TRWS app

Ok we are going to begin to get into web services. Most apps now a days use data from the web. Whether that data comes from a private (business back end) or public (weather services) source, most apps use them.

So for this app we will take baby steps towards reading data off the web and parsing it (a fancy name for converting it to our iOS side) into usable iOS data for our app. Then we will worry about what to do with our data inside our app.

We have already experienced this with out Twitter app, but we will make it a bit more useful this time…as well as entertaining. With great power comes great responsibility, so this means you will have to delve into a bit of server programming. It will not be too complicated I promise, but it will be very enlightening. Thus one of the requisites for this tut is that you have access to some public web server (or you could use a webserver for mac such as MAMP, which you can download from here: http://www.mamp.info/en/mamp/.

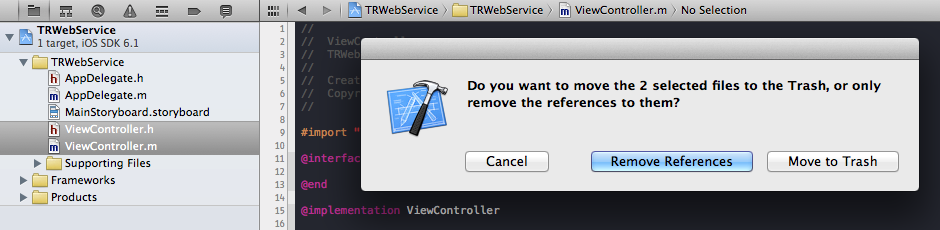
MOCK DATA – Example Free Web Service Flickr

I have a website setup where you can read data for testing purposes. In your web browser go to:

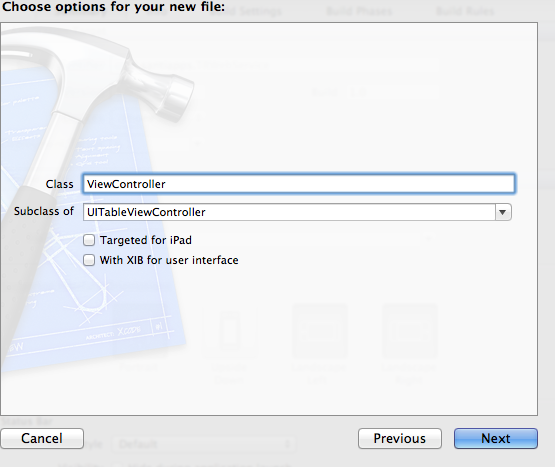
http://www.flickr.com/services/rest/?method=flickr.test.echo&format=json&api\_key=8038f7f7d7151ccbf6df2aa10b1b35ae&nojsoncallback=1

This displays a json response. It’s a little atypical because it has a leading string in there, but the rest (beginning with the first “(“) is typical. Ok so let’s try to fetch this data from within our app. Create a new Single View Project with ARC and Storyboards and call it TRWebService.

Select the Class Files for ViewController and click Delete. When you are prompted for this:



Go ahead and click on Move To Trash. When you select Remove References you are basically just removing the existing XCode references to those files, but the files are still there. So if you think you may need files later, just maybe not for this particular project, then Remove References and you can later move the actual files from the TRWebService XCode project folder to some other folder. If you are positive you will never use a file again, go ahead and Move To Trash. Now let’s add a New File and make it a tableview controller:



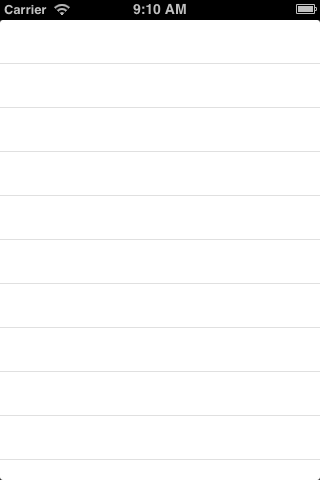
I did this for two reasons; first, it shows how to remove files and add files. Second, because this way we have XCode prepare a UITableViewController for us which gives us those boilerplate methods that come with all UITVCs instead of having to remember what they are called or how they are coded ☺. As to why I chose UITVC, I guess I just like scrolling cells!

UITVCs are nice because they come with some methods, which we can use to explore the viewcontroller lifecycle. Let’s look at some of these methods now:

-initWithStyle: This method is used to add a particular styling to tableviews. If we build and run the app now we get this…CRASH…hahaha! I forgot to replace the scene in storyboard. But I decided to leave the crash in the tutorial, it’s a good way to get used to them, because they are great for learning. Let’s analyze the error shall we:

**'NSInternalInconsistencyException', reason: '-[UITableViewController loadView] loaded the "2-view-3" nib but didn't get a UITableView.'**

XCode is telling us that UITableViewController’s loadView method tried to load a tableview from the nib, but didn’t get one. Of course, it didn’t. There isn’t one! Our app came with a storyboard which included a UIViewController scene, not a UITVC one. So let’s erase the UIViewController scene from storyboard. Select the scene until there is a blue rectangle around it and simply delete it. Now drag a UITVC from the Object Library onto the storyboard. Don’t forget to set its Class Type in the Identity Inspector to ViewController. Now let’s run the app again ☺. Great! Now we get this:



So in this method you can modify some style settings. It gets called only once, at the very beginning of the viewcontroller’s life. Other such methods that get called only once and at the very beginning are; initWithCoder, initWithFrame and viewDidLoad.

-viewDidLoad is a method that you will see quite a lot because it is used as a starting point for many other tasks. After all, once the view has loaded, the user can beging to see things on the screen.

-viewWillAppear, as well as its other closely related cousins, viewDidAppear, viewWillDisappear are methods which are called every time the view will do this or that. The difference is that views can appear many times but they are only loaded once. So for example, if you want the view to be refreshed each time the user loads an app running in the background, or switches tabs withing a UITabBarController etc…you want to use the methods called every time the view appears. If you refresh data inside viewDidLoad, of the other previous methods such as initWithStyle/Coder/Frame/NibName, the data will only be refreshed once!

-nOSIT, nORIS, cFRAIP and dSRAIP are tableview methods and I use those abbreviations to refer to them. They are not standard acronyms so be prepared for people to go “HUH!?”. numberOfSectionsInTableView, numberOfRowsInSection, cellForRowAtIndexPath and didSelectRowAtIndexPath are for the most part, required methods in a tableview. nORIS and nOSIT get called only once but cFRAIP gets called many many times, each time new cells appear in view for whatever reason, scrolling, loading of the view, deletion, editing etc.

Ok, enough about boilerplate methods of UITVC. Other VCs have different methods based upon what they do and which protocols they conform to.

So now we want to get some data off the web before we put it into our iOS app. We do this in three steps; create a URL, create a URLRequest and finally make a URLConnection. Since these are all Mac methods they are called NSURL, NSURLRequest and NSURLConnection. Our code will look something like this:

NSURL \*myURL = [NSURL URLWithString:@"http://www.flickr.com/services/rest/?method=flickr.test.echo&format=json&api\_key=8038f7f7d7151ccbf6df2aa10b1b35ae&nojsoncallback=1"];

NSURLRequest \*myRequest = [NSURLRequest requestWithURL:myURL];

NSURLConnection \*myConnection = [NSURLConnection connectionWithRequest:myRequest delegate:self];

Now the way this works is that it creates a connection which fetches and returns a response and some data if the response is positive. NSURLConnection sets the current view controller as its delegate so this viewcontroller must implement the delegate methods that will receive the response and the data. I just wanted to review the concept about not blocking the main thread. This is so important because what happens is that we want to have all of our data fetched by the time the user interacts with out viewcontrollers. So you would be tempted to go as far back as the AppDelegate’s applicationDidFinishLaunchingWithOptions method to put this code and the respective delegate methods. We can do this for this example because the response and data is returned so quickly that it doesn’t pose a problem. But what if the internet connection was missing or slow?

So let’s move it to our ViewController. Go ahead and make your viewDidLoad method look like this:

- (void)viewDidLoad{

[super viewDidLoad];

NSURL \*myURL = [NSURL URLWithString:@"http://www.flickr.com/services/rest/?method=flickr.test.echo&format=json&api\_key=8038f7f7d7151ccbf6df2aa10b1b35ae&nojsoncallback=1"];

NSURLRequest \*myRequest = [NSURLRequest requestWithURL:myURL];

NSURLConnection \*myConnection = [NSURLConnection connectionWithRequest:myRequest delegate:self];

}

and add these methods below:

- (void)connection:(NSURLConnection \*)connection didReceiveResponse:(NSURLResponse \*)response{

NSHTTPURLResponse \*httpResponse = (NSHTTPURLResponse\*) response;

int errorCode = httpResponse.statusCode;

NSString \*fileMIMEType = [[httpResponse MIMEType] lowercaseString];

NSLog(@"response is %d, %@", errorCode, fileMIMEType);

}

- (void)connection:(NSURLConnection \*)connection didReceiveData:(NSData \*)data{

NSLog(@"data is %@", data);

}

- (void)connection:(NSURLConnection \*)connection didFailWithError:(NSError \*)error {

// inform the user

NSLog(@"Connection failed! Error - %@ %@",

[error localizedDescription],

[[error userInfo] objectForKey:NSURLErrorFailingURLStringErrorKey]);

}

- (void)connectionDidFinishLoading:(NSURLConnection \*)connection {

// do something with the data

// receivedData is declared as a method instance elsewhere

NSLog(@"Succeeded!");

}

Now don’t flip just yet. They look intimidating but theyre quite easy to understand. If we receive a response then the method didReceiveResponse is called and we can do something with that response. Have you ever seen the infamous HTTP 404 error codes when loading a page that no longer exists? Well HTTP is a protocol much like any other and as such, when a resource is not found, it displays a standardized response, called response code, which is #404. We can get the code responded by the flickr server as well as the return file type and log them in our console.

If the response received is good, which by the way is known as an HTTP 200 code, the server begins to send data. That data is received in the didReceiveData method. If there is an error in the connection then the didFailWithError method is called. You can test this one out by turning off your internet connection and running the app. You will see the actual error returned by the Flickr server.

Finally, once all the data from the connection has been received, the connectionDidFinishLoading method is called and we can manipulate our data. So before we get to playing with data, let’s go ahead and run the app. Look in the console and you should get something like this:

**2013-04-10 09:56:22.336 TRWebService[1032:c07] response is 200, text/javascript**

**2013-04-10 09:56:22.336 TRWebService[1032:c07] data is <6a736f6e 466c6963 6b724170 69287b22 6d657468 6f64223a 7b225f63 6f6e7465 6e74223a 22666c69 636b722e 74657374 2e656368 6f227d2c 2022666f 726d6174 223a7b22 5f636f6e 74656e74 223a226a 736f6e22 7d2c2022 6170695f 6b657922 3a7b225f 636f6e74 656e7422 3a223830 33386637 66376437 31353163 63626636 64663261 61313062 31623335 6165227d 2c202273 74617422 3a226f6b 227d29>**

**2013-04-10 09:56:22.337 TRWebService[1032:c07] Succeeded!**

As you can see there is our response code and our file type. Now let’s try to take this data and put it into an iOS object we can use. We know the type returned is text, so we might be tempted to put this data into a string. To do this we need will change the didReceiveData method to this:

- (void)connection:(NSURLConnection \*)connection didReceiveData:(NSData \*)data{

NSLog(@"data is %@", data);

NSString \*myString = [[NSString alloc] initWithData:data encoding:NSUTF8StringEncoding];

NSLog(@"string is %@", myString);

}

Now run the app and you will get the same result you got in your web browser, in your console!

**string is {"method":{"\_content":"flickr.test.echo"}, "format":{"\_content":"json"}, "api\_key":{"\_content":"8038f7f7d7151ccbf6df2aa10b1b35ae"}, "nojsoncallback":{"\_content":"1"}, "stat":"ok"}**

Great! We are one step closer to getting our data. Now we just need to put it into an iOS object so we can pass it around and do whatever we want with it. So modify your method to this:

- (void)connection:(NSURLConnection \*)connection didReceiveData:(NSData \*)data{

NSLog(@"data is %@", data);

NSString \*myString = [[NSString alloc] initWithData:data encoding:NSUTF8StringEncoding];

NSLog(@"string is %@", myString);

NSError \*e = nil;

NSDictionary \*flickrDictionary = [NSJSONSerialization JSONObjectWithData:data options:NSJSONReadingMutableContainers error:&e];

NSLog(@"dictionary is %@", flickrDictionary);

}

And now you should get this as well:

**dictionary is {**

**"api\_key" = {**

**"\_content" = 8038f7f7d7151ccbf6df2aa10b1b35ae;**

**};**

**format = {**

**"\_content" = json;**

**};**

**method = {**

**"\_content" = "flickr.test.echo";**

**};**

**nojsoncallback = {**

**"\_content" = 1;**

**};**

**stat = ok;**

**}**

This means it is now an NSDictionary and we can use the data in our app. Not so painful after all! ☺

Ok so let’s first make our flickrDictionary into an ivar instead of a local variable of the didReceiveData method. Add it to the interface in the ViewController.m file, up top. Go ahead and add a mutable array as well so we can parse the dictionary and put the objects into the array for use in the tableview.

So now the top of your ViewController.m file will look like this:

#import "ViewController.h"

@interface ViewController () {

NSDictionary \*flickrDictionary;

NSMutableArray \*cFRAIPArray;}

@end

@implementation ViewController…

and you will change your connection:didReceiveData line to this:

flickrDictionary = [NSJSONSerialization JSONObjectWithData:data options:NSJSONReadingMutableContainers error:&e];

Now look carefully at what we have. It’s a dictionary of dictionaries. The first dictionary entry is “api\_key”, whose value is set to a “\_content” dictionary. The second entry is “format” whose key is also a “\_content” dictionary, so on and so forth. What we want to do is take the \_content value of each and put it into our array so we can use the array to populate our cells. Since we want to parse the dictionary only once its fully populated from the web fetch let’s change our connectionDidFinishLoading method to this:

- (void)connectionDidFinishLoading:(NSURLConnection \*)connection {

cFRAIPArray = [[NSMutableArray alloc] initWithCapacity:5];

for (NSString \*key in flickrDictionary) {

NSLog(@"the key is %@", flickrDictionary[key]);

id object = flickrDictionary[key]; // this will be apikey,format,method etc...

if ([object isKindOfClass:[NSDictionary class]]) { //first 4 are arrays-crsh-dicts

NSLog(@"object valueForKey is %@", [object valueForKey:@"\_content"]);

[cFRAIPArray addObject:[object valueForKey:@"\_content"]];

} else {

[cFRAIPArray addObject:flickrDictionary[key]];

}

}

NSLog(@"cfraiparray %@", cFRAIPArray);

[self.tableView reloadData];

}

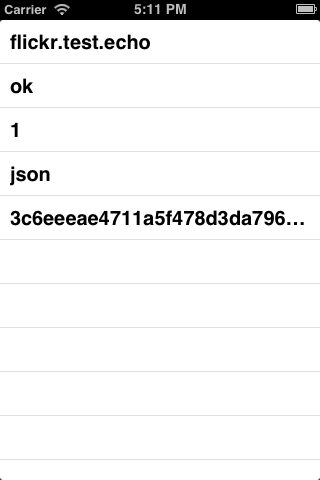
So we declare that our cFRAIPArray will contain 5 objects. We then loop through all its keys and put them into an id object. Now we have to test if those objects are dictionaries (as the first 4 are) or else, like the last entry, which is just a string. In the case the object IS an NSDictionary grab the value for the key called “\_content” and add it to our cFRAIPArray. Finally we reload the tableview.

Pay close attention to the subtle difference between each if/else code branch. In the first, we test if flickrDictionary’s key is a dictionary, and if it is, we put THAT key into a new dictionary called “object”. We then get the value for the “\_content” key of the OBJECT dictionary (NOT of the flickrDictionary) and add it to the cFRAIPArray. Then in the last entry, we simply take the flickDictionary key and add it to the cFRAIPArray.

Then in our cFRAIP method we simply say:

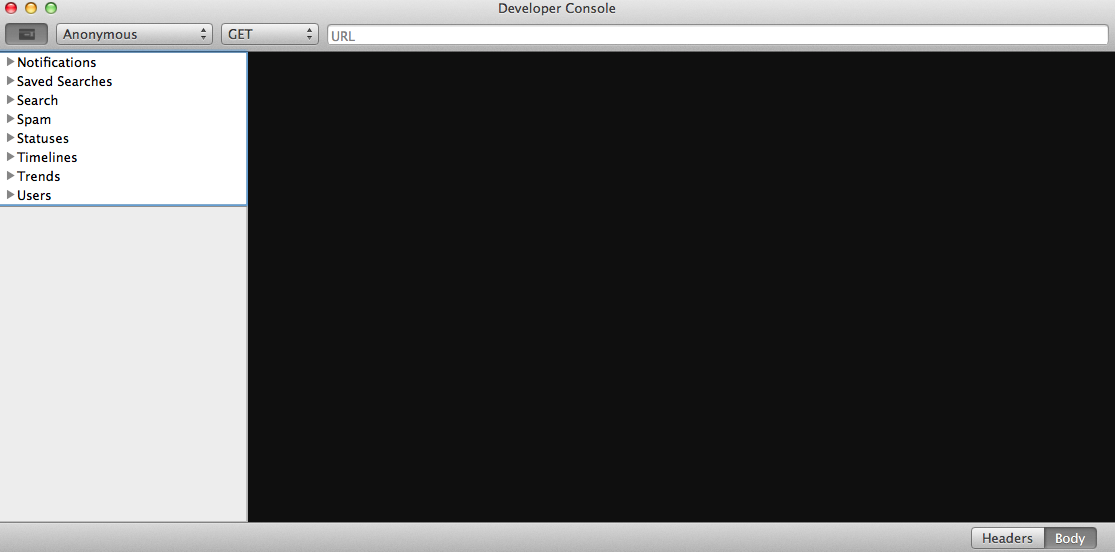
cell.textLabel.text = [cFRAIPArray objectAtIndex:indexPath.row];

Now Build & Run and you have a perfectly populated tableview.



Well that wasn’t so hard! There is a lot you can do with 3rd party APIs such as flickr, twitter and others. Some services such as Twitter even have a developer console you can access directly from your Twitter app. Simply open up the Twitter app on your Mac and go to Twitter>Preferences>Developer and check the Show Developer Menu.

Now from the Develop menu, which is added to the menu bar, select the only option, the Console option. You should get this window:



You can play around with these. Some you can use as Anonymous but others you would need to authenticate for.

Another option is to use web services such as Parse.com, which allow you to update data in many formats and then request that data via their custom calls. This is very convenient since you don’t need to worry about taking care of servers and uptime etc.

Yet another option is to create your own web service. This of course is the most complicated options but you can make it as complex or simple as you wish. This is also the most flexible option. In the next part we will roll our own so stay tuned!