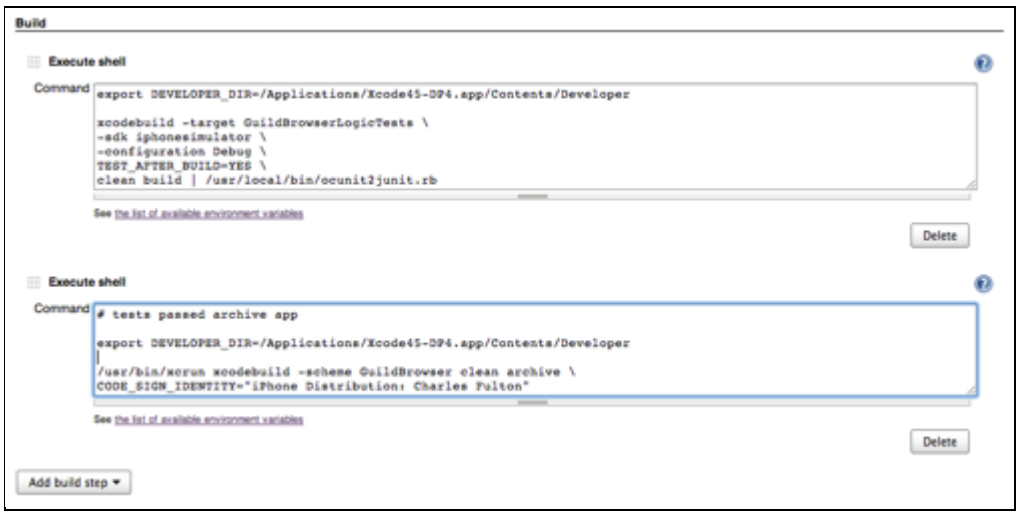
Enter the following, replacing the `CODE_SIGN_IDENTITY` with your distribution certificate name:

```
# tests passed archive app

export DEVELOPER_DIR=/Applications/Xcode.app/Contents/Developer

/usr/bin/xcrun xcodebuild -scheme GuildBrowser clean archive \
CODE_SIGN_IDENTITY="iPhone Distribution: Charles Fulton"
```

Your build section should now look similar to this:



If you save your changes and build again via Jenkins, you should now see the latest archive in the Xcode organizer for this project. In Xcode, open **Window\Organizer** and switch to the **Archives** tab, and you should see the **GuildBrowser** project listed on the left.

When you click on the project, you will see all of your archived builds. This is great, because now you will know that you are building and testing the same archive that will be submitted to the App Store!

> **Note**: If the archiving fails, you will not see an archive listed in the Xcode Organizer, nor will you see any immediate indication from Jenkins as to the archive failure. You would need to check the build logs to see if the archiving actually succeeded.
>
> Usually, the archive process fails because the `CODE_SIGN_IDENTITY` is not correctly specified, or because it doesn't match the Bundle ID for the project. So if you run into any archival failures, those would be the items to check. One solution here would be to set the `CODE_SIGN_IDENTITY` to iPhone Distribution, since that will match the default distribution profile.
>
> Also remember that if you make any project changes to fix the above, you need to commit to Git and push to GitHub. Otherwise, Jenkins won't pick up your changes for the next build. ☺

Next you will add sending this archived build to TestFlight, and keeping track of the artifact (which is Jenkins terminology for the results of the build) in Jenkins.

Uploading the archive to TestFlight

One of the best things about the iOS development community is the variety of awesome frameworks and services that have emerged over the past few years.

Back in the day, it was quite an effort to get a beta build to your testers. You would have to get your IPA file to them by email, have them drag that to iTunes, then connect their device to sync up with iTunes just to get it on their device. You also had to send them an email asking for the UDID of their device, scribble that down or copy it up to create the new provisioning profile, then create the new build. It was a nightmare to keep track of which device belonged to what user, what iOS version they were running, etc.

Enter TestFlight! This is a website that makes distribution and testing of beta versions of apps a breeze. ☺ Before TestFlight, the only way to distribute builds to your beta testers was via ad hoc builds. The ad hoc builds still remain, since TestFlight works within the ad hoc mechanism, but it makes distribution and management of these builds much simpler.

Testflight will also allow you to set up their TestFlight SDK packaging in your app for crash log analysis, usage analysis, and more!

We are going to focus on the auto upload and distribution pieces that TestFlight offers.

PackageApplication

Here is a sweet little Perl script included by Apple in the Xcode.app bundle. You can take a peek at it (no touching!) by opening:

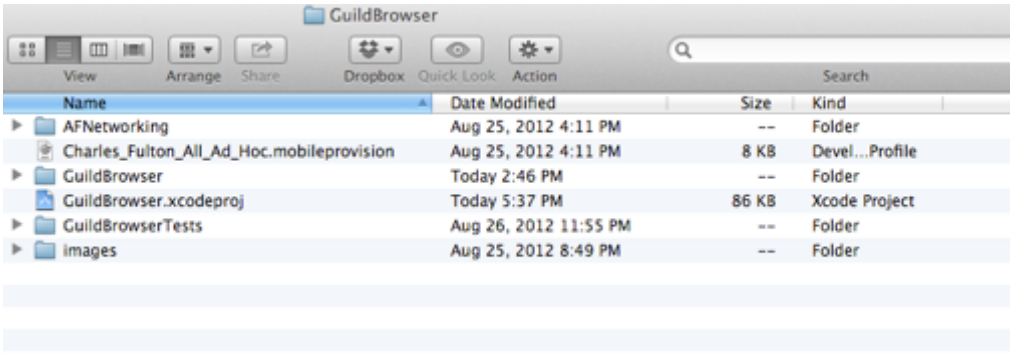
```
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/PackageApplication
```

This is the tool that will allow you to do the following from the latest archived build, the one you just created in the previous step. You will be modifying your Jenkins archive step to include:

- Creating the GuildBrowser.app bundle;
- Embedding your ad hoc provisioning profile;
- Codesign with your distribution certificate.

To make sure you're ready, download your latest **ad hoc provisioning profile** from the iOS Provisioning Portal and add it to the top level of your project folder.

After downloading it, your project should look like the image below. Notice my ad hoc provisioning profile named **Charles_Fulton_All_Ad_Hoc.mobileprovision**:



Note: Your .mobileprovision file can be located anywhere. You just have to make sure to give it the full absolute path – no relative paths. For example:

~/Library/MobileDevice/Provisioning\ Profiles/

Must be:

/Users/charlie/Library/MobileDevice/Provisioning\ Profiles/

I like to keep mine in Git, so that when new devices are added, I just check in a new provisioning profile. Then I can do a manual build in Jenkins and it's ready to go.

Make sure to **commit** and **push** to GitHub after adding the file, so Jenkins can see it.

If you are using Xcode to commit to Git, then note that you would need to add the mobile provisioning profile to your project before Xcode sees the new file. Otherwise, you will not be able to commit it to Git. If you use the command-line Git tools or a separate Git client, this issue should not arise.

Let's edit your Jenkins job again. Go to the **Jenkins Dashboard\GuildBrowser job\Configure.** Add a new Build step of type **execute shell**:

```
export DEVELOPER_DIR=/Applications/Xcode.app/Contents/Developer

#
# Setup
#
# 1
PROJECT="GuildBrowser"
SIGNING_IDENTITY="iPhone Distribution: Charles Fulton"
PROVISIONING_PROFILE="${WORKSPACE}/Charles_Fulton_All_Ad_Hoc.mobil
eprovision"

# 2
# this is the latest archive from previous build step
ARCHIVE="$(ls -dt
~/Library/Developer/Xcode/Archives/*/${PROJECT}*.xcarchive|head -
1)"
# 3
IPA_DIR="${WORKSPACE}"
DSYM="${ARCHIVE}/dSYMs/${PROJECT}.app.dSYM"
APP="${ARCHIVE}/Products/Applications/${PROJECT}.app"


#
# PackageApplication
#

# package up the latest archived build
/bin/rm -f "${IPA_DIR}/${PROJECT}.ipa"

# 4
/usr/bin/xcrun -sdk iphoneos PackageApplication \
-o "${IPA_DIR}/${PROJECT}.ipa" \
-verbose "${APP}" \
-sign "${SIGNING_IDENTITY}" \
--embed "${PROVISIONING_PROFILE}"

# zip and ship
/bin/rm -f "${IPA_DIR}/${PROJECT}.dSYM.zip"

# 5
/usr/bin/zip -r "${IPA_DIR}/${PROJECT}.dSYM.zip" "${DSYM}"
```

There is a lot going on in this build script. You know what that means.

Break it down!

1. These are the settings for what certificate and provisioning profile to use when creating your IPA file. You should change the `SIGNING_IDENTITY` and `PROVISIONING_PROFILE` to use your ad hoc distribution profile.
2. This bit of shell trickery finds the latest archive build location. This gives you a sneaky way to get the latest archive result directory. To make more sense of this one, open a terminal and run the command:

```
ls -dt
~/Library/Developer/Xcode/Archives/*/GuildBrowser*.xcarchive|head
-1
```

You should see output like this:

```
/Users/charlie/Library/Developer/Xcode/Archives/2012-08-
27/GuildBrowser 8-27-12 10.37 AM.xcarchive/
```

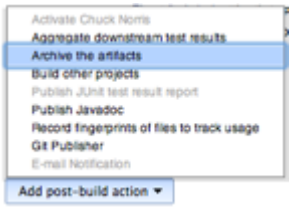
3. You can now use the `ARCHIVE` variable to create the variables `APP` and `DSYM` saving the absolute paths to send to `PackageApplication`. Take a peek inside by trying this command:

```
ls –l "/Users/charlie/Library/Developer/Xcode/Archives/2012-08-
27/GuildBrowser 8-27-12 10.37
AM.xcarchive/Products/Applications/GuildBrowser.app"
```

4. Here you are calling the `PackageApplication` script. Notice that Jenkins gives you some nice environment variables in `$WORKSPACE`. The `$WORKSPACE` variable lets you get an absolute path to the Jenkins job. You can now create artifacts in Jenkins of exactly what gets sent to your users.
5. Compress the `dSYMs` from the archive. `dSYMs` are used to symbolicate crash logs so that you can find out which source file, method, line, etc. had an issue instead of getting memory addresses that would mean nothing to you.

Before saving and building the updated script, let's add a step to create artifacts of all successfully-created `.apps` and `dSYMs`.

Go to the **Post-build Actions** section, and select **Archive the artifacts** from the **Add post-build action** menu.



Enter ***.ipa, *.dSYM.zip** in the **files to archive** text field.

Click **Save** and select **Build Now**. Once the build completes, you should see this:

If something fails at this point, it usually is because the code signing information wasn't correct, or because the ad hoc provisioning profile isn't in the project root folder. So check the build logs to see what is going on.

Mission completion

OK, let's send those artifacts to TestFlight and notify your users of the new build.

> **Note:** This section assumes you already have a TestFlight (testflightapp.com) account. You will need your TestFlight team and API tokens.
>
> You can get your API token here: https://testflightapp.com/account/#api
>
> Your team token can be found by clicking on the team info button while logged into TestFlight.

Edit your Jenkins job again by going to the **Jenkins Dashboard\GuildBrowser job\Configure.** You will be editing the script you added in the previous step.

Add this code, filled out with your own TestFlight info, after the `export DEVELOP_DIR` line:

```
# testflight stuff
API_TOKEN=<YOUR API TOKEN>
TEAM_TOKEN=<YOUR TEAM TOKEN>
```

Add this to the end of the existing script:

```
#
# Send to TestFlight
#
/usr/bin/curl "http://testflightapp.com/api/builds.json" \
  -F file=@"${IPA_DIR}/${PROJECT}.ipa" \
  -F dsym=@"${IPA_DIR}/${PROJECT}.dSYM.zip" \
  -F api_token="${API_TOKEN}" \
  -F team_token="${TEAM_TOKEN}" \
  -F notes="Build ${BUILD_NUMBER} uploaded automatically from
Xcode. Tested by Chuck Norris" \
  -F notify=True \
  -F distribution_lists='all'

echo "Successfully sent to TestFlight"
```

Click **Save** and do another **Build Now.**

Now when the job completes, your build should have been sent to TestFlight! And your users should have received an email telling them that a new build is available. This will allow them to install your app right from the email and begin testing for you!

Where to go from here?

You should now be equipped to set up an automated building, testing, and distribution system for all of your iOS apps!

Let's recap what you did in this chapter:

- First you learned how to set up a remote repo on Github, giving you a spot to share and test your code.
- After that, you took a look at continuous integration with Jenkins, and created a nice build script step-by-step, first building, then testing, and finally uploading your archived app to Testflight.

- You also looked at how to include a "bottom up" approach to unit testing your code. If you're interested in learning more about unit testing in iOS, I highly recommend the book *Test-Driven iOS Development* by Graham Lee. I also encourage all of you to submit a radar to apple to make it easier to run the application unit tests from scripts, without hacks!

In the next chapter, you are going to take the same app and do some "top down" unit testing by creating a cool little testing robot. This robot will use instruments and a UI Automation script to drive some UI interactions in the GuildBrowser app. ☺