Setting up your 'Snapshot' environment for Buzztouch projects

Document version: v1.006

Snapshot. What a great piece of software. It automates screenshots. It may not seem like much, but when you think that you have to cover up to 6 devices, even a few screens can easily add up to 30 or more. And if you consider multiple languages, it starts to get rather tedious, and time consuming.

Enter Snapshot. It's pretty simple to install and use, but for those who aren't familiar with some of the methods, it might be a bit daunting. Let's try and simplify it, so everyone can utilize this fantastic method of automation.

Snapshot is a part of 'Fastlane', a complete automation solution for parts of Android and iOS app prep. I suggest you get Fastlane, and use Snapshot as a part of it, but this tutorial just focuses on the issues I've seen setting up Snapshot.

I hope it's not too confusing; I'm writing about 'snapshot', but "I" am using it as part of fastlane, and not alone. Please try to understand the differences when looking at the examples. However, launching from the command line is pretty similar.

Installing Snapshot or Fastlane

You install snapshot from the command line. Instructions (that should be used in conjunction with these instructions) are available at:

https://github.com/fastlane/fastlane/tree/master/snapshot

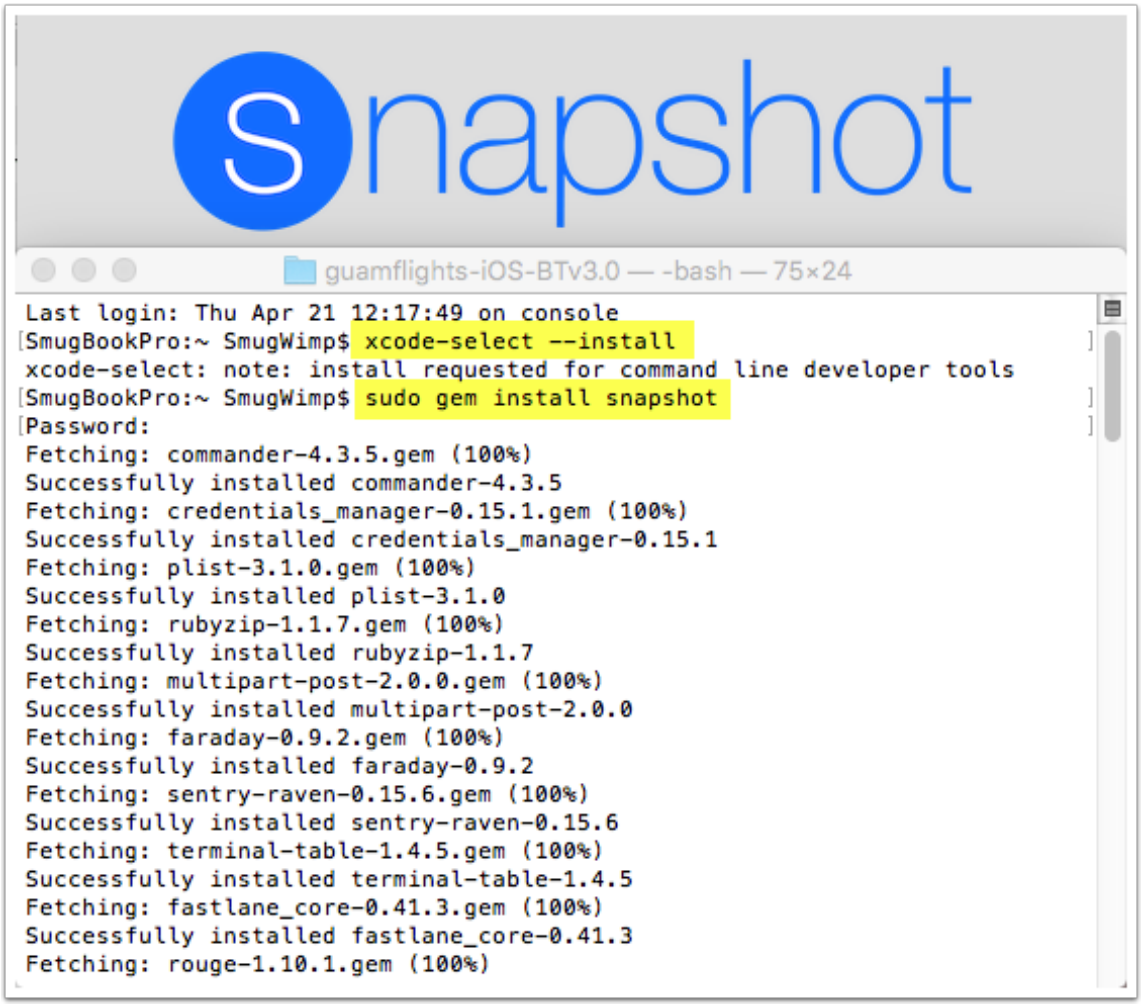
Follow the same instructions as the installation notes;

1) using terminal, type 'xcode-select --install' <return>

It takes a few minutes, depending on machine, internet, etc...

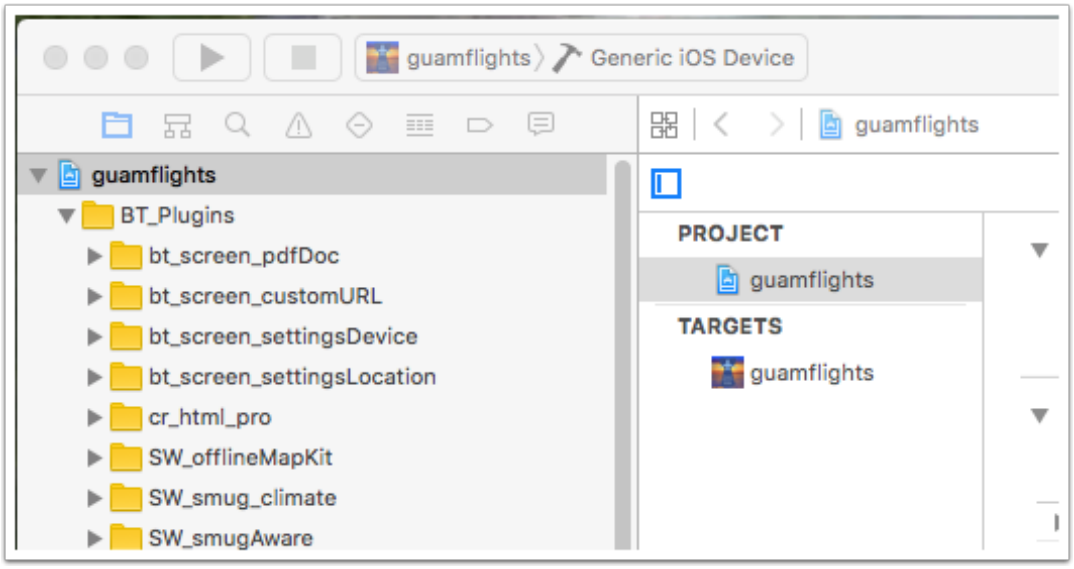
2) using terminal, type 'sudo gem install snapshot' <return>

This takes a few minutes, but after everything is installed it should have a message like 'XX gems installed'. Mine said '12 gems installed'. yours may be different, depending on how few/many other kinds of packages you've installed in the past.

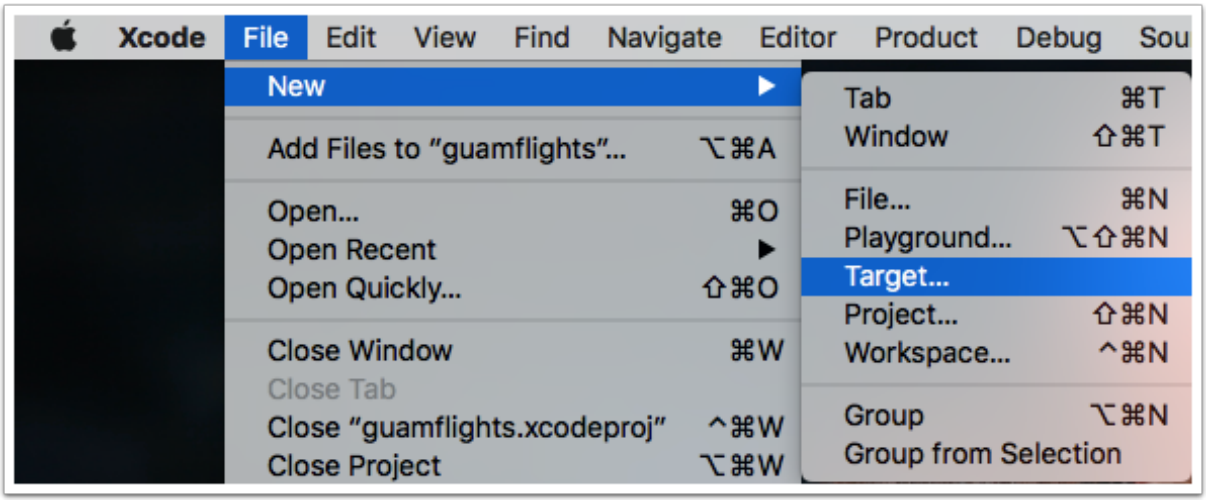


Add a 'test' target to your desired project

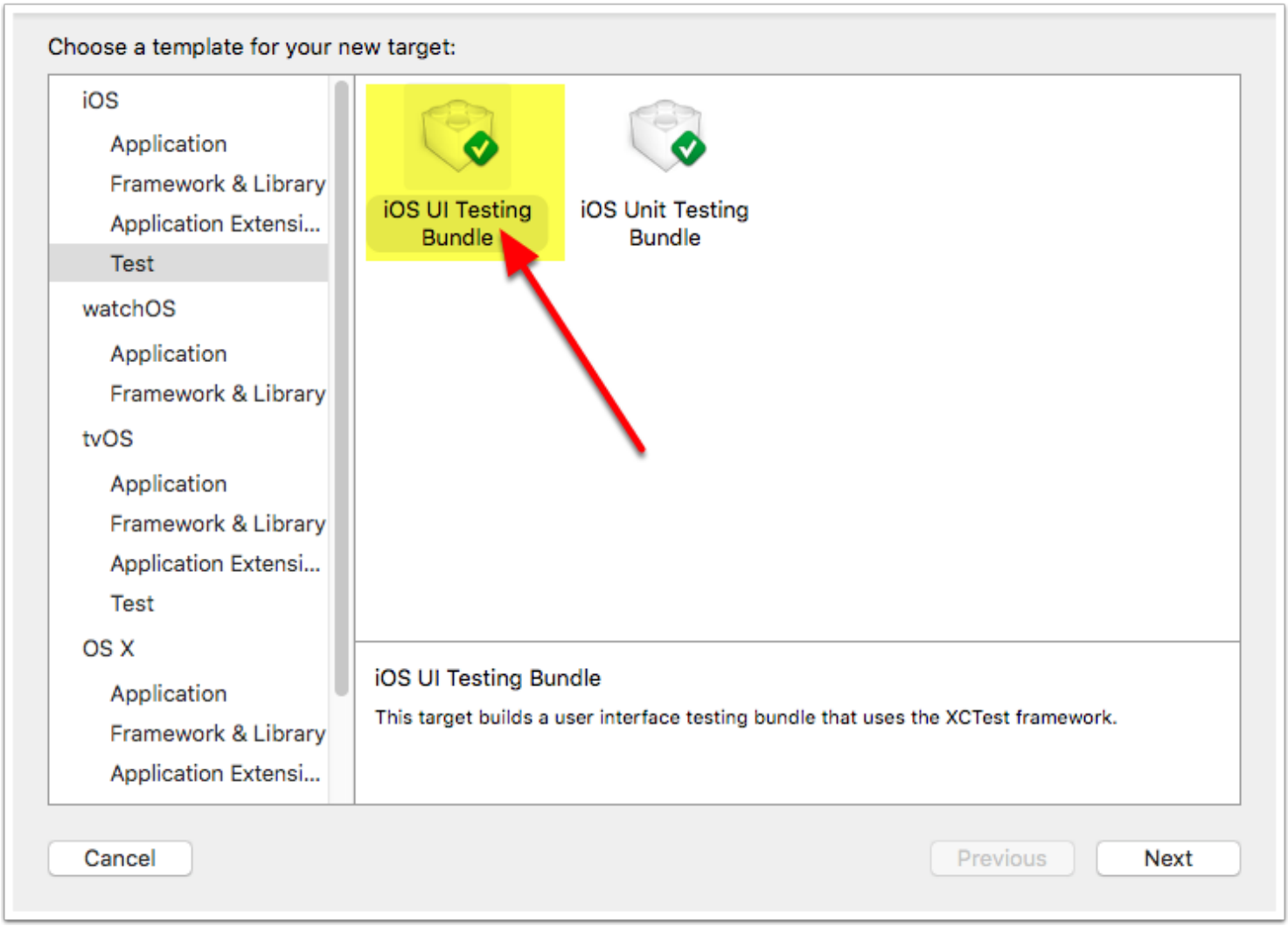
You'll need to do this for each project you want to create snapshots with.



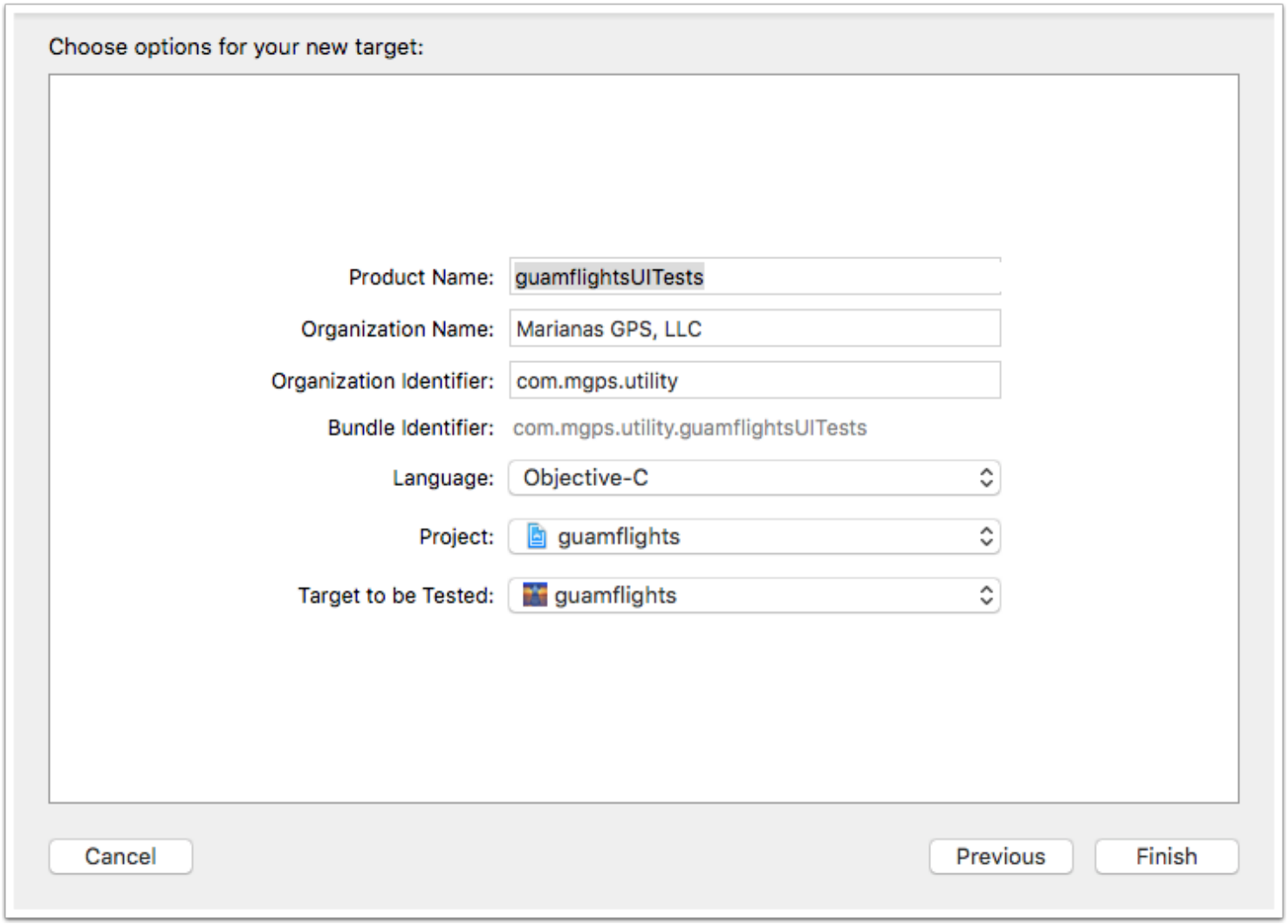
Add New Target



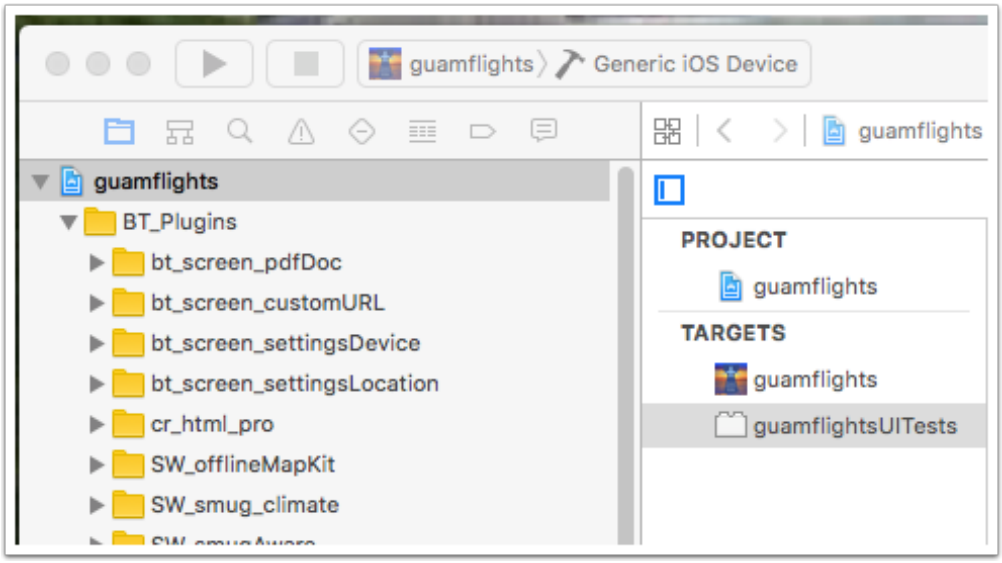
Choose iOS UI Testing Bundle



Leave as default



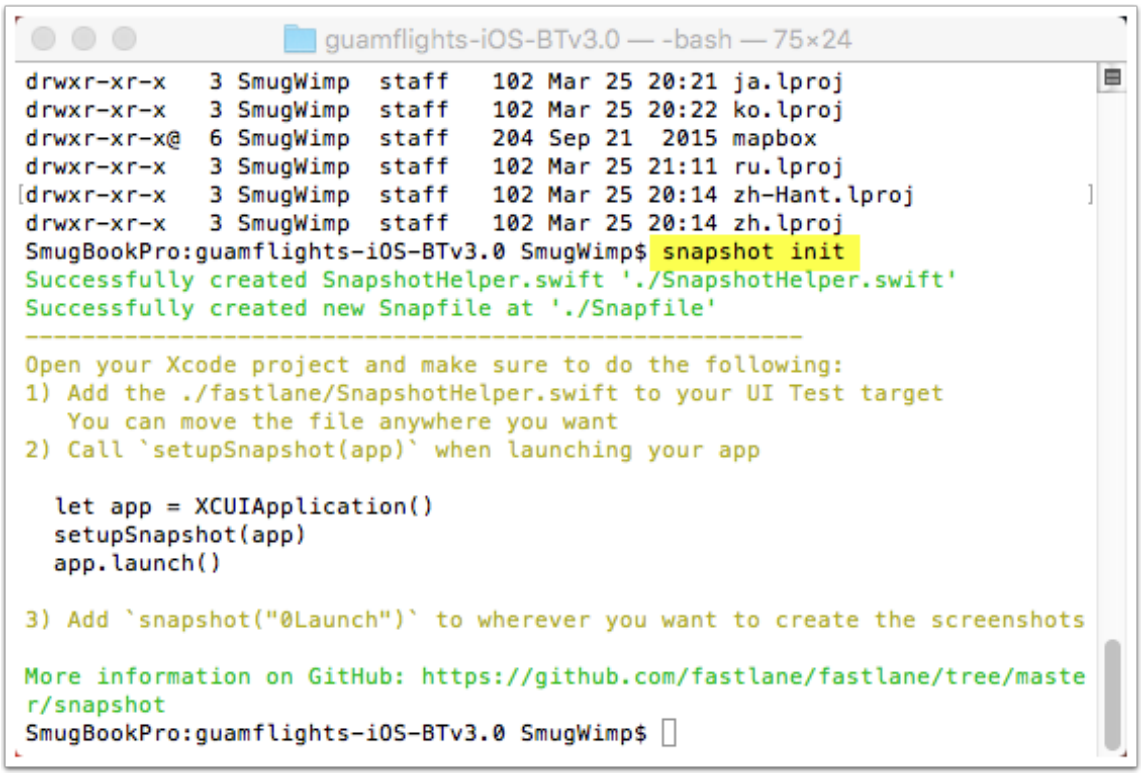
You now have a 'test' target. Congrats. Have a cool refreshing drink.



Back to the command line...

In your terminal window, type 'snapshot init' in your project root directory (the same directory that has 'appName.xcodeproj)

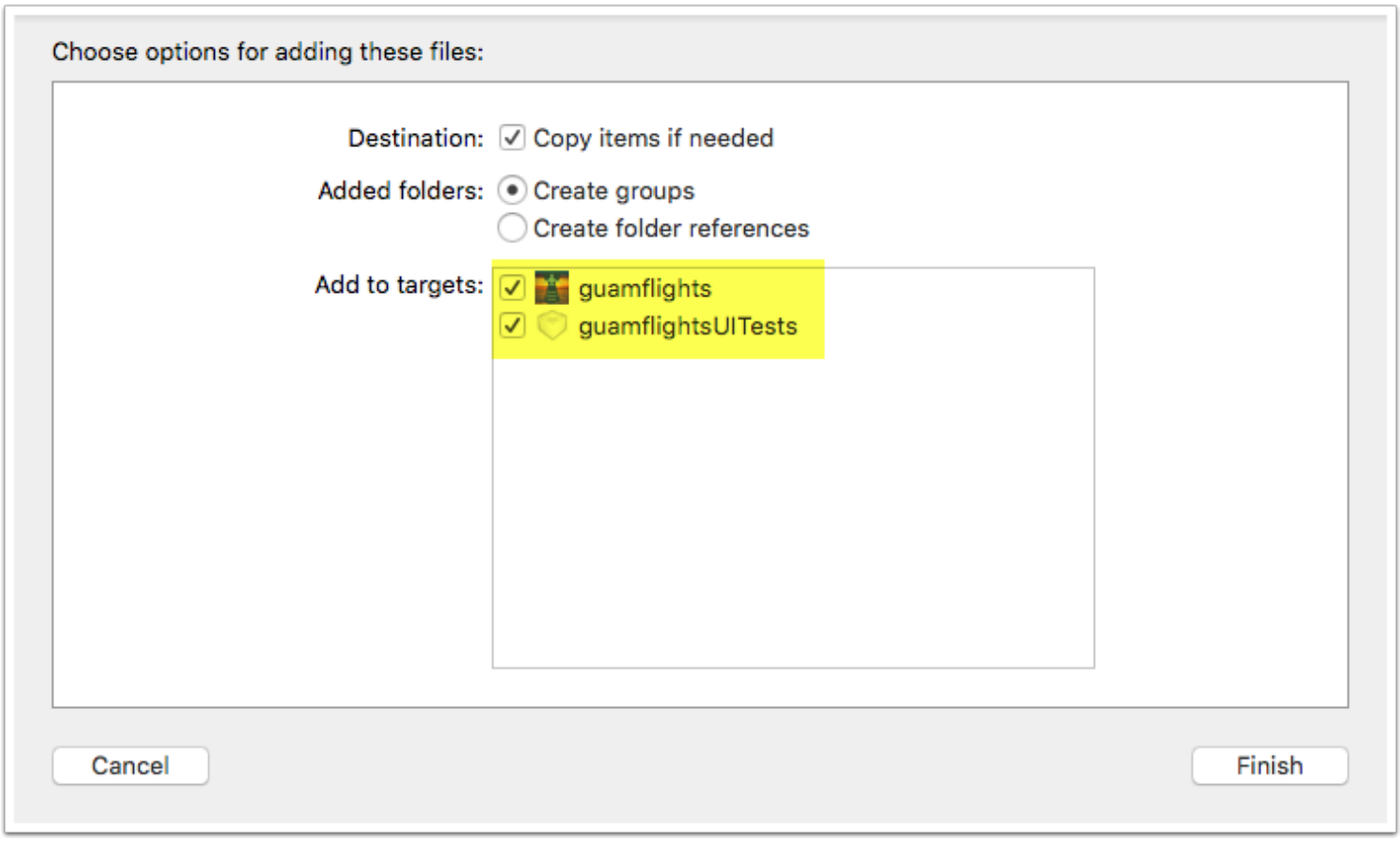
Your results 'should' be similar to those in the image. If not, please go back and check that you have not skipped any steps.



Adding your Snapshot files to your project

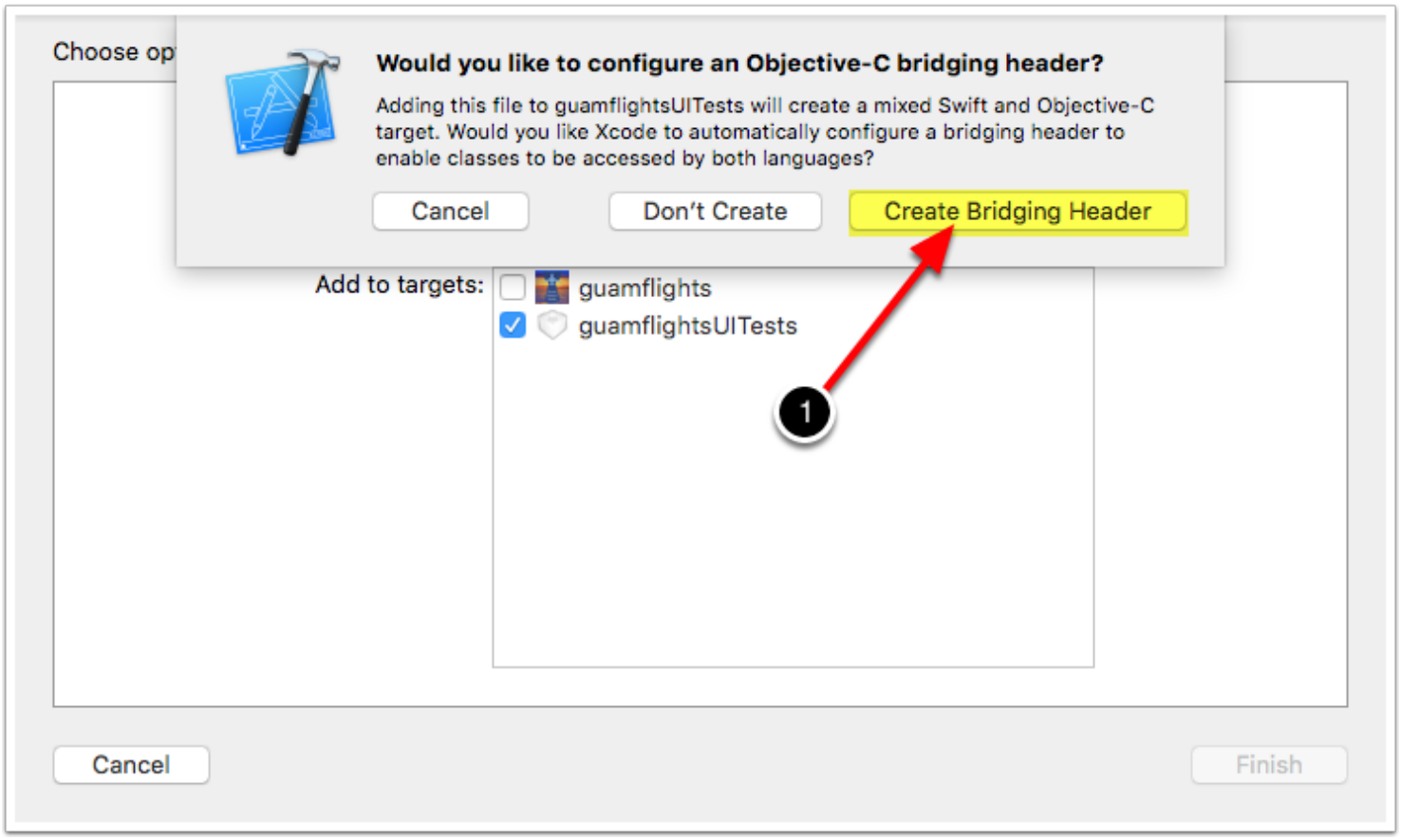
After running the command, it will generate two files; 'SnapshotHelper.swift' and 'Snapfile'.

Add the 'SnapshotHelper.swift' file to your project. Be sure to deselect your 'production' app as target, and select the 'UITests' target. Where do you put it? Pretty much anywhere. I put mine in my <projectname>UITests directory, so I'd remember it was there. You're welcome to put yours in a different location.



Create a Bridging Header between ObjC and Swift.

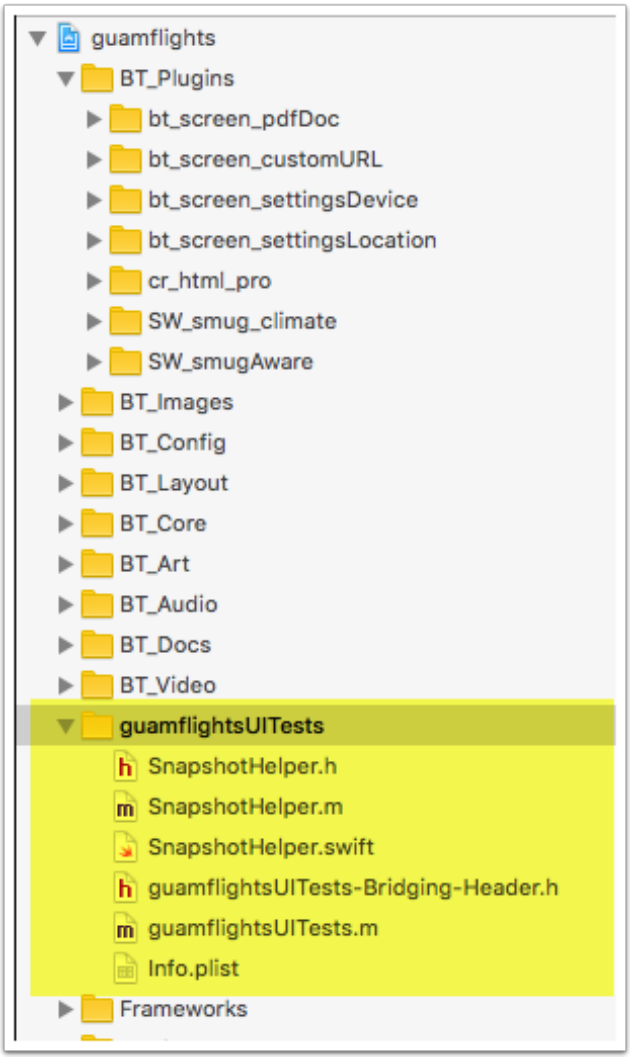
1) If added correctly, it will prompt you to create a 'bridging header' to tie the ObjC and Swift code together, sort of. Yes, Create Bridging Header.



Your app project should reflect 'similar' structure.

Wait... Where did that "SnapshotHelper.h" and "SnapshotHelper.m" file come from?

It came from a problem I had; during compile, regardless of adding 'import', it could not find my swift file. After googling the problem, I discovered that sometimes it works, sometimes it doesn't. In my case, it didn't. So I had to add the h/m file and reference it, even though the swift file is still required. You can grab a copy off my server: https://www.marianasgps.com/public/SnapShotHelper.zip

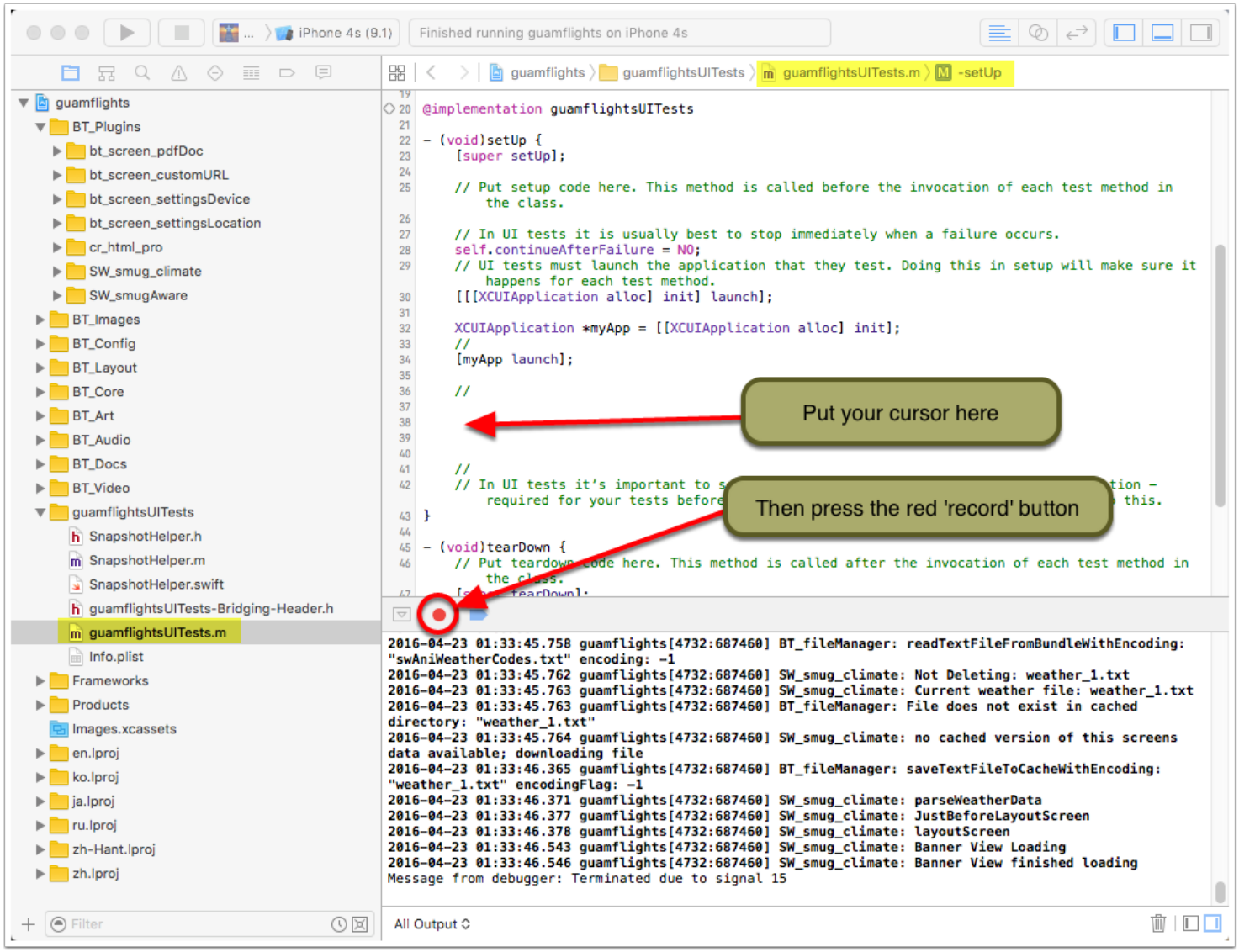


Ok, now what?

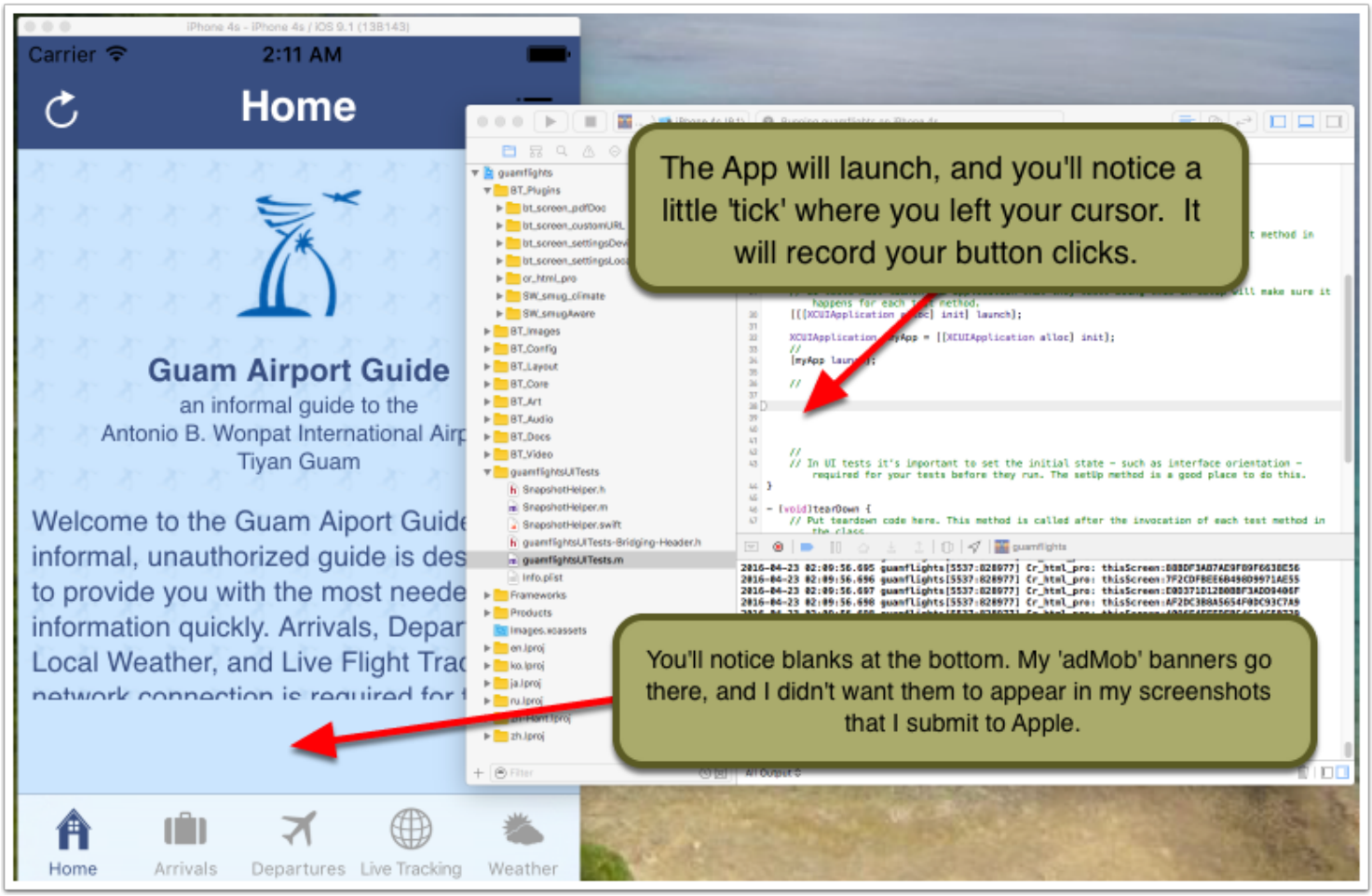
Now you start to 'record' the screens that you want to have created. You do this in the test environment, NOT the traditional way.

1) Place your cursor in the middle of the 'setup' method, after 'launch' is called

2) Press the red 'record' button. It will launch the app.



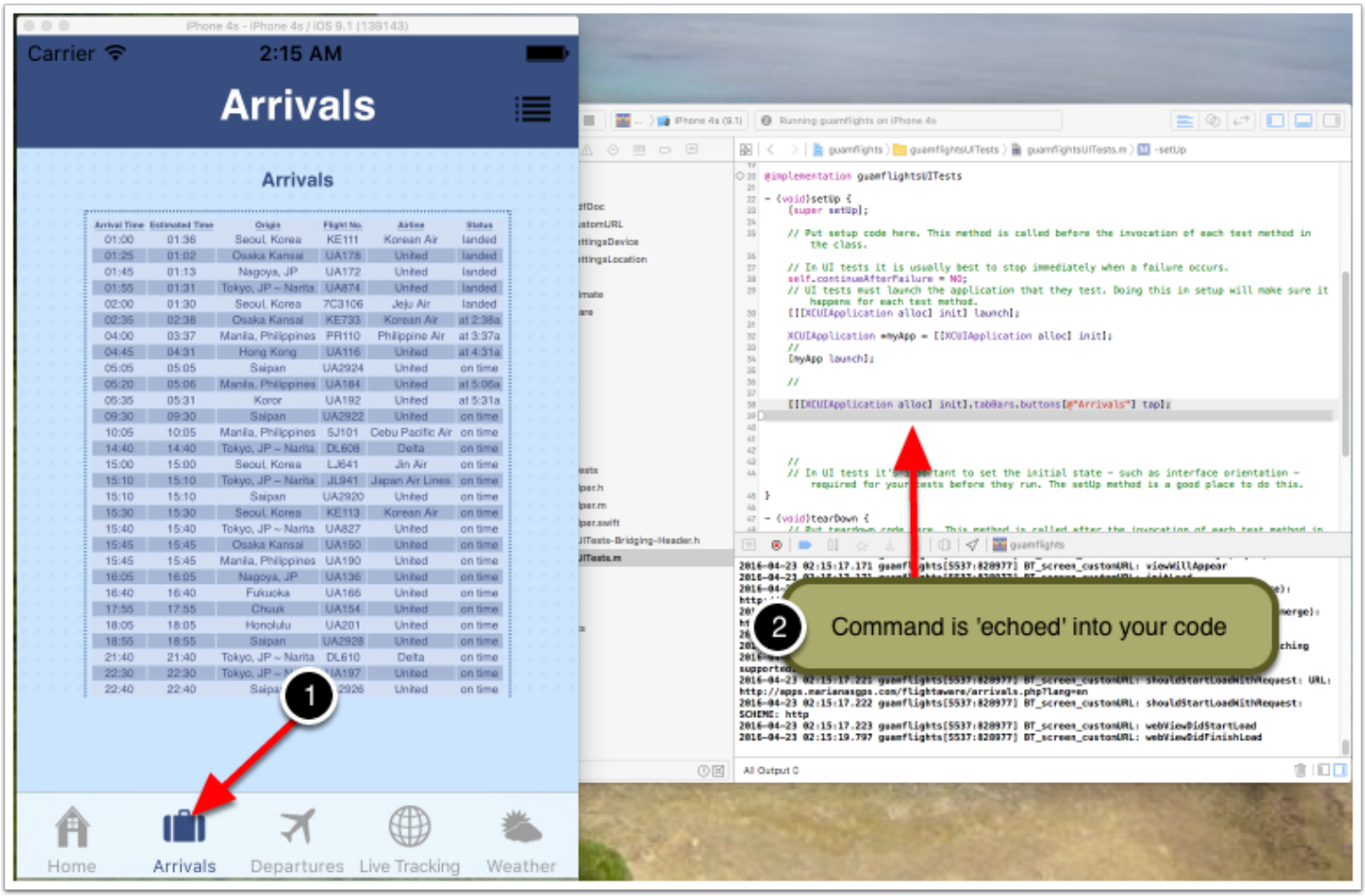
Launch Test App



Start your clicking

So, now just 'navigate' to the screens you want images of. Each click will echo a command that is inserted into your 'setup' method.

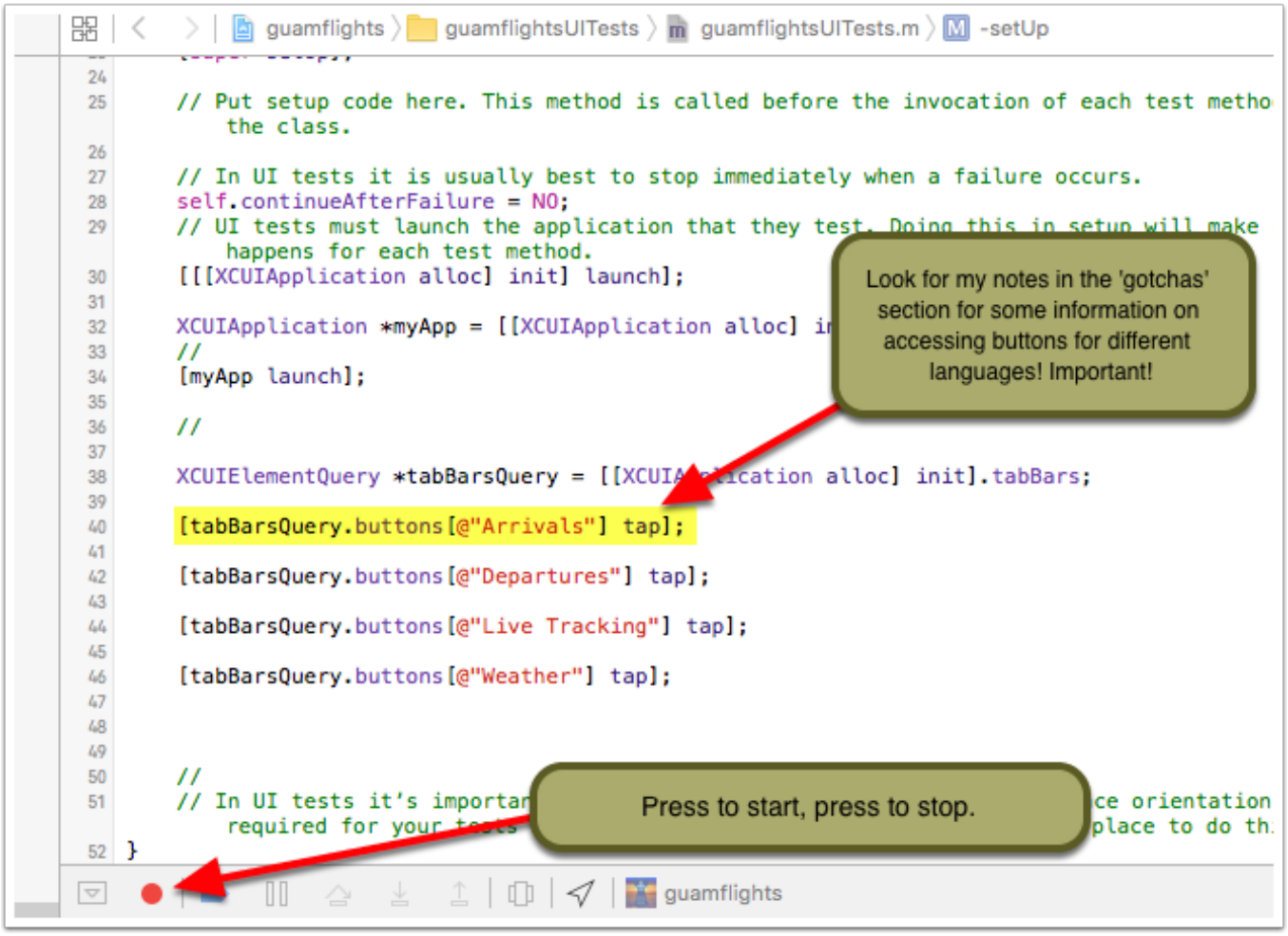
1. You click on a screen element
2. The command to emulate that screen click is placed in your projectNameUITests.m file



Finished? Not yet.

Now, you have the commands to replicate navigating to the screens. Now we need to add the commands to take a screen shot.

Press the red 'record' button again to halt the recording.



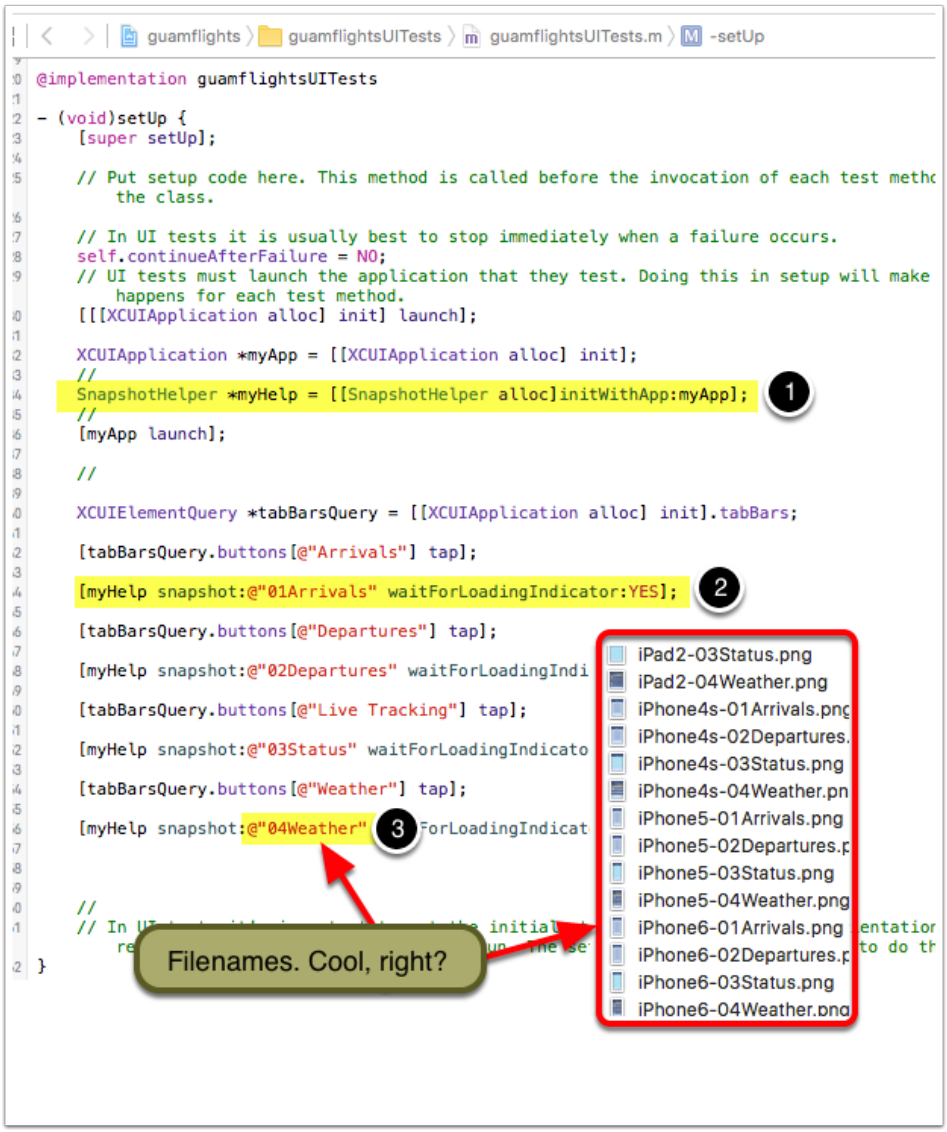
Creating an Instance of 'SnapShotHelper' and using it to create your shots at the right screens...

This is no definitive manual, but I hope it will get you started enough to where a little trial and error will get you where you need to go.

1. Create an Instance of SnapShotHelper, and place it below the app declaration, but before launch.
2. Add a 'screenshot' command to each screen you wish to create.
3. If you're going to use 'Deliver', one of the scripts in Fastlane, remember that they are uploaded in alphabetical order. If you want your screen shots to be automatically positioned the way you desire, it's best to ensure that the name of the file can be easily sorted the way you want. Adding a numerical order to the name makes it pretty simple.

What is happening: during the 'test', the script 'steps' through each line of your setup. some can be done semi-automatically. Others must be manually inserted, such as the command for your screenshots.

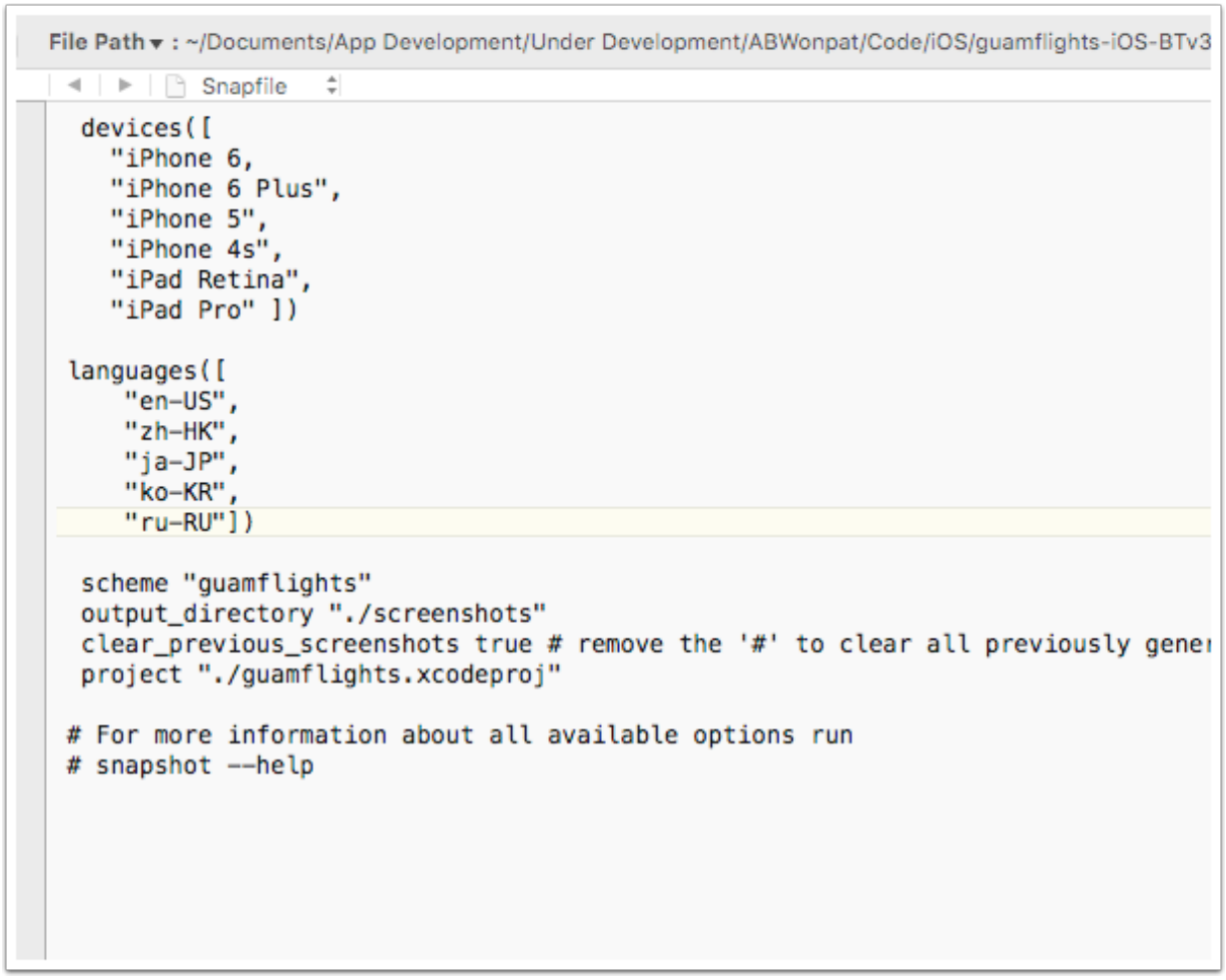
Setting up your 'Snapshot' environment for Buzztouch projects



Your Snap File

Your snapfile is the basic configuration that you're going to use for your project screen shots. Device types and Languages go here. If it looks a little like json, good. Remember that the last item in the list doesn't trail with a comma.

Your snapfile is created when you typed 'snapshot init' (or fastlane init) in your project. It should be located in your project root, or in your fastlane directory. edit it to your needs.

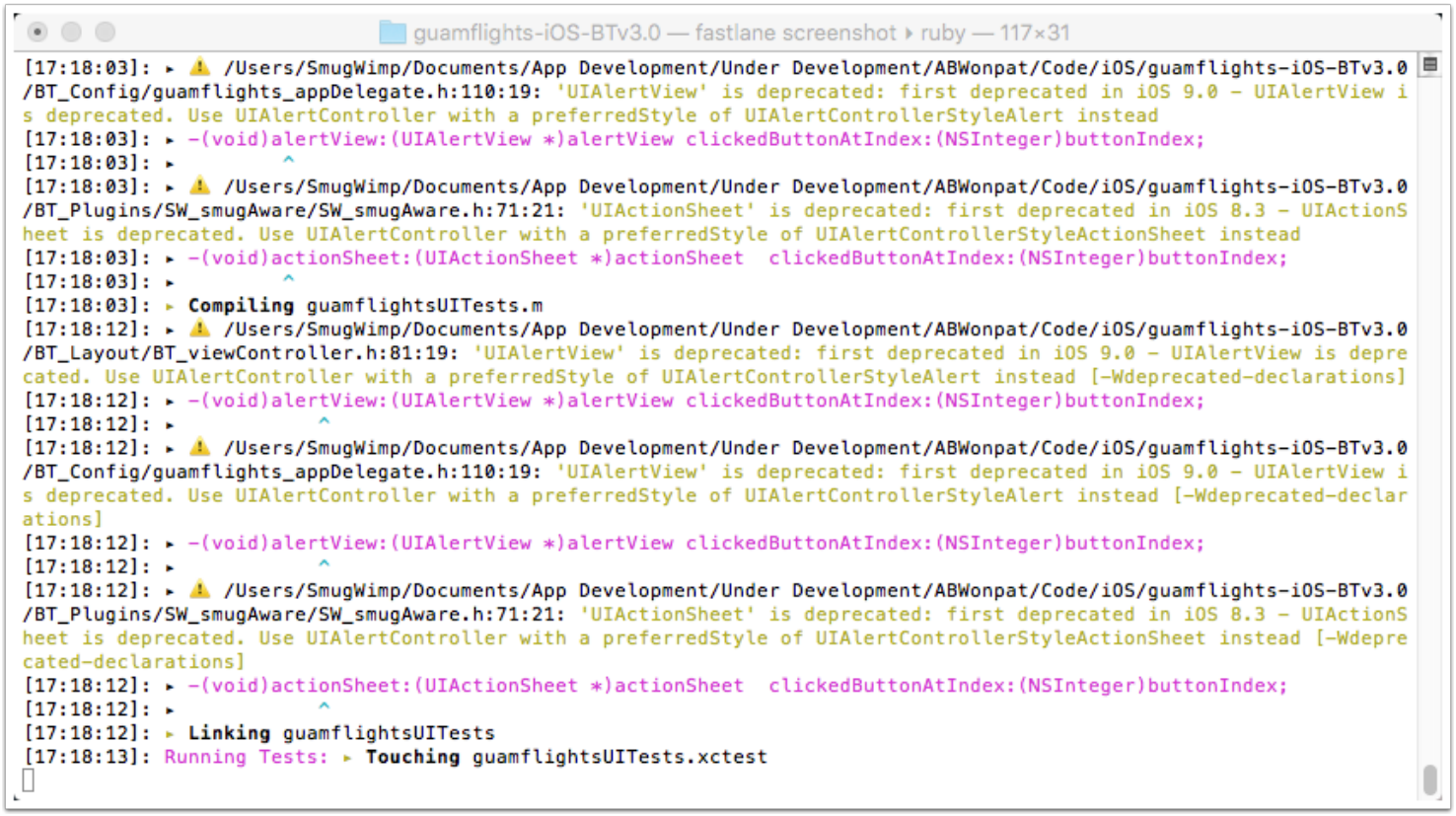


Now try it.

Now that we think we're ready to go, open up 'terminal' and from within your project directory, type 'snapshot' (or 'fastlane snapshot', or however you configured your install to react. my fastlane 'lane' [or scriptchannel] is called 'screenshot', so my command reflects this). If you have no errors, it will start to launch each device, for each screen, for each language and capture the screenshot.

Ignore the warnings... for now.

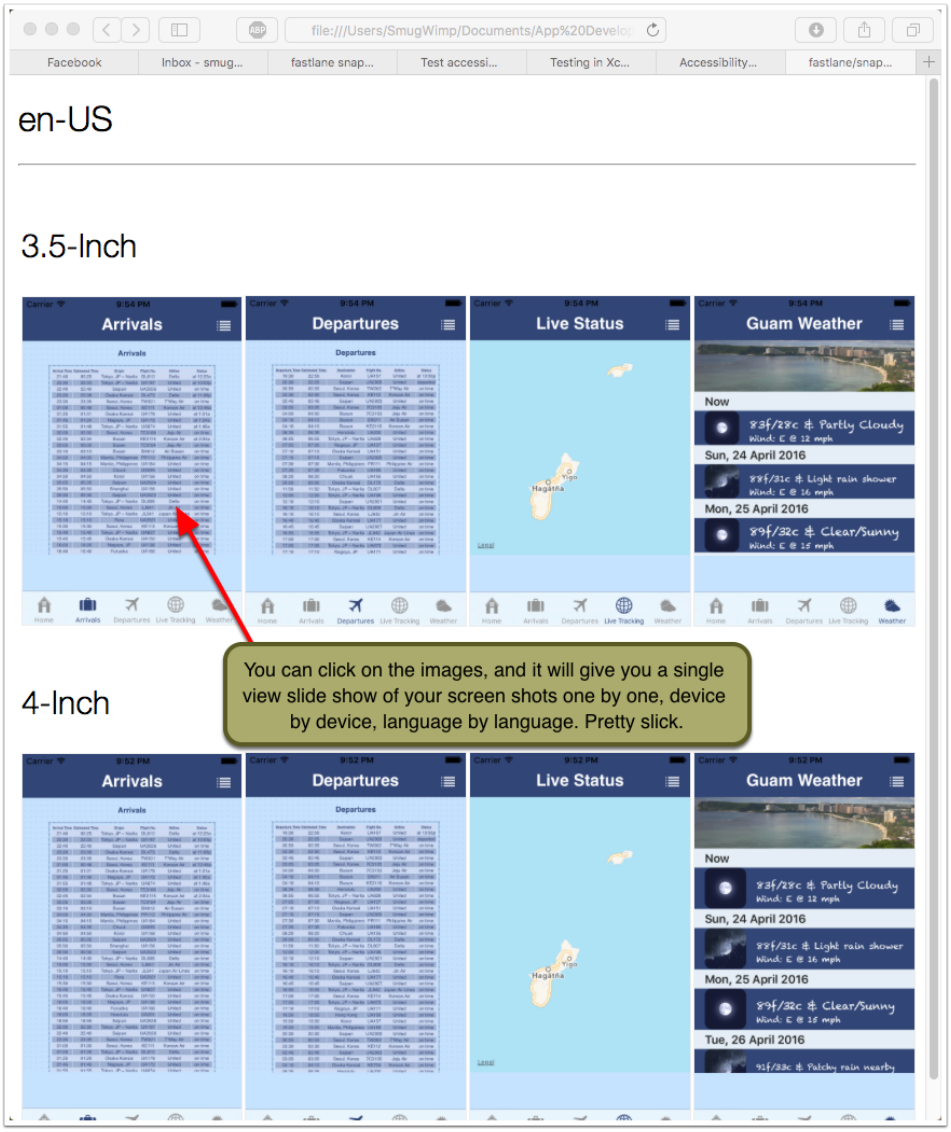
Face it. You're going to have warnings. But they're just that. 'Warnings'. 'Hey, this could be a problem eventually'... but for now, just get it to work. Go back and address the warnings later. Errors are in red. You must clear errors. Warnings? Not so much. Important, but not Critical. These are the same warnings you get in xcode when you're doing all of this by hand, so the warnings here, are the same warnings from there. Nothing new to see here.



Sweet!

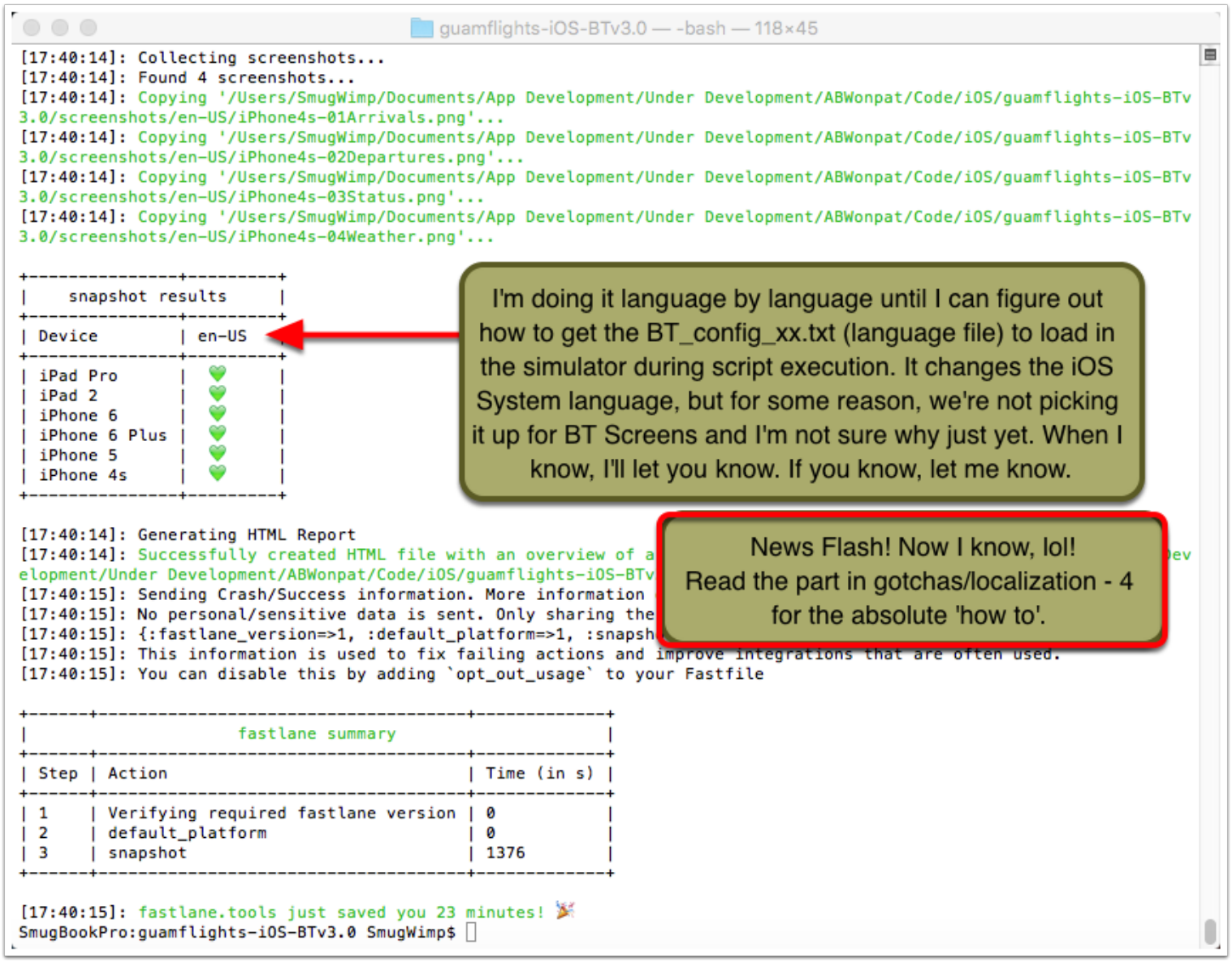
When it's complete, it creates and displays an HTML page with all of the languages, and all of the device sizes that it created screen shots for.

Is this freakin' cool, or what!?!?!?



Sweet! - 2

This is what your terminal results could look like, with a well configured setup. Look into the 'gotchas' for how I figured out the way to get mine all green.



Ok, that's how it's supposed to work. Does it really do all that?

As with everything iOS (and even more so on Android), some assembly is required. This means it's up to you to take care of some details to enable 'your particular configuration' to work with the package. Sometimes simple, sometimes not so simple.

I've already run into a few things that have me scratching my head. And as I work through them, I'll try and document it so we all don't have to freak out. In the meantime, here are a few things that immediately stood out:

Some gotchas -- Screenshot Timing; too fast?

A few of my screens load from the Web. As such, the screen shot came and went before the screen finished loading. Not good. So what I did was, I went into the 'SnapShotHelper.m' file and adjusted the wait time. I set it to 10 seconds rather than 1 second. True, it's a while, but it allowed all of the screens to load. And since it's automatic, I figured slow and steady always wins the race. You may need this, you may not. Remember, it's going to have to be redone each project. All of this stuff will.

Some gotchas -- Unknown Type? It's known. It just needs to be documented.

I wish I had kept better documentation. And when I do my next project, I'll keep this in mind. In my initial project setup, I had troubles with snapshot giving errors about a "unknown" type... This was things like, UIView, BT_item, etc...

All I can say is, add the 'header' file ('#import BT_item.h', etc...) to the file that is giving the error. I remember it happening in the BT_viewController, and a couple of other files. The errors were all similar, but not the same. Adding 'import' to the files fixed the error. It works in BT, because of the subclass hiarchy. But not with Snapshot, I guess. Good luck.

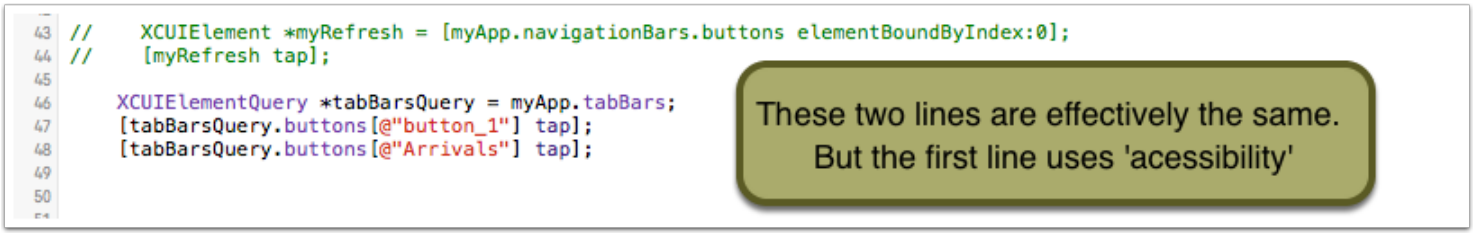
Some gotchas -- Localization (Languages) - 1

This stumped me for a few days. I thought I'd never figure it out. And I'm sure it 'must' be documented somewhere, because all the 'pros' kept talking about using accessibility labels for elements that were in different languages, but they never spelled out 'how' to use them. Here's how.

1) give everything you're going to 'click on' an Accessibility profile. This means

- telling xcode that the object will be an accesibility object
- giving the object an accessibility identifier (if applicable)
- giving the object an accessibility label
- giving the object an accessibility value

Do this for everything you're going to need to see, feel or touch in every language to get the screens you want to appear. Menus, tables, buttons, everything. And it seems to be hierchial (I could be wrong, but) so if it's the right navbar button, you have to setup the button, and the navbar. because it accesses the navbar first. So it can be tedious. But if it were easy, everyone would be doing it ;)



Some gotchas -- Localization (Languages) - 2

Acessibility allows you to set a 'common identifier' to distinguish different elements. So, how do you implement it?

1) First you set it up (when I say 'set it up' I mean add Accessibility properties to your code)

2) Then you record your actions. AFTER you've set everything up that needs to be accessibility friendly.

3) By all means not complete, however I did record a few of the 'kinds' of changes and where they should be made in this quick text file:

https://www.marianasgps.com/public/snapshotNotesForBT.txt

Some gotchas -- Localization (Languages) - 3

Edit your 'snapfile' to include:

- erase_simulator true
- reinstall_app true
- clean true
- (by omission, 'clear_previous_screenshots' is false, which is what we want. If 'clear_previous_screenshots' is present, either remove it, or change the value to 'false'.)

What we're doing here is setting the app to use any language file as default; that will be setup in code. We also erase any simulator, so it starts off without any previous copies of the app, and install a fresh copy on the simulator. This forces the 'first' BT_config to be the language config. We must do this once per language, to ensure that we get the correct screen display for the screen shot. It's a little slower, but still better than doing it manually.

Once 'all' of the languages are complete, the 'aggregate' will be available for upload to the itunes store, just as if it had occured in one complete run.

```
scheme "guamflights"
output_directory "./screenshots"
project "./guamflights.xcodeproj"
erase_simulator true
reinstall_app true
clean true
```

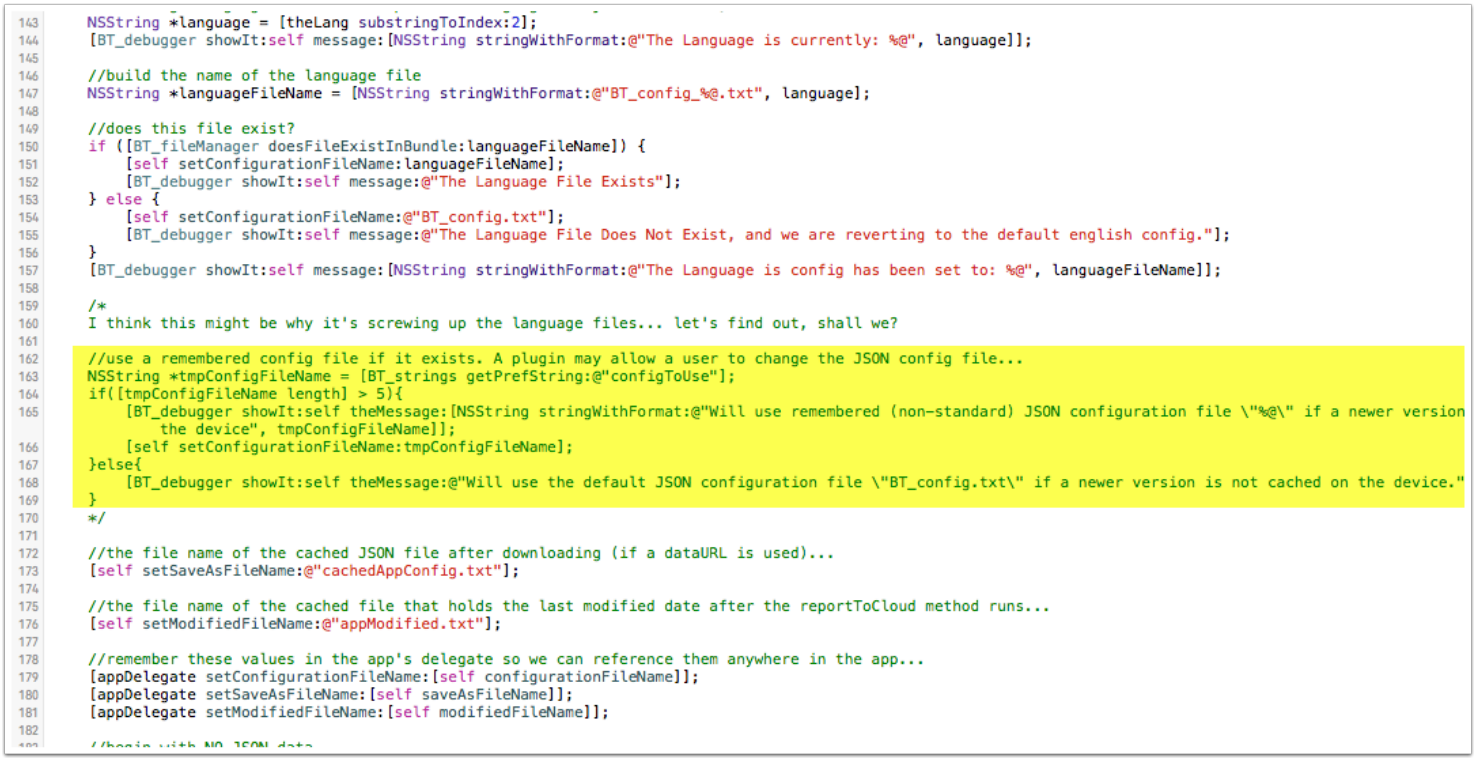
Some gotchas -- Localization (Languages) - 4

Finally, REMOVE (or comment out) This section of your BT_loadConfigDataViewController.m file

THIS is what has been keeping my tests from being able to dynamically switch languages. If you don't use a 'configToUse' property, then get rid of this. If you do use the 'configToUse' property, then figure it out yourself. I don't need it, so my brain stops here. ;)

Now we can use a list of languages in our Snapfile, across all devices. Now *this* is efficiency!

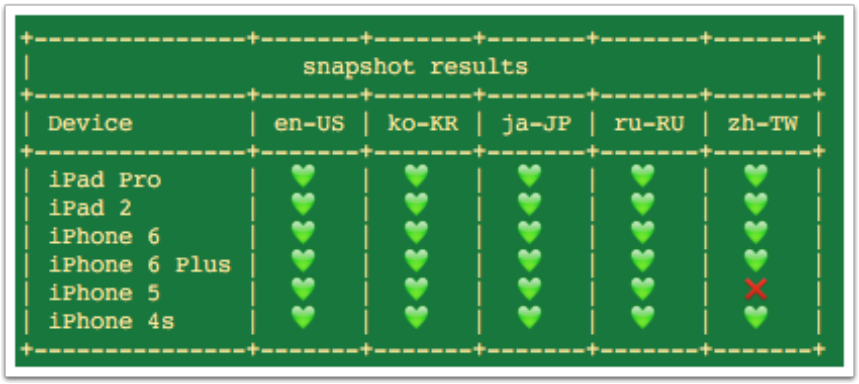
Granted, it was a little work on the outside, but now, we know what we need to know to do what we need to do efficiently.



```
    NSString *language = [theLang substringToIndex:2];
    [BT_debugger showIt:self message:[NSString stringWithFormat:@"The Language is currently: %@", language]];

    //build the name of the language file
    NSString *languageFileName = [NSString stringWithFormat:@"BT_config_%@.txt", language];

    //does this file exist?
    if ([BT_fileManager doesFileExistInBundle:languageFileName]) {
        [self setConfigurationFileName:languageFileName];
        [BT_debugger showIt:self message:@"The Language File Exists"];
    } else {
        [self setConfigurationFileName:@"BT_config.txt"];
        [BT_debugger showIt:self message:@"The Language File Does Not Exist, and we are reverting to the default english config."];
    }
    [BT_debugger showIt:self message:[NSString stringWithFormat:@"The Language is config has been set to: %@", languageFileName]];

    /*
    I think this might be why it's screwing up the language files... let's find out, shall we?

    //use a remembered config file if it exists. A plugin may allow a user to change the JSON config file...
    NSString *tmpConfigFileName = [BT_strings getPrefString:@"configToUse"];
    if([tmpConfigFileName length] > 5){
        [BT_debugger showIt:self theMessage:[NSString stringWithFormat:@"Will use remembered (non-standard) JSON configuration file \"%@\" if a newer version
            the device", tmpConfigFileName]];
        [self setConfigurationFileName:tmpConfigFileName];
    }else{
        [BT_debugger showIt:self theMessage:@"Will use the default JSON configuration file \"BT_config.txt\" if a newer version is not cached on the device."
    }
    */

    //the file name of the cached JSON file after downloading (if a dataURL is used)...
    [self setSaveAsFileName:@"cachedAppConfig.txt"];

    //the file name of the cached file that holds the last modified date after the reportToCloud method runs...
    [self setModifiedFileName:@"appModified.txt"];

    //remember these values in the app's delegate so we can reference them anywhere in the app...
    [appDelegate setConfigurationFileName:[self configurationFileName]];
    [appDelegate setSaveAsFileName:[self saveAsFileName]];
    [appDelegate setModifiedFileName:[self modifiedFileName]];
```

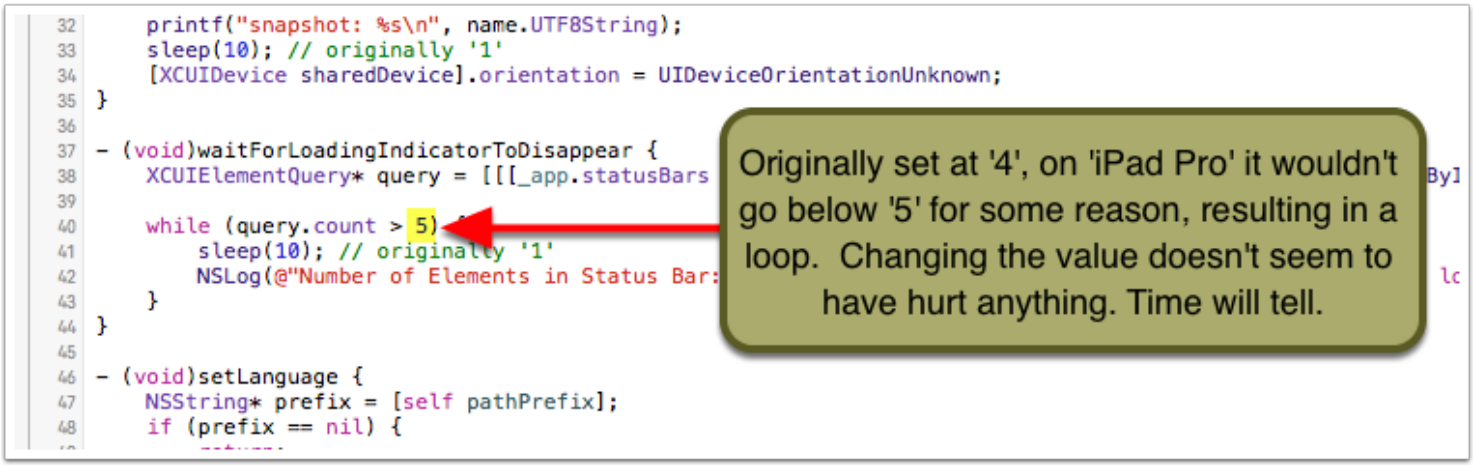
Some gotchas -- Localization (Languages) - 5

Granted, sometimes it's going to drop the ball... it's a fairly complex script. But it's a simple task now to just edit out what I don't need in my Snapfile, and rerun the script to get the botched screens. Voila!



Some gotchas -- My script seemed to stop/is hung...?

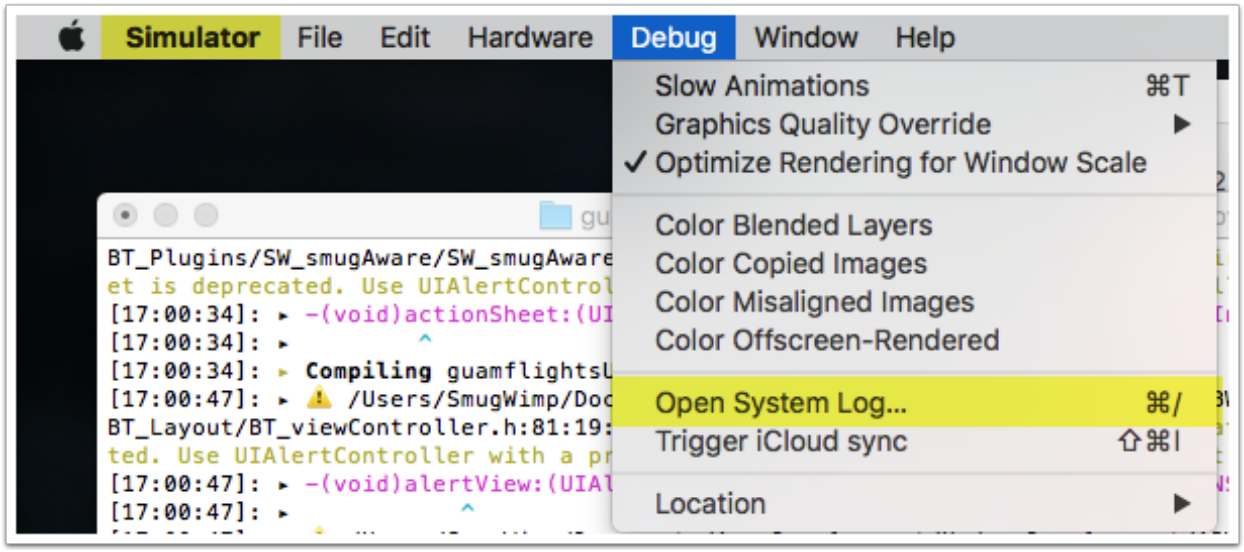
If it's hung up, chances are there is an answer in the logs. What logs? I'm glad you asked.



Some gotchas -- Where are the logs? - 1

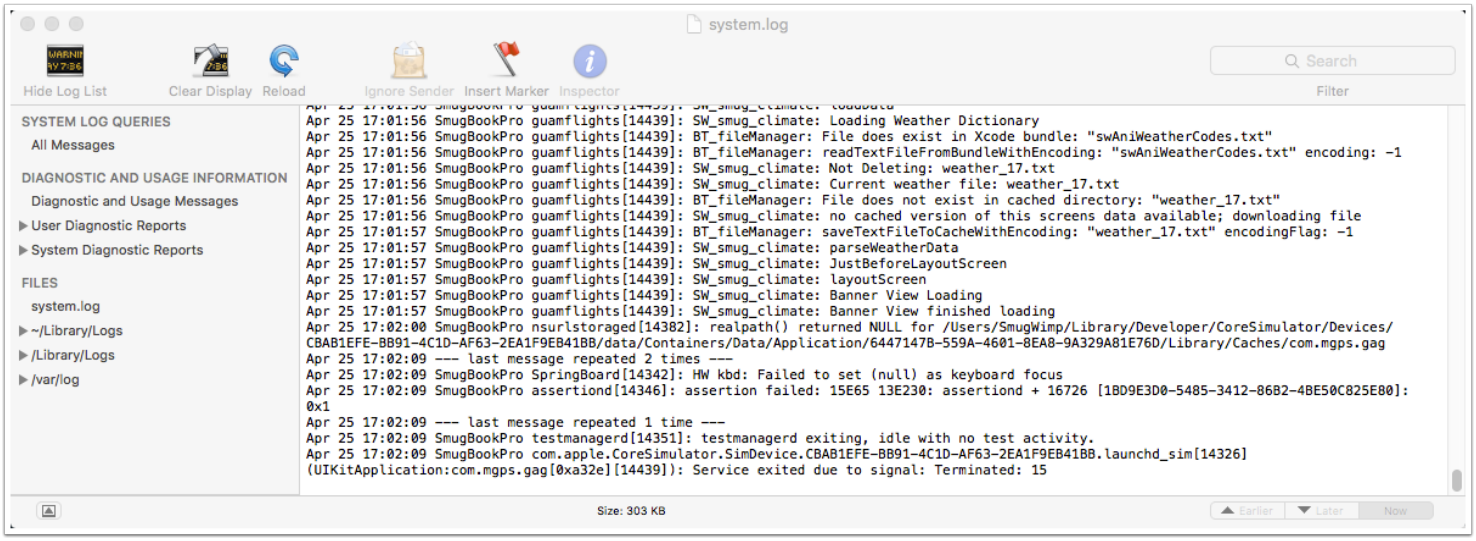
Normally you would be looking in your xcode interface for the logs, debugger, whatever you want to call it.

Since we're running a script that launches xcode as a command line thing, we have to look elsewhere for our debugger output. Fortunately, we just bring the simulator to be the front active app. In the Simulator's 'Debug' menu, select 'Open System Log'.



Some gotchas -- Where are the logs? - 2

And in the system log window are the same familiar log entries you know and love. BT_debugger, NSLog, etc... they all show up here, just as if you were using xcode. Well, you are, but... anyway. That's where the logs are. Look there if you have problems. That's how I discovered that it was stuck on 5 elements in a 4 element loop. Logs, dude. Logs.



Some gotchas -- Oops! I need to stop the script! What do I do!?!?

It's running on Python. Just hit 'control + c' (both keys at the same time) and your script should stop.

Some gotchas -- Editing your Snapfile 1

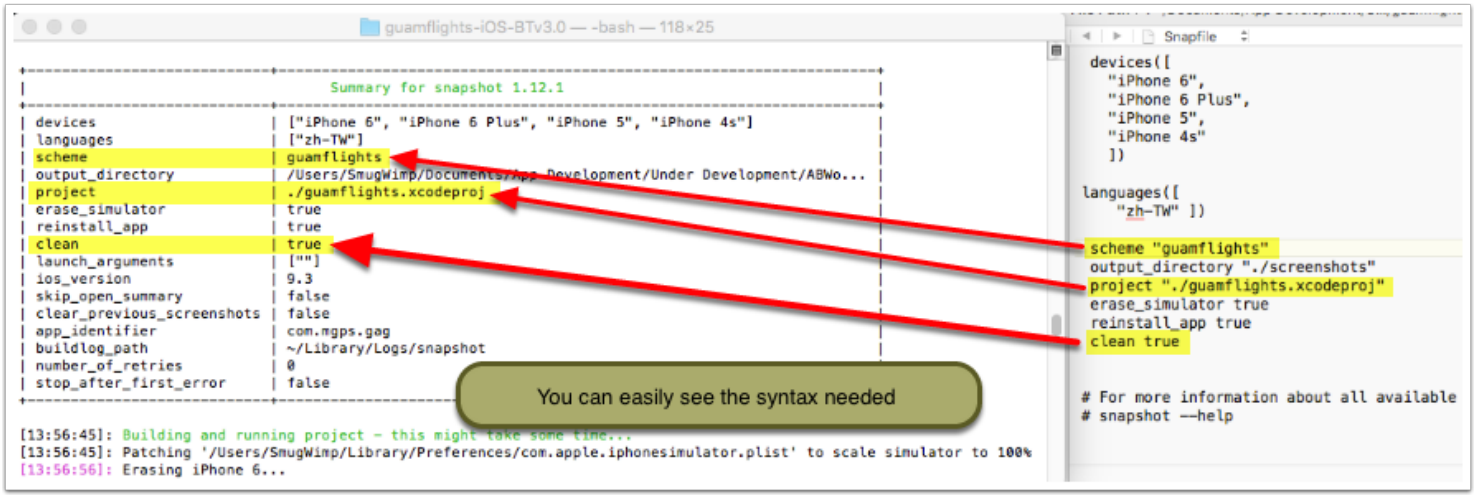
It's pretty easy, but it's not exactly laid out for understanding. Basically you use the same commands that you see 'on screen'.

If you have 'nothing' in your snapfile, the answer is assumed to be 'false'. So a lot of things are false, merely because you didn't specify them. That also means that unless you specify them as 'true', they will be 'false'. For most things, this is cool. But if you want to change something, it doesn't say exactly how to do it. It's simple; just add the command, and the value.

example: skip_open_summary true

Some gotchas -- Editing your Snapfile 2

Not all of the changes were highlighted, to keep some space between arrows, lol! but you can see the old snapfile in other screen shots, and the changes that are made here. It's just more of the same BS, with different names. In the array sections (devices, languages) keep it 'json' looking, with a comma between values, except for the last value. If the value is a string, enclose it in quotes.



Some gotchas -- Editing your Snapfile 3

So, I ran the script loaded. 6 devices, 5 languages. And it went through each and every one of them, but it dropped the ball in Chinese on the iPhone 5 screenshots. So I have to redo them. I was very pleased to see we can throw in pound signs (#) to comment out a part, rather than cut/paste things around. So now the simulation is only running on Chinese, iPhone 5 to make the rest complete. Yaay.

Some gotchas -- Sometimes you just can't sudo.

Regardless of getting successful screenshots, and html generation, the script always ended in an error. I'm not sure why. Well, I'm sure 'now', but at the time it was puzzling. In Short: Take control! Take ownership!

- Using the 'finder', navigate to the parent folder of your BT Project.
- Right click; select 'get info' (or command-I, or whatever)
- Authenticate, if you must, to change sharing permissions
- Select your own account/username
- Ensure it allows for 'Read & Write'. if not, change it.
- Using the 'gear' icon menu, select "Apply to enclosed items" "yes" to everything.
- close it up.

The next time you run the script under your own account (which is where you should be) it won't error on you when it creates the finishing files. What we just did was ensure that the project directory and all subdirectories (remember, we had to sudo to install fastlane/snapshot, so it might not have been using our credentials, right?) have 'our' account listed with read/write privileges, which means any script under our account has read/write priviledges.

```
[17:02:09]: Variable Dump:
[17:02:09]: {:DEFAULT_PLATFORM=>:ios, :PLATFORM_NAME=>:ios, :LANE_NAME=>"ios screenshot"}
[17:02:09]: Tests failed - check out the log above
-----------------------------------------------------------------------------------------
😰 An error occured. Please enable crash reports using `fastlane enable_crash_reporting`.
👍 This makes resolving issues much easier and helps improve fastlane.
🔒 The reports will be stored securely on getsentry.com.
👴 More information about privacy: https://github.com/fastlane/fastlane/releases/tag/1.33.3
-----------------------------------------------------------------------------------------
/Library/Ruby/Gems/2.0.0/gems/fastlane-1.81.0/lib/fastlane/documentation/docs_generator.rb:39:in `write': [!] Permissi
on denied - ./fastlane/README.md (Errno::EACCES)
        from /Library/Ruby/Gems/2.0.0/gems/fastlane-1.81.0/lib/fastlane/documentation/docs_generator.rb:39:in `run'
        from /Library/Ruby/Gems/2.0.0/gems/fastlane-1.81.0/lib/fastlane/lane_manager.rb:55:in `cruise_lane'
        from /Library/Ruby/Gems/2.0.0/gems/fastlane-1.81.0/lib/fastlane/command_line_handler.rb:30:in `handle'
        from /Library/Ruby/Gems/2.0.0/gems/fastlane-1.81.0/bin/fastlane:38:in `block (2 levels) in run'
        from /Library/Ruby/Gems/2.0.0/gems/commander-4.3.5/lib/commander/command.rb:178:in `call'
        from /Library/Ruby/Gems/2.0.0/gems/commander-4.3.5/lib/commander/command.rb:178:in `call'
        from /Library/Ruby/Gems/2.0.0/gems/commander-4.3.5/lib/commander/command.rb:153:in `run'
        from /Library/Ruby/Gems/2.0.0/gems/commander-4.3.5/lib/commander/runner.rb:428:in `run_active_command'
        from /Library/Ruby/Gems/2.0.0/gems/fastlane_core-0.41.3/lib/fastlane_core/ui/fastlane_runner.rb:23:in `run!'
        from /Library/Ruby/Gems/2.0.0/gems/commander-4.3.5/lib/commander/delegates.rb:15:in `run!'
        from /Library/Ruby/Gems/2.0.0/gems/fastlane-1.81.0/bin/fastlane:168:in `run'
        from /Library/Ruby/Gems/2.0.0/gems/fastlane-1.81.0/bin/fastlane:174:in `<top (required)>'
        from /usr/bin/fastlane:23:in `load'
        from /usr/bin/fastlane:23:in `<main>'
```