Titanium Mobile: Module Development

iOS Module Developer's Guide

appcelerator

Modules

August 18, 2010

Appcelerator, Inc. 444 Castro Street, Suite 818, Mountain View, California 94041

Contents

Introduction

Titanium provides the ability to extend the built-in functionality through a series of optional extensions, called Titanium+Plus Modules. We'll reference them as Modules in this guide as a short abbreviation.

This document is part of the Titanium Module Developer's Guide series, covering the Steps 0 through 3 of iOS Module development.

- Step 0 covers your existing computer environment, specifying how to set up your environment for development of iOS based Modules.
  See Step 0: Setting Up your Module Environment on page 4.
- In Step 1, you'll use Objective-C to create your first module.
  See Step 1: Creating your First Module on page 5.
- In Step 2, you'll learn about the basic architecture of modules.
  See Step 2: Basic Module Architecture on page 12.
- In Step 3, you'll learn how to use your module and distribute it to others.
  See Step 3: Packaging your Module for Distribution on page 32.

iOS Module Prerequisites

To develop an iOS-based Module, you'll need all of the following basic prerequisites:

1. Intel-based Macintosh running OSX 10.5 or above
2. XCode 3.2 or above
3. iOS 4.0 SDK or above
4. Titanium 1.4 Mobile SDK or above

iOS Modules support both iPhone and iPad based extensions.

In addition to the following environment prerequisites, you must have a general knowledge of Objective-C development.

Step 0: Setting Up your Module Environment

To set up your Module environment on OSX, you'll need to place a helper script on your path or set up an environment alias to the file. In this document, we recommend you create an alias.

Note: You'll need to determine where your Titanium 1.4.0 SDK (or more recent version) is installed. It may be installed in /Library/Application\ Support/Titanium, or it may be in ~/Library/Application\ Support/Titanium. The steps below assume that it is below /Library. If it is below ~/Library instead, you'll need to make the appropriate change to the line in step 3.

1. Open the Console application. The Console application is located under /Applications/Utilities/Console.
2. Edit the file named .bash_profile in your $HOME directory.
3. In this file, add the following line at the end of the file:

```
alias titanium='/Library/Application\ Support/Titanium/mobilesdk/osx/1.4.0/titanium.py'
```

This will create an alias to the titanium.py helper script. You should substitute the version above (1.4.0) with the version of Titanium mobile SDK you're using.

4. Save the file.

5. Type the following command in the console:

```
> source ~/.bash_profile
```

6. Type the following command in the console:

```
> titanium
```

You should see the following output (or substantially similar):

```
Appcelerator Titanium
Copyright (c) 2010 by Appcelerator, Inc.

commands:
   create      - create a project
   run         - run an existing project
   help        - get help
```

7. Keep open the console as you'll need it in subsequent steps.

Step 1: Creating your First Module

Before building your first Module, you'll want to create at test Module to ensure that your environment is set up correctly and that you can use the Module in a test application.

To create a module, you'll need to run the following command in your console:

```
> titanium create --platform=iphone --type=module --dir=~/tmp --name=test --id=com.test
```

After running this command, you should see the following:

```
Appcelerator Titanium XCode templates installed
Created iphone module project
```

This will create a new module in the directory ~/tmp/test. Let's examine the contents of the newly created module.

```
> cd ~/tmp/test
> ls -la
```

You should see the following contents (or substantially similar):

```
total 80
drwxr-xr-x   16 jhaynie  staff   544 Jul 30 15:08 .
drwxr-xr-x@ 254 jhaynie  staff  8636 Jul 30 15:08 ..
-rw-r--r--    1 jhaynie  staff    20 Jul 30 15:08 .gitignore
drwxr-xr-x    7 jhaynie  staff   238 Jul 30 15:08 Classes
-rw-r--r--    1 jhaynie  staff    62 Jul 30 15:08 ComTest_Prefix.pch
-rw-r--r--    1 jhaynie  staff    78 Jul 30 15:08 LICENSE
-rw-r--r--    1 jhaynie  staff  4877 Jul 30 15:08 README
drwxr-xr-x    3 jhaynie  staff   102 Jul 30 15:08 assets
-rwxr-xr-x    1 jhaynie  staff  5490 Jul 30 15:08 build.py
drwxr-xr-x    3 jhaynie  staff   102 Jul 30 15:08 documentation
drwxr-xr-x    3 jhaynie  staff   102 Jul 30 15:08 example
```

```
drwxr-xr-x    7 jhaynie  staff   238 Jul 30 15:08 hooks
-rw-r--r--    1 jhaynie  staff   373 Jul 30 15:08 manifest
-rw-r--r--    1 jhaynie  staff   777 Jul 30 15:08 module.xcconfig
drwxr-xr-x    3 jhaynie  staff   102 Jul 30 15:08 test.xcodeproj
-rw-r--r--    1 jhaynie  staff   478 Jul 30 15:08 titanium.xcconfig
```

Let's explain what the files and directories are for:

Filename / Directory	Description / Purpose
.gitignore	A special .gitignore file used by the revision control system git to instruct it to ignore certain temporary files and directories. This file is not distributed with your module. This file can be safely ignore if you are not using git.
Classes	The directory where you Objective-C headers and implementation classes should go. These files are not distributed with your module and are used by the Xcode compiler.
ComTest_Prefix.pch	The pre-compiled header file. This file is not distributed with your module and is used by the Xcode compiler.
LICENSE	The file for describing the contents of your module's license text. This file is distributed with your module to end-users.
README	The file that gives a short explanation of the module project. This file is not distributed with your module.
assets	The directory where you should place module assets such as images.
build.py	This file is a script that will compile and package your module for usage and distribution.
documentation	The directory where you should place your module documentation for end-users. The file index.md is a Markdown-formatted template file that you should use when writing your module documentation. This is only required for module distributed in the Titanium+Plus Marketplace. You can safely ignore this directory if you do not intend to distribute your module.
example	The directory where your module example(s) should go. The file app.js will be generated to include a sample loading of your module in a test window. This file can be used for quickly testing your module as well as give an example to end-users on how to use your module. This directory is distributed with your module.
hooks	The directory is currently ignored and reserved for future use.
manifest	A special file that describes meta-data about your module and used by the Titanium compiler. This file is required and is distributed with your module.
module.xcconfig	A special file used by Xcode when your module is compiled in an end-user Titanium application which references your module. This file is a Xcode configuration file which can setup special compiler and linker directives that are specific to your module. This file is distributed with your module.
test.xcodeproj	The directory which contains your Xcode project. This directory is not distributed with your module.
titanium.xcconfig	A special file used by Xcode when your module is compiled when building your module. This file is a Xcode configuration file which can setup special compiler and linker directives that are specific to your module. This file is not distributed with your module and only used during module development.

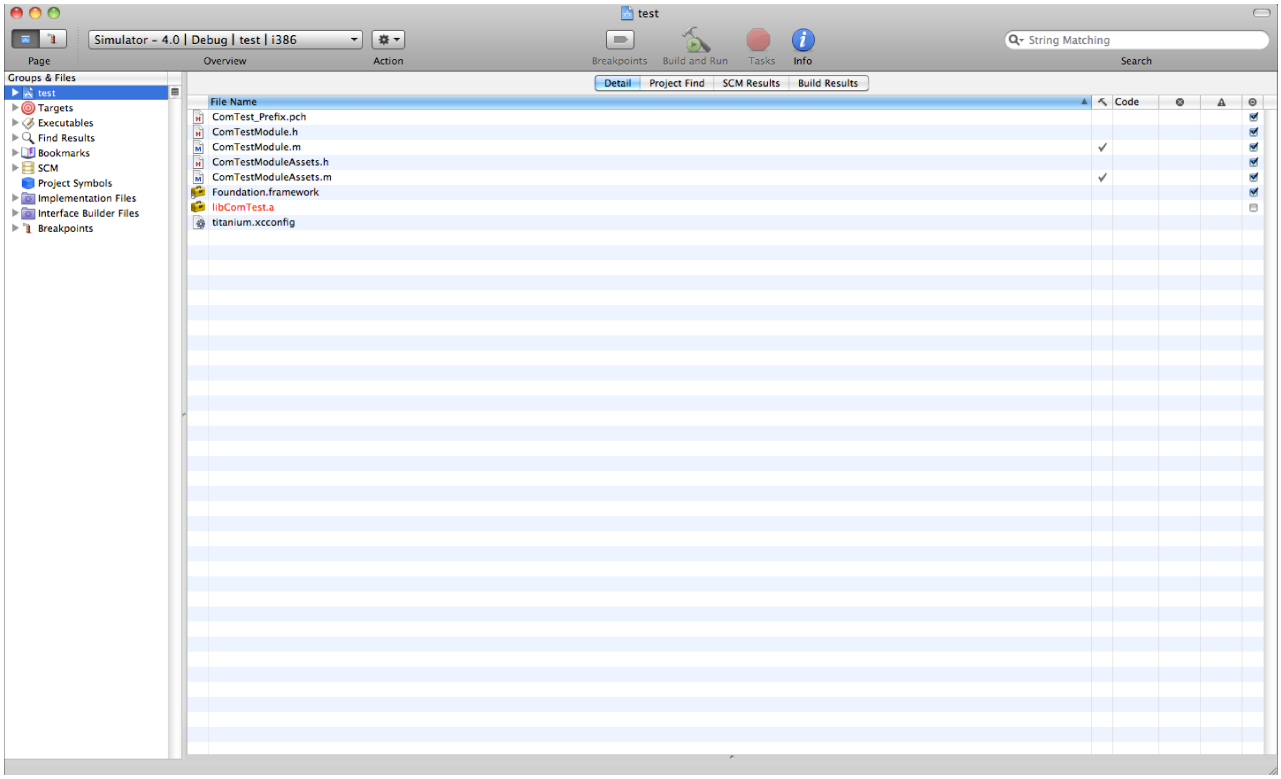
Open the Test Module in Xcode

In this step, we're going to ensure that you can open your newly created Test module in Xcode.

Open the test.xcodeproj in Xcode. From the console, you can simply type the following to open the project:

```
> open test.xcodeproj
```

This should start Xcode (if not already running) and open your test project. You should see the following window (if running XCode 3. XCode 4 has a new UI window appearance).



The main Module files are defined in ComTestModule.h (header) and ComTestModule.m (implementation).

You should ignore the files ComTestModuleAssets.h (header) and ComTestModuleAssets.m (implementation) files as these are system generated when your module is packaged. Any changes to these files will be overwritten each time your module is built.

At this point, you should be able to build your module in Xcode using the Build command under the Build menu.

If your build succeeded, you're ready to package your module and test it in a real Titanium application.

Packaging your Module

To package your module, you'll need to run the build.py script from the console.

```
> ./build.py
```

This will cause the console to emit a bunch of compiler statements to standard output and may take a few seconds to build and package. Once the build completes successfully, you should see the following at the end of output:

```
> ** BUILD SUCCEEDED **
```

Note: As mentioned in the note on page 4, your Titanium SDK may have been installed under ~/Library instead of /Library. Your titanium.xcconfig file must refer to the correct directory, else the build will fail.

Note: The markdown2 module is not on the PythonPath by default for the Mac's pre-installed Python interpreter. Without this installed, build.py will fail when it tries to generate the module docs. If you have not already done so, then follow these steps to install markdown2:
1. Download the latest release from http://code.google.com/p/python-markdown2/
2. Unzip the resulting download
3. Using the setup.py script in the directory that was created from expanding the zip file you downloaded, install with `sudo python setup.py install`
4. Re-run build.py

You should also find the file named com.test-iphone-0.1.zip in the current directory (~/tmp/test). If you do not find this file, your module did not properly build and package.

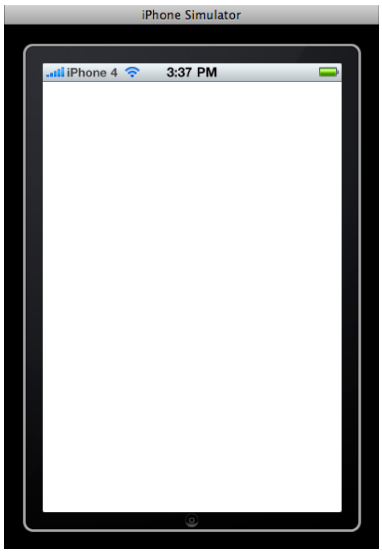
Testing your Module

To test your module, we'll first want to simply run the Module test harness. The Module test harness will create a temporary project, copy in your example app.js file (pre-generated for you) and load your module. You can perform this test by running the following in console from the current directory (~/tmp/test):

```
> titanium run
```

This will cause the console to emit a bunch of logging to standard output and make take several seconds to build, package and launch the iPhone simulator.

If successful, you should see the iPhone simulator and a white screen:



You should see something like the following near the end of your Console output:

```
[INFO] [object ComTestModule] loaded
[DEBUG] loading: /var/folders/4n/4nhl7xlPFsaSrlAGHhPwg++++TI/-Tmp-/mNdElWkti/test/
 Resources/com.test.js, resource: var/folders/4n/4nhl7xlPFsaSrlAGHhPwg++++TI/-Tmp-/
 mNdElWkti/test/Resources/com_test_js
[INFO] module is => [object ComTestModule]
[DEBUG] application booted in 82.601011 ms
```

The important two statements to look for are:

```
[INFO] [object ComTestModule] loaded
[INFO] module is => [object ComTestModule]
```

The first statement can be found if you look at the method named startup in the file ComTestModule.m.

```
-(void)startup
{
    // this method is called when the module is first loaded
    // you *must* call the superclass
    [super startup];

    NSLog(@"[INFO] %@ loaded",self);
}
```

We'll explain this method in more detail in a later section.

The second statement can be found if you look at the file app.js under the example directory.

```
var test = require('com.test');
Ti.API.info("module is => "+test);
```

When loading a module from JavaScript, you'll first need to import it by requiring it using the require('module_id') command like above.

Testing your Module in a test application

Now, we need to test your Module in a test application.

Before we can test the module in a real application, we have to install it. Since we already packaged the module in the previous step, we can skip building your module. However, remember that as you change your module code, you will need to repeat the build and install steps.

To install your Module so that it can be used in a real application, you will need to install it. In the console, run the following:

```
> cp com.test-iphone-0.1.zip /Library/Application\ Support/Titanium/
```

This will copy your Module distribution file into your Titanium directory. The Titanium compiler is smart enough to automatically expand your module into the correct directory when referenced in your application.

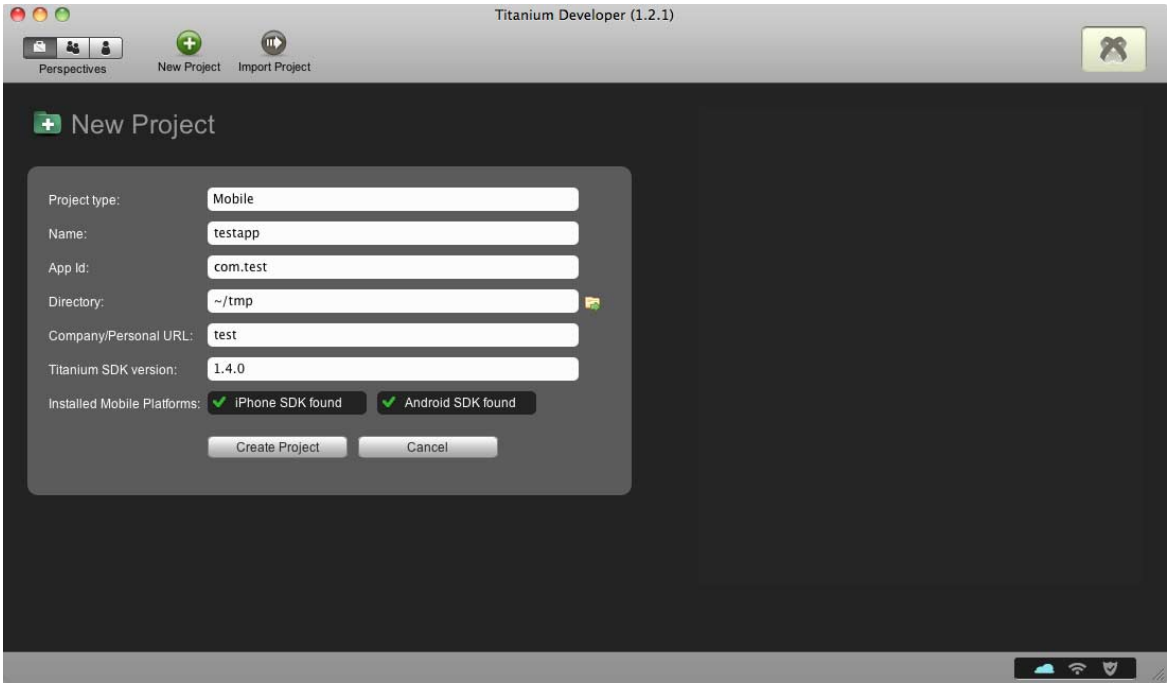
Your module will be expanded into the following directory:

```
/Library/Application Support/Titanium/modules/iphone/com.test/0.1
```

This directory allows multiple versions of the same module to co-exist at the same time.

Note: As mentioned in the note on page 4, you'll need to adjust the paths above if your Titanium SDK was installed below ~/Library instead of /Library.

Next, we need to create a test application in Titanium Developer.



Once your application project is created, navigate to your test application folder and edit the file named tiapp.xml in a text editor.

You will need to add the following to this file near the end of the file:

```
<modules>
    <module version="0.1">com.test</module>
</modules>
```

This is an important step as it will instruct the Titanium compiler that your module needs one or more modules defined and which version(s). In this test case, we simply need to reference our com.test module and the default version, 0.1.

The final XML file should look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<ti:app xmlns:ti="http://ti.appcelerator.org">
    <id>com.test</id>
     <name>testapp</name>
    <version>1.0</version>
    <publisher>not specified</publisher>
    <url>not specified</url>
    <description>not specified</description>
    <copyright>not specified</copyright>
    <icon>appicon.png</icon>
    <persistent-wifi>false</persistent-wifi>
    <prerendered-icon>false</prerendered-icon>
    <statusbar-style>default</statusbar-style>
    <statusbar-hidden>false</statusbar-hidden>
```

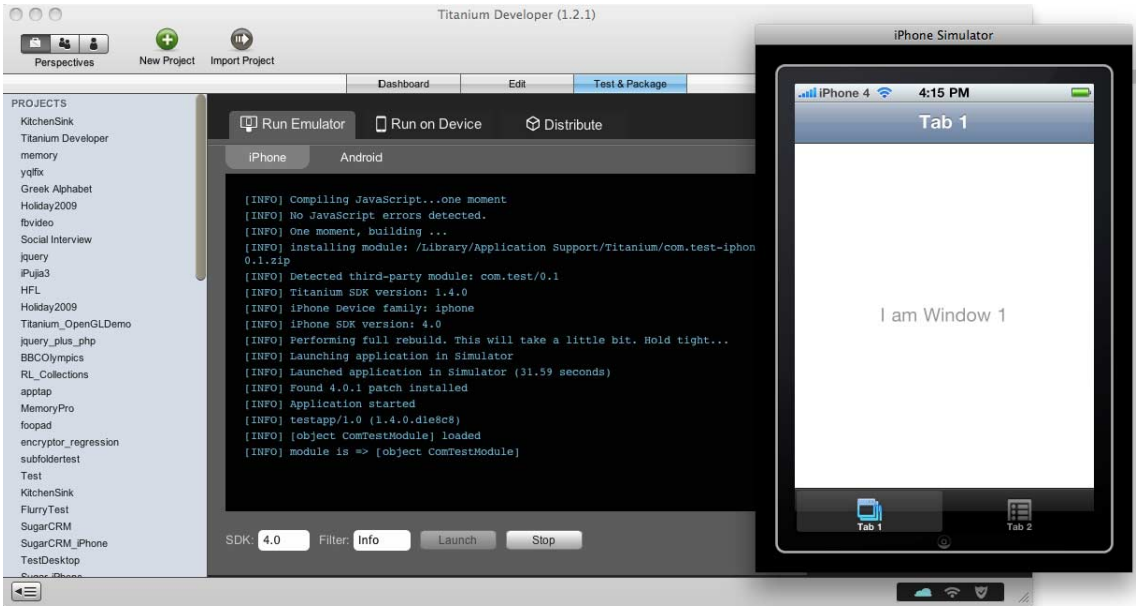
```
    <fullscreen>false</fullscreen>
    <navbar-hidden>false</navbar-hidden>
    <analytics>true</analytics>
    <guid>c347a9a1-44bd-48c8-a6bf-7762f2582f50</guid>
    <modules>
        <module version="0.1">com.test</module>
    </modules>
</ti:app>
```

Note: The guid value will be different that above since it is a globally-unique id and generated each time.

Now, we need to edit our app.js to load our module. Edit the app.js file in the Resources directory of our test application and add the following at the end of the file:

```
var test = require('com.test');
Ti.API.info("module is => "+test);
```

Launch the application in the simulator and you should see a normal test application:



The Titanium Developer console should output some important key logging statements that will help ensure that your module loaded:

```
Detected third-party module: com.test/0.1
```

This statement should be printed for each module referenced in the tiapp.xml file.

We should also see the following based on the code we added to app.js:

```
[INFO] [object ComTestModule] loaded
[INFO] module is => [object ComTestModule]
```

If you see this statements and no errors in the logging console, your Module loaded and works. Your system is set up and ready to go!

Step 2: Basic Module Architecture

The Titanium platform is based on a modular architecture and while you can create third-party Modules as described in this document, Titanium itself uses the same module SDK internally for built-in APIs.

The Module architecture contains the following key interface components:

Proxy	A base class that represents the native binding between your JavaScript code and native code.
ViewProxy	A specialized Proxy that knows how to render Views.
View	The visual representation of a UI component which Titanium can render.
Module	A special type of Proxy that describes a specific API set, or namespace.

When building a Module, you can only have one Module class but you can have zero or more Proxies, Views and ViewProxies.

For each View, you will need a ViewProxy. The ViewProxy represents the model data (which is kept inside the proxy itself in case the View needs to be released) and is responsible for exposing the APIs and events that the View supports.

You create a Proxy when you want to return non-visual data between JavaScript and native. The Proxy knows how to handle any method and property dispatching and event firing.

Proxy

To create a Proxy, your interface declaration must extend TiProxy. Import "TiProxy.h" in your import statement. For example, your interface would be:

```
#import "TiProxy.h"
@interface FooProxy : TiProxy {
}
@end
```

Your implementation would then be:

```
#import "FooProxy.h"
@implementation FooProxy
@end
```

By convention, Titanium requires the use of the suffix Proxy to indicate that the class is a Proxy.

Proxy Methods

Proxies expose methods and properties by simply using standard Objective-C syntax.

To expose a method, a Proxy must have one of the following valid signatures:

```
-(id)methodName:(id)args
{
}
```

This signature is used when the method returns a value to the caller. The returned result can be either a valid NSObject, nil or a Proxy (or subclass thereof).

```
-(void)methodName:(id)args
{
}
```

This signature should be used if your method returns no value.

Both signatures take a single argument (args) as a NSArray. However, generally, we recommend using the id declaration. This will make it easier to typecast the incoming value to another value, which we'll describe below.

If you function takes no arguments, you can simply ignore the incoming args value.

Titanium provides several convenience macros for typecasting incoming values to a specific type and extracting them from the array.

```
ENSURE_SINGLE_ITEM(args,type)
```

• The first parameter is the name of the argument.

• The second parameter is the type name that the value should be.

This macro will do two actions: (1) it will pull out the first argument (i.e. [args objectAtIndex:0]), and (2) it will cast the return value to the type passed in the second argument.

This macro can only be used for single-argument methods. If you have multiple arguments, you should simply used the normal array accessor methods.

```
ENSURE_UI_THREAD_0
```

This macro can be used to ensure that the current method only runs on the UI thread (main thread). If the method is invoked on a non-main thread, it will simply re-queue the method on the UI thread. This method is equivalent to [NSThread performSelectorOnMainThread].

You can only use this method if you have no return result.

```
ENSURE_UI_THREAD_1(arg)
```

This macro can be used to ensure that the current method, with argument, only runs on the UI Thread. It is the same as the previous macro with the exception that it will ensure that the arguments are passed along, too.

Titanium provides a Utility library for converting and checking certain values. This library is in TiUtils and can be imported with:

```
#import "TiUtils.h"
```

Some common conversion utility examples:

```
CGFloat f = [TiUtils floatValue:arg];
NSInteger f = [TiUtils intValue:arg];
NSString *value = [TiUtils stringValue:arg];
NSString *value = [TiUtils stringValue:@"key" properties:dict def:@"default"];
TiColor *bgcolor = [TiUtils colorValue:arg];
```

Proxy Properties

To expose a JavaScript property in a Proxy, you can simply define a Objective-C property:

```
@property(nonatomic,readwrite,assign) NSString* propertyName;
```

You can optionally just define a getter and/or setter property.

```
-(void)setPropertyName:(id)value
{
}

-(id)propertyName
{
   return @"foo";
}
```

In the setter method, Titanium will pass a single value as the converted value as the first argument (instead of an NSArray like a method).

In the getter method, your property method must return a value as either a NSObject, nil or TiProxy (or sub-class).

Returning Object Values

The following Objective-C types can be returned without conversion:

NSString	NSDictionary	NSArray
NSNumber	NSDate	NSNull

Returning Primitive Values

To return a primitive value from either a method or property, you should return an NSNumber with the appropriate wrapped primitive value. Titanium provides a set of macros to make this easier:

Macro	Description
NUMINT	Equivalent to [NSNumber numberWithInt:value]
NUMBOOL	Equivalent to [NSNumber numberWithInt:value]
NUMLONG	Equivalent to [NSNumber numberWithLong:value]
NUMLONGLONG	Equivalent to [NSNumber numberWithLongLong:value]
NUMDOUBLE	Equivalent to [NSNumber numberWithDouble:value]
NUMFLOAT	Equivalent to [NSNumber numberWithFloat:value]

Returning Complex Values

There are two approaches to returning complex values:

- The first approach is to set values in a NSDictionary and it. When returning a NSDictionary, Titanium converts this to a JavaScript object with each key/value being mapped into JavaScript object property/values.
- The second approach is to create a specialized Proxy. The Proxy should then be returned which will be exposed as a JavaScript object with functions and properties. Invocation against the returned Proxy will be invoked against your returned proxy instance. When you return a Proxy instance, you must autorelease it if you created it in your method.

Returning Files

To return a reference to a filesystem file, you should return an instance of a TiFile proxy. This will automatically handle exposing the native file and it's methods and properties. To return a file, you can use the following example:

```
    return [[[TiFile alloc] initWithPath:path] autorelease];
```

Returning Blobs

To return a reference to a blob data (such as a NSData), you should return an instance of a TiBlob. This will automatically handle exposing some native blob operations. To return a blob, you can use the following examples:

```
    return [[[TiBlob alloc] initWithData:nsdata mimetype:@"application/octet-stream"]
autorelease];
    return [[[TiBlob alloc] initWithImage:image] autorelease];
    return [[[TiBlob alloc] initWithFile:file] autorelease];
```

In the first example above, the second argument should map to the mime type of the raw data content. If the data is binary, you can use the "application/octet-stream" value.

Returning CGRect

To return a CGRect, Titanium provides a proxy named TiRect. You can use the following example:

```
TiRect *result = [[[TiRect alloc] init] autorelease];
[result setRect:rect];
return result;
```

Returning CGPoint

To return a CGPoint, Titanium provides a proxy named TiRect. You can use the following example:

```
    return [[[TiPoint alloc] initWithPoint:point] autorelease];
```

Returning NSRange

To return an NSRange, Titanium provides a proxy named TiRange. You can use the following example:

```
    return [[[TiRange alloc] initWithRange:range] autorelease];
```

Returning UIColor

To return a UIColor, Titanium provides a proxy named TiColor. You can use the following example:

```
    return [[[TiColor alloc] initWithColor:color name:@"#fff"] autorelease];
```

The second argument (name) should be the textual name (if provided) of the UIColor.

Setting Proxy Values

When you create a proxy from JavaScript, you typically pass an optional dictionary of key/value pairs. By using this pattern, Titanium provides built-in functionality to help make proxy programming easier.

Let's use a simple example:

```
    var my_module = require('com.test');
    var foo = my_module.createFoo({
      "bar":"123"
    });
```

Titanium will automatically correctly define and dispatch proxy factory creation methods like above. You simply need to define a Proxy with the same name as your module and the name of your method plus the Proxy suffix.

```
    ComTestFooProxy.h
    ComTestFooProxy.m
```

Your module does not need to define the createFoo method. Using this convention, Titanium already knows how to do that for you.

In your proxy, you would then want to define a setter method to handle the bar property. Upon construction (the init), Titanium will automatically call the setter for all properties passed in the constructor.

```
#import "ComTestFooProxy.h"
#import "TiUtils.h"

@implementation ComTestFooProxy

-(void)setBar:(id)value
```

```
{
   NSString *str = [TiUtils stringValue:value];
}
@end
```

In the above example, we simply define the setter and use the TiUtils to convert the value into a stringValue. Using this utility ensures that no matter what type of value passed into the argument, we'll get the string representation. So, if the user was to pass the number 123, it would still return an NSString with the value @"123".

Proxies are also special about how they handle and hold their properties. Proxies will always store values passed in a special internal NSDictionary called 'dynprops'. This means that you can always use the following method to retrieve the value of the proxy without having to define getters and setters for each of your properties:

```
id value = [self valueForUndefinedKey:@"bar"];
```

If you use valueForUndefinedKey, you will always retrieve the original property value. However, if you want to invoke a potential getter (which may or may not return the original property value in JavaScript), you should use the following:

```
id value = [self valueForKey:@"bar"];
```

In the above code example, if we had defined a method like the following below it would be invoked instead of retrieving our internal original property.

```
-(id)bar
{
    return @"456";
}
```

Properties don't have to be passed in the constructor for them to be internally set and your setter invoked.

```
foo.bar = 789;
```

When you invoke the property of a proxy, the following will happen:

• If you have defined a setter, it will be invoked.

• If you have not defined a setter, the property and value will be stored internally in the dynprops.

If you implement a setter, you should also manually store the property yourself in dynprops. You can do this by calling the following method:

```
-(void)setBar:(id)value
{
    [self replaceValue:value forKey:@"bar" notification:NO];
}
```

The third argument (notification) tells Titanium whether you want the setter to be invoked from this property change. Since we're already inside our setter, we don't want an infinite recursion so we pass NO.

Handling Events

Proxies automatically handle firing events and managing event listeners. Internally, when you call addEventListener or removeEventListener from JavaScript against a proxy instance, the proxy will automatically handle the code for managing the event listeners.

If you want to be notified upon an add or remove, you should override the methods:

```
-(void)_listenerAdded:(NSString*)type count:(int)count
{
}
-(void)_listenerRemoved:(NSString*)type count:(int)count
{
}
```

The _listenerAdded method will be invoked with the event name (type) and the total number of existing listener with the same type (including the new event listener). This is convenient, for example, when you would like to enable some action only once at least one listener is listening for the event. This can be useful for conserving system resources.

The _listenerRemoved method will be invoked with the event name (type) and the total number of remaining listeners with the same type (excluding the removed listener). This is useful when you want to cleanup system resources once no listeners are actively listening for events. To send an event to any event listener, you use the convenience method:

```
-(void)fireEvent:(NSString*)type withObject:(id)obj;
```

The first argument is the default and most common way to fire an event. The first argument (type) is the event name. The second argument (obj) is an NSDictionary of event properties. The second argument can also be nil if no additional event properties are needed. The event properties will be part of the event argument which is the first argument in all event functions.

For example:

```
-(void)mymethod
{
    NSDictionary *event = [NSDictionary
        dictionaryWithObjectAndKeys:@"foo",@"name",nil];
    [self fireEvent:@"foo" withObject:event];
}
```

In this example, we'd adding one additional event property named 'name' with the value of 'foo'.

In JavaScript, this would be retrieved like:

```
foo.addEventListener('foo',function(e)
{
    alert("name is "+e.name);
});
```

In addition to any additional event arguments passed, Titanium automatically provides the following built-in properties on all events fired:

• source — The source object (proxy) that fired the event

• type — The type of event

You can also check for the existing of a listener (recommended) before firing an event. This is done with the _hasListeners method.

```
(void)fireMyEvent
{
    if ([self _hasListeners:@"foo"])
    {
        // fire event
    }
}
```

It is generally recommended that you only construct your event object and fire the event if you have listeners to conserve processing power.

Memory Management

Proxies act like any Objective-C class and all memory management rules must be considered. When returning a new proxy instance from a method, you must autorelease the instance. Titanium will retain a reference to the proxy which maps to a reference to the resulting JavaScript variable reference. However, once the JavaScript variable is no longer referenceable, it will be released and your proxy will be sent the dealloc message.

You must take special care to retain/release your objects in Titanium just like you would in any Objective-C based programs. Improper retain/release will cause crashes and undesired results.

View Proxy

A View Proxy is a specialization of a Proxy that is used for Views — objects which interact with the UI to draw things on the screen.

The View Proxy holds the data (model) and acts like a controller for dispatching property changes and methods against the view. The View handles the internal logic for interacting with UIKit and handling UIKit events. The View is a model delegate to the View Proxy and, as long as referenced, receives property changes.

The View Proxy does not always retain a valid reference to a View. The View is only created on demand, as needed. Once the View is created and retained by the View Proxy, all property change events on the View Proxy will be forwarded to the View.

The View property methods are named using a special, required convention.

```
-(void)setFoo_:(id)value
{
}
```

Note: The suffix of the property name must end with an underscore. In the case of the View property methods, invocations against the View will always be on the UI main thread.

When you have properties in your View Proxy that should be handled by an active View, you don't need to define them in your View. Instead, you can define them using the syntax above in your View and they be automatically dispatched. In the event that your View is attached after properties have been set, the View Proxy will automatically forward all property change events (results in calling each setter method) to the View upon construction.

However, the View must have and dispatch any methods that it wants the View to also handle. This is normally done with the following example code:

```
(void)show:(id)args
{
    [[self view] performSelectorOnMainThread:@selector(show:)
       withObject:args waitUntilDone:NO];
}
```

You can also use a convenience macro that does the equivalent:

```
USE_VIEW_FOR_UI_METHOD(methodName)
```

The following code example is the same as the show method above:

```
USE_VIEW_FOR_UI_METHOD(show)
```

To get a generic reference to the View Proxy's view, the method 'view' can be called and you can cast the view result to your View class.

View

A View implementation must extend the TiUIView class. The TiUIView class extends UIView and provides Titanium specific functionality.

To define a new View, for example:

```
#import "TiUIView.h"
@interface ComTestFooView : TiUIView
{
}
@end
```

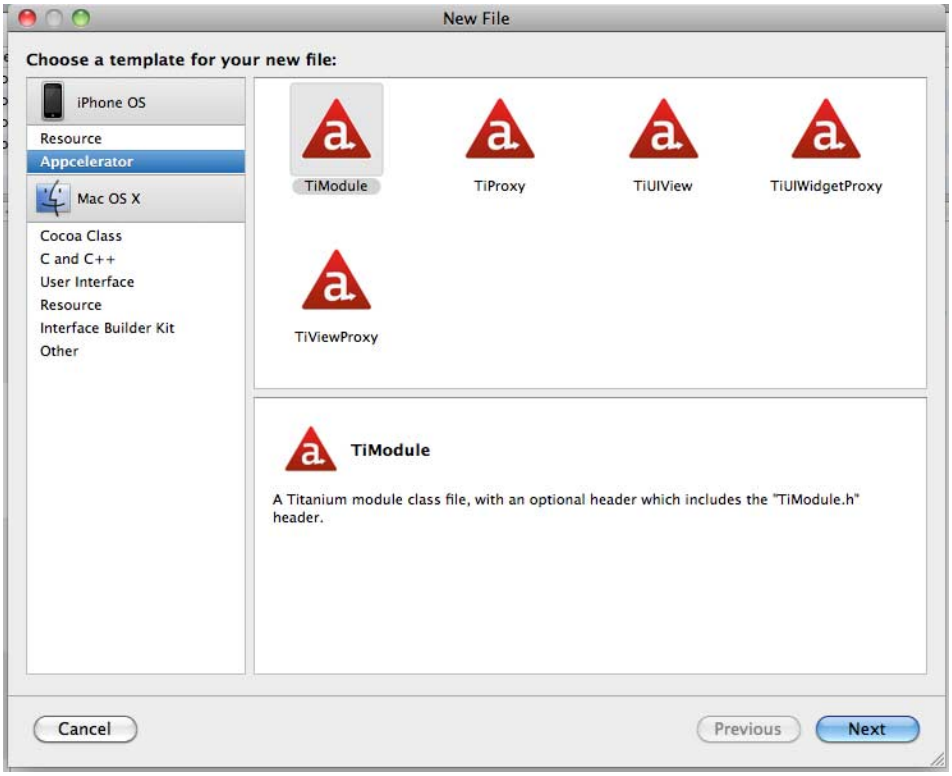
To define the View implementation, for example:

```
#import "ComTestFooView.h"

@implementation ComTestFooView

@end
```

The View will be attached to the UIView hierarchy as needed by the View Proxy. However, if you have subviews you will need to attach them to yourself (TiUIView) as necessary. This is typically done by assigning your view to an instance variable and keeping a reference to it and only creating and attaching when the reference is nil.

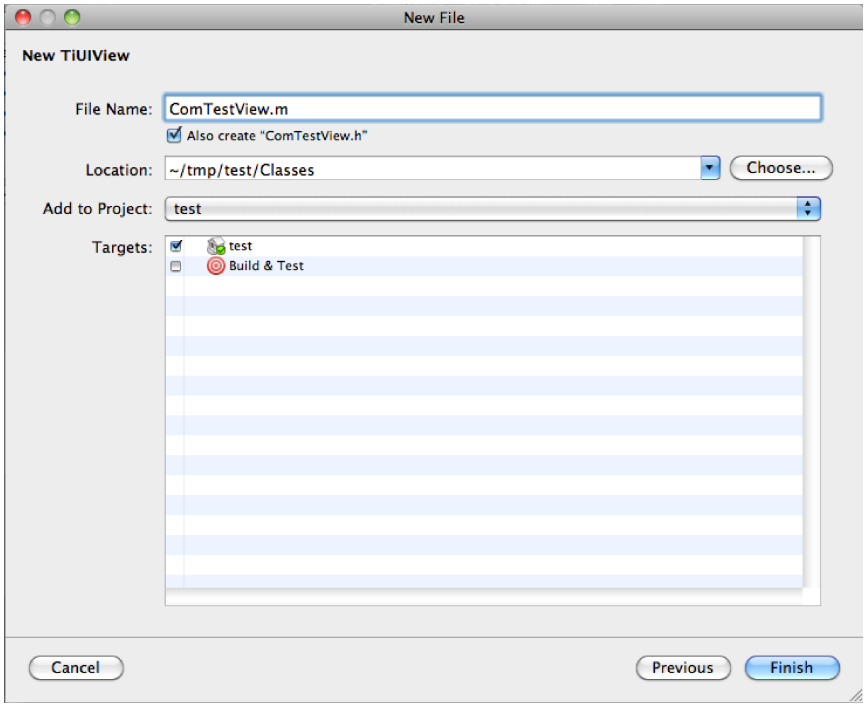
Creating a View and View Proxy

To create a view, in Xcode, select the menu File->New File... and the following File chooser dialog should be shown:

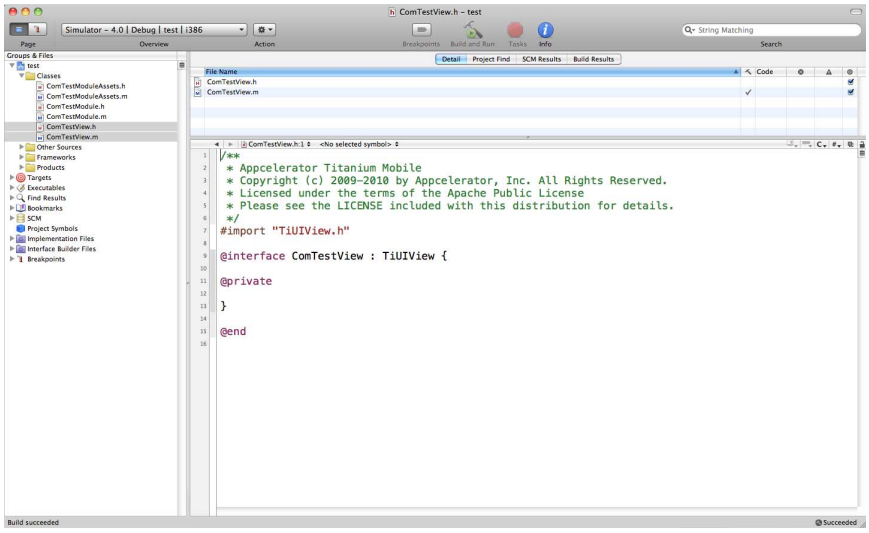


You should see an Appcelerator item under iPhone OS in the left hand window pane. Each of the icons in the right hand window pane provide the various types of files that can be quickly created from an Appcelerator-specific template. Choose "TiUIView" and click the "Next" button.

In the next dialog, you should enter the filename "ComTestView.m" and select you project directory and the "Classes" subdirectory. You can use the "Choose..." button to select using a file chooser.



At this point, you should see two new files in your Xcode project.



You'll probably want to change the default Copyright boilerplate to meet your own specific needs.

At this point, we're ready to change the code to make it do something useful.

In this example, we'll create a View that will simply attach a square as a child subview when the color property is set. Use the following code in ComTestView.h:

```
#import "TiUIView.h"
@interface ComTestView : TiUIView
{
    UIView *square;
}
@end
```

And then change ComTestView.m to the following code:

```
#import "ComTestView.h"
#import "TiUtils.h"

@implementation ComTestView

-(void)dealloc
{
    RELEASE_TO_NIL(square);
    [super dealloc];
}

-(UIView*)square
{
    if (square==nil)
    {
        square = [[UIView alloc] initWithFrame:[self frame]];
        [self addSubview:square];
    }
    return square;
}

-(void)frameSizeChanged:(CGRect)frame bounds:(CGRect)bounds
{
    if (square!=nil)
    {
        [TiUtils setView:square positionRect:bounds];
    }
}

-(void)setColor_:(id)color
{
    UIColor *c = [[TiUtils colorValue:color] _color];
    UIView *s = [self square];
    s.backgroundColor = c;
}

@end
```

We've created our view class that represents the visual part of our example. We'll now need to create a View-Proxy that knows how to map our JavaScript bindings into native code. Select the menu "File->New File..." again and create a "TiViewProxy".

Since this is a simple example, we can simply leave the template code as-is for now.

To test the new view, we need to write the application logic in JavaScript. Since we're testing a simple view, we can use the titanium command line tool to quickly test. Add the following JavaScript code to the file under your project examples directory in the app.js file:

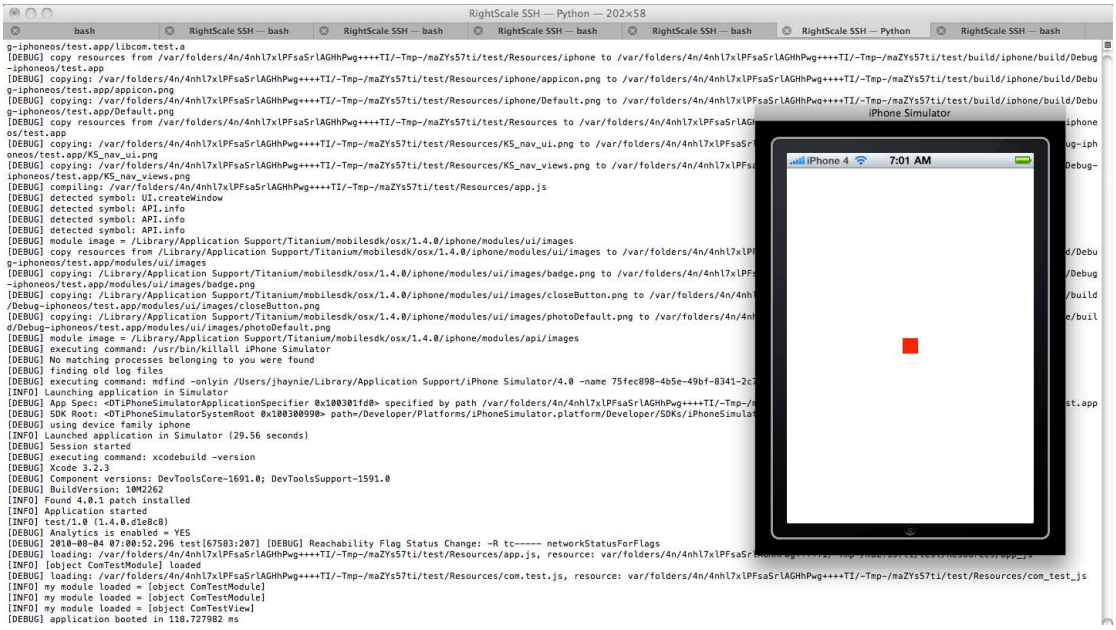
```
var my_module = require('com.test');

var foo = my_module.createView({
     "color":"red",
     "width":20,
     "height":20
});

window.add(foo);
```

This will load our module, create a view proxy (and View) and attach our View to the window as a 20 x 20 red square.

To run it, in the console, simply type the following from your module project directory:

```
> titanium run
```

You should see a lot of console logging as the project is compiled, packaged and the test application built and run. After about 30 seconds, you should see the following:



A few notes on our test code:

- In Titanium views, we typically recommend storing a reference to your main UIView as a class member variable (in this example, "square") and then load the view and attach it to yourself (your TiUIView) as needed. In this example, we only attach it when a property is set.

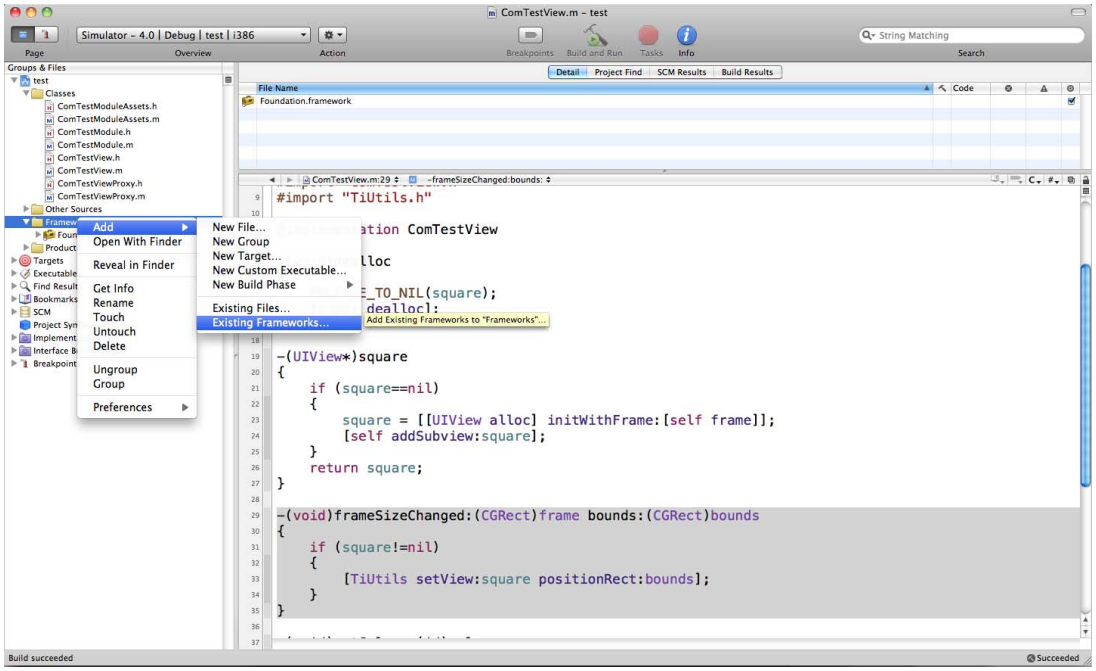
- You must implement a special method called "frameSizeChanged:bounds:". This method is called each time the frame/bounds/center changes within Titanium. Titanium has a special layout engine and you should use this method to signal changes to your frame and bounds. You should not override "setBounds:", "setFrame:" or "setCenter:". When you override this method, you must call the special method "setView:positionRect:" against the TiUtils helper class. This method will correctly layout your child view within the correct layout boundaries of the new bounds of your view.
- Since we used a View naming pattern for creation (the method "createView" against our module), we don't need to define a function in our module. Titanium will automatically do this for us when using this naming convention. However, if you instead wanted to create a method named "createFooView", you would need to define this method yourself in your module and create the proxy directly.

Adding Special Compiler Directives

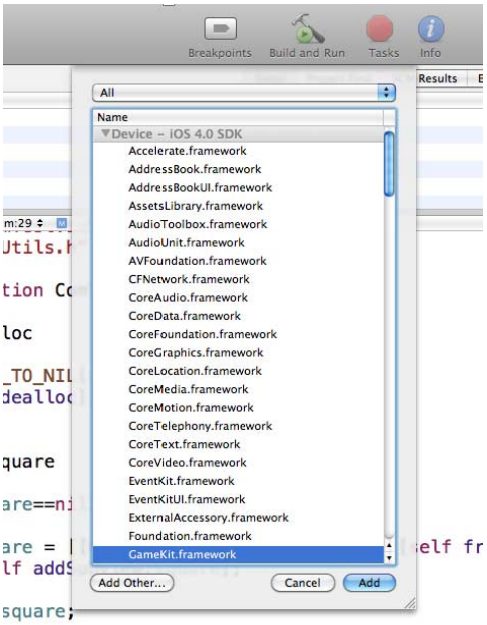
If your module needs a special Framework or other system library or special compiler directives, you can use the module's xcconfig file to define them. Titanium automatically creates a file named module.xcconfig and one named titanium.xcconfig. The titanium.xcconfig is used when compiling your module for packaging. However, module.xcconfig is used by the application compiler when the application is built and your module is referenced. This allows you to control the compiler directives used during this process, too.

To demonstrate this, let's walk through a simple example. Let's add a Framework.

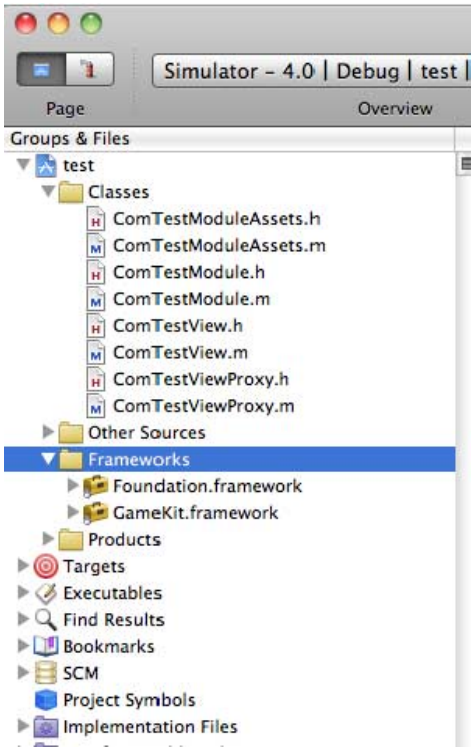
First, we'll need to add the Framework in Xcode. In this example, we'll simply add GameKit. Select Frameworks in your project folder, right click "Add -> Existing Frameworks" and select the menu item.



This will bring up the following dialog window:



And choose "GameKit.framework" and click the "Add" button. You should now see the new Framework reference under the Frameworks folder:



If you compile your project, you should now have no errors.

Now that we've added the Framework to the project, we still need to set it up so that the application compiler can also import this Framework during compile. To do that, edit the module.xcconfig and add the following line at the bottom:

```
OTHER_LDFLAGS=$(inherited) -framework GameKit
```

Now, we want to actually define a couple of methods that will actually use GameKit so we can demonstrate that it's working. In this example, we want to create a GameKit session and display the UI peer chooser.

Let's start with the application code. In Titanium when designing an API that support choosing a value from a dialog, we typically recommend passing an object with the function callbacks defined as values to the keys "success", "error" and "cancel".

In code, this would look like:

```
my_module.start('Jeff');

my_module.showChooser({
  success:function(e)
  {
    alert("peer choosen: "+e.peer);
  },
  cancel:function(e)
  {
    alert("cancelled");
  }
});
```

The first function (start) simply will start a GameKit session with my display name ("Jeff"). The second function (showChooser) will display a GameKit peer chooser UI dialog. We pass two function callbacks, "success" and "cancel". We want to call "success" when a peer is chosen and "cancel" when the user cancels the selection dialog. You might also want to define an "error" callback in the case that you encounter a setup error, etc.

Now, let's look at adding the native code. We'll do this code in our ComTestModule.m and ComTestModule.h files. Let's start with the implementation code. Paste the following after the methods already defined and before the "@end" statement:

```
#pragma mark Internal

-(void)dealloc
{
    if (session!=nil)
    {
        [session disconnectFromAllPeers];
        RELEASE_TO_NIL(session);
    }
    RELEASE_TO_NIL(peer);
    if (controller!=nil)
    {
        [controller performSelectorOnMainThread:@selector(dismiss) withObject:nil waitUn-
tilDone:NO];
        RELEASE_TO_NIL(controller);
    }
    [super dealloc];
}
```

```
#pragma Public APIs

-(void)start:(id)name
{
    ENSURE_SINGLE_ARG(name,NSString);
    NSString *sessionid = [TiUtils createUUID];
    [[GKSession alloc] initWithSessionID:sessionid
                           displayName:name
                           sessionMode:GKSessionModePeer];
}

-(void)stop:(id)args
{
    if (session!=nil)
    {
        [session disconnectFromAllPeers];
        RELEASE_TO_NIL(session);
    }
}

-(void)showChooser:(id)args
{
    ENSURE_UI_THREAD_1_ARG(args);
    ENSURE_SINGLE_ARG(args,NSDictionary);

    id success = [args objectForKey:@"success"];
    id cancel = [args objectForKey:@"cancel"];

    RELEASE_TO_NIL(successCallback);
    RELEASE_TO_NIL(cancelCallback);

    successCallback = [success retain];
    cancelCallback = [cancel retain];

    controller = [[GKPeerPickerController alloc] init];
    controller.delegate = self;
    [controller show];
}

#pragma mark Delegates

/* Notifies delegate that a connection type was chosen by the user.
 */
- (void)peerPickerController:(GKPeerPickerController *)picker didSelectConnection-
Type:(GKPeerPickerConnectionType)type
{
}
```

```
/* Notifies delegate that the connection type is requesting a GKSession
 */
- (GKSession *)peerPickerController:(GKPeerPickerController *)picker sessionForConnection-
Type:(GKPeerPickerConnectionType)type
{
    return session;
}

/* Notifies delegate that the peer was connected to a GKSession.
 */
- (void)peerPickerController:(GKPeerPickerController *)picker didConnectPeer:(NSString
*)peerID toSession:(GKSession *)session_
{
    RELEASE_TO_NIL(peer);
    peer = [session_ retain];

    if (successCallback)
    {
        NSDictionary *event = [NSDictionary dictionaryWithObjectsAnd-
Keys:peerID,@"peer",nil];
        [self _fireEventToListener:@"success" withObject:event listener:successCallback
thisObject:nil];
    }
}

/* Notifies delegate that the user cancelled the picker.
 */
- (void)peerPickerControllerDidCancel:(GKPeerPickerController *)picker
{
    if (cancelCallback!=nil)
    {
        [self _fireEventToListener:@"cancel" withObject:nil listener:cancelCallback this-
Object:nil];
    }
}
```

Let's examine this code. First, dealloc simply cleans up once the module is unloaded. We simply release our memory. Titanium defines a macro RELEASE_TO_NIL that will conveniently call release against an NSObject and also set the reference to nil. It's safe to call against a nil value.

The next method, start, will start our GKSession and retain our reference in our member variable, session. The start method uses our macro ENSURE_SINGLE_ARG to convert our argument and unpack it as the first argument into the variable "name".

The next method, stop, simply will disconnect our session from many connected peers and release the reference.

The next method, showChooser, will be responsible for displaying our UI dialog. Module methods run on a non-UI thread, however, all UIKit methods must be executed on the main UI thread. In this case, we use our macro ENSURE_UI_THREAD_1_ARG to ensure that the method is running on the UI thread. The second macro, ENSURE_SINGLE_ARG will convert the argument passed to a NSDictionary as the first argument.

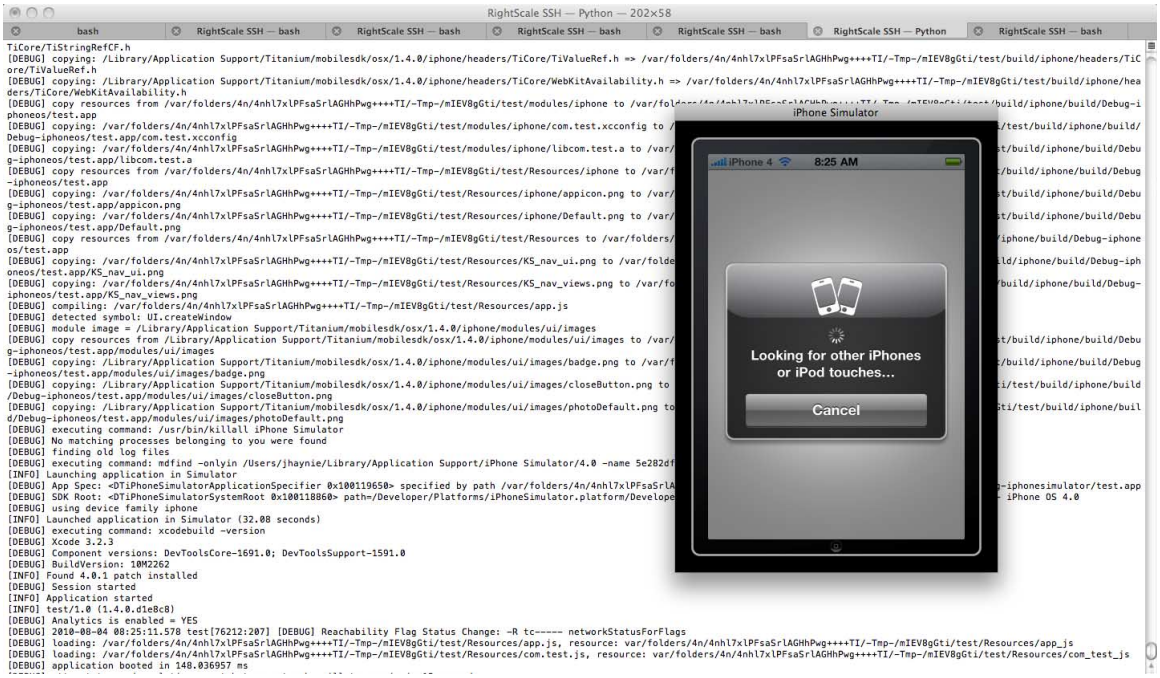
We then pull out the callbacks, release any memory potentially already held and retain our arguments as member variables of our class. We'll use those later when we want to invoke the functions from native. The remaining GameKit code simply creates a controller, assign ourself as a delegate and shows the chooser.

The remaining methods are optional GameKit chooser delegate methods that we implement. The "peerPickerController" method is called when a peer is chosen. In this case, we want to fire an event with some data to our "success" callback. Data passed in a JavaScript event should be passed as a NSDictionary of key/value pairs. Each of the keys will be properties of the event object passed to the callback function. We can call the proxy method "_fireEventToListener:withObject:listener:thisObject:" to invoke the callback method object and pass data to it. You can pass nil to "withObject" if you do not have any additional event data.

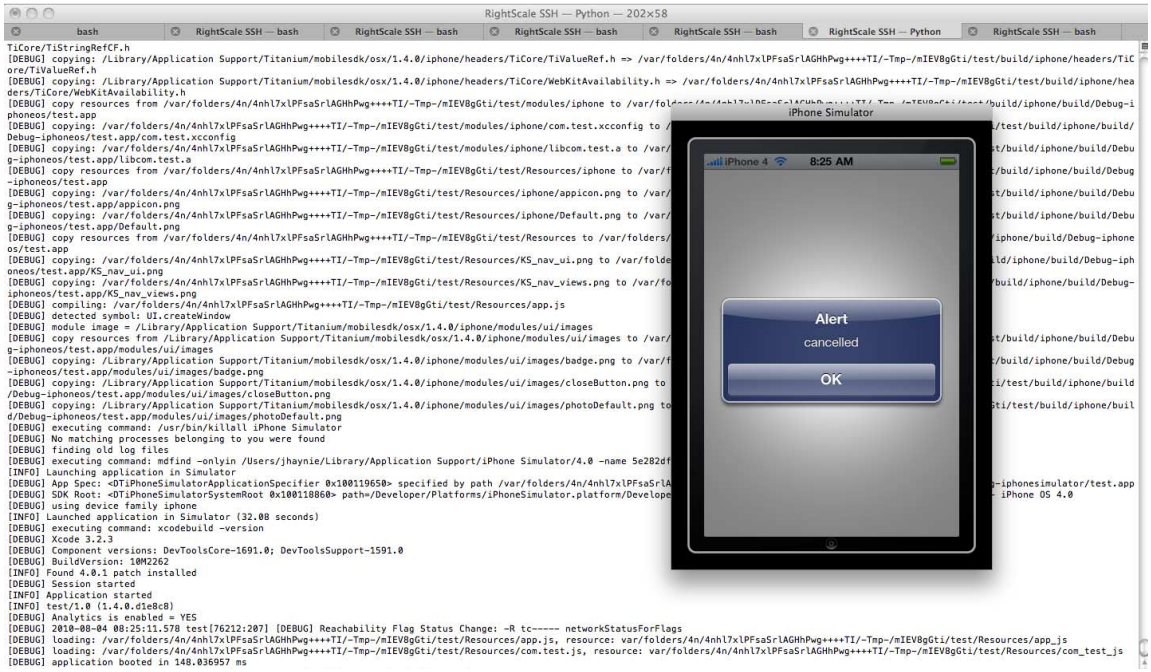
Let's now add the interface code in ComTestModule.h to complete the exercise:

```
#import "TiModule.h"
#import <GameKit/GameKit.h>

@interface ComTestModule : TiModule <GKPeerPickerControllerDelegate>
{
    GKSession *session;
    GKSession *peer;
    GKPeerPickerController *controller;
    KrollCallback *successCallback;
    KrollCallback *cancelCallback;
}

@end
```

Now that we've added our interface and implementation, compile in Xcode and make sure that we have no compiler errors. Assuming you have no errors, now execute titanium run from console and you should get something like the following:



Now, let's hit the "Cancel" button to invoke our cancel callback. You should see the following:



If you see the cancel, good news, it works!

Bundling Module Assets

To distribute module assets with your module distribution, you must place them in the "assets" directory of your project. Any assets within this folder (with the exception of JavaScript files, see below) will be distributed and copied into the folder pattern "module/<moduleid>" in the application bundle. You can then load them using this relative path from your Objective-C code. For example, assuming you had a module image named "foo.png". You could load that using the following example:

```
NSString *path = [NSString stringWithFormat:@"modules/%@/foo.png",[self moduleId]];
NSURL *url = [TiUtils toURL:path proxy:self];
UIImage *image = [TiUtils image:url proxy:self];
```

Building JavaScript Native Modules

One of the nice things about Titanium modules is that they can be either fully native or fully JavaScript. Up until now, we've talked primarily about fully native modules. Let's now talk about JavaScript modules.

Sometimes you want to create a native module but implement them written in JavaScript and distribute them as compiled modules. In Titanium, you would write your module code in a file named "<module_id>.js" in your "assets" directory. Using our sample project, this would be "com.test.js".

The module file must use the CommonJS format for declaring a module and it's exports. Let's start with a very simple example that defines one property and one function:

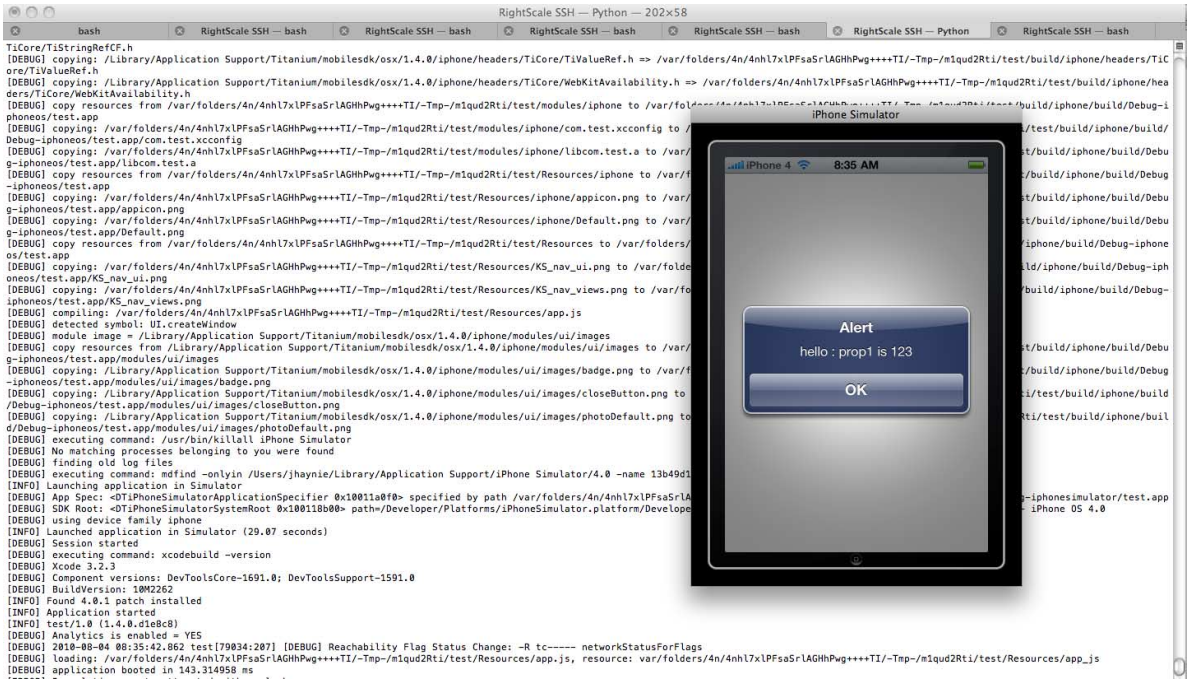
```
exports.prop1 = 123;
exports.func = function(v)
{
    return "hello : " + v;
};
```

Copy this code into a new file named "com.test.js" under the "assets" folder.

Now, let's modify our test application code to use the module. Remove our existing code changes in this file and replace with:

```
var my_module = require('com.test');
var result = my_module.func( "prop1 is " + my_module.prop1 );
alert(result);
```

Now, in the console, type "titanium run" to run the project. You should see the following:



CommonJS allows you to cleanly export one or more properties/functions by defining them to the special "exports" object (which is pre-defined before loading your module). Any methods or properties defined outside of the "exports" object will be considered module scoped and won't be visible outside of the module.

Step 3: Packaging your Module for Distribution

Titanium modules are created so that they can be easily built for distribution - either internally within your own distribution mechanism or externally, using the upcoming Titanium+Plus marketplace.

Describing your Module

Titanium module metadata is described in a special text file named "manifest". This file is a simple key/value property format. Here's an example of our test module:

```
#
# this is your module manifest and used by Titanium
# during compilation, packaging, distribution, etc.
#
version: 0.1
description: My module
author: Your Name
license: Specify your license
copyright: Copyright (c) 2010 by Your Company

# these should not be edited
name: test
moduleid: com.test
guid: 5bfdfd9b-12fe-42a4-bf2f-6ce7841fb4ae
platform: iphone
minsdk: 1.4.0
```

Before you distribute your module, you must edit this manifest and change a few values. Some of the values are pre-generated and should not be edited - these are noted with the comment before them. In the manifest file, any line starting with a hash character (#) is ignored. The following are the descriptions of each entry in the manifest:

Filename/Directory	Description/Purpose
version	This is the version of your module. You should change this value each time you make major changes and distribute them. Version should be in the dotted notation (X.Y.Z) and must not contain any spaces or non-number characters.
description	This is a human-readable description of your module. It should be short and suitable for display next to your module name.
author	This is a human-readable author name you want to display next to your module. It can simply be your personal name, such as "Jeff Haynie" or an organizational name such as "Appcelerator".
license	This is a human-readable name of your license. You should use a short description such as "Apache Public License" or "Commercial".
copyright	This is a human-readable copyright string for your module. For example, "Copyright (c) 2010 by Appcelerator, Inc."
name	This is a read-only name of your module that is generated when you created your project. You must not edit this value.

Filename/Directory	Description/Purpose
moduleid	This is a read-only module id of your module that is generated when you created your project. You should not edit this value. NOTE: you must generate a unique id. We recommend using your reverse-DNS company name + module_name as a pattern to guarantee uniqueness. The Titanium Marketplace will only allow unique module ids when distributing modules. If you must edit this value, you must also edit the value in your module implementation file.
guid	This is a read-only unique module id for your module that is generated when you created your project. You must not edit this value.
platform	This is a read-only platform target of your module that is generated when you created your project. You must not edit this value.
minsdk	The is a generated value for the minimum Titanium SDK version that was used when creating your module. This is currently not used and reserved for future use.

Distributing your Module Manually

To distribute your module manually, you can simply build your module using the "build.py" script and distributing the zip file directly. Modules can simply be copied into the /Library/Application\ Support/Titanium directory to be installed (once referenced and compiled in a project).

Note: As mentioned in the note on page 4, you'll need to determine whether your Titanium SDK was installed below ~/Library instead of /Library. If so, you would need to copy your module to ~/Library/Application\ Support/Titanium instead of /Library/Application\ Support/Titanium.

Distributing your Module through the Titanium+Plus Marketplace

To distribute your module through the Titanium+Plus Marketplace, you'll first need to package normally. Once you have tested your module locally and are ready to distribute it, you can then submit it to the marketplace for distribution. There are several prerequisites you'll need before you can distribute:

- You must have a valid Titanium developer account.
- You must have fully completed filling our your manifest values.
- You must have a valid license text in the LICENSE file in your project.
- You must have a valid documentation file in the index.md file in your documentation directory of your project.
- You must specify some additional metadata upon upload such as the price (which can be free).
- If you are charging for your module, you must establish a payment setup with Appcelerator so we can pay you.
- You must accept the Titanium+Plus Marketplace terms of service agreement.

Once you have upload your module and completed the necessary submission steps, your module will be queued for submission and availability in the marketplace directory.

The first time you submit a module, we will review your module for the basic requirements above. However, once approved, your module(s) will be immediately submitted without subsequent approval required.

Revision History

8/18/2010	Initial release