IcontinuousIntegration (continuous integration for iOS applications)

by Alexander Dodatko. 2011-06-07

Table of Contents

Abstract

The continuous integration process is a common technique in desktop and enterprise software development process. However, there are some obstacles and challenges for iOS mobile development.

Desktop	iOS
Products can run at the machine they are built	Products can run ONLY at the device or simulator
Explicit input-output control (test reports, creation, benchmarking, etc.)	Output data is stored at specific locations. (~/Library/Application Support/ iPhone Simulator/4.3.2/Applications/ 31629FD8-DDF0-4E4C-A2D8- FC2D2BE2D07D/ Caches/...)
Good integration between build tools and CI servers.	Almost no integration. Xcode batch build commands must be invoked from command line.

This manual will help you to solve these problems and may give some ideas how to solve other ones. Here we'll describe the whole process for a simple hello-world program. We'll also cover static libraries creation and usage (both plain ones and so-called **“universal binaries”**)

Our build process will be based on the following software:

1. Xcode 4.0.2 (iOS SDK 4.3.3)
2. Xcode 3.2.6 (because Xcode 4 is still has some bugs about project management)
3. **GHUnit** unit test framework
4. **Hudson CI** build server

Selecting a Framework for Unit Testing

There are 3 major opportunities for ObjectiveC unit testing. They are:

1. SenTesting Kit
2. Google toolbox
3. GHUnit

SenTestingKit

SenTestingKit is a default framework. It is shipped with xCode and has good integration with it. Its main advantage is build failure initiation if some tests are not passed correctly. However, it has some **disadvantages**:

1. Logical tests run ONLY on the simulator.
2. Logical tests cannot be debugged.
   NOTE : actually, you can debug them but you have to invoke them manually.
3. Functionality tests run ONLY on the device. They are intended to test the entire application.
4. The framework does not support UIKit and bundles.

Google Toolbox

Google toolbox has the following advantages :

1. xCode integration.
2. Backward compatibility with SenTesting Kit (and all its advantages)
3. GUI snapshots comparisson (using raster ***.png**, ***.jpeg** graphics)

However, it still lacks debugging opportunities, bundles and UIKit support.

GHUnit

On the contrary, GHUnit tests are packaged in a usual iOS application. It has full support of the ObjC and CocoaTouch framework. You can run it on both the device and simulator. You can even deploy it to the App Store! However, it lacks xCode integration support.

Test Frameworks Comparisson

You can see a brief comparisson of the testing frameworks in a table below :

	SenTest	**Google**	**GHUnit**
Xcode integration (auto navigate to failure, fail build on failed test)	+	+	---
UIKit Support	---	---	+
Bundles support	---	---	+
Xml reports	---	---	+ (lack of support for hudson CI)
Runs on device	+- (Runtime tests only)	+- (Runtime tests only)	+
Runs on simulator	+- (logic tests only)	+- (logic tests only)	+
Debugging (out of box)	---	---	+
UI snapshots comparing	---	+	---

In our application we do a lot of network data exchange. Hence, we need some mock test data to check our protocol related classes. That's why bundles support is critical for us. So, **GHUnit** is our choice.

If bundles and debugging are not so important for you, we suggest using google toolbox for better xCode integration and more balanced features scope.

Setting Up GHUnit for Continuous Integration.

The main advantage of GHUnit is the fact that tests are packaged into a usual iOS application. It is easy to run and debug it in an everyday development cycle.

However, it causes some disadvantages for continuous integration. Here are some problems which are hardly noticeable for desktop applications but are somewhat difficult for iOS :

	Desktop	Embedded / Mobile (iOS)
Pass test data to the application	Desktop applications can just pull the databases and other test files from a nearby directory on the file system.	For iOS applications we should put those files into a bundle or create a mock web server.
Launch the application	The OS will run it with no problem	Simulator or device is required to execute.
Terminate the application	Usually, a unit test is a plain linear command line program.	IOS applications do not actually terminate. They just "**go to the background mode**" (in no doubt, for iOS4 and upper)
Collect test results	Just **fread** an xml report, generated after run.	Each iOS application is executed in a separate sandbox. So, it may be a challenge as well

Passing Data to the Application

We use bundles to pass test data to the iOS application.

Running a Test

A test can be executed on the simulator with the **iphonesim** utility.

```
:~ Oleksandr_Dodatko$ iphonesim
Usage: iphonesim <options> <command> ...
Commands:
  showsdks
  launch <application path> [sdkversion] [family] [uuid]
```

These arguments are case sensitive.

Example (legacy Xcode3 style has been used) :

```
iphonesim launch "$TEST_PROJECTS_PATH/CITest/build/Release-iphonesimulator/CITest.app" 4.2 ipad
```

For details please consider "**Xcode 4.x command line tools reference**" ***article*** and **GHUnit *documentation***.

Terminating an Application

The application, produced by GHUnit, requires some user interaction.
However, there are some configuration flags that allow to execute batch runs.

1. **GHUNIT_AUTORUN** – starts test runners immediately. Without user interaction,
2. **GHUNIT_AUTOEXIT** – terminates the app after all tests are executed.
3. **WRITE_JUNIT_XML** – creates xml report and puts it into the temporary directory.

Note : **GHUNIT_AUTOEXIT** does not work for iOS devices because of a bug. Hopefully, it will be fixed soon. At the moment you can download a fork with this issue fixed from https://github.com/dodikk/gh-unit

```
setenv( "GHUNIT_AUTORUN" , "YES", 1 );
setenv( "WRITE_JUNIT_XML", "YES", 1 );
setenv( "GHUNIT_AUTOEXIT" , "YES", 1 );
```

Extracting Test Results

In order to get results, you must :

1. Set up **WRITE_JUNIT_XML** flag.

```
setenv( "GHUNIT_AUTOEXIT" , "YES", 1 );
```

2. Locate test reports output directory

```
TEMP_DIR=$(/usr/bin/getconf DARWIN_USER_TEMP_DIR)
TEST_DIR_NAME=test-results
TEST_RESULTS_DIR=$TEMP_DIR$TEST_DIR_NAME
```

3. Move to this directory and copy files to the desired location

```
cd "$TEST_RESULTS_DIR"
  pwd
  cp *.xml "$TEST_PUBLISH_DIR"
cd "$LAUNCH_DIR"
```

Now let's do some coding fun in the next chapter.

Defining the Project Structure

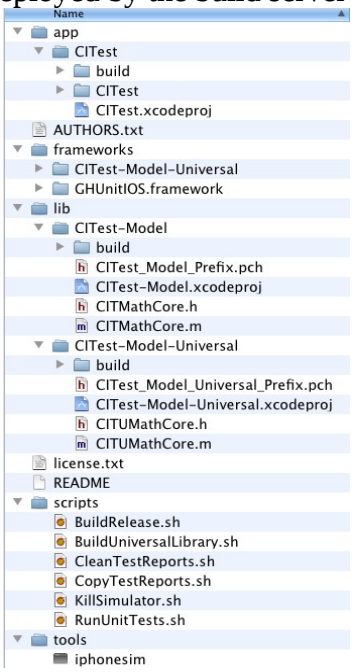
Our project will consist of three parts :

1. Main project – contains unit tests
2. Usual library project
3. Universal binary library project

Main project must be digitally signed to be installed on the device. For our projects we keep those certificate files under version control as well.

That's why we suggest using the following repository structure :

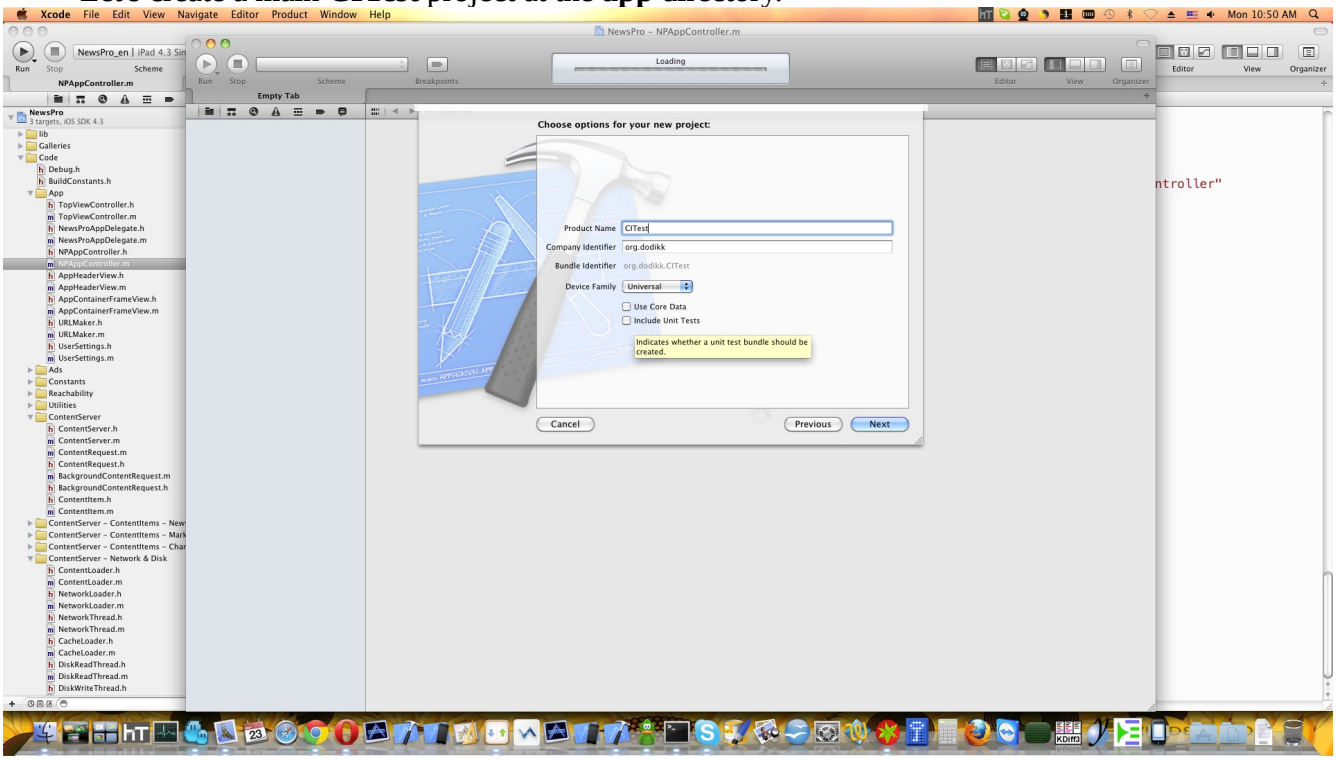
1. **app** – a directory for main sources (there may be more than one main project)
2. **lib** – a directory for library sources. It contains only self-written libraries.
3. **frameworks** – all third-party libraries go here. It is also used as a directory for universal binaries deployment.
4. **scripts** – a directory for build scripts source code. We prefer using explicit script files instead of embedding scripts into the ***.xcodeproj*** directory.
5. **tools** – a directory for tools, used for continuous integration. It contains only ready-to-use binaries.
6. **test** – contains the code of unit tests. Since the main application is a unit-test itself, we do not use this one in the current sample project.
7. **certificates** – contains Apple provision and developer profiles.
8. **deployment** – this directory is NOT supposed to be under version control. It contains build artifacts that will be deployed by the build server or in some other way.



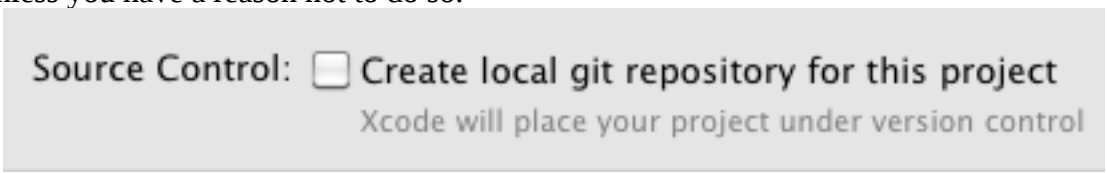
Creating a Main Unit Test Project

Firstly, let's suppose that we have already created a directories structure, described above and have a built GHUnit framework at the "**frameworks**" directory. Let's also suppose that we have deployed our **certificates** and **tools** correctly.

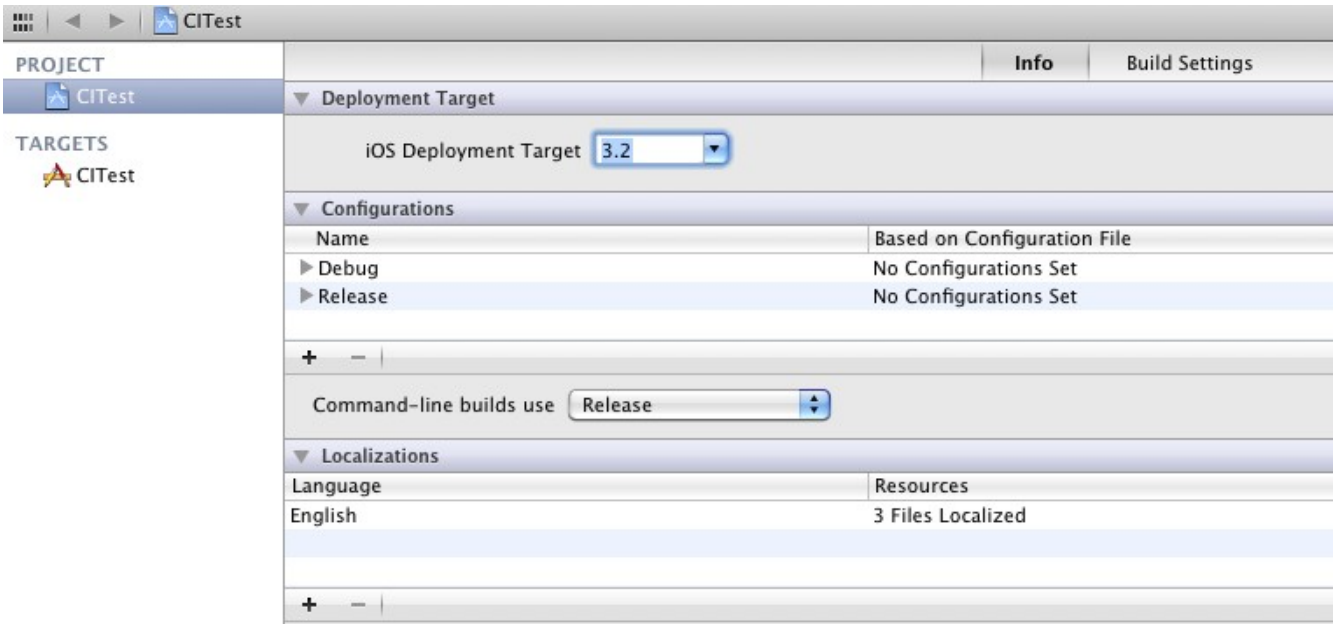
Let's create a main **CITest** project at the **app** directory.

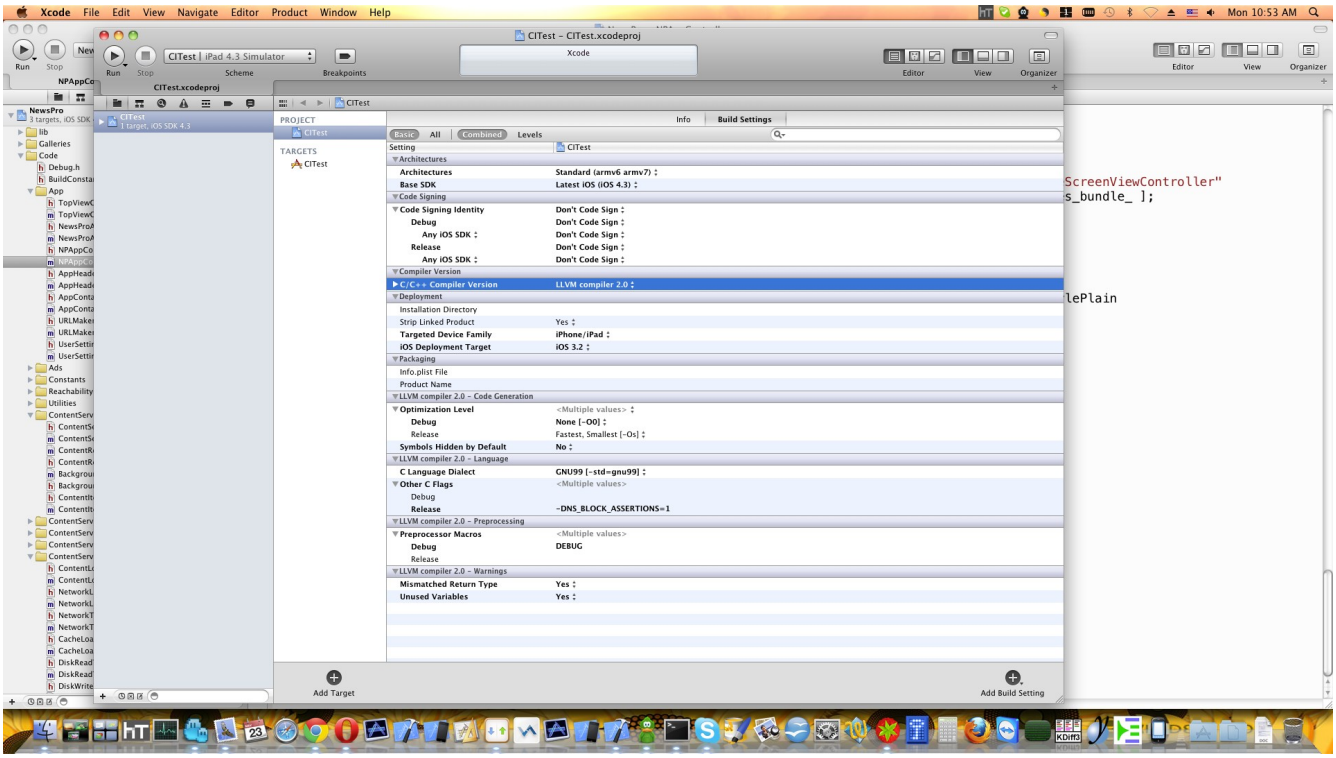


We refused from local git storage as we were going to deploy to github. We suggest using this option unless you have a reason not to do so.



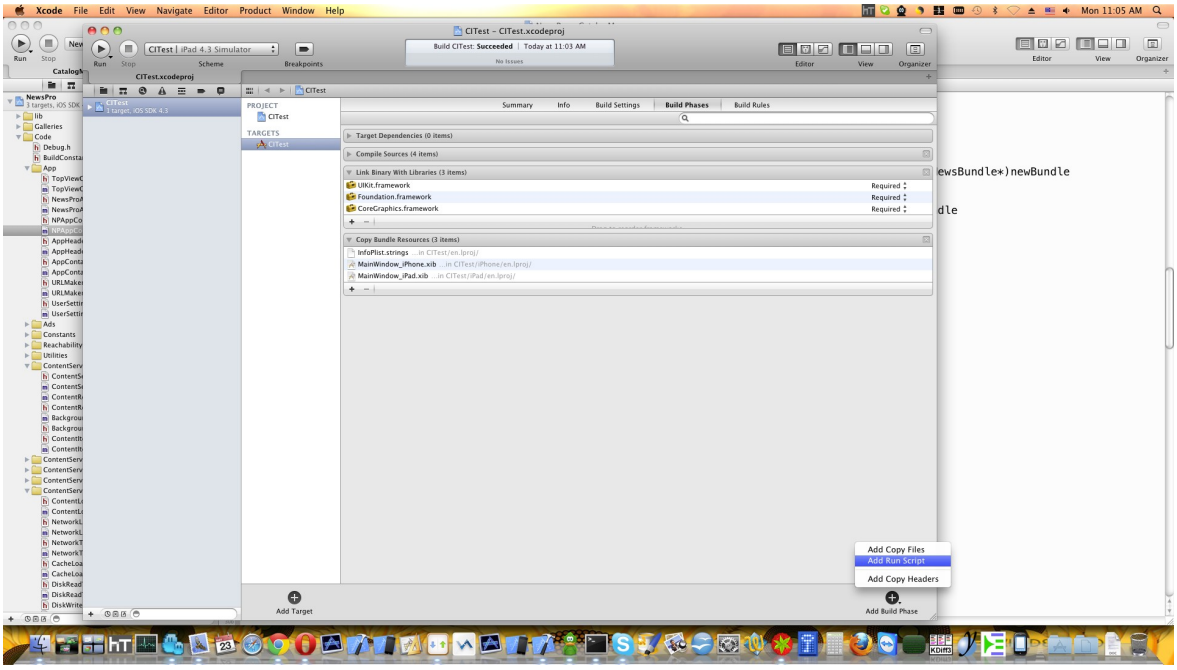
After that let's set the deployment target to iOS3.2 and compiler to LLVM.

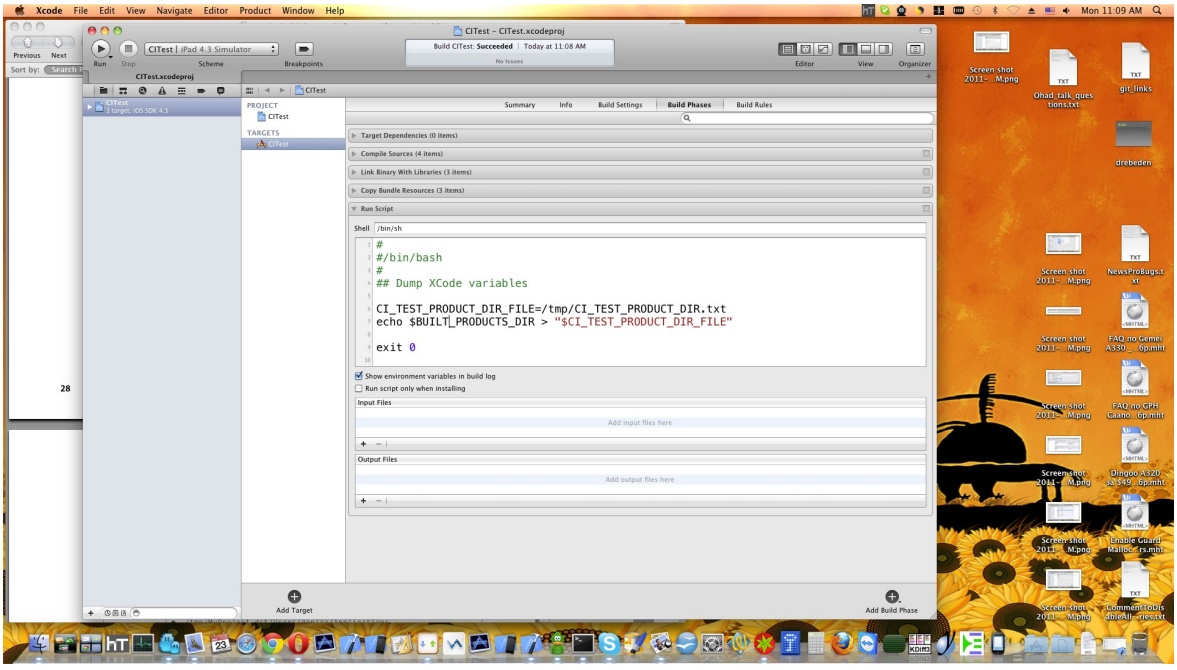




In order to deploy built products we have to get the path to them. As described in "**Xcode 4.x command line tools reference**" ***article,*** we cannot rely on relative to the project "**build**" directory and its structure. However, those variables are available at build time within the scripts, initiated by xCode.

That's why we have to dump those path entries into a temporary file and read them later. We'll add a "**run script**" build step to achieve this.



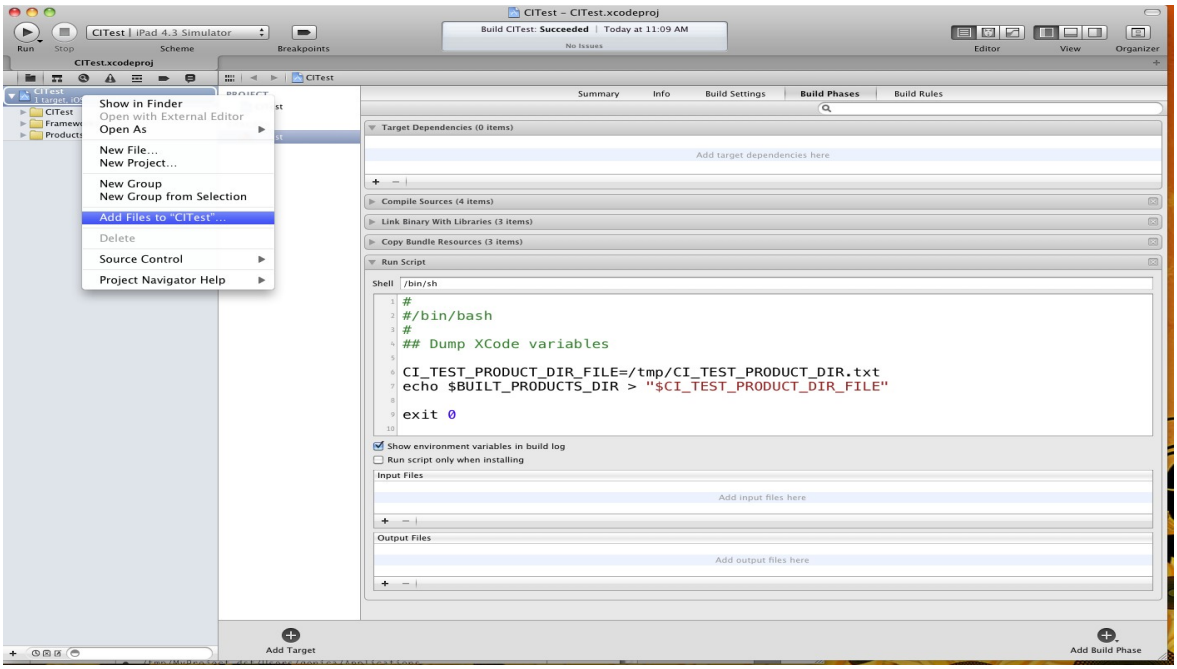


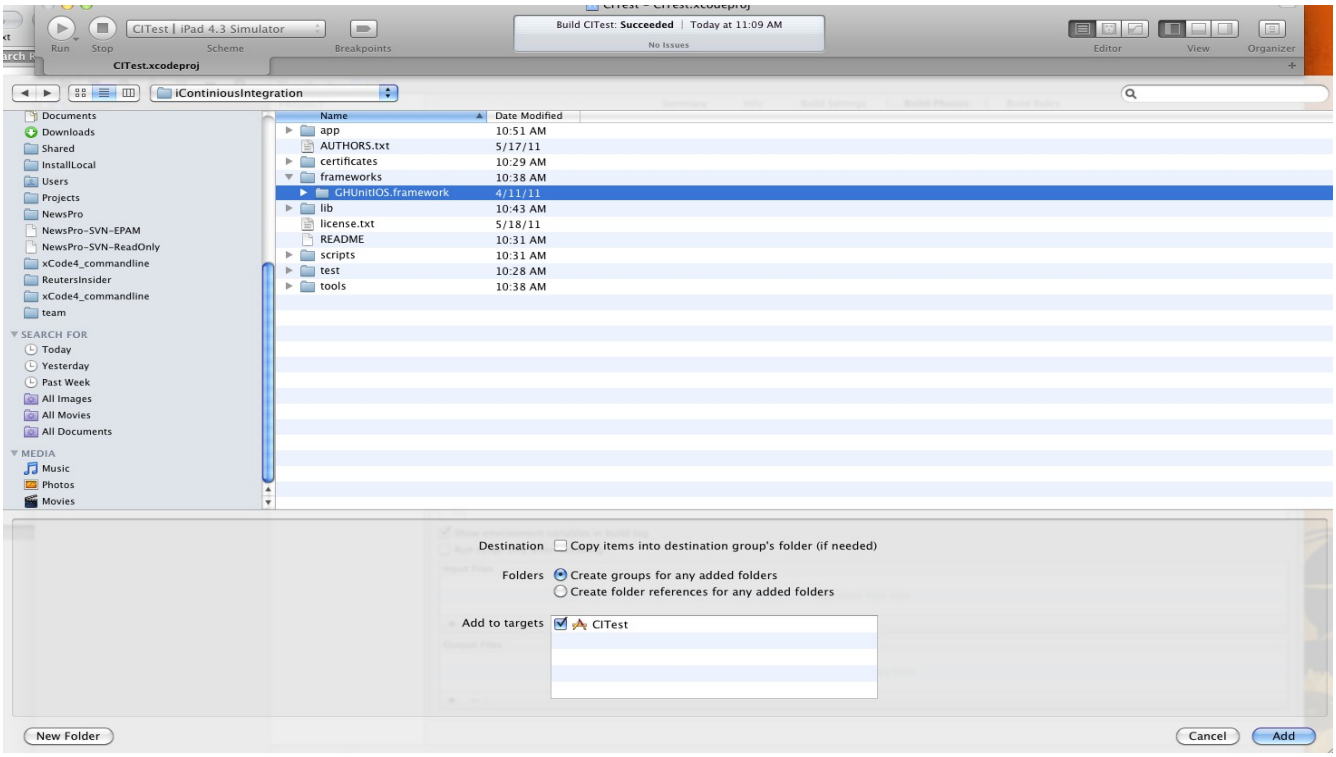
```
#
#/bin/bash
#
## Dump XCode variables

mkdir -p "/tmp/CITestBuild"
CI_TEST_PRODUCT_DIR_FILE=/tmp/CITestBuild/CI_TEST_PRODUCT_DIR.txt

cd "$BUILT_PRODUCTS_DIR"
cd ..
echo $PWD > "$CI_TEST_PRODUCT_DIR_FILE"

exit 0
```

We should add this step to all projects that will be deployed.

Adding GHUnit Framework

GHUnit framewok is added just like any other framework. The only difference is that you have to specify the path to it manually.



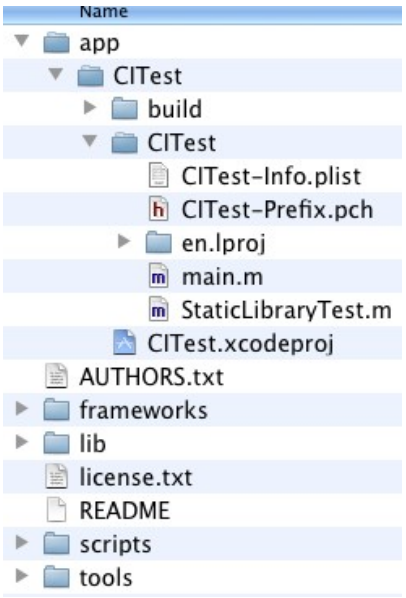


The second step is modifying the “main.m” file. You can find its contents in the ***GHUnit examples***.

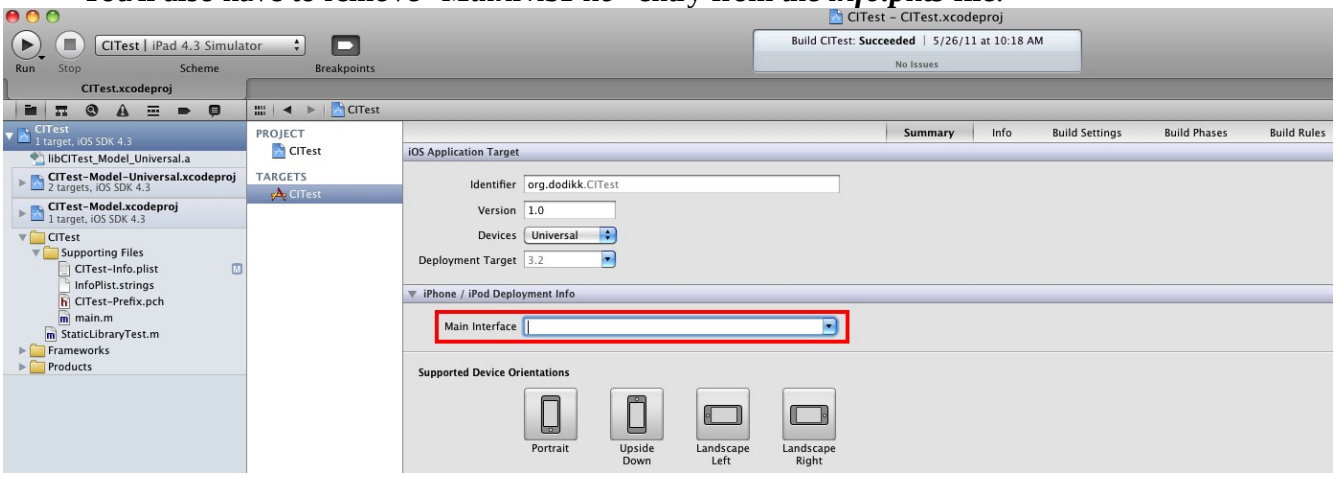
```
#import <UIKit/UIKit.h>

// If you are using the framework
#import <GHUnitIOS/GHUnit.h>
// If you are using the static library and importing header files manually
//#import "GHUnit.h"

// Default exception handler
void exceptionHandler(NSException *exception)
{
  NSLog(@"%@\n%@", [exception reason], GHUStackTraceFromException(exception));
}

int main(int argc, char *argv[])
{
  setenv( "GHUNIT_AUTORUN" , "YES", 1 );
  setenv( "WRITE_JUNIT_XML", "YES", 1 );
  setenv( "GHUNIT_AUTOEXIT" , "YES", 1 ); // Not supported in the official GHUNIT
  NSSetUncaughtExceptionHandler(&exceptionHandler);

  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

  // Register any special test case classes
  //[[GHTesting sharedInstance] registerClassName:@"GHSpecialTestCase"];

  int retVal = 0;
  // If GHUNIT_CLI is set we are using the command line interface and run the tests
  // Otherwise load the GUI app
  if (getenv("GHUNIT_CLI"))
  {
    retVal = [GHTestRunner run];
  }
  else
  {
    retVal = UIApplicationMain(argc, argv, nil, @"GHUnitIPhoneAppDelegate");
  }

  [pool release];
  return retVal;
}
```

You must also remove all view controllers and nib files, generated by the wizard. The remaining files are :

1. main.m
2. *.pch – precompiled headers
3. *-Info.plist



You'll also have to remove "***MainNibFile***" entry from the ***info.plits*** file.



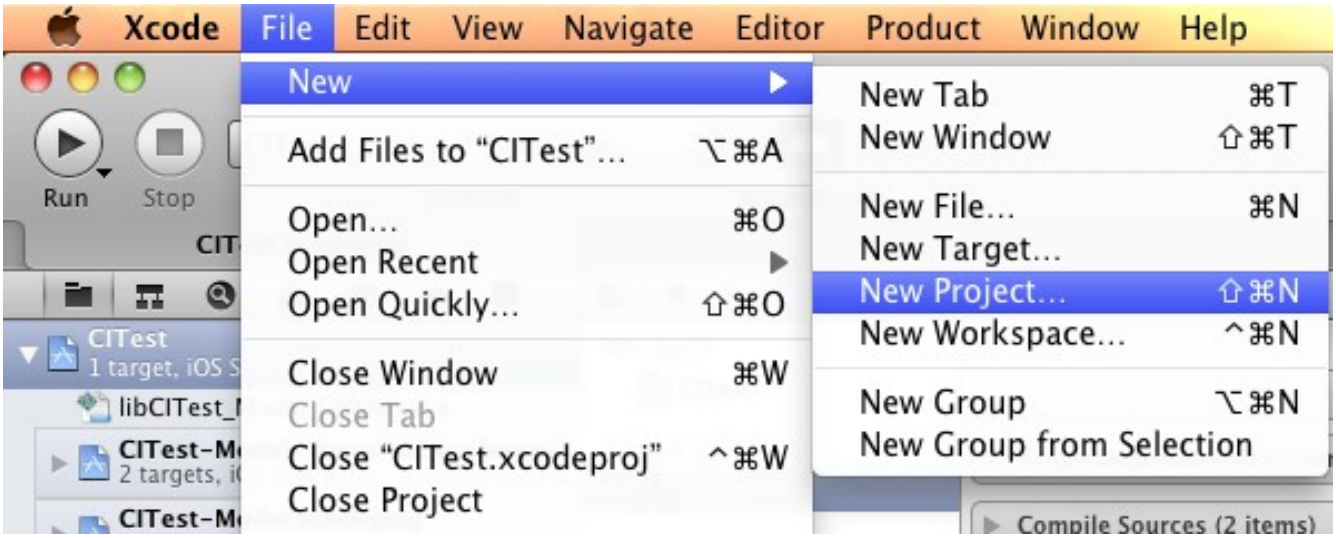
Now we are ready to add test cases. A test case is typically stored in a singe ***.m** file (both declaration and implementation). It may contain the "-(void)setUp" and "-(void)tearDown" methods to manage common test context. Test methods are started with the "**test**" word and should not be declared at the ***@interface*** section. They will be recognized by name, starting with "**test**".

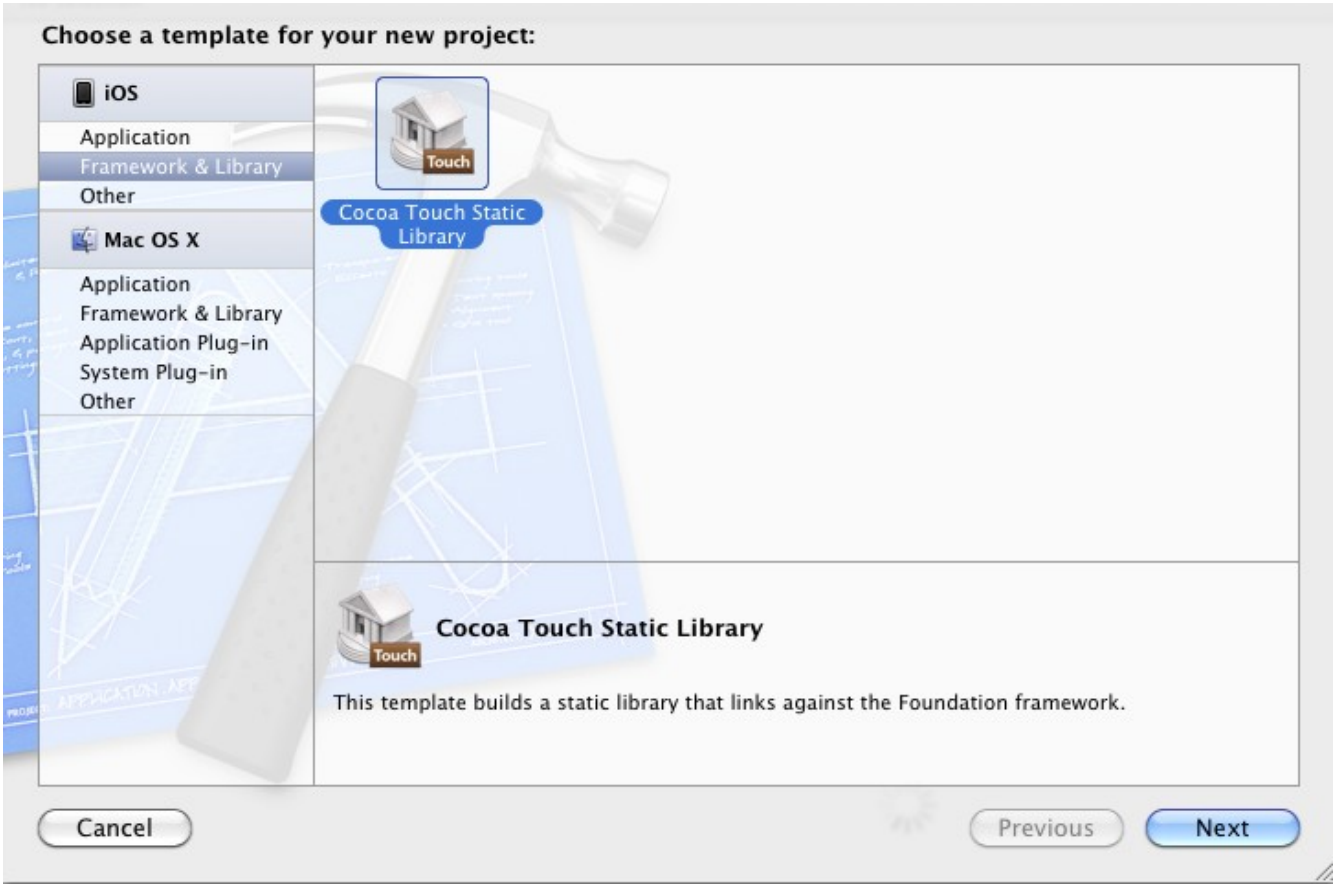
For example:

```
@interface StaticLibraryTest : GHTestCase
@end


@implementation StaticLibraryTest

-(void)testAdd
{
    // put test code here
}
```

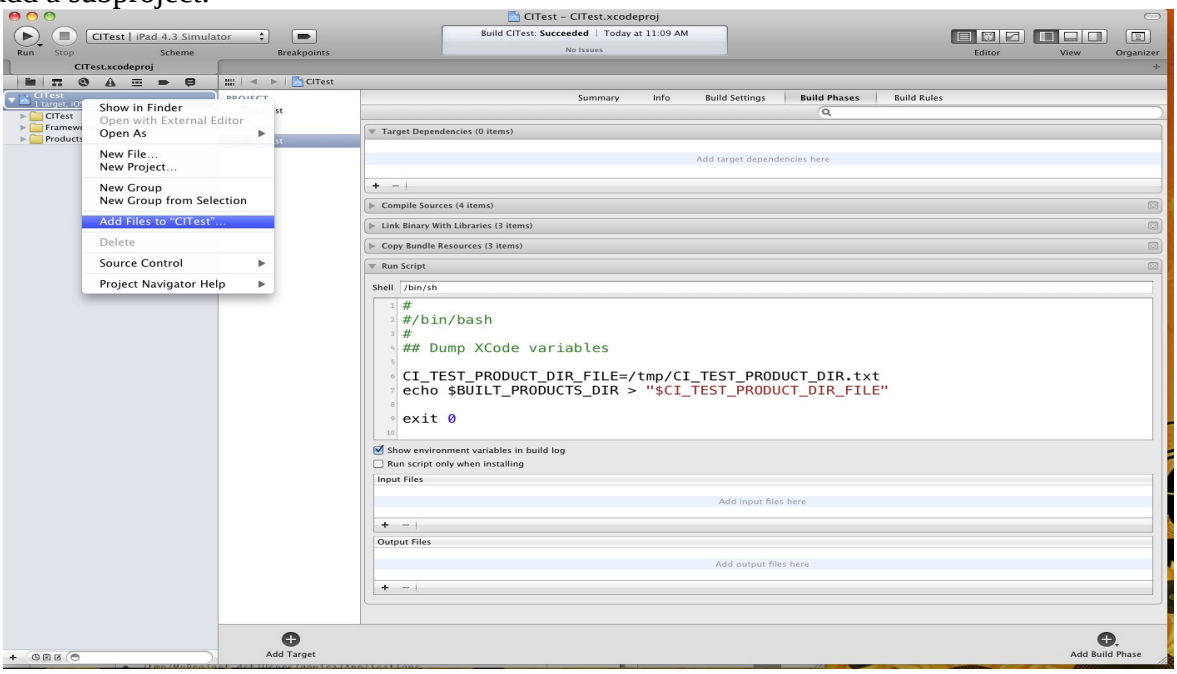
Adding a “Plain Library” Project

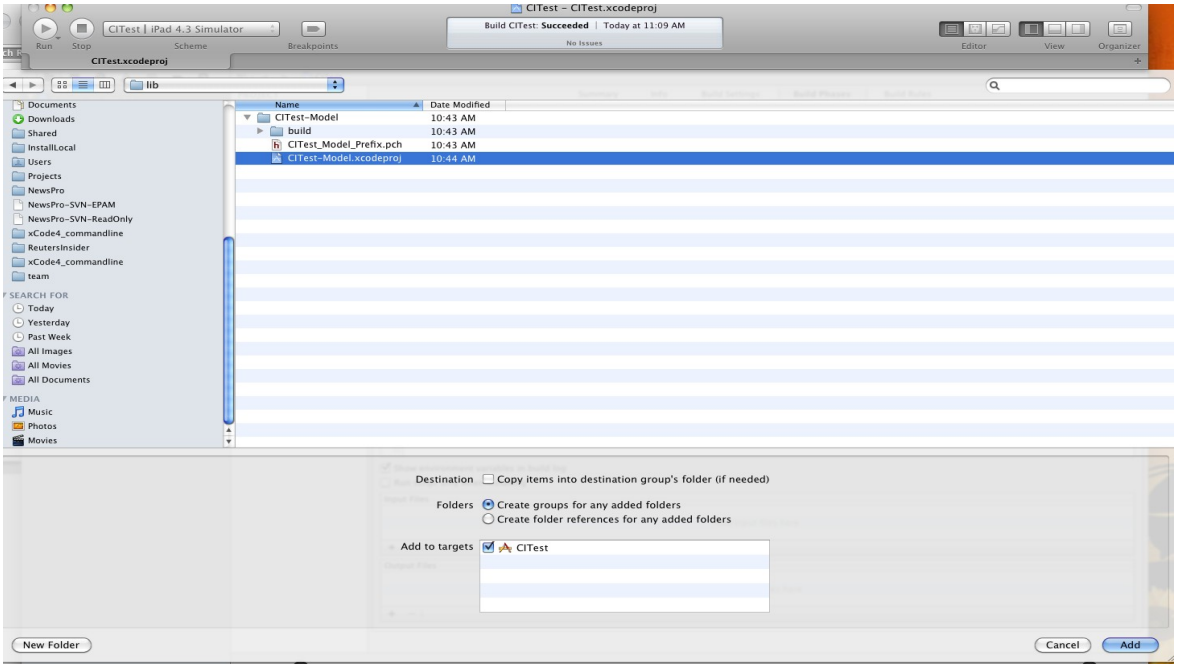
In order to create a library project, just use a wizard within the xCode.





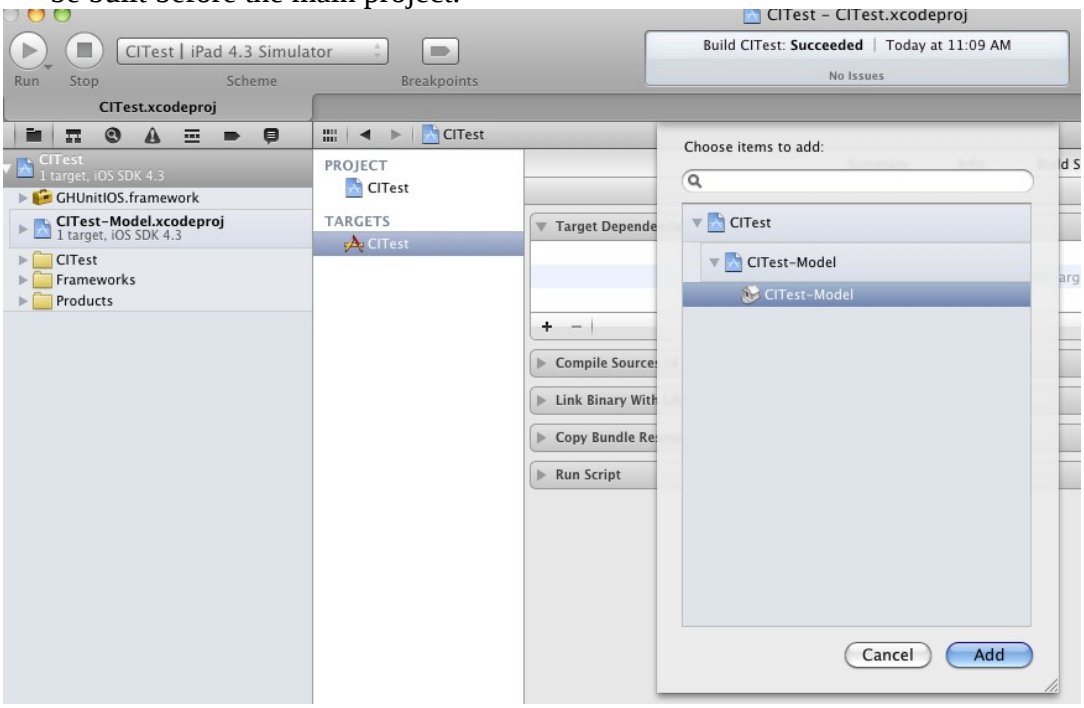
Once it has been created, we'll add some sample code and link it to our main project. First of all, we'll add a subproject.



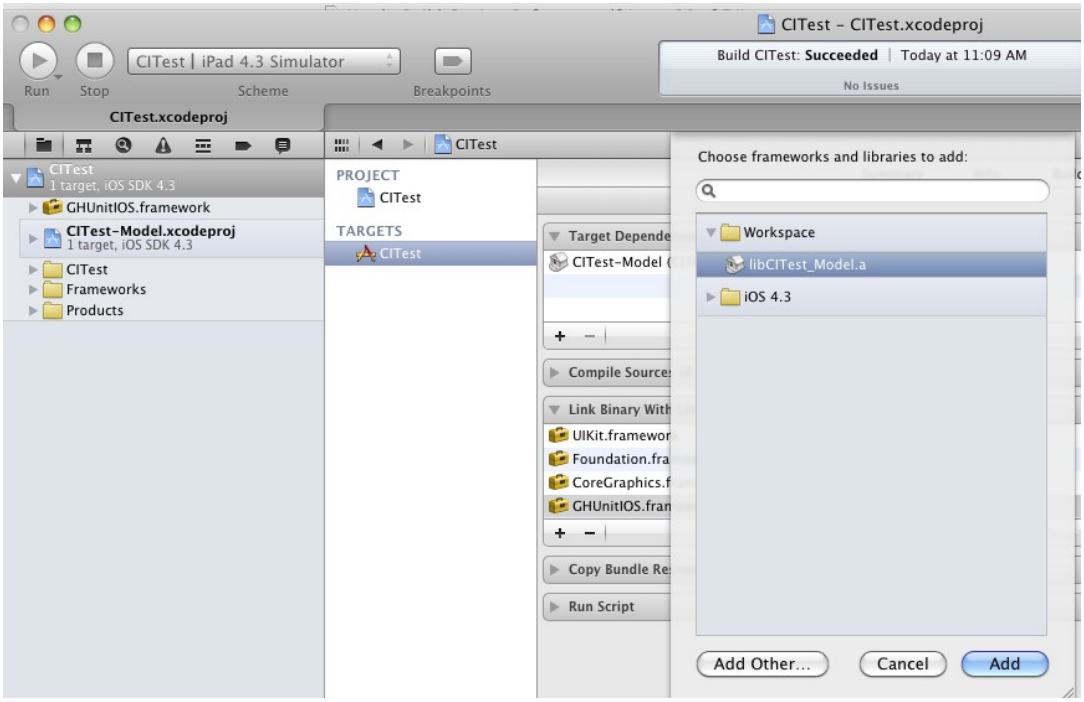


Once added, we'll have to set up the dependencies to it. Dependency types are :

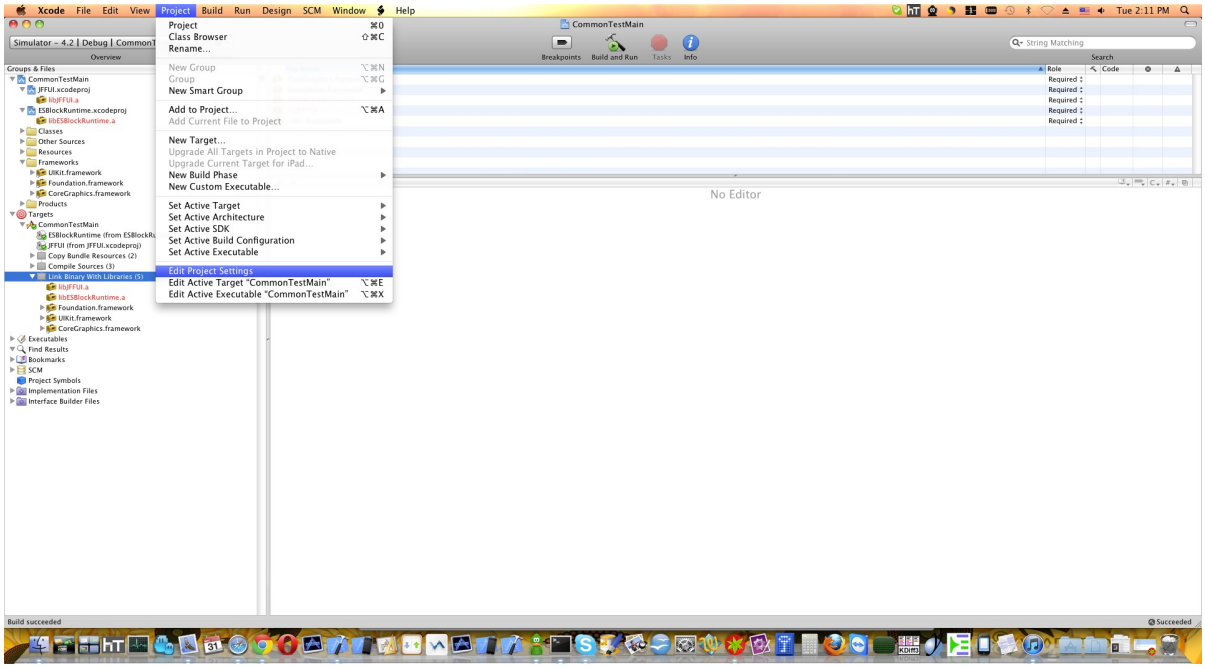
1. **Build time dependency** – we have to make xCode know that a referenced project should be built before the main project.

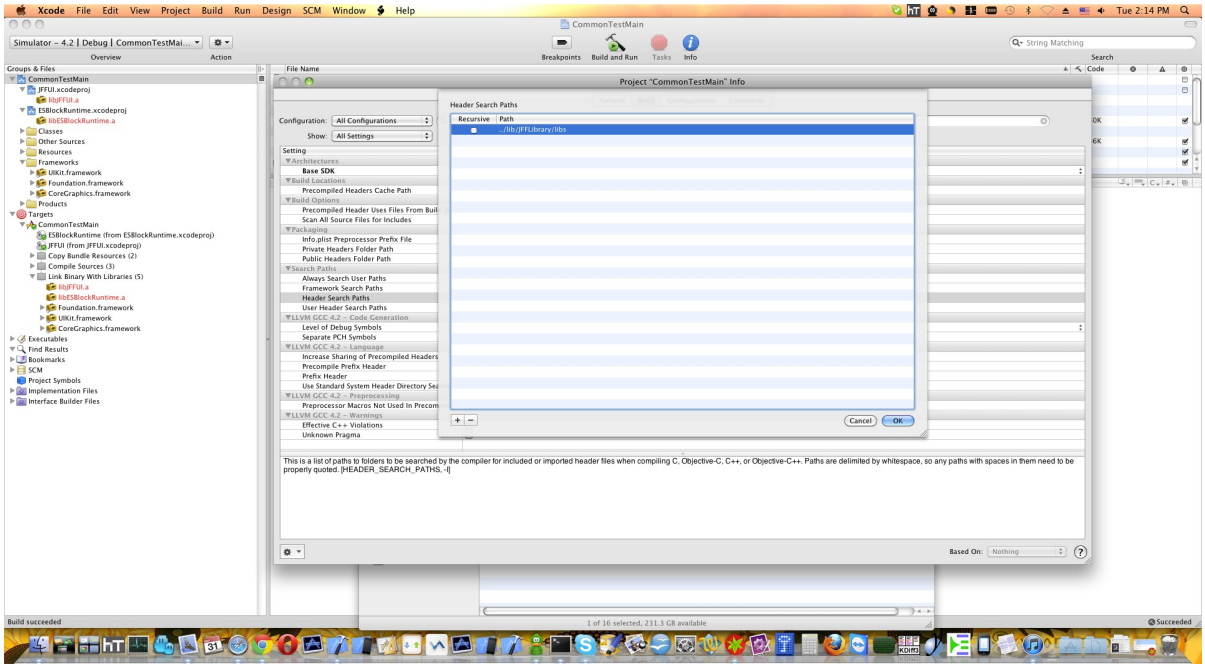


2. **Link dependency** – the library must be linked with the main project. Otherwise the symbols will be inaccessible. Hence, the application will either fail to build or crash at runtime.



3. **Header files include dependency** – we need to access header files with the interface declarations.





And that's it! Now we can use code from the library. xCode will now automatically ensure that library subproject is built at the correct time and that its correct version is linked.

Creating a “Universal Binary” Library

The Purpose of “Universal Binaries”

Normally, xCode builds two versions of a library. One is for real devices. The other one is used for the simulator enabled application builds. It is very different from desktop projects where you have only one version of the library.

This is not a problem since you have library sources and can compile it by yourself. However, the library may contain some know-how code you will not want to distribute in a text form.

(** NOTE : we are not covering anti-disassembling and reverse engineering techniques in this article **).

In this case you may want to distribute it just as you distribute any desktop targeted library. You provide your customers a set of headers and a binary. Meaning that you can link this to both the device and the simulator applications.

It is not a good idea to still have two separate library binaries because of some peculiarities for libraries management. xCode just won't handle them properly.

Apple uses the same approach for its “frameworks”. However, frameworks are officially unavailable for iOS. Still, some libraries are packaged and successfully used in this way. An example is **GHUnit.ftramework**.

We won't cover framework deployment in this article. However, we'll show how to make a universal binary and properly deploy it.

Creating a Universal Library

Firstly, you have to create a ***plain static library***. Just as described above.

The next step is to combine device and simulator versions. We'll use a shell script for this purpose. The script will:

1. Build a library version for the device.
2. Build a library version for the simulator.
3. Combine them to a single binary
4. Deploy universal library to the “**frameworks**” directory.

We build device and simulator versions using **xcodebuild** command line interface.

Device version :

```
xcodebuild -project CITest-Model-Universal.xcodeproj -target CITest-Model-Universal -configuration Release -sdk iphoneos4.3 build
```

Simulator version:

```
xcodebuild -project CITest-Model-Universal.xcodeproj -target CITest-Model-Universal -configuration Release -sdk iphonesimulator4.3 build
```

For details, see “xcode4 command line manual” article. Or just use “***man xcodebuild***”.

Once specific (device/simulator) versions of the library are ready, we need to locate them on the file system. We do this by reading the path entry, dumped by the xCode script, described above.

```
LIB_BUILD_DIR=$(cat /tmp/CITestBuild/CI_TEST_UNIVERSAL_LIB_PRODUCT_DIR.txt)
```

(*** Note : you can safely dump project path entries to any other locations. ***)

Device and simulator library versions are stored at "$LIB_BUILD_DIR/Release-iphoneos" and "$LIB_BUILD_DIR/Release-iphonesimulator" respectively.

In order to combine them into a single binary we use the following command :

```
lipo -create "${LIB_BUILD_DIR}/Release-iphoneos/libCITest_Model_Universal.a" "${LIB_BUILD_DIR}/Release-iphonesimulator/libCITest_Model_Universal.a" -output "../frameworks/CITest-Model-Universal/Lib/libCITest_Model_Universal.a"
```

(*** Note: the order of libraries is important here.
A library for the device goes first. The one for the simulator goes next ***)

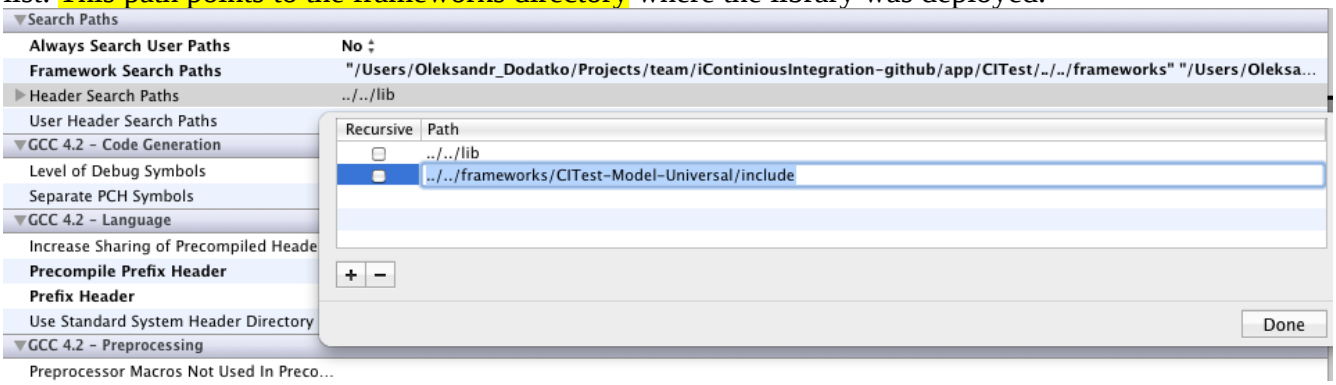
Now we only need to copy the sources to the location beside the binary.

```
cd ../ lib/CITest-Model-Universal
  cp *.h "../frameworks/CITest-Model-Universal/include"
cd "$LAUNCH_DIR"
```

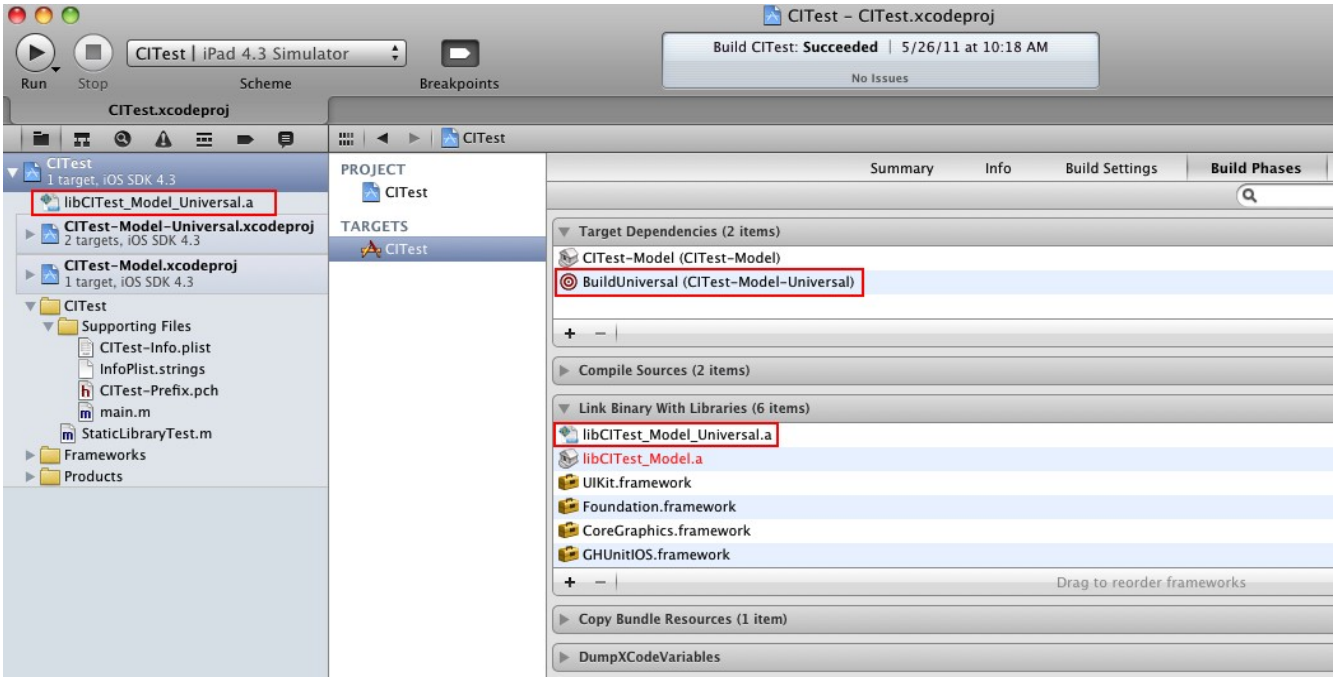
Still, you must make sure that your deployment directory exists and the results can be saved in it.

Using Universal Binaries

In order to use a universal binary, you must add the file system path to its headers to the search list. This path points to the frameworks directory where the library was deployed.

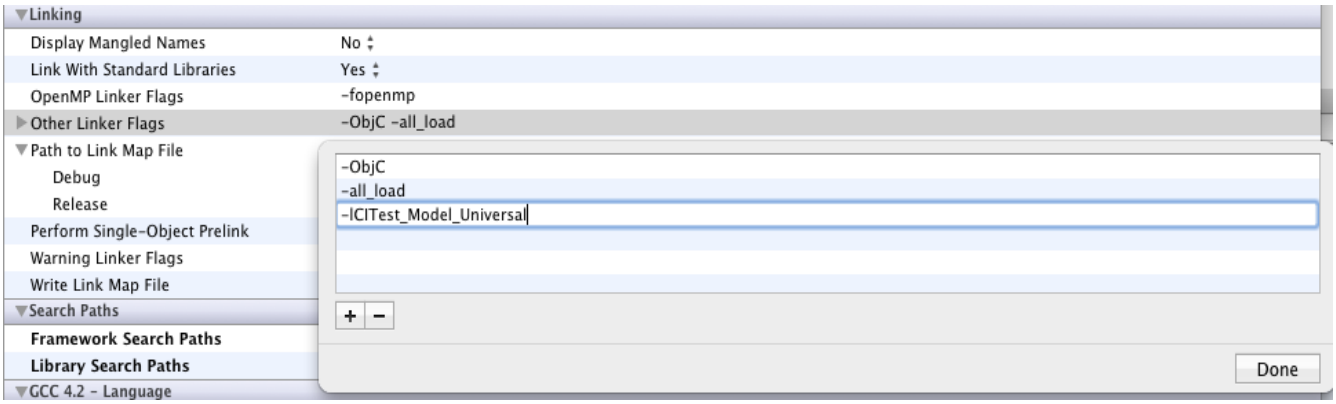


You should also add the library file to the project and to its linker dependencies.



Alternatively, you could link it with the console linker flag **-l<library name>**.

For example, **-lCITest_Model_Universal** . Please note that the filename is **libCITest_Model_Universal.a** . So, **lib** prefix and **.a** suffix have been omitted.



Deploying Main Application

We deploy our applications for the simulator by sending our testers the ***.app** bundle, built for the simulator. It can be launched and tested with the help of the **iphonesim** utility as described above. It is produced by the xCode. So, no further steps should be taken.

For the device we must create a digitally signed ***.ipa** file from the device ***.app** bundle.

In order to do this, we must do the following things :

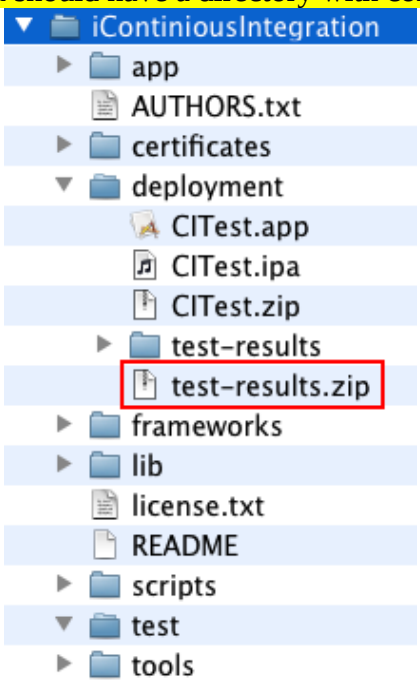
1. Build the product.
   Just as described above for the universal library.

2. Locate a bundle on the file system
   Once again, we do it by reading the contents of the file, dumped by the xCode script.
   ```
   BUILD_DIR=$(cat /tmp/CITestBuild/CI_TEST_PRODUCT_DIR.txt)
   ```

3. Prepare information about provisioning certificates
   ```
   DEVELOPER_NAME="iPhone Developer: Oleksandr Dodatko (ABCDEFG123456)"
   PROVISONING_PROFILE=../certificates/CITest.mobileprovision
   ```
   (*** Note : this should be adjusted for your provisioning ***)

4. Finally – create the ***.ipa** file
   ```
   /usr/bin/xcrun -sdk iphoneos PackageApplication -v "${BUILD_DIR}/Release-iphoneos/CITest.app" -o "${DEPLOYMENT_DIR}/CITest.ipa" --sign "${DEVELOPER_NAME}" --embed "${PROVISONING_PROFILE}"
   ```

After successful deployment you should have a directory with contents like this:



Creating a Hudson Job

Once you are able to build the project on your local machine, you should set up a job at the build server.